ESCUELA SUPERIOR POLITÉCNICA DE CHIMBORAZO

FACULTAD DE INFORMÁTICA Y ELECTRÓNICA

ESCUELA DE INGENIERÍA ELECTRÓNICA EN CONTROL Y REDES INDUSTRIALES

ESTUDIO COMPARATIVO DE LOS LENGUAJES HDL Y SU APLICACIÓN EN LA IMPLEMENTACIÓN DEL LABORATORIO DE SISTEMAS DIGITALES AVANZADOS MEDIANTE FPGAS EN LA EIE-CRI.

TESIS DE GRADO

Previa a la obtención del título de

INGENIERO EN ELETRÓNICA CONTROL Y REDES INDUSTRIALES

PRESENTADO POR:

ALDO ALEJANDRO APARICIO OLMEDO

NEISSER FERNANDO PONLUISA MARCALLA

RIOBAMBA – ECUADOR
2014

AGRADECIMIENTO

Lograr culminar mi carrera profesional después de estos años ha sido gracias al esfuerzo y dedicación pero sobre todo fue gracias a Dios por eso le agradezco la fortaleza que me dio para alcanzar esta meta.

Agradezco a mi madre (Paulita) que ha sabido guiar y apoyar cada locura, cada proyecto, cada instante y cada momento de mi vida. A mi padre (papi ILA) que me ha enseñado a vivir a pesar de los obstáculos y adversidades que se presenten en la vida. A mis hermanos (Danilo y Vivian) mis sobrinos (Abraham, Bryan, Alan, Elías, Aron) pilares y motivadores para la consecución de esta meta.

A los docentes de la Escuela de Ingeniería Electrónica por haber depositado en mí todos sus conocimientos. En especial a la Ingeniera. Janet Arias y al ingeniero Wilson Baldeón por su apoyo incondicional y desinteresado en el desarrollo de esta tesis.

Finalmente te doy gracias mi dios por haber permitido entrar en mi vida a las personitas que han dado el último impulso en mí para la culminación de esta meta mi esposa (Delia Guillin) y mi primogénito (Jared Alejandro (Maysito)) los amores de mi vida.

ALDO APARICIO OLMEDO

AGRADECIMIENTO

Expreso mi agradecimiento a Dios por haberme regalado la vida y permitirme seguir avanzando en la vida con mis metas propuestas.

De igual forma agradezco a la Escuela Superior Politécnica de Chimborazo, a la Faculta de Informática y Electrónica, a la Escuela de Ingeniería Electrónica en Control y Redes Industriales, por haberme dado la oportunidad de formarme profesionalmente, que por medio de los Docentes ayudaron a cumplir una meta más en mi vida.

De manera especial nuestra Tutora la Ing. Janeth Arias por su apoyo incondicional y al Ing. Wilson Baldeon por brindarnos sugerencias e información para el desarrollo de nuestro trabajo de Investigación.

Sin dejar de agradecer a toda mi Familia y Amigos quienes siempre me apoyaron de manera incondicional.

NEISSER FERNANDO PONLUISA M.

DEDICATORIA

Esta tesis va dedicada con todo el amor del mundo para mi pequeño “Maysito” mi primogénito, mi MAYcriado para ti mi vida todo este esfuerzo y dedicación que tuve que poner para alcanzar esta meta.

Hoy, mañana y para el resto de esta vida seguirás siendo la razón por la cual cada proyecto de vida tenga su razón de ser.

ALDO APARICIO OLMEDO

DEDICATORIA

Este trabajo de tesis refleja el esfuerzo constante para llegar a ser un profesional, por ende dedico esta tesis:

A mi madre Eugenia Ponluisa quien ha sido mi pilar fundamental de apoyo incondicional, con su amor, sus consejos, palabras de aliento, por ser una mujer luchadora que ha permitido que Yo pueda llegar a mi Meta sin desfallecer. A mi Padrastro Raúl Burbano que de alguna manera me brindó su apoyo sin desmayar, a mi Tío Holger quien siempre estuvo ahí para brindarme su mano para alcanzar mi meta. A mis Hermanas Jessica por ser fuente de mi inspiración hacia mi meta, Leticia y Evelyn, a mis hermanos David, Elian y Heber quienes también me motivaron en mi camino a ser profesional.

A mis Amigas y Amigos por brindarme su amistad sincera, en los buenos y malos momentos que pase en mi vida estudiantil siendo ellos mi Familia fuera de mi casa, y a todas las personas que de alguna u otra aportaron en mi vida con sus buenos deseos.

NEISSER FERNANDO PONLUISA M.

FIRMAS DE RESPONSABLES Y NOTA

NOMBRE	**FIRMA**	**FECHA**
ING. IVÁN MENES **DECANO DE LA FACULTAD DE INFORMÁTICA Y ELECTRÓNICA**	**.....**	**……..**
ING. ALBERTO ARELLANO **DIRECTOR DE LA ESCUELA DE INGENIERÍA ELECTRÓNICA EN CONTROL Y REDES INDUSTRIALES**	**.....**	**……..**
ING. JANETH ARIAS **DIRECTOR DE TESIS**	**.....**	**……..**
ING. WILSON BALDEON **MIEMBRO DEL TRIBUNAL**	**.....**	**……..**
TLGO. CARLOS RODRÍGUEZ **DIRECTOR DPTO DOCUMENTACIÓN**	**.....**	**……..**

NOTA DE LA TESIS .....

RESPONSABILIDAD DE LOS AUTORES

"Nosotros, **ALDO ALEJANDRO APARICIO OLMEDO y NEISSER FERNANDO PONLUISA MARCALLA**, somos los responsables de las ideas, doctrinas y resultados expuestos en esta tesis de grado, y el patrimonio intelectual de la misma le pertenece a la Escuela Superior Politécnica de Chimborazo".

Aldo Alejandro Aparicio Olmedo

Neisser Fernando Ponluisa Marcalla

ÍNDICE DE ABREVIATURAS

ACRÓNIMO	SIGNIFICADO
AHDL	Lenguajes de Descripción de Hardware de Altera
AMS	Señal Analógica y Mixta.
ASCII	Estándar Americano de Intercambio de Información
ASICS	Circuito integrado de aplicación específica
BCD	Decimal Codificado en Binario.
CAD	Diseño asistido por computadora
CLB	Bloques de Lógica configurable
CPDL	Dispositivo Lógico Programable Complejo
CPU	Unidad Central De Proceso
DAE	Estándar de Atributos Derivados
DUT	Diseño a prueba
DSP	Procesador de Señal Digital
EDA	Diseño Electrónico Automático
EEPROM	Memoria de Solo Lectura eléctricamente programable y Borrable.
EIE-CRI	Escuela de Ingeniería Electrónica en Control y Redes Industriales.
ESPOCH	Escuela Superior Politécnica de Chimborazo.
FLASH	Derivada de la memoria EEPROM
FPGA	Arreglo de Compuertas Programable en el Campo
HDL	Lenguaje de Descripción de Hardware
HW	Hardware
IPS	Entrada de Circuito Programable
IEEE	Instituto de Ingenieros Eléctricos y Electrónicos.
IOB	Bloques de Entrada y Salida
JTAG	Joint Test Action Group
MEF	Máquina de Estado Finito
MIM	Metal-Insulator-Metal
NETS	Interconexiones
ONO	Oxide-Nitride-Oxide
OTP	Una Vez Programable

PAL	Arreglo Lógico Programable
PC	Computador Personal.
PCB	Tarjeta de circuito Impreso
PLD	Dispositivos Lógicos programables
PLICE	Elemento de Circuito de Interconexión Programable de la lógica
PROM	Memoria de Sólo Lectura Programable
RAM	La memoria de acceso aleatorio
ROM	Memoria de solo lectura
RTL	Nivel de registros de transferencia
SiO2	Dióxido de Silicio
SPI	Serial Periférico de Interfaz
SRAM	Memoria Estática de Acceso aleatorio
UDP	Primitivos Definido por el Usuario.
ULA	Unidad Aritmética Lógica
USB	Bus Universal en Serie
VCD	Verilog Change Dump
VLSI	Integración a muy alta escala
VHDL	Circuitos Integrados de muy Alta Velocidad.

ÍNDICE GENERAL

INDICE DE FIGURAS

INDICE DE TABLAS

INTRODUCCION

Los lenguajes de descripción de hardware o HDLs son lenguajes informáticos especializados que se utiliza para describir la estructura, diseño y operación de los circuitos electrónicos complejos y más comúnmente circuitos lógicos digitales.

Existen varios tipos de HDL como son los de nivel bajo, medio y alto pero esta investigación se basa en el estudio de los lenguajes HDL de nivel alto (VERILOG, VHDL), con el uso de la tarjetas FPGA para simular de manera física que los diseños digitales avanzados son correctos y fáciles de implementar sus circuitos electrónicos.

La tarjeta FPGA (Field Programmable Gate Array), son circuitos lógicos que permiten al usuario programarle a su manera, estos chips tienen unos componentes básicos que se pueden unir según las necesidades de diseño, esta configuración se encuentra almacenada en una memoria RAM interna y se carga desde el exterior del chip, lo cual necesita de herramientas que permitan cargar el programa, como el software de desarrollo y el dispositivo grabador. La configuración o programación de estos dispositivos se hace en tiempos relativamente (milisegundos).

El objetivo principal de este trabajo de investigación del ESTUDIO COMPARATIVO DE LOS LENGUAJES HDL Y SU APLICACIÓN EN LA IMPLEMENTACIÓN DEL LABORATORIO DE SISTEMAS DIGITALES AVANZADOS MEDIANTE FPGA EN LA EIE-CRI, es determinar el HDL más óptimo que permita un fácil ambiente de programación y el fortalecimiento de los conocimientos en la parte práctica en el estudiante de las asignaturas relacionadas con el diseño digital, dotando de equipos de aprendizaje didáctico e interactivo para realización de guías prácticas para los estudiantes de la EIE-CRI.

El presente trabajo de investigación contiene los siguientes capítulos:

En el capítulo I se presenta los antecedentes, justificación, objetivos, hipótesis, constituyen el marco referencial para el desarrollo de nuestra tesis de investigación.

En el capítulo II se definen conceptos específicos relacionados a la investigación como: HDL, VERILOG, VHDL.

En al capítulo III se describe las tarjetas FPGAs, su configuración en el software, proveedores.

En el capítulo IV se efectuara el diseño de módulos didácticos e interactivos para la realización de prácticas de programación y el uso de la parte física del diseño digital en la tarjeta FPGA.

En el capítulo V se realiza el análisis comparativo de los lenguajes HDL mediante el parámetro de la sintaxis de programación de cada uno de los lenguajes dando como resultado el lenguaje más óptimo para fortalecer los cocimientos teóricos y prácticos en los estudiantes del área de diseño digital.

CAPITULO I

MARCO REFERENCIAL

1.1 ANTECEDENTES

Hacia fines de los 80 mientras los diseños de circuitos integrados de aplicación específica (ASICS) crecían en complejidad los sistemas gráficos para describir estos empezaban a mostrar limitantes. El desarrollo la visualización, depuración y mantenimiento de estos sistemas era cada vez más complicada ya que mientras los diseños alcanzaban las 5000 compuertas lógicas, la cantidad de hojas de esquemas gráficos crecían de igual forma. El seguir planos esquemáticos con decenas de hojas daba paso a que surjan muchos errores y que el proceso se vuelva muy lento para depurar estos errores.

Es así que en 1985 una compañía llamada Automated Integrated Design Systems (sistemas automatizados de diseño integrado) renombrada después como Gateway Design Automation en 1986, introduce un nuevo producto llamado Verilog, era el primer simulador lógico que integraba un lenguaje de alto nivel y simulación a nivel de compuertas lógicas. En 1984 se crea VHDL por Texas Instrumenst.

HDL se utiliza para escribir especificaciones ejecutables de un cierto pedazo de hardware. Puede describir la operación del circuito, su diseño y organización, y pruebas para verificar su operación por medio de simulación. Se pueden hacer pruebas físicas mediante el uso de las FPGA. (1)

Las FPGA significan Field Programmable Gate Array en español Arreglos de Compuertas Programable en el Campo. Como su nombre indica, se trata de un dispositivo compuesto por una serie de bloques lógicos (puertas, registros, memorias, flip/flops, etc) programables, es decir, la interconexión entre estos bloques lógicos y su funcionalidad no viene predefinida sino que se puede programar y reprogramar.

Las FPGA tienen muchas aplicaciones en la actualidad que incluyen a los DSP (Digital Signal Processor), radio definido por software, sistemas aeroespaciales y de defensa, prototipos de los ASIC, sistemas de imágenes para medicina, etc.

Todo ingeniero de diseño debe conocer un Lenguaje Descriptivo de Hardware, ya que la tendencia tecnológica apunta en esos rumbos.

En la Escuela de Ingeniería Electrónica en Control y Redes Industriales hace un año y medio se incrementó la asignatura de Diseño Digital con el nombre de VHDL la cual comenzó con ciertas limitaciones, debido a falta de un laboratorio de sistemas digitales avanzado y siendo una materia optativa en la EIE-CRI. La asignatura de Diseño Digital en HDL está en pleno auge en países desarrollados, siendo de vital importancia para todo Ingeniero Electrónico tener conocimiento en las nuevas tendencias tecnológicas en desarrollo de lenguajes HDL y el uso de tarjetas FPGA con sus innumerables aplicaciones en la actualidad.

La asignatura siendo aún nueva en la EIE-CRI tiene una buena acogida por parte de los estudiantes lo que permite seguir desarrollándose a pesar de tener muchas limitaciones, para lo cual esta tesis de estudio comparativo de los lenguajes HDL permitirá escoger un mejor lenguaje para fortalecer los conocimientos teóricos dictados en clases y hacer prácticas usando las tarjetas FPGA que servirán para desarrollar nuevas aplicaciones de diseño digital para circuitos electrónicos desde niveles básicos hasta los más complejos.

1.2 JUSTIFICACION DEL PROYECTO DE TESIS

Debido a las limitaciones que posee la asignatura de diseño digital en la EIE-CRI se propone realizar un estudio comparativo de lenguajes HDL con la finalidad de desarrollar un laboratorio de sistemas digitales que ayude a fortalecer los conocimientos prácticos, ya que por ahora solo cuentan con desarrollo de habilidades de programación que representa la parte teórica de la asignatura.

En esta investigación proporcionará a los estudiantes del área de diseño digital que podrán llevar a cabo prácticas de circuitos electrónicos básico hasta llegar a los circuitos electrónicos más complejos. En esta investigación, se implementará arquitecturas de diseños digitales como sumadores, multiplicadores, restadores, etc. De una manera más fácil de implementar, no como hace años atrás se implementaba circuitos no difíciles de implementar, pero sí laboriosos de realizar, lo que en ocasiones daba como resultado diseños con pobre desempeño, debido a que apenas procesaban números binarios de pocos bits y con mayor costo de construcción de sus diseños digitales.

Con las tarjetas FPGA fortalecerán los conocimientos prácticos de los diseños digitales realizados en los lenguajes HDL permitiendo que los estudiantes de la asignatura de diseño digital diseñen, simulen, sinteticen y finalmente descarguen sus programas en la tarjeta

entrenadora (FPGA), para comprobar el correcto funcionamiento de la solución propuesta para un problema específico de Electrónica.

1.3 OBJETIVOS

1.3.1 OBJETIVO GENERAL

- Analizar y evaluar los lenguajes de descripción de hardware VHDL y Verilog mediante el parámetro de sintaxis de programación para determinar el mejor lenguaje, también desarrollar guías prácticas de manera adecuada para la implementación del laboratorio de sistemas digitales avanzados mediante FPGA en la EIE-CRI.

1.3.2 OBJETIVOS ESPECIFICOS

- Describir las principales características de los Lenguajes de descripción de Hardware, VHDL, VERILOG.
- Estudiar los dispositivos de lógica programable FPGA a utilizar en el laboratorio de sistemas digitales avanzados.
- Implementar los módulos guías de programación, para el uso de los estudiantes de la asignatura de diseño digital.
- Realizar pruebas físicas y de interconexión interna sobre FPGA.
- Comparar los lenguajes de descripción de hardware VHDL y Verilog.

1.4 HIPOTESIS

El estudio comparativo mediante la sintaxis de programación de los lenguajes VHDL y Verilog permitirá demostrar el lenguaje más óptimo enfocado a guías prácticas de sistemas digitales avanzados usando FPGA en la EIE-CRI.

CAPITULO II

LENGUAJES HDL

2.1 HDL

2.1.1 HISTORIA DE LOS LENGUAJES DE DESCRIPCION DE HARDWARE (HDL)

El aumento de la dificultad del diseño electrónico, así como los costos de desarrollo tecnológico y tiempo de lanzamiento de productos impulso al desarrollo de herramientas de diseño electrónico automático (EDA, Electronic Design Automation). Estas herramientas son de gran utilidad a los ingenieros en las diferentes etapas del diseño digital. Los lenguajes de descripción de hardware (HDL) son el principal componente que ha contribuido a complementar estas herramientas para el diseño digital.

Hacia fines de los 80 mientras los diseños de circuitos integrados de aplicación específica (ASICS) crecían en complejidad, los sistemas gráficos para describir estos empezaban a

mostrar limitantes. El desarrollo, visualización, depuración y mantenimiento de estos sistemas era cada vez más complicada ya que mientras los diseños alcanzaban las 5000 compuertas lógicas, la cantidad de hojas de esquemas gráficos crecían de igual forma. El seguir planos esquemáticos con decenas de hojas daba paso a que surjan muchos errores y que el proceso se vuelva muy lento para depurar estos errores.

Es así que en 1985 una compañía llamada Automated Integrated Design Systems (renombrada después como Gateway Design Automation en 1986), introduce un nuevo producto llamado Verilog, era el primer simulador lógico que integraba un lenguaje de alto nivel y simulación a nivel de compuertas lógicas. En 1984 se crea VHDL por Texas Instrumenst. (1)

2.1.2 DEFINICION

Un lenguaje HDL para descripción de hardware (HDL: Hardware Description Language) es una herramienta que nos permite describir el uso y la estructura de sistemas de diseño digital usando un esquema textual de codificación.

Usando HDL el diseñador puede describir la operación del sistema con diferentes niveles de abstracción:

- **Nivel de conmutadores** (switch level), utilizado para describir el circuito en términos de transistores y cables.
- **Nivel de compuertas** (gate level), para describir el circuito en términos de compuertas lógicas y elementos de almacenamiento como flip-flops.
- **Nivel de flujo de datos**, que describe el circuito en términos de flujo de datos entre registros.

- **Nivel algorítmico** o comportamental similar a un programa en un lenguaje de alto nivel como C. Incluye instrucciones de alto nivel tales como lazos, comandos de decisión y otros. (10)

2.1.3 ANALOGIA CON LENGUAJE DE DESCCRIPCION DE SOFTWARE

Al realiza una analogía entre los lenguajes de descripción de software y los lenguajes de descripción de hardware como se muestra en la figura II.1:

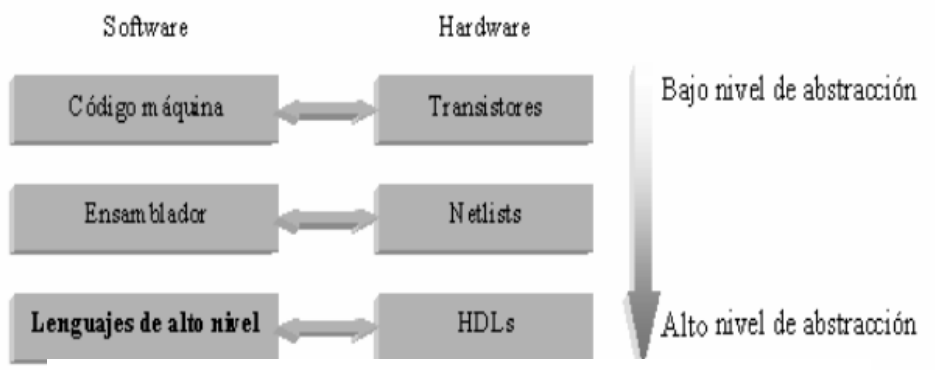


FIGURA II.1: Analogía con lenguaje de descripción de software
Fuente: *http://oretano.iele-ab.uclm.es/~miniesta/intro%20hdl.pdf*

2.1.4 CARACTERISTICAS DE LOS HDL

- Describe actividades que ocurren en forma simultánea (CONCURRENCIA).
- Permite describir módulos con acciones que serán evaluadas en forma secuencial (procedural), donde todo el módulo será visto como una acción concurrente más.
- Posibilita la construcción de una estructura jerárquica, donde es posible combinar descripciones estructurales y de flujo de datos con descripciones de comportamiento (BEHAVIOR)

- ❖ Permite modelizar el concepto de “tiempo”, fundamental para la descripción de sistemas electrónicos. (2)

2.1.5 ESQUEMA TIPICO DE UN DISEÑO DIGITAL

En la Figura II.2 se puede observar el esquema típico de diseño de un sistema digital. Las áreas sombreadas en la figura representan los procesos en el flujo de diseño, las áreas sin sombra los distintos niveles de representación del diseño.

Este esquema representa el típico flujo de diseño utilizado en un diseño basado en HDL. En todo diseño la especificación debe ser descrita primero, esta describe de forma abstracta la funcionalidad, interface y arquitectura global del diseño a ser escrito. En este punto no es necesario pensar de forma detallada la manera de la implementación del circuito.

La descripción en comportamiento se crea con fines de evaluar el diseño en términos de funcionalidad, desempeño y otras características de alto nivel. Esto se logra utilizando lenguajes de alto nivel (HDL).

La descripción a nivel de registros de transferencia (RTL) se crea manualmente en HDL. El diseñador debe describir el flujo de datos que implementará el circuito digital. Desde este punto en adelante el diseño se lo hace con la ayuda de una herramienta EDA (Electronic Design Automation). Generalmente estas herramientas son provistas por los fabricantes de los chips o por empresas aliadas.

Las herramientas de síntesis lógica convierten la descripción de RTL a lista de interconexión de Nets o puntos en un nivel de compuertas lógicas. Esta es una lista de interconexiones entre las compuertas lógicas.

Las herramientas de síntesis lógica aseguran que las conexiones entre nets cumplan con los tiempos, áreas y especificaciones de potencia.

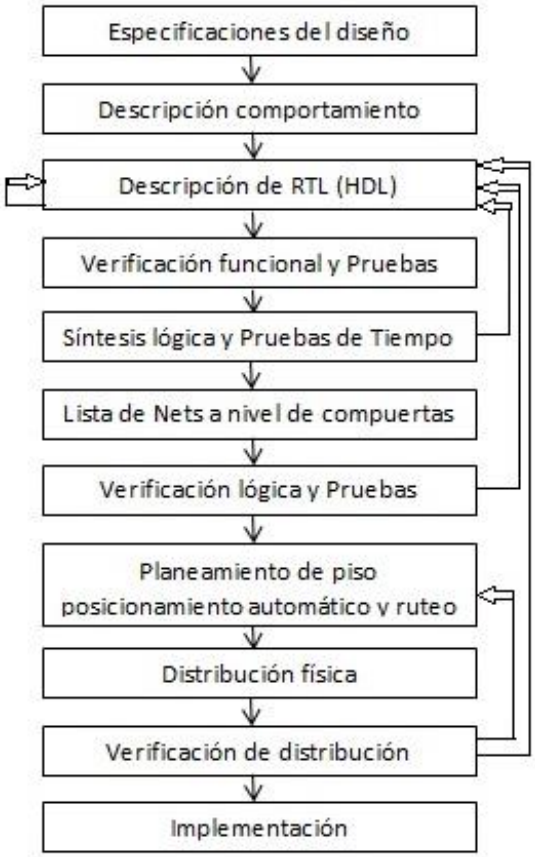


FIGURA II.2: Esquema típico de un diseño digital.

Fuente: *http://www15.online-onvert.com/downloadfile/7f35571353b5f5a26646887b151b2b8d*

La lista de Nets a nivel de compuertas es entonces alimentada a una herramienta automática de posicionamiento y ruteo la cual crea el "Layout" o distribución de las compuertas. Este layout es verificado y después implementado físicamente en un chip.

La mayor parte del tiempo de diseño se concentra en la optimización manual de la descripción del circuito a nivel de RTL. Una vez que esta está completa y ha satisfacción del diseñador, las herramientas EDA se utilizan para procesos posteriores. El diseño a nivel de RTL ha reducido los tiempos de diseño de años a unos pocos meses.

Con la aparición de herramientas de síntesis descriptiva se puede crear descripciones de RTL desde una descripción algorítmica o de comportamiento del circuito. Mientras estas herramientas mejoran su desempeño más se parece el diseño digital a una programación de

alto nivel de un computador. Los diseñadores implementan el algoritmo en un HDL a un nivel muy abstracto y las herramientas EDA ayudan a convertir e implementar este algoritmo en un chip o FPGA.

Es muy importante notar que mientras las herramientas EDA ayudan a acortar los periodos de diseño notablemente, estas dependen de las personas para que su trabajo sea eficiente. El diseñador debe entender bien los conceptos involucrados en el diseño digital de tal suerte que el resultado sea óptimo. (1)

2.1.6 TIPOS DE HDL

Existen tres tipos de lenguajes HDL:

1. **De bajo nivel:** Permiten definir un circuito a nivel de arquitectura (FlipFlops, compuertas básicas, ecuaciones lógicas) como son los lenguajes:
   - PALASM, CUPL, ABEL
2. **De nivel medio:** permiten definir un circuito en modo jerárquico, así como la generación condicional/iterativa de hardware; en ciertos permiten el uso de descripciones de comportamiento (funciones aritméticas, máquinas de estado), como es el lenguaje:
   - AHDL
3. **De alto nivel:** no sólo posibilitan mayor nivel de abstracción, sino que también son usados para la simulación, para la síntesis del generador de estímulos y el monitor de salidas (2), como son los lenguajes:

   -VHDL, VERILOG HDL

2.1.7 VENTAJAS DEL HDL

Aquí enumeramos las ventajas que tiene usar HDL para sistemas de diseño digital.

2.1.7.1 Herramientas de especificación

- Es posible su uso para la de especificación general de un sistema, tanto a nivel de hardware como de software.
- Permite describir el hardware, tanto a nivel de sistemas y subsistemas, como de componentes.

2.1.7.2 Herramientas de diseño

- Mejor documentación y facilidad de reuso
- Posibilidad de parametrización.
- Portabilidad de un diseño. Independencia tecnológica.

2.1.7.3 Herramientas de simulación

- Disponibilidad de modelos de distintos componentes de fabricantes variados en HDL normalizados.
- Facilidad para la generación de vectores de test complejos. [2]

2.1.7.4 Herramientas de sencillez

- Como la descripción se centra más en la funcionalidad que en la implementación, resulta más sencillo para una persona comprender qué función realiza el diseño a partir de una descripción HDL que a partir de un esquemático de interconexión de puertas, como por ejemplo:

```
begin
        Y <= (A and B) or C;
end a;
```

2.1.7.5 Ahorro de tiempo

- Facilita las corregir al diseño digital realizado debido a fallos de diseño o cambio de especificaciones.
- La existencia de herramientas comerciales automáticas (sintetizadores RTL) que permiten crear descripciones gate-level a partir de los modelos a nivel RTL
- Si bien, el diseño final no suele estar tan optimo como si lo hubiera realizado un humano, la mayoría de las veces es necesario sacrificar un mínimo en las prestaciones, para poder llevar a cabo el proyecto. Para ello se necesita la disponibilidad de dichas herramientas, las librerías de síntesis del fabricante y sus archivos de tecnología. (3)

2.1.8 INCONVENIENTES DEL USO HDLs

- Supone un esfuerzo de aprendizaje, ya que prácticamente se puede considerar como nueva metodología y hay q aprender cómo usarla.
- Necesaria la adquisición de nuevas herramientas:
  - Simuladores.
  - Sintetizadores de HDL, teniendo que mantener el resto de las herramientas para otras fases del diseño digital.
- El uso de estos lenguajes de descripción de hardware hace que involuntariamente se pierda un poco de control sobre el aspecto físico del diseño, dándole una mayor importancia a la funcionalidad de dicho diseño. (3)

2.2 LENGUAJE VHDL

2.2.1 INTRODUCCION

El lenguaje de descripción de hardware VHDL (Very High Speed Integrated Circuits) es un conjunto de recursos que permite describir el modelado de circuitos digitales con estructuras jerárquicas, desde puertas lógicas hasta algoritmos de programación de sistemas de diseño digital avanzados.

En la década de 1980, los rápidos avances en la tecnología de los circuitos integrados impulsaron el desarrollo de prácticas estándar de diseño para los circuitos digitales. VHDL se creó como parte de tal esfuerzo y se convirtió en el lenguaje estándar industrial para describir circuitos digitales, principalmente porque es un estándar oficial de la IEEE. En 1987 se adoptó la norma original para VHDL, llamada *IEEE* 1076. En 1993 se adoptó una norma revisada, la IEEE 1164.

En sus orígenes, VHDL tenía dos propósitos centrales. Primero, servía como lenguaje de documentación para describir la estructura de circuitos digitales complejos. Como estándar oficial del IEEE, ofreció una forma común de documentar los circuitos diseñados por varias personas.

Segundo, VHDL aportó funciones para modelar el comportamiento de un circuito digital, lo que permitió emplearlo como entrada para programas que entonces se usaban para simular la operación del circuito.

En años recientes, aparte de usarlo para documentación y simulación, VHDL también se volvió popular para el ingreso de diseño en sistemas CAD. Las herramientas CAD se utilizan para sintetizar el código de VHDL en una implementación de hardware del circuito descrito.

VHDL es un lenguaje complejo y refinado muy útil para la programación de sistemas de diseño digital HW, para ellos no se necesita ser un experto en el diseño VHDL, sino conocer su sintaxis de descripción para circuitos síncrono y asíncronos. (4) Para realizar esto debemos:

- Pensar en puertas y biestables, no en variables ni funciones.
- Evitar bucles combinacionales y relojes condicionados.
- Saber qué parte del circuito es combinacional y cuál secuencial.

En particular VHDL permite tanto una descripción de la estructura del circuito (descripción a partir de subcircuitos más sencillos), como la especificación de la funcionalidad de un circuito utilizando formas familiares a los lenguajes de programación.

La misión más importante de un lenguaje de descripción HW es que sea capaz de simular perfectamente el comportamiento lógico de un circuito sin que el programador necesite imponer restricciones. (5)

2.2.2 CARACTERISTICAS GENERALES DE VHDL

En los apartados que se exponen a continuación se revisan las características generales del lenguaje VHDL en referencia a su léxico, clases, formato y tipos de datos, así como aquellos conceptos significativos del lenguaje por su orientación para modelar hardware**.**

2.2.2.1 SINTAXIS DE ARCHIVOS VHDL

Una descripción VHDL puede constar de uno o varios archivos. Cada archivo debe estar constituido por el siguiente conjunto restringido de elementos del código ASCII-7:

TABLA II.I: ELEMENTOS RESTRINGIDOS

Mayúsculas	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Minúsculas	a b c d e f g h i j k l m n o p q r s t u v w x y z
Números	0123456789
Caracteres especiales	“ # & ‘ () * + , - . / : ; < = > _ | $ % @ ? { } \ ] [^ ` ~
Códigos de formato	Espacio (20), LF(0A), CR(0D), FF(0C), HT(09), VT(0B)

Con esos elementos se pueden confeccionar secuencias de caracteres que forman elementos fundamentales que no pueden fraccionarse en otros. En VHDL no se diferencia entre mayúsculas y minúsculas. Los archivos se componen de esos elementos fundamentales denominados elementos léxicos, de los que existen varios tipos:

a) SEPARADORES

Se permite un número variable de separadores entre elementos léxicos adyacentes, antes del primero y después del último en una línea de texto. Son los códigos de formato.

b) DELIMITADORES

Caracteres que se usan para separar elementos léxicos y que tienen algún significado específico en VHDL. Son los caracteres:

& ‘ ( ) * + , - . / : ; < = > |

Por ejemplo, todas las sentencias en VHDL terminan con el carácter delimitador;

Cuando se habla de delimitadores compuestos se hace referencia a secuencias de delimitadores con un significado especial en VHDL, por ejemplo:

=> ** := /= >= <= <> --

c) COMENTARIOS

Son elementos léxicos precedidos del delimitador -- y que finalizan con la línea donde están. Los comentarios son los últimos elementos léxicos de una línea. Pueden seguir a una sentencia VHDL existente en la línea o ser el único elemento léxico en la línea, por ejemplo:

código VHDL -- comentarios precedidos de doble guión

d) IDENTIFICADORES

Son palabras reservadas en VHDL o bien los nombres que el diseñador asigna a los objetos que utiliza en sus descripciones, por ejemplo, para nombrar constantes, variables, señales, identidades, arquitecturas, bloques, subprogramas, etc.

Los identificadores son secuencias de caracteres que empiezan por una letra, pueden incluir letras, números y caracteres de subrayado aislados o sin duplicar.

En el código VHDL no se diferencian letras mayúsculas y minúsculas, salvo que estén entre comillas, en cuyo caso son elementos diferentes de cadenas literales. Debe evitarse el uso de palabras acentuadas, incluso en los comentarios.

e) PALABRAS RESERVADAS

Son elementos léxicos con algún significado especial en las expresiones del lenguaje, por lo que no pueden ser utilizados fuera del contexto para el que están reservados. Estas palabras reservadas del lenguaje VHDL se muestran en la siguiente tabla II.II:

TABLA II.II: PALABRAS RESERVADAS EN VHDL

Abs	Else	linkage	procedure	Then
Access	Elsif	literal*	process	To
After	End	Loop	pure*	Transport
Alias	Entity	Map	Range	Type
All	Exit	Mod	record	unaffected*
And	File	Nand	register	Units
Architecture	For	New	reject*	Until
Array	function	Next	Rem	Use
Assert	generate	Nor	report	variable
Attribute	generic	Not	return	Wait
Begin	group*	Null	rol*	When
Block	guarded	Of	ror*	While
Body	if	On	select	With
Buffer	impure*	Open	severity	xnor*
Bus	in	Or	signal	Xor
Case	inertial*	others	shared*	
Component	inout	Out	Sla	
Configuration	is	package	sll*	
Constant	label	Port	sra*	
Disconnect	library	postponed*	srl*	
Downto			subtype	

Se han marcado con un asterisco las añadidas con la revisión VHDL-1993.

f) CARACTERES LITERALES

Están formados por un solo carácter entre dos delimitadores apóstrofo (‘ ’).

‘X’ ‘x’ ‘%’ ‘Z’ ‘z’ ‘;’

Nótese que al ir entre apóstrofos, aquí sí se distingue entre ‘X’ y ‘x’.

g) CADENAS LITERALES

Similar al anterior, con secuencias de caracteres entre delimitadores comillas (“ ”).

"esto es una cadena"

Si la cadena excede una línea, deberá fraccionarse y concatenarse por el delimitador &.

h) CADENAS BIT LITERALES

Son elementos léxicos formados por cadenas de dígitos de base hexadecimal (X), binaria (B), u octal (O), delimitados por comillas. Se pueden usar caracteres de subrayado aislados para separar cadenas largas de bits y mejorar la legibilidad del dato.

Independientemente de la base utilizada para expresar la cadena, VHDL lo interpreta siempre como el valor de la cadena de bits.

La interpretación de las cadenas binarias queda al criterio del diseñador: "1010" puede interpretarse como (+10) o como (-6), en función del contexto del dato. Dos ejemplos de cadenas bits equivalentes son:

B"11110000"	B"100011001"	X"F0"
B"1111_0000"	B"100_011_001"	O"431"

i) LITERALES NUMERICOS

Elementos léxicos con un valor numérico asociado. Pueden ser enteros o reales.

La base de numeración es decimal o especificada con número decimal, entre 2 y 16.

- Los reales tienen punto (.). La coma (,) no está permitida.
- Con notación exponencial se utiliza E o e. El exponente es siempre en base 10.
- Solo se permiten exponentes negativos en números reales.
- El primer carácter de un número real debe ser un número decimal.
- Se puede utilizar un carácter de subrayado para mejorar legibilidad.
- No se permiten espacios entre caracteres.

- Los números significativos siguen a la base especificada, van entre caracteres #.

- En base hexadecimal, los números superiores a 9 se expresan con A~F o a~f.

Ejemplos correctos:

7 2E5 007 2e1 262_144 3.1416 03.1416 0.31416E1 0.31416e+1
31.416 E-1 0#255# 16#fF# 2#1111_1111#

Ejemplos incorrectos:

1,000 .5 3E-2 5E -1 0.5 e0 16# fF # 2# 1111 1111 #

2.2.2.2 CLASES DE OBJETOS: DECLARACION E INICIALIZACION

En VHDL, un objeto es un contenedor o portador en el que se pueden almacenar valores de cierto tipo.

Todos los objetos tienen un tipo que determina la clase de valor que puede asignarse al objeto. Al declararlos, se les asigna nombre y tipo. El nombre debe cumplir las reglas mencionadas para ENTITYs y los tipos deben haber sido declarados previamente, salvo que sean predefinidos. Hay tres clases de objetos en VHDL:

a) CONSTANTES

Tienen un valor fijo, asignado al compilarlas, no alterable durante la simulación.

Deben ser declaradas con el siguiente formato:

CONSTANT nombre_de_constante**: TIPO** [: = valor inicial];

Ejemplos:

CONSTANT voltaje: **REAL: =** 5.0;
CONSTANT pulso: **TIME: =** 100 NS;
CONSTANT tres: **BIT_VECTOR:=** "0011";

b) VARIABLES

Son objetos normalmente utilizados como portadores temporales de valor alterable.

Solo son declarables en áreas secuenciales, como en procesos y subprogramas.

Deben declararse con el siguiente formato:

> **VARIABLE** nombre1,.. nombre_n : **TIPO** [RESTRICCIONES][:= VALOR INICIAL];

Ejemplos:

> **VARIABLE** temporal: **INTEGER RANGE** 0 **TO** 10 := 5;
> **VARIABLE** frecuencia, ganancia: **REAL RANGE** 1.0 **TO** 10.0;
> **VARIABLE** octeto: **BIT_VECTOR** (0 **TO** 7) := X"FF";

Las variables, a diferencia de las señales, no pueden pasar valores entre subprogramas.

c) SEÑALES

Son portadores del valor de un parámetro común a varias unidades de diseño, utilizándose en descripciones para comunicar cambios del parámetro, normalmente asociados a un tiempo de simulación. Su relación con el hardware es evidente y, a diferencia de las variables, solo pueden ser declaradas en áreas concurrentes.

Su formato de declaración y algunos ejemplos son como sigue:

> SIGNAL nombre: tipo [restricciones][:= valor inicial];SIGNAL clk: BIT := '0';
> SIGNAL databus: BIT_VECTOR (7 DOWNTO 0) := B"0000_1111" ;
> SIGNAL dos_bytes: octales (0 TO 1):= (B"0000_1111", B"1111_1111") ;

Las señales no deben declararse dentro de Procesos, pero se usan dentro de ellos y como objetos para pasarles valores.

2.2.2.3 ASIGNACION DE VALORES

Son expresiones utilizadas para cambiar el valor que tienen las variables y las señales.

a) ASIGNACION A VARIABLES

Las variables sustituyen inmediatamente el valor que tienen con el que se les asigna usando el delimitador compuesto (:=):

```
octeto := b”1111_1110”;
frecuencia := 2.0;
temporal := temporal + 1;
```

b) ASIGNACION A SEÑALES

La asignación simple de un valor a una señal indica el nuevo valor que será asignado en un instante futuro. Nótese que, a diferencia de la asignación inmediata que ocurre en variables, en las señales hay un retardo que, si no se especifica un valor determinado, tiene una duración infinitesimal denominada retardo delta.

El delimitador empleado para asignar valor a señales es (<=):

```
clk <= ‘0’;      clk <= ‘1’ after 5 ns;          databus <= x”0f” after 10 ns;
```

El valor asignado a la señal debe ser del tipo con el que la señal fue declarada.

Dada la importancia del concepto señal en cualquier circuito a modelar, las sentencias de asignación de valor a señales son elementos clave y con importantes matices.

2.2.2.4 TIPOS DE OBJETOS

Un tipo es un conjunto de valores con un nombre que identifica al tipo.

El tipo de un objeto especifica los valores que puede tener y limita las operaciones que pueden realizarse con sus datos a aquellas definidas para el tipo y su rango de valores.

En VHDL todos los datos que se utilizan tienen un tipo que es necesario especificar al declarar el objeto que lleva el valor. La compatibilidad entre tipos al asignar valores es un requisito en VHDL.

El paquete STANDARD, en la biblioteca STD, define tipos básicos como BIT ó INTEGER, pero el usuario de VHDL puede definir tipos y operadores específicos para la aplicación en uso, existiendo cuatro clases de tipos definibles en VHDL:

1. **Tipos escalares**

   Son tipos con valores simples y aislados.

   Existen tres tipos escalares: ENUMERADOS, NUMÉRICOS y FÍSICOS.

2. **Tipos compuestos**

   Tienen valores compuestos por conjuntos de valores. Son los ARRAYS y RECORDS.

3. **Tipos archivo**

   Son tipos especiales utilizados para definir objetos relacionados con la escritura y lectura de archivos de datos contenidos en el sistema donde se hace la simulación del modelo. Estos archivos, cuyo formato puede ser especial o de texto, pueden utilizarse como entradas para simulación del modelo o como archivos en los que se archivan las salidas del mismo. Por su relación al paquete TextIO.

a) TIPOS ENUMERADOS

Su declaración consiste en la enumeración simple de los valores que pueden tener. El paquete ESTÁNDAR contiene la declaración de los cuatro tipos de enumeración básicos de VDHL: boolean, bit, carácter y severity_level.

Para declaraciones a nivel flujo de datos, donde se describen buses y sus señales de control, es necesario manejar señales que no pueden ser descritas con lógica binaria, por lo que se podría definir, por ejemplo, un tipo escalar que usara cuatro valores lógicos:

TYPE cuad IS ('0','1','Z','X');

Donde '0' es el valor por defecto, 'Z' = 'alta impedancia' y 'X' = 'valor indefinido'.

Para poder utilizar el tipo cuad en las descripciones se puede:

- Declararlo en la parte declarativa de la arquitectura.
- Incluirlo en un paquete, al que se llamaría con USE para habilitarlo.

La 2ª opción es la más empleada normalmente en diseños.

b) TIPOS NUMERICOS

Denominación asignada a los tipos INTEGER y REAL del paquete STANDARD. Los rangos máximos están definidos en el paquete, pero el usuario puede definir o limitar los rangos de estos dos tipos en sus descripciones, de forma similar ha como lo hacen los subtipos NATURAL y POSITIVOS también definidos en el mismo paquete STANDARD, por ejemplo:

TYPE centenas **IS RANGE** 0 **TO** 100; TYPE probabilidad **IS RANGE** 0.0 **TO** 1.0

c) TIPOS FISICOS

Se pueden definir y declarar tipos relacionados con magnitudes físicas que son de uso frecuente en descripciones de hardware como resistencias, capacidades, frecuencia, potencia, etc. Un ejemplo se tiene en el tipo TIME, declarado en el paquete STANDARD por su utilidad en VHDL para modelar retardos. Para disponer de tipos físicos en modelos relacionados con resistencias y condensadores se podrían definir los tipos:

TYPE resistencia **IS RANGE** 0 **TO** 1E16
Units
mo;-- miliohms (unidad)
ohms = 1000 mo;
kohms = 1000 ohms;
END units;

TYPE capacidad **IS RANGE** 0 **TO** 1E16
Units
ffr;-- femto faradios (unidad)
pfr = 1000 ffr;
nfr = 1000 pfr;
END units;

d) ARRAYS

Son tipos compuestos por elementos homogéneos, con igual subtipo. Un ejemplo se tiene en los tipos STRING y BIT_VECTOR del PACKAGE.STD. STRING es un array de caracteres y BIT_VECTOR es un array de bits. Los arrays se puede considerar y definir como tipos indexables y multidimensionales, pudiendo dejarse sin definir sus dimensiones o rangos. El formato y algunos ejemplos son como sigue:

TYPE nombre **IS ARRAY** (rangos o dimensiones) **OF** tipo de los elementos;
TYPE cuad_nibble **IS ARRAY** (3 **downto** 0) **OF** cuad;
TYPE cuad_byte **IS ARRAY** (7 **downto** 0) **OF** cuad;
TYPE cuad_word **IS ARRAY** (15 **downto** 0) **OF** cuad;
TYPE cuad_2por4 **IS ARRAY** (1 **downto** 0, 0 **to** 3) **OF** cuad;

Nótese que los elementos del array pueden indexarse de dos formas, en las que se especifica el índice dentro del rango de (izquierda a derecha).

El array puede tener dimensiones abiertas, útil para describir diseños genéricos:

TYPE bit_vector **IS ARRAY** (natural range <>) **OF** bit; -- (ver PACKAGE.STD).

El delimitador compuesto <> indica que es un array con rango abierto, cuyos límites vendrán especificados posteriormente en la aplicación que use el tipo así declarado.

Pueden declararse tipos enumerados en relación a los índices de un array, por ejemplo:

TYPE orden **IS** (uno, dos, tres, cuatro, cinco);
TYPE orden_de_array **IS ARRAY** (orden **RANGE** uno **TO** cinco) **OF INTEGER**;

Que nos indica que los valores almacenados de “uno” a “cinco” son enteros, que el primer elemento del array tiene por nombre “uno” y que con él nos podemos referir a ese elemento, sea para asignarle un valor o para leerlo después.

Una vez que los array están declarados, o llamado el paquete en que se hallan incluidos, se puede asignar valores a los elementos de los arrays o utilizar estos elementos para asignar valores a objetos de tipo compatible, por ejemplo:

TYPE dispositivos **IS** (ram, micro, fpga, resistor, condensador, bobina);
TYPE array_activo **IS** ARRAY (**dispositivos** RANGE ram **TO** fpga) **OF INTEGER**;
TYPE activos_pasivos I**S ARRAY** (dispositivos **RANGE** <>) **OF INTEGER**;

Permiten asignar valores a señales declaradas como:

SIGNAL pasivos : activos_pasivos (range resistor to bobina);
SIGNAL databus : cuad_byte := "ZZZZZZZZ";

"ZZZZZZZZ" es el valor inicial forzado en la declaración, en lugar del valor “00000000” que tendría si no hubiese inicialización, ya que el valor por defecto del tipo cuad es '0'.

En los ejemplos que siguen pueden verse posibles formas de asignación de valores a señales relacionadas con los tipos array declarados anteriormente:

SIGNAL s1:cuad;	s1 <= s16 (3);
SIGNAL s4: cuad_nibble;	s4 <= s16 (2) & s16(3) & s16(4) & s16(5);
SIGNAL s8: cuad_byte;	s8 <= s16 (15 **downto** 8);
SIGNAL s16: cuad_word;	s16 (7 **downto** 0) <= s4 & s8(3 **downto** 0);

Además de la indexación por enteros, si la declaración del array especifica un tipo o tipos como indicación del rango discreto de ese array, la indexación está definida por una lista o tabla cuyos elementos estan indexados por filas y columnas ordenadas igual que los elementos de los tipos que describen los rangos discretos del nuevo tipo, al igual que los elementos a_{ij} de una matriz de i filas y j columnas:

TYPE cuad_2dim **IS ARRAY** (cuad,cuad) **OF** cuad;

Este TYPE estará definido por una tabla de (4 x 4), siendo el orden de filas y columnas el definido para el tipo cuad, es decir, ('0','1','Z','X'), ('0','1','Z','X').

La inicialización de los valores de una señal del tipo array multidimensional se efectúa dentro de paréntesis anidados, separados por comas, en el mismo orden o dirección en que están declarados los elementos del array.

```
SIGNAL s_2_4: cuad_2por4:= (('0','1','1','Z'), ('Z','0','0','0'));
```

Algunas herramientas de síntesis no soportan los arrays multidimensionales, por lo que se hace necesario descomponerlos en arrays unidimensionales, por ejemplo:

```
CONSTANT memdim : INTEGER := 3;
TYPE memdato IS ARRAY (0 TO memdim, 7 DOWNTO 0) OF BIT;
```

El array bidimensional memdato puede sustituirse por dos arrays unidimensionales como

```
TYPE palabras        IS ARRAY (7 DOWNTO 0) OF BIT;
TYPE memdato         IS ARRAY (0 TO memdim) OF palabras;
```

Con lo visto hasta aquí, y suponiendo que se tienen declarados los tipos anteriores apropiadamente en el paquete “modelos” sería posible modelar una sencilla memoria ROM como sigue:

```
USE WORK. modelos. ALL;
ENTITY memrom IS
        PORT (direc : IN INTEGER;
                datos : OUT palabras);
END memrom;
ARCHITECTURE minima OF memrom IS
CONSTANT romdatos : memdato := (    (‘0’, ‘0’, ’0’, ‘0’, ‘0’, ‘1’, ‘1’, ‘1’),
                                    (‘1’, ‘1’, ‘1’, ‘1’, ‘1’, ‘0’, ‘0’, ‘0’),
                                    (‘0’, ‘0’, ‘0’, ‘1’, ‘1’, ‘0’, ‘0’, ‘0’),
                                    (‘1’, ‘1’, ‘1’, ‘0’, ‘0’, ‘1’, ‘1’, ‘1’)
BEGIN
        datos <= romdatos( direc ) AFTER 25 ns;
END minima;
```

2.2.2.5 SUBTIPOS

Son subconjuntos de los valores de un tipo predefinido, al que se llama tipo base, del cual se obtiene fijando alguna restricción. Los distintos subtipos obtenidos de una misma base son totalmente compatibles entre sí y con su base, lo que permite usar con ellos todas las funciones definidas para el tipo base.

Todos los tipos son subtipos de ellos mismos, por lo que subtipo se usa a veces para referirse a todos los tipos y subtipos declarados de una misma base.

Los subtipos se declaran de forma similar a los tipos, indicando después de la palabra clave SUBTYPE el nombre que se le asigna, seguido de IS y del nombre del tipo base del que se derivan, por ejemplo:

```
SUBTYPE cuarteto IS bit_vector ( 3 DOWNTO 0);
SUBTYPE decadas IS integer RANGE 0 TO 9;
```

Otros ejemplos se tienen en el paquete STANDARD:

```
TYPE INTEGER IS RANGE -2147483648 TO 2147483647;
SUBTYPE NATURAL IS INTEGER RANGE 0 TO INTEGER'HIGH;
SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGH;
```

Existen tres razones principales para el uso de subtipos:

Limitar el número de “casos posibles” en ciertas sentencias, como en las de asignación selectiva de valor a señales o en la sentencia secuencial CASE.

Facilitar la creación de funciones de resolución para la asignación de valor a una señal con múltiples drivers.

Ser innecesaria la conversión de tipos al hacer asignaciones entre objetos con base común, por ejemplo, si tuviésemos definido un tipo como:

```
TYPE semibyte IS ARRAY (3 DOWNTO 0) OF BIT;
```

Para asignar el valor de un objeto de ese tipo a otro del tipo BIT_VECTOR, sería necesario hacer una conversión para pasar los elementos del tipo "semibyte" al tipo BIT_VECTOR, lo cual sería innecesario si el objeto driver fuese del tipo “cuarteto” definido arriba.

La asignación de valores de un subtipo a objetos de su tipo base es siempre posible, ya que es un subconjunto de la base. Por razón similar, un valor del tipo base puede no ser asignable a un objeto del subtipo, por ejemplo, si se tienen los subtipos:

SUBTYPE triple **IS** cuad **RANGE** '0' **TO** 'Z' ;
SUBTYPE binar **IS** cuad **RANGE** '0' **TO** '1' ;

Los objetos del subtipo "binar" son directamente asignables a objetos "triple"o "cuad" y viceversa, pero cuidando no provocar asignaciones que sobrepasen la dimensión definida para el subtipo destino, ya que se tendría un mensaje de error por asignación fuera de rango. Nótese, por contra, que el subtipo "binar" no es compatible con el tipo BIT definido en el paquete STANDARD. Aunque sus elementos son los mismos, en "binar" la base es "cuad".

2.2.2.6 TIPOS STD_LOGIC

Es evidente que para modelar o simular hardware digital, el tipo BIT del paquete

STANDARD es insuficiente ya que solo dispone de los valores 0 y 1. Por otra parte, si cada usuario define sus propios tipos y paquetes resultarán incompatibilidades que dificultarán los desarrollos compartidos o reutilizables. Para evitar estos inconvenientes, el IEEE desarrolló un nuevo paquete denominado STD_LOGIC_1164, donde se declara el tipo std_ulogic con nueve valores.

El paquete STD_LOGIC_1164 está dentro de una biblioteca denominada IEEE, por lo que, a diferencia de los tipos bit y bit_vector, que por estar en el paquete y biblioteca

STANDARD siempre están visibles, para usar los tipos std_ulogic es necesario iniciar la descripción con las sentencias:

libarary IEEE;

use IEEE. std_logic_1164.all;

Que, al hacer visibles la biblioteca y el paquete, permiten utilizar los tipos allí declarados.

El tipo base declarado en el paquete std_logic_1164 es el tipo std_ulogic, conocido también como MVL9 - multi value logic - y tiene sus nueve valores declarados como:

TYPE std_ulogic IS se muestran en la tabla II.III:

TABLA II.III: MULTI VALORES LÓGICOS

Variables	**Descripción**
'U'	No inicializado
'X'	Forzando desconocido
'0'	Forzando 0
'1'	Forzando 1
'Z'	Alta impedancia
'W'	Débil desconocido
'L'	Débil 0
'H'	Bajo 1
'-'	No importa

Y, al igual que en el paquete STANDARD, inmediatamente después se declara el tipo array

TYPE std_ulogic_vector **IS ARRAY** (NATURAL RANGE <>) **OF** std_ulogic;

El valor 'U', uninitialized, valor no inicializado, se emplea para comprobar que la lógica, particularmente la secuencial, ha sido inicializada correctamente y no tiene valores imprevistos. Por defecto, los dispositivos secuenciales estarán inicializados en 'U'. Cuando comienza la simulación deben inicializarse a '0' o '1', pero si por olvido o fallo no sucediese, el simulador avisará, ya que 'U' es el más fuerte y ningún otro le hará cambiar.

El valor ‘X’ se reserva para conflictos de señales motivados por asignación múltiple de valores diferentes a una misma señal, por lo que se diferencia de ‘U’ reservado únicamente para inicialización por defecto, en espera del valor impuesto por el diseñador. El valor ‘-’ se usa exclusivamente en aplicaciones de síntesis. Durante procesos de simulación se asimila a ‘Z’.

2.2.2.7 ATRIBUTOS PREDEFINIDOS DE TIPOS Y ARRAY'S

Los atributos son valores específicos tales como datos, funciones, tipos o rangos asociados con los objetos a que se refieren. Los atributos permiten un uso más eficiente del lenguaje al disponer de codificaciones adicionales que simplifican las descripciones

El formato tipo es:

nombre_ de_objeto ‘'nombre_ de_atributo

a) ATRIBUTOS DE TIPOS

En los tipos enumerados y en los numéricos, INTEGER O REAL, permiten encontrar el valor de los elementos, su posición, elemento que sigue, elemento que precede, etc. En estos atributos predefinidos se considera que al declarar un tipo enumerado se asocia a cada elemento un índice que indica su posición en la lista, siendo 0 para el primer elemento enumerado e incrementando en 1 para cada elemento sucesivo. Los ejemplos muestran los atributos predefinidos del tipo días que se declara como:

TYPE dias **IS** (lun, mar, mie, jue, vie, sab, dom);

Como se muestra en la siguiente tabla II.IV:

TABLA II.IV: ATRIBUTOS PREDEFINIDOS DE TIPO DIA

ATRIBUTOS	**DESCRIPCION**
BASE	Proporciona el nombre del tipomar'BASE = días
LEFT	Da el valor del elemento izquierdo del tipodias'LEFT = lun
RIGHT	Entrega el valor del elemento con índice más altodias'HIGH = dom
HIGH	Da el valor del elemento derecho del tipodias'RIGHT = dom
LOW	Entrega el valor del elemento con índice más bajodias'LOW = lun
POS	Suministra el índice del elemento referidodias'POS(jue) = 3
VAL	Suministra el valor del elemento con el índice dadodias'VAL(3) = jue
SUCC	Indica el valor de la base que sigue al referidodias'SUCC(jue) = vie
PRED	Indica el valor de la base que precede al referidodias'PRED(vie) = jue
LEFTOF	Valor del elemento a la izquierda del referidodias'LEFTOF(jue) = mie
RIGHTOF	Valor del elemento a la derecha del referidodias'RIGHTOF(lun) = mar

Otros ejemplos se obtienen del paquete STANDARD, a partir de los tipos CHARACTER y BIT:

character'POS(NUL) = 0 ; character'POS('0') = 48

bit'LEFT = '0'bit'RIGHT = '1' bit'POS('0') = 0bit'VAL(0) = '0'

Nótese que NUL va directamente, mientras que el carácter '0' va entre apóstrofos.

De forma similar, para tipos INTEGER como los subir y bajar definidos a continuación, se tienen los ejemplos siguientes, en los que deben apreciarse las diferencias que existen, se muestran en la tabla II.V:

TABLA II.V ATRIBUTO PREDEFINIDO DE TIPO ENTERO

TYPE subir **IS RANGE** 0 **TO** 9;	**TYPE** bajar **IS RANGE** 9 **DOWNTO** 0;
subir'LEFT = 0 subir'RIGHT =9 subir'PRED(5) = 4 subir'SUCC(5) = 6 subir'LOW = 0 subir'LEFTOF(3) = 2 subir'RIGHTOF(4) = 5	bajar'LEFT = 9 bajar'RIGHT = 0 bajar'PRED(5) =4 bajar'SUCC(5) = 6 bajar'LOW = 0 bajar'LEFTOF(3) = 4 bajar'RIGHTOF(4) = 3

b) ATRIBUTOS DE ARRAYS

Devuelven valores relativos al rango, longitud o límites del array. Dado que pueden existir arrays multidimensionales, los atributos pueden opcionalmente hacer referencia a una de las dimensiones del array. El formato general es:

nombre_de_array ' atributo[(n)]

Existen los siguientes atributos de arrays:

TABLA II.VI: ATRIBUTO PREDEFINIDO DE TIPO ARRAYS

ATRIBUTO	**DESCRIPCION**
array'LEFT(n)	Devuelve el límite izquierdo del subarray de índice n.
array'RIGHT(n)	Devuelve el límite derecho del subarray de índice n.
array'HIGH(n)	Devuelve el límite alto del subarray de índice n.
array'LOW(n)	Devuelve el límite bajo del subarray de índice n.
array'LENGTH(n)	Da la longitud del subarray de índice n.
array'RANGE(n)	Da el rango del subarray de índice n
array'REVERSE RANGE(n)	Da el rango inverso del subarray de índice n.

Para arrays definidos con rangos ascendentes:

array'LEFT = array'LOW
array'RIGHT = array'HIGH

Para arrays definidos con rangos descendentes:

array'LEFT = array'HIGH

array'RIGHT = array'LOW

Aplicados a un array bidimensional como

```
TYPE brillo is BIT_VECTOR( 7 DOWNTO 0 );
TYPE pantalla IS ARRAY ( 1 TO 640, 480 DOWNTO 1 ) OF brillo;
```

Se tendría como se muestra en la tabla II.VII:

TABLA II.VII: ATRIBUTO PREDEFINIDO DE TIPO ARRAYS BIDIMENSIONAL

ATRIBUTO	**RANGO**	**ATRIBUTO**	**RANGO**
pantalla'LEFT(1)	1	pantalla' LEFT(2)	480
pantalla'LOW(1)	1	pantalla'LOW (2)	1
pantalla'RIGHT(1)	640	pantalla'RIGHT(2)	1
pantalla'HIGH(1)	640	pantalla'HIGH(2)	480
pantalla'LENGTH(1)	640	pantalla'LENGTH(2)	480
pantalla'RANGE(1)	1 to 640	pantalla'RANGE(2)	480 DOWNTO 1
pantalla'REVERSE RANGE(1)	640 DOWNTO 1	pantalla'REVERSE RANGE(2)	1 TO 480

Un posible uso de los atributos 'LOW y 'HIGH de arrays se ve en el ejemplo siguiente:

```
FUNCTION convertir_a_entero ( dato: BIT_VECTOR ) RETURN INTEGER IS
    VARIABLE resultado : INTEGER := 0;
BEGIN
    FOR n IN dato'LOW TO dato'HIGH LOOP
        IF dato(n) = '1' THEN
            resultado := resultado + ( 2** n);
        END IF;
    END LOOP;
RETURN resultado;
END convertir_a_entero;
```

2.2.2.8 OPERADORES Y EXPRESIONES

Asociados en grupos en los que todos los operadores tienen igual prioridad y ordenados los grupos por relación de precedencia, se tiene la clasificación siguiente mostrada en la tabla II.VIII:

TABLA II.VIII OPERADORES Y EXPRESIONES

OPERADOR	**SIMBOLOGIA**
LÓGICOS	AND OR NAND NOR XOR
RELACIONALES	= /= < <= > >=
DE ADICIÓN	+ - &
DE SIGNO	+ -
DE MULTIPLICAR	* / mod rem
MISCELÁNEOS	** abs NOT

a) OPERADORES LOGICOS: AND OR NAND NOR XOR

Definen operaciones sobre variables o señales de tipo BIT o BOOLEANO y se obtienen resultados del mismo tipo que tengan los operandos. Dado que todos los operadores del grupo tienen igual precedencia, según las expresiones que se manejen, puede ser necesario el empleo de paréntesis para precisar la operación a realizar con los operandos, por ejemplo:

```
a := b AND c OR d AND e                    -- expresión illegal
```

Es una expresión ilegal que puede expresarse correctamente como:

```
a := ( b AND c) OR ( d AND e )
```

Nótese que los operadores AND y OR, por ser asociativos, permiten escribir indistintamente

```
a := b AND c AND          O BIEN          a := c AND d AND b
z := b OR c OR d          O BIEN          z := c OR b OR d
```

Mientras que los operadores NAND y NOR, no asociativos, requieren el empleo del operador NOT con más de dos operandos, por ejemplo, para representar la función NOR de tres entradas, no se puede formular como :

```
z <= x NOR y NOR w                -- es ilegal
```

ni tampoco como

```
z <= ( x NOR y ) NOR w            -- legal, pero no es la función NOR
```

debiendo expresarse como

```
z <= NOT ( x OR y OR w )
```

que justifica la mayor prioridad de NOT.

b) OPERADORES RELACIONALES: = /= < <= > >=

Deben comparar objetos del mismo tipo y el resultado de operadores de este tipo es siempre tipo booleano: cierto o falso, TRUE o FALSE. Ejemplos:

```
tempext := 10;tempint := 18;
    ( tempext < tempint )         -- resultado es TRUE
segnal_A <= 3;segnal_B <= 5;
    ( segnal_B <= segnal_A )      -- resultado es FALSE
```

En este último ejemplo puede verse el doble significado del delimitador compuesto <= , cuyo sentido correcto es deducido por el contexto de la expresión donde figure.

c) OPERADORES DE ADICION: + - &

+ y - son operadores aritméticos, para operandos de igual tipo: entero, real o físico.

El resultado es del mismo tipo que los operandos.

El operador de concatenacion, &, enlaza dos o más elementos del mismo tipo en una única unidad o elemento nuevo. Se utiliza en arrays unidimensionales de igual tipo base, obteniéndose un nuevo array resultado cuya longitud es la suma de las longitudes de los arrays operandos. Los ejemplos que siguen muestran los resultados de concatenar:

```
segnal_A <= '0';segnal_B <= '1';
segnal_A & segnal_Btiene por valor "01"
VARIABLE marca: STRING ( 1 TO 4 );:= "opel";
VARIABLE modelo: STRING ( 5 DOWNTO 1);:= "corsa";
VARIABLE color: STRING ( 1 TO 3 );:= "roj";
VARIABLE coche: STRING ( 1 TO 14 );
coche := marca & ' ' & modelo & ' ' & color; -- coche tiene valor "opel corsa roj"
```

Nótese en este ejemplo el uso de apóstrofo y dobles comillas, ya que el espacio ' ' es un carácter y los STRING son arrays de caracteres.

d) OPERADORES DE PRODUCTO: * / MOD REM

* Y /, multiplicación y división, están definidos para tipos entero y real.

Los objetos de tipo físico, puestos a la izquierda del operador * o /, pueden ser multiplicados o divididos por enteros o reales y el resultado será del tipo físico. Dos objetos del mismo tipo físico pueden ser divididos y el resultado es de tipo entero, como se demuestra en los ejemplos:

```
VARIABLE t1, t2, t3: TIME: = 10 ns; r1:= v1*r2;            -- r1 vale 50 kohms
VARIABLE r1, r2, r3: resistencia:= 10 kohms; t1:= 5.0*t2;          -- t1 vale 50.0 ns
VARIABLE v1, v2: entero:= 5;  v2:= r1/r2;                  -- v2 vale 5
                        r1 := r2*r3;  v1 := r1/v2;                  -- son ilegales
```

mod y rem están definidos solo para tipo INTEGER.

mod está definido por la expresiónA = B*N + (A mod B)

Donde (A mod B) tiene el signo de B y un valor absoluto menor que el de B. N es un número entero para el que se debe cumplir la expresión anterior.

rem está definido por la expresiónA = (A/B)*B + (A rem B)

Donde (A rem B) tiene el signo de A y un valor absoluto menor que el de B.

e) OPERADORES DE SIGNO: + -

Solamente afectan al signo de los operandos numéricos. Por la precedencia de los operadores, no pueden ir en las expresiones inmediatamente detrás de los operadores de multiplicación ni misceláneos, por lo que deberán ir entre paréntesis en tales casos.

```
TYPE temperatura is INTEGER RANGE -50 TO 100;
VARIABLE referencia : temperatura := 3 ;
SIGNAL radiador, congelador : temperatura;
congelador <= - referencia;-- congelador vale -3
referencia := -radiador;-- referencia vale 50
radiador/ -congelador-- es ilegal
radiador/(-congelador)-- es correcto
radiador** -congelador-- es ilegal
radiador**(-congelador)-- es correcto
```

f) OPERADORES MISCELANEOS: ** ABS NOT

** operador de exponenciación.

La base debe ser de tipo entero o real y el exponente de tipo entero.

El exponente solo puede ser negativo con bases de tipo real.

El resultado es del mismo tipo que la base.

ABS Operador unario que obtiene el valor absoluto del operando.

NOT Operador lógico NOT, para tipos BIT y booleanos. Algunos ejemplos son :

VARIABLE X,Y,Z: **INTEGER** := 10;

```
VARIABLE A: REAL := 5.0;
SIGNAL S1,S2,S3 : BIT := '1';

Z := X**2;                          --Z vale 100;
A := 0.5 ** (-1);                   -- A vale 2.0
Z := X**(A);                        --es ilegal
X := A** 3;                         --es ilegal
Y := X**( -2 );                     --es ilegal
X := ABS ( Z - ( Y**2)) ;           -- x vale 90
s1 <= s2 AND NOT s3                 -- s1 recibe el valor '0'
```

2.2.2.9 SOBRECARGA DE TIPOS Y OPERADORES

En VHDL el concepto de sobrecarga - overloading - se refiere a la posibilidad de tener objetos o elementos con más de un significado, por ejemplo, en el paquete STANDARD tenemos ya sobrecarga en los caracteres '0' y '1' que se definen en la segunda declaración del paquete como tipo BIT y en la tercera como tipo CHARACTER.

Igualmente, el usuario puede definir tipos sobrecargados, por ejemplo:

```
TYPE colores_primarios IS (rojo, verde, azul);
TYPE leds IS (rojo, ambar, amarillo, verde, azul);
```

De la misma forma, se puede tener sobrecarga de operadores, por ejemplo, si se tienen las señales w, x, y, z de tipo BIT, se podría escribir la sentencia:

```
w <= ( x AND y ) OR z;
```

Si posteriormente las mismas variables se tuviesen de tipo cuad, definido previamente, sería posible mantener la expresión anterior si antes se hubieran definido los operadores AND y OR para los tipos cuad, ya que estos operadores solo están predefinidos para los tipos BIT y BOOLEAN. La expansión o sobrecarga de los operadores AND y OR, se hace definiendo un nuevo operador, con igual nombre, por medio de una función que devuelve resultados de tipo cuad a partir de operandos cuad .

Las situaciones de sobrecarga se resuelven en VHDL por contexto, es decir, si los operandos son, por ejemplo, del tipo cuad y hay una expresión donde se llama a un subprograma existente para aplicarles la función AND u OR, el VHDL aplicará la función sobrecargada para dichos tipos y devolverá un resultado que será acorde con el que se espera que, puede ser de tipo BIT, INTEGER, cuad u otro cualquiera definido, siempre que exista la función o subprograma sobrecargado que corresponda al tipo de parámetros de entrada y salida especificados al invocarlo. Es decir, el concepto de sobrecarga se extiende a tipos, operadores y subprogramas.

Cuando se sobrecargan operadores, al declararlos y definirlos, estos van entre comillas, por ejemplo "AND" y "OR", para diferenciarlos de las funciones. La razón es que los operadores van entre los operandos mientras que los subprogramas se llaman con los parámetros entre paréntesis. En el paquete "modelos" se tienen algunos ejemplos de operadores sobrecargados.

A veces, existen expresiones en las que puede existir una ambigüedad que VHDL no puede resolver por la sobrecarga que tengan los objetos, por ejemplo, en la expresión

... IF ('Z' < 'X') ...

y en un contexto donde están visibles los tipos cuad, aparte de los CHARACTER, siempre visibles por ser declarados en el paquete STANDARD, se tendría una situación cuyo resultado depende del tipo que se considere para los elementos sobrecargados, por ejemplo:

('Z' < 'X') es VERDADERO si los elementos son del tipo cuad ('0', '1', 'Z', 'X')

('Z' < 'X') es FALSO si los elementos son del tipo CHARACTER

Para evitar situaciones como la del ejemplo, se deben cualificar las expresiones susceptibles de ambigüedad. Esto se hace por marcado de tipos o expresiones cualificadas que consisten en forzar el tipo de los elementos según el formato de expresión siguiente:

TIPO’ (expresión con tipos sobrecargados)

así , la expresión ambigua anterior dejaría de serlo expresándola como :

... **IF** (cuad’(‘Z’) < cuad’(‘X’)) [6]

2.2.3 ELEMENTOS BASICOS VHDL

El diseño digital está compuesto por las entradas, salidas y la relación que existe entre las mismas.

En VHDL se describe el aspecto exterior del diseño digital en las entradas y salidas con la relación de sus entradas con las salidas. El aspecto exterior, cuántos puertos de entrada y salida tenemos, es lo que denominaremos identidad (entity) y la descripción del comportamiento del circuito arquitectura (architecture), toda arquitectura está asociada a una identidad.

Las bibliotecas y paquetes que vamos a utilizar, deben ser declaradas lo que nos indicará que tipos de puertos y operadores podemos utilizar. Antes de empezar a declarar la identidad y la arquitectura se debe declarar las bibliotecas a usar.

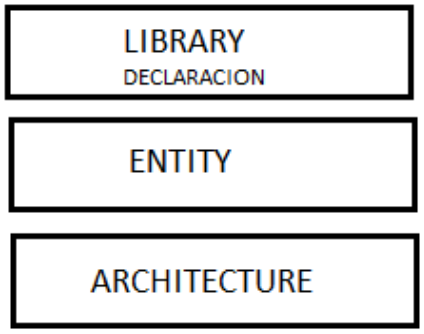


FIGURA II.3: Estructura del código VHDL

Fuente: *Los autores*

2.2.3.1 DECLARACION DE BIBLIOTECA (LIBRARY)

Las bibliotecas son recursos o paquetes que se disponen en la descripción del programa para cumplir el diseño digital.

En la librería de VHDL existen mínimos tres paquetes de librerías diferentes que se utilizan en el diseño:

- ieee.std_logic_1164 (Esta librería define la extensión de los operadores lógicos).
- standard (Esta librería está incluida lo que no es necesario llamarla).
- work (Esta librería es donde se almacenan datos en ejecución y al ser utilizada de esta manera no es necesaria llamarla).

Estas librerías nos permiten verificar que las tareas estén siendo escritas el código de forma correcta.

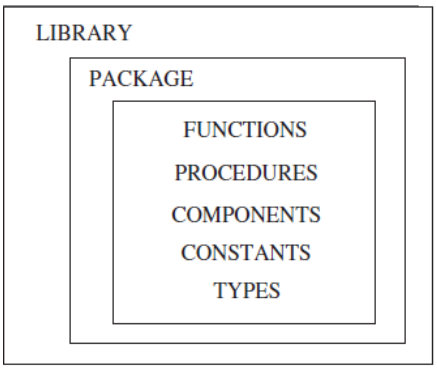


FIGURA II.4: Partes fundamentales de las librerías

Fuente: *Volnei A. Pedroni, Circuit Desing with VHDL*

2.2.3.2 DECLARACION DE LA INDENTIDAD (ENTITY)

Una entidad es la abstracción de un circuito, ya sea desde un complejo sistema electrónico o una simple puerta lógica. La entidad únicamente describe la forma externa del circuito, en ella se enumeran las entradas y las salidas del diseño. Una entidad es análoga a un símbolo

esquemático en los diagramas electrónicos, el cual describe las conexiones del dispositivo hacia el resto del diseño.

- Define externamente al circuito o subcircuito.
- Está constituido por nombre y número de puertos, tipos de datos de entrada y salida.
- Tiene toda la sintaxis general de la entidad necesaria para conectar el circuito a otros circuitos.

```
entity nombre is
        port (
                        port_name : signal_mode signal_tipe;
                        port_name : signal_mode signal_tipe;
                …);
end nombre;
```

Una entidad puede tener tres tipos de puertos:

- Entrada **on** (solo se puede leer y no se puede modificar su valor).
- Salida **out** (Solo se puede escribir y nunca tomar decisiones).
- Entrada-salida **inout o buffer** (Si es estrictamente necesario escribir sobre un puerto a la vez que se tiene que tener en cuenta su valor).

2.2.3.3 DECLARACION DE LA ARQUITECTURA (ARCHITECTURE)

Los pares de entidades y arquitecturas se utilizan para representar la descripción completa de un diseño. Una arquitectura describe el funcionamiento de la entidad a la que hace referencia, es decir, dentro de la arquitectura tendremos que describir el funcionamiento de la entidad a la que está asociada utilizando las sentencias y expresiones propias de VHDL.

- Define internamente el circuito.

- Señales internas, funciones, procedimientos, constantes.
- La descripción de la arquitectura puede ser la sintaxis general de programación.

```
architecture arch_name of entity_name is
        -- declaraciones de la arquitectura:
        -- tipos
        -- señales
        -- componentes
begin
        -- código de descripción
        -- instrucciones concurrentes
        -- ecuaciones booleanes
        -- componentes
        process (lista de sensibilidad)
        begin
         -- código de descripción
        end process;
end arch_name;
```

El código VHDL propiamente dicho se escribe dentro de arquitectura. Cada arquitectura va asociada a una entidad y se indica en la primera sentencia. A continuación, y antes de begin se definen en todas las variables (señales) internas que vas a necesitar para describir el comportamiento de nuestro circuito, se definen los tipos particulares que necesitamos utilizar y los componentes, otros circuitos ya definidos y compilados de los cuales conocemos su interfaz en VHDL (su entidad).

Desde begin hasta end escribiremos todas las sentencias propias de VHDL, pero no todas pueden utilizarse en cualquier parte del código. Así pues aquellas sentencias de VHDL que tengan definido un valor para cualquier valor de la entrada (y que nosotros denominamos sentencias concurrentes) podrán ir en cualquier parte del código pero fuera de la estructura process. [5]

2.2.4 ESTRUCTURA BASICA DE UN ARCHIVO FUENTE EN VHDL

Como hemos visto los modelos VHDL están formados por dos partes: la entidad (entity) y la arquitectura (architecture); es en esta última donde se escriben las sentencias que describen el comportamiento del circuito, a este modelo de programación en VHDL se le suele denominar behavioral.

```
architecture circuito of nombre is
-- señales
begin
-- sentencias concurrentes
process (lista de sensibilidad)
begin
-- sentencias secuenciales
-- sentencias condicionales
end process
end architecture circuito;
```

Dentro de la arquitectura se encuentra:

i) Tipos y señales intermedias necesarias para la descripción del comportamiento.

ii) Sentencias de asignación que deben realizarse siempre así como sentencias concurrentes.

iii) Uno a varios process que tienen en su interior sentencias condicionales y/o asignaciones a señales que dan lugar a hardware secuencial.

2.2.4.1 SENTENCIA PROCESS

VHDL presenta una estructura particular denominada process que define los límites de un dominio que se ejecutará (simulará) si y sólo si alguna de las señales de su lista de sensibilidad se ha modificado en el anterior paso de simulación.

Un process tiene la siguiente estructura:

```
process (lista_de_sensibilidad)
-- asignacion de variables
-- opcional no recomendable
begin
-- Sentenicas condicionales
-- Asignaciones
end process;
```

La sentencia process es una de las más utilizadas en programación con VHDL ya que tanto las sentencias condicionales como la descripción de HW secuencial se realiza dentro de él. Pero a la vez es, para aquellos que se acercan por primera vez a la simulación y síntesis con VHDL, el principal problema para un correcto diseño.

2.2.4.2 SENTENCIAS CONCURRENTES

Las sentencias concurrentes son sentencias condicionales que tiene al menos un valor por defecto para cuando no se cumplen ninguna de las condiciones. Aunque podría utilizarse una sentencia común como un if con obligación de else, los desarrolladores de VHDL han preferido utilizar dos sentencias particulares:

a) La primera es When – Else:

```
WHEN – ELSE
señal_a_modificar <=        valor_1 when condición_1 else
     valor_2 when condición_2 else
     ...
     valor_n when condición_n else
     valor_por defecto;
```

En esta sentencia siempre modificamos el valor de una misma señal, pero las condiciones pueden ser independientes (actuar sobre distintas señales cada una), dónde la colocación de las condiciones indica la preferencia de unas sobre otras, es decir, la condición 1 tiene preferencia sobre el resto, la condición 2 sobre todas menos la 1 y así sucesivamente.

b) La segunda es With – Select – When:

WITH – SELECT – WHEN

with señal_condición select
señal_a_modificar <= valor_1 **when** valor_1_señal_condición,
valor_2 **when** valor_2_señal_condición,
...
valor_n **when** valor_n_señal_condición,
valor_por_defecto **when** others;

Esta sentencia es menos general que la anterior. En este caso se modificará el valor de una señal dependiendo de los valores de una señal condición, aparecerán como máximo tantas líneas como valores posibles pueda tener la señal condición.

2.2.4.3 SENTENCIAS CONDICIONALES

VHDL permite utilizar otro tipo de sentencias condicionales más parecidas a los lenguajes de programación usados. Las sentencias condicionales más comunes en VHDL son las siguientes:

a) IF – THEN – ELSE

process (lista de sensibilidad)
begin
if condición **then**
-- asignaciones
elsif otra_condición **then**
-- asignaciones
else
-- asignaciones
end if;
end process;

La sentencia if-else permite cualquier tipo de combinación y encadenamiento, exactamente igual que ocurre en C o PASCAL o cualquier otro lenguaje de programación de alto nivel.

b) CASE – WHEN

```
process (lista de sensibilidad)
begin
case señal_condición is
    when valor_condición_l =>
    -- asignaciones
    …
    when valor_condición_n =>
    -- asignaciones
    when others =>
    -- asignaciones
end case;
end process;
```

Dentro de las asignaciones pueden parecer también sentencias if-else. Es necesario que aparezca en la estructura when others, pero no es necesario que tenga asignaciones, se puede dejar en blanco.

2.2.4.4 TIPOS DE BUCLES

Igual que en los lenguajes software, existen distintos tipos de bucles:

a) FOR – LOOP

```
process (lista de sensibilidad)
begin
for loop_var in range loop
  -- asignaciones
end loop;
end process;
```

Para el for range puede ser 0 to N o N downto 0.

b) WHILE – LOOP

```
process (lista de sensibilidad)
begin
while condición loop
```

```
    -- asignaciones
  end loop;
  end process;
```

El bucle tipo for está soportado si el rango del índice es estático (0 to N ó N downto 0, donde N posee siempre el mismo valor) y el cuerpo no contiene sentencias de espera. Los bucles de tipo while en general no están soportados.

2.2.4.5 DESCRIPCION ESTRUCTURAL

Esta descripción utiliza para la creación de la arquitectura de la entidad entidades descritas y compiladas previamente, de esta manera en VHDL podemos aprovechar diseños ya realizados, o realizar diseños sabiendo que se utilizarán en otros más complicados. Así se ahorra trabajo al diseñador-programador.

Se declaran los componentes que se van a utilizar y después, mediante los nombres de los nodos, se realizan las conexiones entre los puertos. Las descripciones estructurales son útiles cuando se trata de diseños jerárquicos botton-up.

```
architecture circuito of nombre is
  component subcircuito
     port (…);
  end component;
  -- señales
begin
  chip_i: subcurcuito port map (...);
  -- Se puede combinar con la descripción
  -- behavioral (por comportamiento)
end circuito;
```

2.3 LENGUAJE VERILOG

2.3.1 HISTORIA VERILOG

Verilog HDL se originó en la empresa Automated Integrated Design Systems (más tarde rebautizado como Gateway Design Automation) en 1985. La empresa fue una empresa privada fundada en ese momento por el Dr. Prabhu Goel, el inventor del algoritmo de generación de pruebas Podem. Verilog HDL fue diseñado por Phil Moorby, quien luego se convertiría en el jefe de diseño de Verilog-XL y el primer Fellow Corporativa de Cadence Design Systems. Pasarela Design Automation creció rápidamente con el éxito de Verilog-XL y finalmente fue adquirida por Cadence Design Systems, San Jose, CA en 1989.

Verilog fue inventado como lenguaje de simulación. El uso de Verilog para la síntesis era una idea de último momento.

Tal vez debido a estas presiones del mercado, Cadence Design Systems decidió iniciar el lenguaje Verilog al público en 1990, y por lo tanto OVI (Open Verilog International) nació. Verilog fue un lenguaje propietario, siendo propiedad de Cadence Design Systems. Cuando OVI se formó en 1991, una serie de pequeñas empresas comenzaron a trabajar en simuladores de Verilog, incluyendo Chronologic Simulación, Frontline Design Automation, y otros. El primero de ellos llegó al mercado en 1992, y ahora hay simuladores de Verilog maduros disponibles de muchas fuentes.

Como resultado, el mercado de Verilog ha crecido sustancialmente. El mercado de herramientas de Verilog relacionados en 1994 fue de más de $ 75.000.000, lo que es el lenguaje descripción de hardware más comercial en el mercado.

Un grupo de trabajo IEEE fue establecido en 1993 por el Sub-Comité de Diseño de automatización a Verilog como estándar IEEE 1364.

Como un estándar internacional, el mercado Verilog siguió creciendo. En 1998 el mercado de los simuladores de Verilog solo era más de $ 150 millones; continuando en ser el lenguaje de descripción de hardware líder en el mercado.

El grupo de trabajo IEEE publicó una norma revisada en marzo de 2002, conocida como IEEE 1364-2001. Errores de publicación significativas estropearon esta versión, y una versión revisada fue lanzado en 2003, conocido como IEEE 1364-2001 Revisión C.

Posteriormente, se formó un nuevo grupo de trabajo, IEEE P1800, para construir sobre el lenguaje IEEE 1364, junto con contribuciones adicionales de Accellera. A mediados de 2004 el comité de IEEE 1364 se disolvió, y en el mantenimiento del estándar fue considerado por el grupo de trabajo IEEE 1800. (7)

2.3.2 INTRODUCCION DE VERILOG

Es utilizado para diseñar sistemas electrónicos, verilog soporta el diseño, prueba e implementación de circuitos analógicos, digitales y de señales mixta a diferentes niveles de complejidad.

Posee una sintaxis similar a la del lenguaje de programación C. El lenguaje tiene un preprocesador como C, y la mayoría de palabras reservadas de control como while, if entre otras son similares.

Verilog es uno de los HDL más utilizados y permite descripciones abstractas y representaciones en bajo nivel es decir puede describir sistemas digitales en base a compuertas, e incluso en base a transistores.

Permite que en un diseño se puedan usar diferentes niveles de descripción de sistemas digitales en un mismo ambiente, las diferentes descripciones pueden ser simuladas para

verificar el funcionamiento y además pueden ser sintetizadas es decir traducidas a la interconexión de componentes básicas de un dispositivo programable.

Además permite la descripción estructural del diseño en base a componentes básicas, y descripciones más abstractas que se enfocan en la conducta del sistema. La conducta puede describirse mediante expresiones lógicas y también empleando procedimientos.

Un diseño basado en descripciones funcionales o de comportamiento puede resultar lento y de gran tamaño. Las descripciones en niveles estructurales permiten un ahorro de los circuitos lógicos para maximizar la velocidad, minimizar el tamaño y más bajo costo.

2.3.3 NIVELES DE ABSTRACCION EN VERILOG

Verilog soporta el diseño de un circuito a diferentes niveles de abstracción, entre los que destacan:

a) **Nivel de puerta.-** Corresponde a una descripción a bajo nivel del diseño, también denominada modelo estructural. El diseñador describe el diseño mediante el uso de primitivas lógicas (AND, OR, NOT, etc...), conexiones lógicas y añadiendo las propiedades de tiempo de las diferentes primitivas. Todas las señales son discretas, pudiendo tomar únicamente los valores '0', '1', 'X' o 'Z' (siendo 'X' estado indefinido y 'Z' estado de alta impedancia).

b) **Nivel de transferencia de registro o nivel RTL**.- Los diseños descritos a nivel RTL especifican las características de un circuito mediante operaciones y la transferencia de datos entre registros. Mediante el uso de especificaciones de tiempo las operaciones se realizan en instantes determinados. La especificación de un diseño a nivel RTL le confiere la propiedad de diseño sintetizable, por lo que hoy en

día una moderna definición de diseño a nivel RTL es todo código sintetizable se denomina código RTL.

c) **Nivel de comportamiento (Behavioral).-** La principal característica de este nivel es su total independencia de la estructura del diseño. El diseñador, más que definir la estructura, define el comportamiento del diseño. En este nivel, el diseño se define mediante algoritmos en paralelo. Cada uno de estos algoritmos consiste en un conjunto de instrucciones que se ejecutan de forma secuencial. La descripción a este nivel pude hacer uso de sentencias no sintetizables, y su uso se justifica en la realización de los denominados testbenches (definidos más adelante). [8]

d) **Nivel Algorítmico**.- Similar a un programa de alto nivel en C.

2.3.4 PALABRAS RESERVADAS DE VERILOG

Son palabras de léxico propio del lenguaje Verilog con algún significado especial en las expresiones que no pueden ser utilizados fuera del contexto para lo que estan reservadas, se describen a continuación en la tabla II.IX:

TABLA II.IX: PALABRAS RESERVADAS

Palabras Reservadas				
Always	Endfunction	Join	pullup	Supply1
And	Endgenerate	Large	Pulsestyle_onevent	table
Assign	Endmodule	Liblist	Pulsestyle_ondetect	Task
automatic	Endprimitive	Localparam	Rcmos	Time
Begin	Endspecify	Macromodule	Real	Tran
Buf	Endtable	Médium	Realtime	Tranif0
Bufif0	Endtask	Module	Reg	Tranif1
Bufif1	Event	Nand	Reléase	Tri
Case	For	Negedge	Repeat	Tri0
Casex	Forcé	Nmos	Rnmos	Tri1
Casez	Forever	Nor	Rpmos	Triand

Cell	Fork	Not	Rtran	Trior
Cmos	Function	Noshowcancelled	Rtrainf0	Trireg
Config	Generate	Notif0	Rtrainf1	Unsigned
Deassign	Genvar	Notif1	Scalared	Use
Default	Highz0	Or	Signed	Vectored
Defparam	Highz1	Output	Showcancelled	Wait
Desing	If	Parameter	Small	Wand
Disable	Ifnone	Pmos	Specify	Weak0
Edge	Initial	Posedge	Specparam	Weak1
Else	Instance	Primitive	Strength	While
End	Inout	Pull0	Strong0	Wire
Endcase	Input	Pull1	Strong1	Wor
Endconfig	Integer	Pulldown	Supply0	Xnor
				Xor

2.3.5 LOS COMPONENTES DE DESCRIPCION VERILOG

Las letras en negrita indican palabras reservadas de Verilog. El símbolo <> indica parámetros opcionales.

1) **Declaración de módulo:**

   Indica el inicio de la definición de módulos. Esta es estrictamente necesaria.

   Sintaxis: module nombre_de_módulo <(lista_de_puertos)>;

2) **Declaración de puerto:**

   Indica la direccion, ancho y nombre del puerto.

   Sintaxis: in/out/inout <[MSB:LSB]> nombre_de_puerto;

3) **Declaración de registros y cables:**

   Indica el ancho y nombre del registro o cable.

   Sintáxis: reg/wire <[MSB:LSB]> nombre_de_registro/nombre_de_cable;

4) **Instancias de componentes:**

   Instancia de subbloque o compuerta. El nombre de la instancia debe ser único.

   Sintaxis: subbloque/nombre_módulo/compuerta nombre_instancia(lista de puertos conectividad)

5) **Assign:**

   Asignación de valores a una conexión (wire)

   Sintaxis: **assign** nombre_de_conexion = <#delay >nombre_del_registro/nombre_de conexión;

6) **Cuerpo del módulo:**

   Es el corazón del código HDL, contiene la descripción comportamental o estructural de toda la lógica combinacional y secuencial. Incluye las declaraciones always e initial, expresiones lógicas y aritméticas, los comandos case y muchos otros.

7) **Declaración de fin de módulo:**

   Indica el fin de la definición de un módulo. Esta es estrictamente necesaria.

   Sintaxis: endmodule.

Ejemplo de los componentes de descripción:

module nombre_de_módulo <(lista_de_puertos)>;

in nombre_de_puerto;

out nombre_de_puerto;

inout nombre_de_puerto;

reg/wire <[MSB:LSB]> nombre_de_registro/nombre_de_cable;

assign nombre_de_conexion = nombre_del_registro/nombre_de conexión**;**

endmodule

2.3.6 REGLAS DE CONECTIVIDAD DE PUERTOS EN VERILOG

Cuando se conectan elementos de hardware considere lo siguiente:

- ❖ Registros (registers) o redes (nets) externos se deben conectar únicamente a entradas (inputs) o redes internas.
- ❖ Salidas (outputs), registros (registers) o redes (nets) internos se deben conectar únicamente a redes (nets) externas.
- ❖ Entrada/salidas (inouts) deben ser conectados únicamente a redes (nets) externas.

2.3.7 USO DE DECLARACIONES BLOQUEADORAS Y NO BLOQUEADORAS

Las asignaciones bloqueadoras son utilizadas para especificar comportamientos combinacionales. Esto significa que al argumento de la derecha se le asigna al argumento de la izquierda en exactamente en el orden en que se escriben las asignaciones.

Su sintaxis es: **variable = expresión;**

Por otro lado, las asignaciones no bloqueadoras son aquellas que se ejecutan en paralelo, esto significa que se ejecutan independientemente de su orden de aparición, y permiten especificar comportamientos secuenciales.

Su sintaxis es: **variable <= expresión;**

El tipo de sentencia que se utiliza tiene un impacto directo sobre el circuito sintetizado, de forma que el programa escrito en el HDL podría no comportarse tal y como se esperaba que lo hiciera. Es por ello que al programar se deben observar las siguientes reglas de oro:

- Nunca mezcle asignaciones bloqueadoras y no bloqueadoras en el mismo bloque de código.

- Use sentencias no bloqueadoras para describir Flip-Flops y hardware similar, pero use sentencias bloqueadoras cuando el código describa lógica combinacional, es decir que el orden de las asignaciones sí cambia el resultado. (9)

2.3.8 ELEMENTOS BASICOS DEL LENGUAJE VERILOG

Antes de comenzar es preciso conocer algunos elementos básicos del lenguaje Verilog.

2.3.8.1 COMENTARIOS

Los comentarios van precedidos de los caracteres //, cuya información hasta el final de la línea es ignorado. También, pueden ir entre /* y */, donde la información entre estos caracteres es ignorada, la diferencia con la anterior alternativa, es que en esta segunda puede haber más de una línea. Ejemplos:

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

2.3.8.2 IDENTIFICADORES

Un identificador está formado por una letra o "_" seguido de letras, números y los caracteres "$" o "_". Este lenguaje si distingue entre mayúsculas y minúsculas.

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

2.3.8.3 NUMEROS

Los números en Verilog pueden especificarse en decimal, hexadecimal, octal o binario. Los números negativos se representan en complemento a2 y el caracter "_" puede utilizarse para una representación más clara del número. La sintaxis para la representación de un número es la siguiente.

<TAMAÑO> <BASE> <VALOR>

a) Tamaño: Es el número de bits expresado en decimal de la cantidad que viene a continuación. Este dato es opcional, en caso de no darse, por defecto el valor es de 32 bits.

b) **Base:** indica la base en la que se va a expresar el valor. Es opcional, por defecto es decimal.

'b Base binaria
'd Base decimal
'h Base hexadecimal
'o Base octal

c) Valor: Verilog tiene 4 valores que cualquier señal puede tomar:

TABLA II.X: VALORES DE SEÑALES

VALOR	SIGNIFICADO
0	Un valor binario de cero. Corresponde a cero voltios.
1	Un valor binario de 1. Dependiendo de la tecnología de fabricación, puede corresponder a +5V, +3.3V, o algún otro valor positivo.
x	Un valor cualquiera los valores x no son ni 0 o 1, y debería ser tratado como un valor desconocido.
z	Un valor de alta impedancia de un bufer de tres estados cuando la señal de control no está definida. Corresponde a un wire que no está conectado, o está flotando.

2.3.8.4 TIPOS DE DATOS

Fundamentalmente existen dos tipos de datos: reg y wire. La sintaxis para declarar estas variables es la siguiente.

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

- **Tipo:** existen varios tipos de variables, aunque las más destacadas son reg ywire.

  reg: Representan variables con capacidad de almacenar información.
  wire: Representan conexiones estructurales entre componentes. No tienen capacidad de almacenamiento.
  integer: Registro de 32 bits.
  real: Registro capaz de almacenar números en coma flotante
  time: Registro sin signo de 64 bits.

- **MSB y LSB:** Por defecto estas variables son de un sólo bit, aunque es posible definir vectores de bits.

- **Nombre:** Indica el nombre de la variable. (10)

A continuación, se muestran varios 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
```

2.3.8.5 TIPOS DE OPERADORES EN VERILOG

Los operadores aritméticos y/o lógicos se utilizan para construir expresiones. Estas expresiones se realizan sobre uno o más operandos. Estos últimos pueden ser nodos, registros constantes, etc…

a) OPERADORES ARITMETICOS

De un operando:

"+" a = +8'h01 Resultado: a = 8'h01
"-" a = -8'h01 Resultado: a = 8'hFF

De dos operandos:

"+" a = b + c;
"-" a = b – c;
"*" a = b * c;
"/" a = b / c; Resultado: Se devuelve la parte entera de la división
"%*" a = b % c; Resultado: Se devuelve el módulo de la división

El resultado del módulo toma el signo del primer operando.

Si algún bit del operando tiene el valor "x", el resultado completo es "x".

Los números negativos se representan en complemento a 2.

b) OPERADORES RELACIONALES

"<" a < b a es menor que b
">" a > b a es mayor que b
"<=" a <= b a es menor o igual que b
">=" a >= b a es mayor o igual que b

El resultado que se devuelve es:

0 si la condición no se cumple
1 si la condición se cumple
x si alguno de los operandos tiene algún bit a "x"

c) OPERADORES DE IGUALDAD

"=="	a == b	a es igual a b
"!="	a != b	a es diferente de b
"==="	a === b	a es igual a b, incluyendo "x" y "z"
"!=="	a !== b	a es diferente a b, incluyendo "x" y "z"

Los operandos se comparan bit a bit, rellenando con ceros para ajustar el tamaño en caso de que no sean de igual ancho. Estas expresiones se utilizan en condicionales. El resultado es:

0 si se cumple la igualdad
1 si no se cumple la igualdad
x únicamente en las igualdades "==" o "!=" si algún operando contiene algún bit a "x" o "z"

d) OPERADORES LOGICOS

"!" negación lógica
"&&" AND lógica
"||" OR lógica

Las expresiones con "&&" o "||" se evalúan de izquierda a derecha. Estas expresiones se utilizan en condicionales. El resultado es:

0 si la relación es falsa
1 si la relación es verdadera
x si algún operando contiene algún bit a "x" o "z"

e) OPERADORES BIT A BIT (BIT-WISE)

"~" negación
"& " AND
"|" OR
"^" OR exclusiva

Los citados operadores pueden combinarse obteniendo el resto de operaciones, tales como ~& (AND negada), ~^ (OR exclusiva negada), etc… El cálculo incluye los bits desconocidos (x) en la siguiente manera:

~x = x
0&x = 0
1&x = x&x = x
1|x = 1
0|x = x|x = x
0^x = 1^x = x^x = x

f) OPERADORES DE REDUCCION

"&" AND Ejemplo: &a Devuelve la AND de todos los bits de a.
"~&" AND negada Igual que el anterior.
"|" OR Ejemplo: |a Devuelve la OR de todos los bits de a.
"~|" OR negada Igual que el anterior.
"^" OR exclusiva Ejemplo: ^a Devuelve la OR exclusiva de todos los bits de a.
"~^" OR exclusiva negada Igual que el anterior

Estos operadores de reducción realizan las operaciones bit a bit sobre un solo operando.

Las operaciones negadas operan como AND, OR o EXOR pero con el resultado negado.

g) OPERADORES DE DESPLAZAMIENTO

"<<" Desplazamiento a izquierda Ejemplo: a = b << 2;
">>"Desplazamiento a derecha Ejemplo: a = b >> 2;

Notas: Los bits desplazados se rellenan con ceros.

h) OPERADOR DE CONCATENACION

"{}" Concatenación de operandos. Los operandos se separan por comas.

Ejemplos: {a,b[3:0],c} Si a y c son de 8 bits, el resultado es de 20 bits

{3{a}} Es equivalente a {a,a,a}

{b,3{c,d}} Es equivalente a {b,c,d,c,d,c,d}

Nota: No se permite la concatenación con constantes sin tamaño.

i) OPERADOR CONDICIONAL

"?" El formato de uso de dicho operador es (condición) ? trae_expr : false_expr

Ejemplos: out = (enable) ? a : b; La salida out toma el valor de a si enable está a 1, en caso contrario toma el valor de b.

Son los tipos de operadores más usados en el lenguaje Verilog. [8]

2.3.9 MODULOS EN VERILOG

2.3.9.1 CARACTERISTICAS

Las características principales de los módulos se pueden recoger en los siguientes puntos.

- Cada módulo dispone de una serie de entradas y salidas, por las cuales se puede interconectar otros módulos, aunque puede no tener entradas ni salidas, como es el caso de los testbenches.
- No existen variables globales.
- Fuera de los módulos solo hay directivas del compilador, que afectarán a partir del punto en donde aparecen.
- A pesar de que se pueden realizar varias simulaciones concurrentes, en general se suele tener un único módulo que emplea los módulos previamente definidos.
- Cada módulo puede describirse de forma arquitectural o de comportamiento.

2.3.9.2 ESTRUCTURA

En Verilog un sistema digital está compuesto por la interconexión de un conjunto de módulos. La estructura general de estos módulos es la siguiente.

module <nombre> (<señales>);
input, output <declaración de señales>
<funcionalidad del módulo>
endmodule

2.3.9.3 INTERFAZ DEL MODULO

Los argumentos del módulo pueden ser de tres tipos, estos argumentos comunicarán el interior o funcionalidad del módulo con otros elementos del propio diseño.

- **Input:** Entradas del módulo, cuyo tipo son wire.
- **Output:** Salidas del módulo. Dependiendo del tipo de asignación que las genere serán wire si proceden de una asignación continua y reg si proceden de una asignación procedural.
- **Inout:** Son a la vez entradas y salidas. Únicamente, son de tipo wire. (10)

2.3.9.4 ASIGNACIONES

Existen dos maneras de asignar valores en Verilog, de forma continua o procedural.

a) ASIGNACION CONTINUA

La asignación continua se utiliza exclusivamente para modelar lógica combinacional, donde no se necesita de una lista de sensibilidad para realizar la asignación. La variable a la que se realiza la asignación continua sólo puede ser declarada de tipo net, es decir wire. Y la asignación sólo puede ser declarada fuera de cualquier proceso y nunca dentro de bloques always o initial. La sintaxis es la siguiente.

assign variable <#delay> = asignación

Un ejemplo de la asignación continua podría ser el siguiente.

assign A = B & C;

En el ejemplo no se ha añadido una temporización en la asignación, por lo tanto en cada ciclo se revisará dicha sentencia. Estas asignaciones también se ejecutan de forma secuencial las cuales se pueden controlar mediante la temporización.

b) ASIGNACION PROCEDURAL

La asignación procedural es la que se ha visto hasta el momento, donde a las variables se le asigna un valor dentro de un proceso always o initial, el tipo de variable a la que se le asigna el valor puede ser de cualquier tipo. Su sintáxis es la siguiente.

<#delay> variable = asignación

Un ejemplo de esta asignación es el siguiente.

```
A = B & C;
```

2.3.9.5 TEMPORIZACIONES

Las temporizaciones se consiguen mediante el uso de retardos. El retardo se especifica mediante el símbolo # seguido de las unidades de tiempo del retardo. Un ejemplo podría ser el siguiente, donde en el instante 0 se ejecutan las dos primeras sentencias, cinco unidades de tiempo más tarde se realiza la tercera asignación y cuatro más tarde se ejecutan las dos últimas.

```
initial
begin
  clk = 0;
   rst = 0;
 #5 rst = 1;
 #4 clk = 1;
  rst = 0;
end
```

En las temporizaciones existen dos tipos de asignaciones procedurales.

- Con bloqueo: La asignación se realiza antes de proceder con la siguiente.
- Sin bloqueo: El término se evalúa en el instante actual, pero no se asigna hasta finalizar dicho instante.

Con el siguiente ejemplo se entiende de mejor manera las asignaciones:

```
initial
    begin
        // Asignación con bloqueo
        a = 5;
        #1 a = a + 1;
        b = a + 1;
        // Asignación sin bloqueo
        a = 5;
        #1 a <= a + 1;
        b = a + 1;
    end
```

En el ejemplo, inicialmente se realiza una asignación con bloqueo, donde el resultado será a = 6 y b = 7, ya que a se ha calculado previamente. Mientras que en la asignación sin bloque a y b serán 6, ya que a no ha sido calculado.

2.3.10 ESTRUCTURAS DE CONTROL

En este apartado se presentan las estructuras más comunes para la descripción de comportamiento en Verilog. A continuación se presentan cada una de ellas por separado indicando, además de un ejemplo, si son sintetizables o no.

2.3.10.1 SENTENCIAS CONDICIONALES IF – ELSE

La sentencia condicional if – else controla la ejecución de otras sentencias y/o asignaciones (estas últimas siempre procedurales). El uso de múltiples sentencias o asignaciones requiere el uso de begin – end. La sintaxis de la expresión es:

```
if (condición) //Condición simple
 sentencias;

if (condición) //Condición doble
 sentencias;
```

```
else
 sentencias;

if (condición1) // Múltiples condiciones
 sentencias;
else if (condición2)
sentencias;
....
else
 sentencias;
```

2.3.10.2 SENTENCIA CASE

La sentencia case evalúa una expresión y en función de su valor ejecuta la sentencia o grupos de sentencias agrupadas en el primer caso en que coincida. El uso de múltiples sentencias o asignaciones requiere el uso de begin – end. En caso de no cubrir todos los posibles valores de la expresión a avaluar, es necesario el uso de un caso por defecto (default). Este caso se ejecutará siempre y cuando no se cumplan ninguno de los casos anteriores. La sintaxis de la expresión es:

```
case (expresión)
 <caso1>: sentencias;
 <caso2>: begin
 sentencias;
 sentencias;
 end
 <caso3>: sentencias;
 ....
 ....
 <default>: sentencias;
endcase
```

2.3.10.3 SENTENCIA CASEZ Y CASEX

Estas sentencias corresponden a versiones especiales del case y cuya particularidad radica en que los valores lógicos z y x se tratan como valores indiferentes.

casez: usa el valor lógico z como valor indiferente

casex: toma como indiferentes tanto el valor z como el valor lógico x

Muestra de un ejemplo del uso del casez.

```
 casez (opcode)
4'b1zzz: out = a; // Esta sentencia se ejecuta siempre que el bit más
                     significativo sea 1.
4'b01??: out = b; // El símbolo ? siempre se considera como "indiferente",
                     luego en  este caso es equivalente a poner 4'b01zz
4'b001?: out = c;
default: $display("Error en el opcode");
endcase
```

2.3.10.4 SENTENCIAS DE BUCLE

Todas las sentencias de bucles en Verilog deben estar contenidas en bloques procedurales (initial o always). Verilog soporta cuatro sentencias de bucles.

a) Sentencia forever

El bucle forever se ejecuta de forma continua, sin condición de finalización. Su sintaxis es:

forever <sentencias>

En caso de englobar mas de una sentencia o asignación, éstas se deben incluir entre begin – end.

```
initial
begin
 clk = 0;
forever #5 clk = !clk;
end
```

b) SENTENCIA REPEAT

El bucle repeat se ejecuta un determinado número de veces, siendo este número su condición de finalización. Su sintaxis es:

repeat (<número>) <sentencias>

En caso de englobar mas de una sentencia o asignación, éstas se deben incluir entre begin – end.

```
if (opcode == 10) //Realización de una operación de rotación
repeat (8) begin
temp = data[7];
data = {data <<1,temp};
end
```

c) SENTENCIA WHILE

El bucle while se ejecuta mientras la condición que evalúa sea cierta. La condición que se especifica expresa la condición de ejecución del bucle. Su sintaxis es:

while (<expresión>) <sentencias>

En caso de englobar más de una sentencia o asignación, éstas se deben incluir entre begin – end.

```
loc = 0;
if (data = 0)) //Ejemplo de cálculo del bit más significativo
loc = 32;
else while (data[0] == 1'b0) begin
loc = loc + 1;
data = data >> 1;
end
```

d) BUCLE FOR

El bucle for es similar a aquellos utilizados en los lenguajes de programación. Su sintaxis es:

for (<valor inicial>,<expresión>, <incremento>) <sentencias>

En caso de englobar más de una sentencia o asignación, éstas se deben incluir entre begin – end. (10)

```
 for (i=0, i<64, i=i+1)) //Inicialización de memoria RAM
ram[i] = 0;
```

2.3.11 FUNCIONES DEL SISTEMA VERILOG

Existen acciones de bajo nivel disponibles denominadas funciones de sistema, suelen variar de una implementación a otra. Enumeraremos varias funciones más usas:

2.3.11.1 $FINISH

Si no se indica lo contrario la simulación es indefinida, como esto no suele ser deseable, esta función indica el final de la simulación.

2.3.11.2 $TIME

Esta función retorna el instante de tiempo en el que se encuentra la simulación.

2.3.11.3 $RANDOM

Esta función retorna un valor aleatorio entero de 32 bits cada vez que se invoca.

Se le puede suministrar como argumento una variable con la semilla que controla la generación de la misma secuencia aleatoria en el caso de repetir la simulación. Esto es, los valores obtenidos de la invocación repetida de esta función son aleatorios entre sí, pero será la misma secuencia si se ejecuta de nuevo.

$display y $write
$display(P1, P2, P3, ...);
$write(P1, P2, P3, ...);

Estas dos funciones de sistema permiten imprimir por la salida estándar, mensajes y variables del sistema. Ambas tienen una funcionalidad similar salvo porque **$display** coloca un salto de línea al final.

Si el argumento es una variable se imprime (en formato decimal), si es una cadena se imprime tal cual salvo que tenga caracteres de escape:

TABLA II.XI:CARACTERES DE ESCAPE

CARÁCTER	CARACTERÍSTICA
\n	Salto de línea
\t	Tabulador
\\	Carácter \
%%	Carácter %

O indicadores de formato:

TABLA II.XII: INDICADORES

CARÁCTER	CARACTERÍSTICA
%d	formato decimal (defecto)
%h	formato hexadecimal
%b	formato binario
%o	formato octal
%t	formato tiempo
%c	carácter ASCII

En este caso, a continuación de la cadena, deben de aparecer tantos argumentos como formatos se especifiquen.

Por omisión el tamaño del formato es el máximo de la variable correspondiente, pudiendo ajustarse al tamaño actual de la variable colocando un cero después del %. Así por ejemplo:

```
reg [15:0] A;
initial begin
A=10;
$display ("%b %0b %d %0d", A, A, A, A);
end
```

Imprimiría:
0000000000001010 1010 22210 10

2.3.11.4 $FDISPLAY Y $FWRITE

a) **$fdisplay**(fd, P1, P2, P3, ...);

b) **$fwrite**(fd, P1, P2, P3, ...);

Estas dos comandos son similares a los del apartado anterior salvo que permiten almacenar el resultado en un fichero, cuyo descriptor **(fd)** se le da como primer argumento.

Las funciones para manipular la apertura y cierre del fichero son:

Declaración	Ejemplo
fd=$fopen**(nombre del fichero); **$fclose(fd);	**integer** fd; **initial** fd=**$fopen**("resultados.dat"); **initial begin** **for**(i=0;**i**<100;**i**=**i**+1) begin #(100) $fdisplay(fd,"%d %h",A,B); **end** **$fclose(fd);** **$finish;** **end**

2.3.11.5 $MONITOR Y $FMONITOR

a) **$monitor**(P1, P2, P3, ...);

b) **$fmonitor**(fd, P1, P2, P3, ...);

Estas funciones permiten la monitorización continua: se ejecutan una única vez, registrando aquellas variables que deseamos ver su cambio. De forma que cada vez que se produzca un cambio en una variable registrada, se imprimirá la línea completa.

2.3.11.6 $DUMPFILE Y $DUMPVARS

Un formato de almacenamiento de los datos de una simulación Verilog es el VCD (Verilog Change Dump). Este formato se caracteriza porque se almacena el valor de un conjunto de variables cada vez que se produce un cambio. Suele ser empleado por los visualizadores y post-procesadores de resultados.

Para crear este fichero son necesarias dos acciones: abrir el fichero donde se almacenaran los resultados y decir que variables se desea almacenar.

```
Initial
 begin
$dumpfile("file1.vcd");
$dumpvars;
End
```

En el ejemplo anterior se desea crear un fichero cuyo nombre es file1.vcd, donde se desean almacenar todas las variables del diseño (que es lo que se almacena en el caso de no proporcionar parámetros).

Pueden restringirse las variables almacenadas empleando:

$dumpvars(levels,name1,name2,...)

Siendo levels el número de niveles de profundidad en la jerarquía que se desean almacenar, y name1, name2... las partes del diseño que se desean almacenar.

Si el primer argumento es 0, significa que se desean todas las variables existentes por debajo de las partes indicadas. Un 1 significa las variables en esa parte, pero no el contenido de módulos instanciados en su interior.

2.3.11.7 $READMEMB Y $READMEMH

Permite leer la información contenida en un fichero en una memoria.

a) **$readmemh**(fname,array,start index,stop index);
b) **$readmemb**(fname,array,start index,stop index);

Donde fname: es el nombre de un fichero que contiene:

Datos binarios ($readmemb) o hexadecimales ($readmemh) (sin especificadores de tamaño o base). Pudiendo emplearse, x o z separadores: espacios, tabulaciones, retornos de carro.

La información se debe de almacenar en una memory array, como por ejemplo:

reg [7:0] **mem** [0:1023];

El resto de argumentos son opcionales:

Ejemplo

```
initial $readmemh(file.dat,mem,0);
initial $readmemh(file.dat,mem,5);
initial $readmemh(file.dat,mem,511,0);
```

En el primer ejemplo se comienza a rellenar desde el índice 0, en el segundo se comienza desde el índice 5. En el último caso se comienza por 511 y descendiendo hasta 0.

Si el número de datos del fichero es diferente al tamaño reservado o especificado para rellenar el simulador da un error.

2.3.12 DIRECTIVAS PARA EL COMPILADOR

Verilog ofrece un conjunto de directivas de compilación que permiten obtener diferentes códigos a partir de una única descripción. A continuación se detallan las más comunes. La sintaxis para la definición de una directiva es la siguiente.

'directiva <nombre> <valor>

2.3.12.1 DEFINE

La directiva define permite definir un valor.

```
'define TD 1
   ........
assign #TD data = A;
```

2.3.12.2 INCLUDE

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

'include "sumador.v"

2.3.12.3 IFDEF

La directiva ifdef permite compilar un código siempre y cuando se haya definido el símbolo al que referencia.

```
'define SIMULATION
    .....
 always @(posedge clk or posedge rst)
  if(rst)
    data <= 0;
  else
   'ifdef SIMULATION
      data <= A;
```

```
'else
   data <= B;
'endif
```

2.3.12.4 TIMESCALE

La directiva timescale permite definir las unidades de tiempo con las que se va a trabajar. La sintaxis de esta directiva es la siguiente. Donde la unidad de tiempo es mayor o igual a la resolución y los valores que pueden tomar los enteros son 1, 10 y 100 y la unidad de medida puede ser: "s", "ms", "us", "ns", "ps" y "fs".

'timescale <unidad de tiempo> / <resolucion>

Un ejemplo de esta directiva es la siguiente.

'timescale 1ns/100ps

En el ejemplo, la unidad de tiempo empleada es el nanosegundo y cualquier retraso fraccional se redondeará al primer decimal debido a la resolución especificada. (10)

2.3.13 PROCESOS DE VERILOG

El concepto de procesos que se ejecutan en paralelo es una de las características fundamentales del lenguaje, siendo ese uno de los aspectos diferenciales con respecto al lenguaje procedural como el lenguaje C.

Toda descripción de comportamiento en lenguaje Verilog debe declararse dentro de un proceso, aunque existe una excepción que trataremos a lo largo de este apartado. Existen en Verilog dos tipos de procesos, también denominados bloques concurrentes.

2.3.13.1 INITIAL: Este tipo de proceso se ejecuta una sola vez comenzando su ejecución en tiempo cero. Este proceso NO ES SINTETIZABLE, es decir no se puede utilizar en una descripción RTL. Su uso está íntimamente ligado a la realización del testbecnh.

2.3.13.2 ALWAYS: Este tipo de proceso se ejecuta continuamente a modo de bucle. Tal y como su nombre indica, se ejecuta siempre. Este proceso es totalmente sintetizable. La ejecución de este proceso está controlada por una temporización (es decir, se ejecuta cada determinado tiempo) o por eventos. En este último caso, si el bloque se ejecuta por más de un evento, al conjunto de eventos se denomina lista sensible. La sintaxis de este proceso es:

always <temporización> o <@(lista sensible)>

Los dos ejemplos muestran la utilización de los procesos initial y always. De estos ejemplos se pueden apuntar las siguientes anotaciones:

```
INITIAL                          always @(a or b or sel)
 BEGIN                           begin //Sobra en este caso
       CLK = 0                     if (sel == 1)
       RESET = 0                         y = a;
       ENABLE = 0                  else
       DATA = 0                          y = b;
END                              end
```

- **Begin, end:** Si el proceso engloba más de una asignación procedural (=) o más de una estructura de control (if-else, case, for, etc…), estas deben estar contenidas en un bloque delimitado por **begin** y **end.**
- **Initial:** Se ejecuta a partir del instante cero y, en el ejemplo, en tiempo 0 (no hay elementos de retardo ni eventos, ya los trataremos), si bien las asignaciones

contenidas entre begin y end se ejecutan de forma secuencial comenzando por la primera. En caso de existir varios bloques initial todos ellos se ejecutan de forma concurrente a partir del instante inicial.

- **Always:** En el ejemplo, se ejecuta cada vez que se produzcan los eventos variación de la variable a o variación de b o variación de sel (estos tres eventos conforman su lista de sensibilidad) y en tiempo 0. En el ejemplo, el proceso **always** sólo contiene una estructura de control por lo que los delimitadores **begin** y **end** pueden suprimirse.
- Todas las asignaciones que se realizan dentro de un proceso initial o always se deben de realizar sobre variables tipo **Reg** y NUNCA sobre nodos tipo **Wire**. [8]

2.3.13.3 EVENTOS DE NIVEL:

Se distinguen dos tipos de eventos:

- **Eventos de nivel**

Este evento se produce por el cambio de valor de una variable simple o de una lista sensible. Ejemplos:

TABLA II.XIII: EVENTO DE NIVEL

Evento	**Descripción**
Always @(x) Z <= x + y	Cada vez que el valor de la señal "x" Cambia, el proceso se evalúa.
Always @(x or y or z) T <= x + y + z	Cada vez que el valor de la señal "x" o "y" o "z" Cambia el proceso se evalúa. En este caso x, y, y z conforman la lista sensible.

- **Eventos de flanco:**

Se producen por la combinación de flancos de subida y/o bajada de una señal simple o de una lista sensitiva. (11)

TABLA II.XIV:EVENTOS DE FLANCOS

Evento	**Descripción**
Always @(**posedge** clk or **posedge** irt) Z <= z + y	Cada vez que se produce un flanco de subida de clk o de irt, el proceso se evalúa
Always @(**posedge** clk or **negedge** irt) Z <= z + y	Cada vez que se produce un flanco de subida de clk o un flanco de bajada de irt, elproceso se evalúa

2.3.14 TESTBENCH

El propósito de un testbench no es otro que verificar el correcto funcionamiento de un Módulo /diseño. La escritura de un testbench es casi tan compleja como la realización en RTL del módulo a verificar, a partir de ahora la denominaremos DUT (Desing Under Test). Una de las ventajas que presenta la escritura de un testbench es la posibilidad de no tener que ser sintetizable y, por tanto RTL.

Para escribir un testbench el diseñador debe tener siempre presente las especificaciones de diseño, en la que quedan reflejadas las funciones del diseño y, por tanto, las funciones a verificar.

2.3.14.1 ESTRUCTURA DE UN TESTBENCH

La estructura de un testbench es la que se refleja en la Figura II.5:

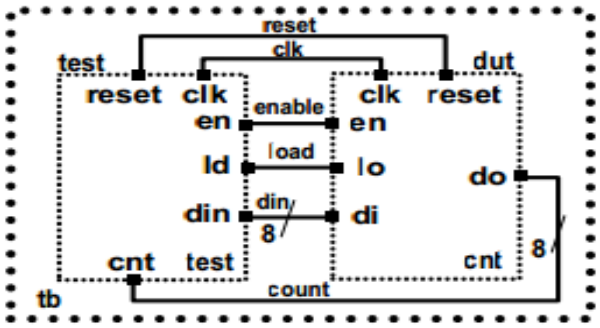


FIGURA II.5: Testbench

Fuente: *http://www.iuma.ulpgc.es/~nunez/clases-FdC/verilog/Verilog%20Tutorial%20v1.pdf*

```
module tb;
 //nodos internas
wire [7:0] din, count;
wire reset,clk,enable,load;
//Dut
cnt dut(.clk (clk),
 .reset (reset).
 .di (din),
 .en (enable),
 .lo (load),
 .do (count));
//Test
test test(.clk (clk),
 .reset (reset),
 .en (enable),
 .din (din),
 .ld (load),
 .cnt (count));
initial
begin
 $shm_open("waves.shm");
 $shm_probe(tb,"AS");
end
endmodule
```

Se compone de:

a) **Módulo dut:** Diseño a verificar

b) **Módulo test:** Módulo generador/analizador. Este módulo es el encargado de generar las señales de entrada al módulo DUT y de analizar sus salidas. Su realización se hace a nivel de comportamiento (behavioral) y no necesariamente utilizando código RTL. Tal y como se indicó anteriormente, la realización de este módulo implica el conocer en detalle el diseño a verificar.

c) **Módulo tb:** Este módulo incluye los módulos anteriores. Se caracteriza por no tener entradas ni salidas. Corresponde a una declaración estructural del conexionado de los módulos dut y test. En el ejemplo de la se ha incluido un proceso (initial) cuya única finalidad es la de abrir una base de datos con las formas de ondas del testbench.

2.3.14.2 Interfaz de entrada/salida

La descripción del módulo de test comienza con la definición de las señales de interfaz. Éstas deben ser iguales, pero de sentido contrario, a las señales de interfaz del módulo dut. La muestra la descripción del interfaz del módulo test para nuestro ejemplo. (8)

```
module test(clk,reset,en,lo,di,do);  //Señales globales
                input clk,reset;
                //Entradas
                input [7:0] di;
                input en;
                input lo;
                //Salidas
                output [7:0] do;
                reg [7:0] do;
        endmodule
```

CAPITULO III

TARJETA FPGA

3.1 HISTORIA TARJETA FPGA

Las FPGA fueron inventadas en el año 1984 por Ross Freeman y Bernard Vonderschmitt, co-fundadores de Xilinx, y surgen como una evolución de los CPLD (Dispositivo Lógico Programable Complejo).

Tanto los CPLD como las FPGA contienen un gran número de elementos lógicos programables. Si medimos la densidad de los elementos lógicos programables en puertas lógicas equivalentes (número de puertas NAND equivalentes que podríamos programar en un dispositivo) podríamos decir que en un CPLD hallaríamos del orden de decenas de miles de puertas lógicas equivalentes y en una FPGA del orden de cientos de miles hasta millones de ellas.

Aparte de las diferencias en densidad entre ambos tipos de dispositivos, la diferencia fundamental entre las FPGA y los CPLD es su arquitectura. La arquitectura de los CPLD es más rígida y consiste en una o más sumas de productos programables cuyos resultados van a parar a un número reducido de biestables síncronos (también denominados flip-flops). La arquitectura de las FPGA, por otro lado, se basa en un gran número de pequeños bloques utilizados para reproducir sencillas operaciones lógicas, que cuentan a su vez con biestables síncronos. La enorme libertad disponible en la interconexión de dichos bloques confiere a las FPGA una gran flexibilidad.

La diferencia importante entre FPGA y CPLD es que en la mayoría de las FPGA se pueden encontrar funciones de altos niveles (como sumadores y multiplicadores) embebidas en la propia matriz de interconexiones, así como bloques de memoria.

Las FPGA son el resultado de la convergencia de dos tecnologías diferentes, los dispositivos lógicos programables (PLD [Programmable Logic Devices]) y los circuitos integrados de aplicación específica (ASIC [Application-Specific Integrated Circuit]). La historia de los PLD comenzó con los primeros dispositivos PROM (Programmable Read-Only Memory) y se les añadió versatilidad con los PAL (Programmable Array Logic) que permitieron un mayor número de entradas y la inclusión de registros. Esos dispositivos han continuado creciendo en tamaño y potencia. Mientras, los ASIC siempre han sido potentes dispositivos, pero su uso ha requerido tradicionalmente una considerable inversión tanto de tiempo como de dinero. Intentos de reducir esta carga han provenido de la modularización de los elementos de los circuitos, como los ASIC basados en celdas, y de la estandarización de las máscaras, tal como Ferranti fue pionero con la ULA (Uncommitted Logic Array). El paso final era combinar las dos estrategias con un mecanismo de interconexión que pudiese programarse utilizando fusibles, antifusibles o celdas RAM y celdas ROM, como los

innovadores dispositivos Xilinx de mediados de los 80. Los circuitos resultantes son similares en capacidad y aplicaciones a los PLD más grandes, aunque hay diferencias puntuales que delatan antepasados diferentes. Además de en computación reconfigurable, las FPGA se utilizan en controladores, codificadores/decodificadores y en el prototipado de circuitos VLSI y microprocesadores a medida.

El primer fabricante de estos dispositivos fue Xilinx y los dispositivos de Xilinx se mantienen como uno de los más populares en compañías y grupos de investigación. Otros vendedores en este mercado son Atmel, Altera, AMD y Motorola. [12]

3.2 INTRODUCCION

Los Dispositivos FPGA (Fiel Programmable Gate Array, en español Arreglo de Compuertas Programable en el Campo), como su nombre lo dice son un arreglo matricial de bloques lógicos programables en cualquier espacio físico, útil para la implementación de los circuitos digitales.

Es un dispositivo semiconductor que posee varios bloques lógicos cuya interconexión y funcionalidad se puede programar por el usuario. La lógica programable puede emitir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica, un sistema combinacional y hasta sistemas complejos.

La tecnología FPGA permite realizar diseños a medida, de bajo coste de desarrollo, incluso para la producción de pocas unidades. Estas características la hacen muy interesante para realizar prototipado rápido. Especialmente tiene un gran interés dentro del campo docente. Sobre FPGA, la mayoría de estas aplicaciones funcionan cerca un orden de magnitud más rápido que en un microcontrolador.

3.3 FPGA VS VARIAS TECNOLOGIAS

3.3.1 FPGA vs CPLD

Los dos chips son de lógica digital pero ellos tienen diferentes características que dependen se sus fabricantes. Las FPGA contienen hasta 100 mil bloques lógicos, mientras que los CPLD contienen a lo sumo 100 macros bloques lógicos, los dos con flip-flops. Además, las FPGA contienen funciones aritméticas como sumadores, comparadores y memoria RAM, lo que soporta a realizar diseños digitales muy grandes. Sin embargo, los CPLD poseen tiempos de entrada/salida más rápidos que las FPGA.

3.3.2 FPGA vs MICROCONTROLADOR

Los microcontroladores están basados en una arquitectura CPU y ejecutan las instrucciones de una manera secuencial. En contra parte, las FPGA son dispositivos de lógica programable y el algoritmo se ejecuta de una manera paralela.

3.3.3 FPGA vs ASIC

Los FPGA son muy utilizados por fabricantes que producen tecnología a baja escala, los cuales no pueden justificar la producción por los bajos debido a los volúmenes de dispositivos que venden son muy bajos. Los FPGA tienen una funcionalidad similar, a costos menores y con una velocidad ligeramente menor, tienen un mayor consumo de potencia y no pueden abarcar sistemas tan complejos como las tarjetas FPGA. Las FPGA tienen las ventajas de ser reprogramables como el usuario la desee (lo que añade una enorme flexibilidad al flujo de diseño), su costo de desarrollo y adquisición son mucho menores para pequeñas cantidades de dispositivos y el tiempo de desarrollo es también menor. También las tarjetas FPGA se utilizan como prototipos, que permiten depurar y

permiten refinar el diseño digital que se esté realizando. Con el software de diseño se puede simular en hardware antes de fabricar el ASIC correspondiente al diseño digital. (14)

3.4 FABRICANTES

A inicio del 2007, el mercado de los FPGA se había colocado en un estado donde hay dos productores (Xilinx, Altera) de FPGA de propósito general que están a la cabeza del mismo, y un conjunto de otros competidores quienes se diferencian por ofrecer dispositivos de capacidades únicas. (13)

Entre las empresas que producen las FPGA, tenemos:

3.4.1 XILINX

Es el líder más grande del mercado en la fabricación de FPGA.

Entre las principales series de FPGA de Xilinx tenemos:

TABLA III.I: Tarjetas FPGAs de XILINX

Series Virtex	**Series Spartan**
Se pueden emplear los dispositivos de esta serie para reemplazar ASIC en muchas aplicaciones, incluyendo redes alámbricas e inalámbricas, telecomunicaciones, almacenamiento, servidores, computación, video, imagen, médico, industrial y de defensa. Dentro de esta serie tenemos las familias de FPGA Virtex-5, Virtex-4, Virtex-II Pro, Virtex-II, and Virtex-E	Hasta 5 millones de compuertas y hasta 784 puertos de entrada/salida y 344 pares de entrada/salida diferencial. Tecnología de impedancia controlada digitalmente XCITE. La tecnología de 90 nm reduce el tamaño y el costo, incrementando la eficiencia de manufactura. Ideales para diseños que requieren FPGA de bajo costo para aplicaciones de procesamiento digital de señales tales como radio militar, cámaras de supervisión o vigilancia, imágenes médicas. Muy útil para aplicaciones donde se requiere de una alta densidad lógica. Dentro de esta serie tenemos las familias de FPGA Spartan-3A DSP, Spartan-3AN, Spartan-3A, Spartan-3E, Spartan-3, Spartan-IIE, Spartan-II,

	Spartan/XL. Cada familia tiene un área de aplicación específica como procesamiento digital de señales, memoria no volátil, entre otros.

La FPGA que se usará para realizar las conexiones físicas revisar Anexo 1 donde se enumeran las partes más importantes de esta tarjeta FPGA como se muestra en la Figura III.1:

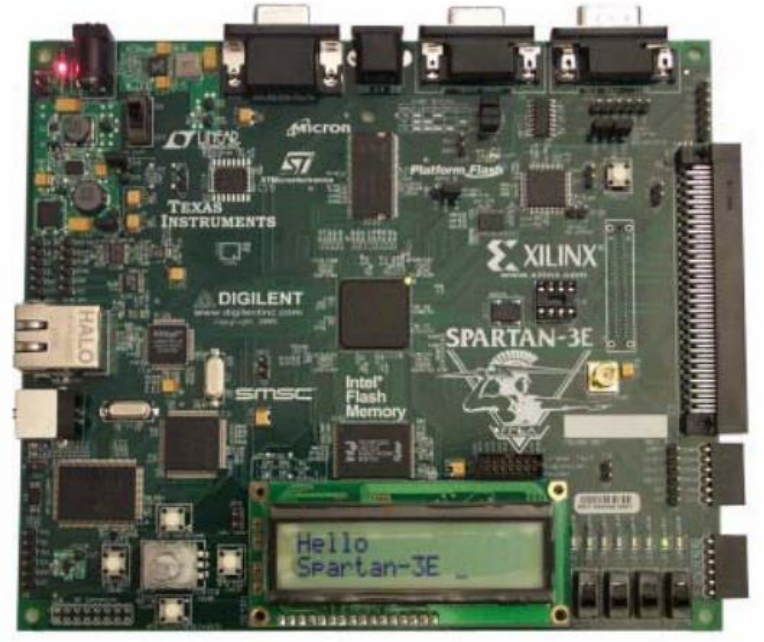


FIGURA III.1: Tarjeta Fpga Xilinx Spartan 3e Fpga (500k Gates)
Analog Devices Adm3232earnz Rs-232 Line Driver/Receiver
Fuente: *MANUAL DE LA TARJETA XILINX SPARTAN 3E*

3.4.2 ALTERA

Es el otro gran líder.

a) **Serie Cyclone:** Poseen hasta 120 mil elementos lógicos y hasta 535 pines de entrada/salida. Dentro de esta serie tenemos las familias de FPGA Cyclone III, Cyclone II y Cyclone, El dispositivo Cyclone III posee 4Mbits de memoria embebida dedicada a circuitería de interfaz de memoria externa, PLLs y

capacidades de entrada/salida diferencial de alta velocidad. Los dispositivos de esta serie son de baja potencia, alta funcionalidad y bajo costo, y además se pueden utilizar en aplicaciones tales como: automotriz, despliegue y procesamiento de imágenes, militares, video, e inalámbrico.

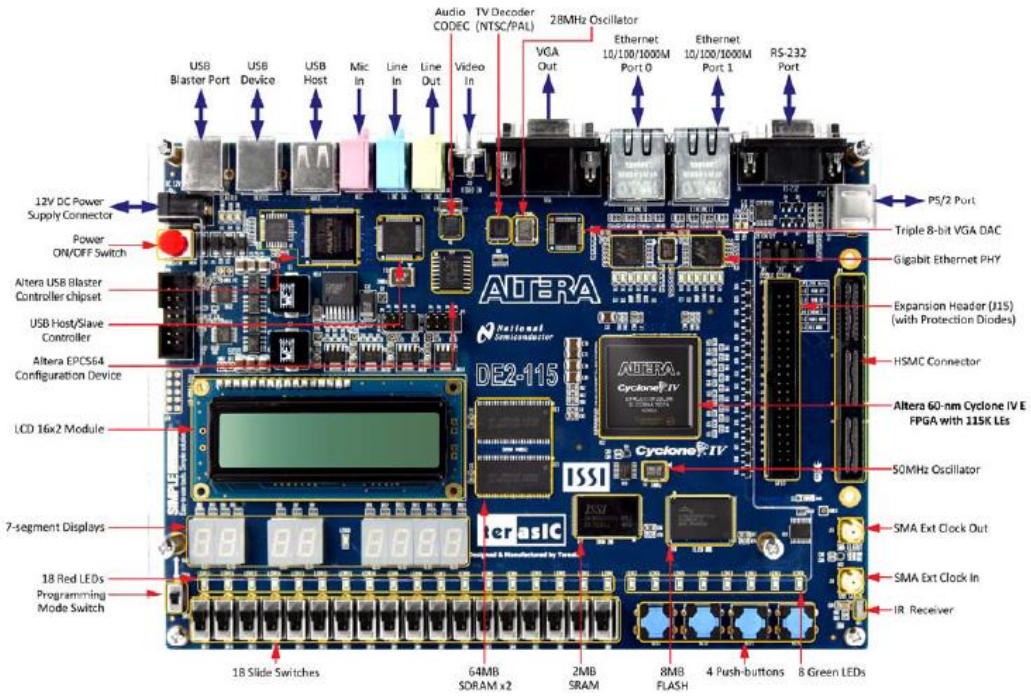


FIGURA III.2: TARJETA FPGA DE2-115 LADO ANVERSO

Fuente: *MANUAL DE USUARIO FPGA DE2-115*

b) Serie Stratix: Poseen hasta 79 mil compuertas, 1200 pines de entrada/salida, 7 Mbits de memoria RAM, 22 bloques de DSP, 176 multiplexores embebidos, optimizados para aplicaciones complejas que requieren alto flujo de datos, interfaces de comunicaciones de alta velocidad incluyendo los estándares 10G Ethernet XSBI, SFI-4, POS-PHY Level 4 (SPI-4 Phase 2), HyperTransport, RapidIO™ y UTOPIA IV. Ofrece una solución de administración de reloj completa con una estructura de reloj jerárquica y hasta 18 PLLs. Los dispositivos de esta serie

ofrecen funcionalidad dedicada para administración del reloj y para aplicaciones con DSP, además de soportar estándares de entrada/salida en modo single-ended y diferencial, permite actualizaciones remotas vía una comunicación de red, utiliza la tecnología on-chip-termination que mejora la calidad de la señal de salida Son ampliamente utilizadas en aplicaciones aeroespaciales y militares donde se requiere un amplio rango de temperatura de operación. Dentro de esta serie tenemos las familias de FPGA Stratix III, Stratix II GX y Stratix II. (14)

3.4.3 LATTICE SEMICONDUCTOR

Lanzó al mercado dispositivos FPGA con tecnología de 90nm. En adición, Lattice es un proveedor líder en tecnología no volátil, FPGA basadas en tecnología Flash, con productos de 90nm y 130nm, figura III.3.

FIGURA III.3: CHIP FPGA LATTICE

Fuente: *http://es.farnell.com/productimages/farnell/standard/1571979-40.jpg*

3.4.4 ACTEL

Tiene FPGA basados en tecnología Flash reprogramable. También ofrece FPGA que incluyen mezcladores de señales basados en Flash, figura III.4.

FIGURA III.4: CHIP FPGA ACTEL

Fuente: *http://www.aldec.com/en/solutions/prototyping/microsemi_prototyping*

3.4.5 QUICKLOGIC

Tiene productos basados en antifusibles (programables una sola vez), figura III.5.

FIGURA III.5: CHIP FPGA QUICKLOGIC

Fuente: *http://static.electronicsweekly.com/news/wp-content/uploads/sites/16/2013/05/QuickLogic.jpg*

3.4.6 ATMEL

Es uno de los fabricantes cuyos productos son reconfigurables (el Xilinx. XC62xx fue uno de estos, pero no están siendo fabricados actualmente). Ellos se enfocaron en proveer microcontroladores AVR con FPGA, todo en el mismo encapsulado, figura III.6.

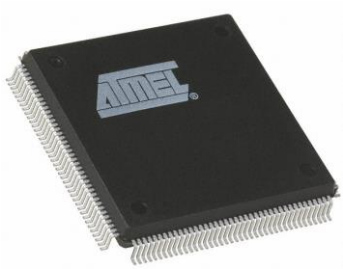


FIGURA III.6: CHIP FPGA ATMEL

Fuente: *http://media.digikey.com/photos/Atmel%20Photos/313-160-PQFP.jpg*

3.4.7 ACHRONIX

Tienen en desarrollo FPGAs muy veloces. Planean sacar al mercado a comienzos de 2007 FPGAs con velocidades cercanas a los 2GHz, figura III.7.

FIGURA III.7: CHIP FPGA ACHRONIX

Fuente: *http://www.achronix.com/wp-content/uploads/img/hd1000_bga.jpg*

3.4.8 MATHSTAR.

Ofrecen FPGA que ellos llaman FPOA (Arreglo de objetos de matriz programable), figura III.8.

FIGURA III.8: CHIP FPGA MATHSTAR

Fuente: *http://www.achronix.com/wp-content/uploads/img/hd1000_bga.jpg*

3.5 LAS PRINCIPALES CARACTERISTICAS DE LA FPGA

- Una un orden de grado de interconexiones programables permite a los bloques lógicos de un FPGA ser interconectados según la necesidad del diseñador del sistema digital, algo parecido a una protoboard (es una placa de uso genérico reutilizable) programable. Estos bloques lógicos e interconexiones pueden ser programados después del proceso de manufactura por el usuario/diseñador, así que el FPGA puede desempeñar cualquier función lógica necesaria.
- Una tendencia que en la actualidad se muestra es combinar los bloques lógicos e interconexiones de los FPGA con microprocesadores y periféricos relacionados para formar un sistema programable en un chip.
- Varias FPGA modernos soportan la reconfiguración parcial del sistema, permitiendo que una parte del diseño sea reprogramada como el usuario lo realice, mientras las demás partes siguen funcionando. Este es el principio de la idea de la computación reconfigurable, o los sistemas reconfigurables, que permiten el ahorro de tiempo.

3.6 ESTRUCTURA GENERAL DE LA FPGA

La estructura de una tarjeta FPGA en la parte interna se muestra en la figura III.9:

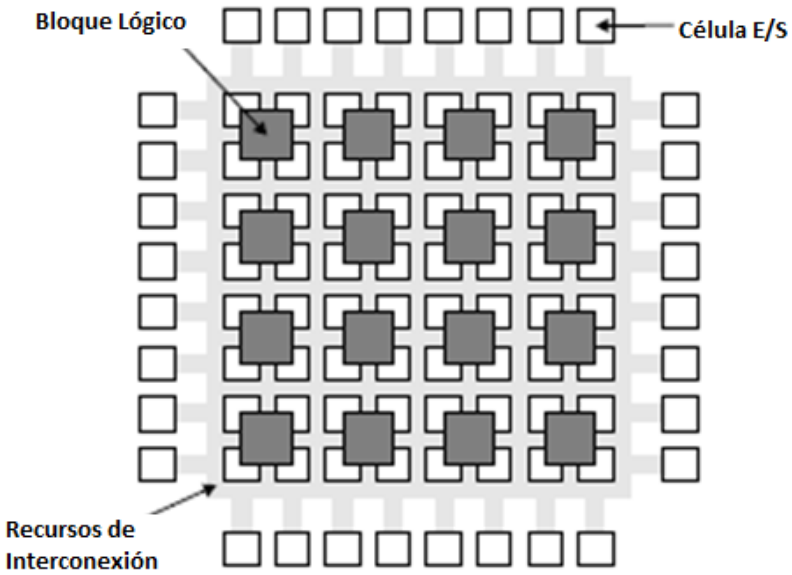


FIGURA III.9: ARQUITECTURA INTERNA DE LA FPGA (XILINX)

Fuente: ***FPGA****: NOCIONES BASICAS DE IMPLEMENTACIÓN – M. L. LÓPEZ VALLEJO Y J. L. AYALA RODRIO*

Los elementos básicos que constituyen una FPGA como las de Xilinx se pueden ver en la Figura III.9 y son las siguientes:

1. **Bloques Lógicos:** La estructura y contenido se denomina arquitectura en la cual está la descripción del sistema. Hay muchos tipos de arquitecturas, que varían principalmente en complejidad (desde una simple puerta hasta módulos más complejos). Suelen incluir biestables para facilitar la implementación de circuitos secuenciales. Otros módulos de importancia son los bloques de entrada/salida.
2. **Recursos de interconexión:** Cuya estructura y contenido se denomina arquitectura de rutado.
3. **Memoria RAM,** que se carga durante el RESET para configurar bloques y conectarlos.

Por supuesto, no todas las FPGA son iguales. Dependen de los fabricantes para poder encontrar con diferentes soluciones. Las FPGA que existen en la actualidad en el mercado

se pueden clasificar como pertenecientes a cuatro grandes familias, dependiendo de la estructura que adoptan los bloques lógicos que tengan definidas. Las cuatro estructuras se pueden ver en la Figura III.10, sin que aparezcan en la misma los bloques de entrada/salida.

1. Matriz simétrica, como son las de XILINX
2. Basada en canales, como ACTEL
3. Mar de puertas, como ORCA
4. PLD jerárquica, como ALTERA o CPLD de XILINX.

En concreto, para explicar el funcionamiento y la estructura básica de este dispositivo programables sólo se considerarán las distintas familias de XILINX. [15]

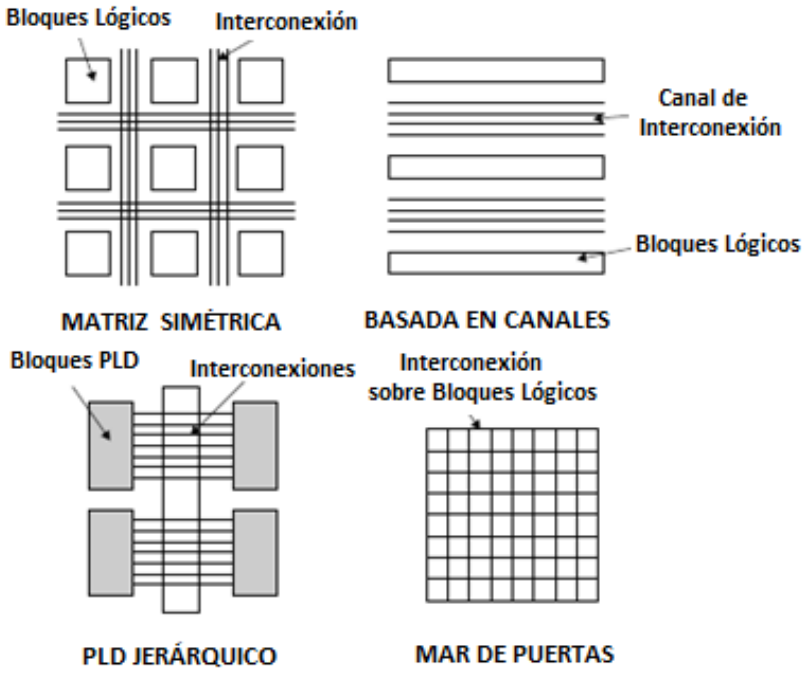


FIGURA III.10: ARQUITECTURA INTERNA DE LOS TIPOS DE FPGA

Fuente*: FPGA: NOCIONES BASICAS DE IMPLEMENTACIÓN – M. L. LÓPEZ VALLEJO Y J. L. AYALA RODRIO*

3.6.1 DIAGRAMA DE BLOQUES GENERAL DE LAS FPGA

El diagrama de bloques se muestra en la figura III.11. El bloque principal lo constituye la FPGA, a la que se añade una circuitería adicional, dividida en los siguientes bloques:

- **Circuito de reloj**, para la realización de diseños síncronos. La frecuencia del oscilador empleado depende de la aplicación de usuario.
- **Circuito de programación interno**, constituido por la memoria EEPROM serie y un multiplexor para que los pines de la EEPROM sean accesibles bien desde la FPGA, para su carga, o bien desde los pines del puerto de control para su programación en el circuito.
- **Circuito de programación externa**, que permite descargar bitstreams desde el PC o desde un sistema externo.
- **Circuito de configuración**: jumper y switch para la configuración de los diferentes modos de trabajo. Mediante el jumper se pueden configurar el modo de trabajo: entrenador o autónomo. Mediante un conmutador se selecciona si la memoria EEPROM se conecta a la FPGA o al puerto de control para programarla desde un sistema externo, sin tener que sacarla del zócalo.
- **Circuito de pruebas**, constituido por un led y un pulsador conectados a los pines de la FPGA, que permiten probar el correcto funcionamiento de la placa, configurando la FPGA con un diseño de prueba que lo use, como por ejemplo una compuerta inversora.
- **Puertos de expansión**. La placa incorpora 6 puertos de expansión, con 8 bits de datos, configurables para entrada o salida, y dos pines para la alimentación, de forma que los circuitos externos conectados se puedan alimentar directamente a través de los cables de bus. [16]

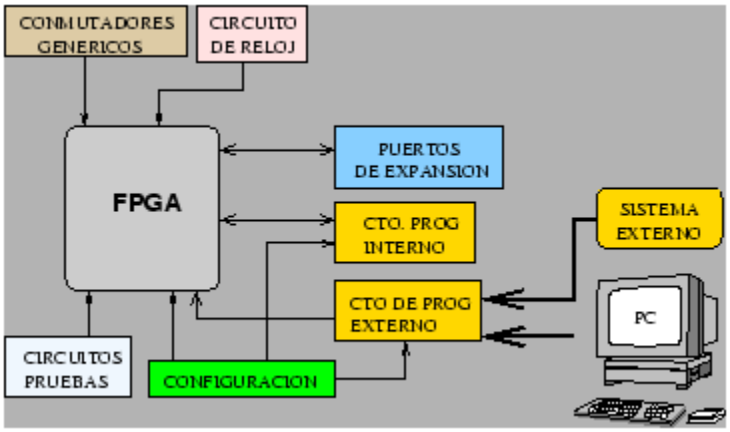


FIGURA III.11: TARJETA ENTRENADORA DE FPGA

Fuente:*http://www.iearobotics.com/personal/juan/publicaciones/art1/html/node5.html*

3.7 CONFIGURACION DE LAS FPGA

Dispositivos electrónicos como microprocesadores, microcontroladores, procesadores DSP y periféricos son totalmente programables a través de una serie de unos y ceros empaquetados en lo que se llama programa. Por eso se dice que un procesador se programa cuando se carga en la memoria de programa un archivo binario que el procesador ejecutará. Por otro lado, los FPGA son dispositivos lógicos reprogramables, pero en una FPGA el proceso de carga o programación es llamado configuración (no programación), usando un archivo de configuración llamado 'bits de configuración' ('configuration bitstream') que define la funcionalidad del FPGA.

La FPGA puede opcionalmente auto-cargar el archivo de configuración, por ejemplo desde una memoria externa no-volátil. Una FPGA puede también ser configurado por un dispositivo tipo microprocesador, DSP procesador, PC o algún otro dispositivo que pueda comunicarse inteligentemente con la FPGA.

Una FPGA puede funcionar como maestro o como esclavo todo depende del tipo de programación que se le realice, a continuación se detallarán los métodos más comunes de configurar una FPGA.

3.7.1 MODO MAESTRO

Cuando el FPGA auto-carga el archivo de configuración desde un dispositivo externo, se dice que el FPGA usa el Modo Maestro (Master Modo) para cargar su configuración. Esta carga se puede efectuar en un modo de transmisión de datos serie o modo paralelo. Normalmente en este modo el archivo de configuración reside en una memoria no-volátil externa al FPGA, y el FPGA genera la señal de reloj, CLK, y controla la comunicación con la memoria.

La Figura III.12 detalla diferentes opciones de configuración del FPGA en modo Master. La Figura III.12.a y Figura III.12.b muestran situaciones similares donde la FPGA extrae de la memoria externa su configuración. La Figura III.12.c detalla una interface SPI muy común para este tipo de configuración pues la memoria es barata, y ocupa muy poco lugar en el PCB. Una configuración de un FPGA tamaño medio usando este método Master-Serial puede tardar unos 500ms entre la comunicación y hasta que el FPGA esté listo para funcionar. Dentro del modo Master también está la opción de comunicarse en modo paralelo con una memoria externa tipo NOR Flash o PROM Flash. Las Figura III.12.d y figura III.12.e detallan éste tipo de interface. En este caso la gran ventaja es la disminución del tiempo de carga del archivo de configuración, mientras que la principal desventaja es la cantidad de líneas necesarias para rutear en el PCB.

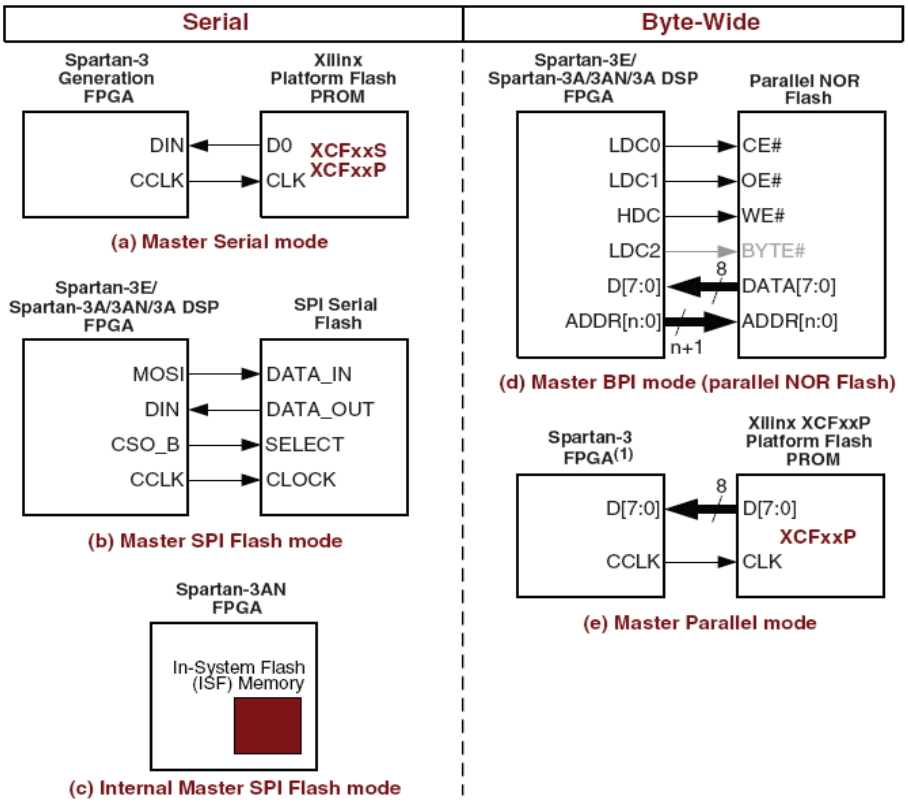


FIGURA III.12: DISTINTAS OPCIONES DE CONFIGURACIÓN DEL FPGA EN MODO MAESTRO

Fuente: *FPGA – Cristian Sisterna*

3.7.2 MODO ESCLAVO

Otra opción de configurar el FPGA es el modo en que el FPGA recibe los datos, el reloj y las señales de control del dispositivo que controla la comunicación entre ambos. En este caso el FPGA actúa como esclavo del otro dispositivo, por ello el modo se llama Modo Esclavo. La Figura III.13.detalla las distintas opciones para este método. Tal como ocurre en el Modo Master, en éste modo también existe una comunicación serie y una paralela. Por lo general el dispositivo maestro es un dispositivo inteligente tipo microcontrolador/microprocesador que se encarga de toda la comunicación. Una de las grandes ventajas de este método es que el archivo de configuración puede residir en cualquier parte del sistema, puede ser en un disco duro, en una memoria conectada a

Internet, o en la memoria que también contiene el archivo de programación del microcontrolador.

El modo de configuración conocido como JTAG, Figura III.13.b, es el modo más usado durante la etapa de debug o prototipo del sistema a implementar en la FPGA. Una vez seguidos todos los pasos para implementar el sistema en la FPGA, el último paso es generar el archivo de configuración de la FPGA. Para probar si el diseño sintetizado funciona correctamente se debe configurar la FPGA y comprobar su funcionamiento. Durante la etapa de prueba o prototipo, el software de desarrollo del fabricante de la FPGA provee una herramienta para configurar el FPGA a través de un cable tipo USB que se conecta por un lado a la PC y por esta conexión estándar es llamada JTAG. De este modo si ls FPGA no funciona como se esperaba, se corrigen los problemas en el código descriptivo del funcionamiento, se sintetiza, se genera un nuevo archivo de configuración y se configura de nuevo la FPGA.

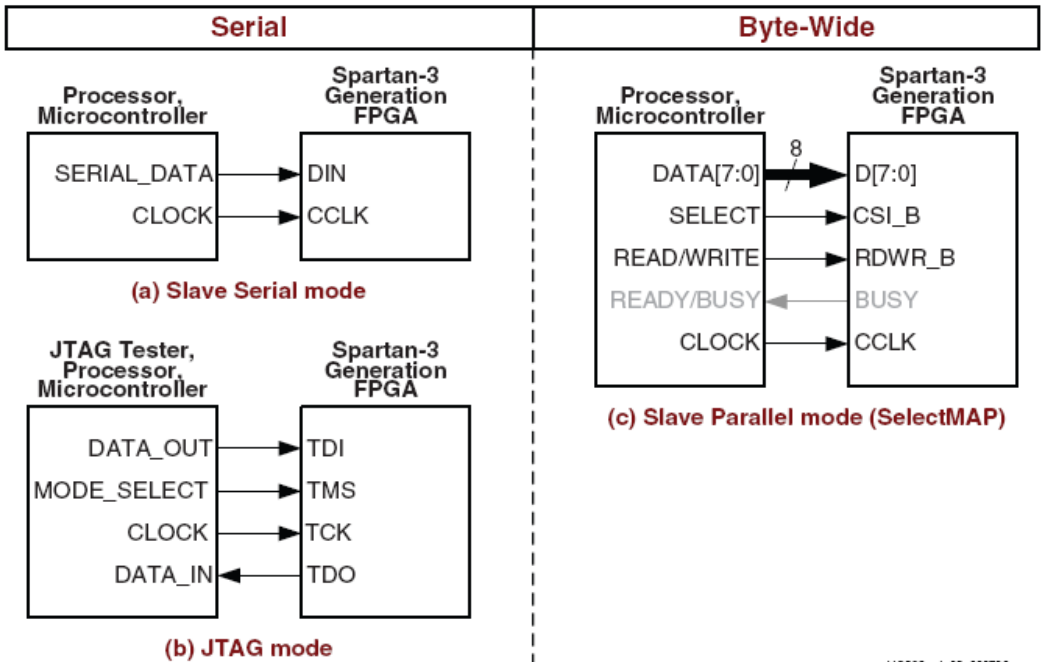


FIGURA III.13: DISTINTAS OPCIONES DE CONFIGURACIÓN DEL FPGA EN MODO ESCLAVO

Fuente: *FPGA – Cristian Sisterna*

3.8 CELDAS BASICAS DE CONFIGURACION

El elemento básico de un FPGA desde el punto de visto no-lógico, es decir que no tiene una función digital lógica, es la celda de configuración. Esta celda es la que va determinar la configuración de cada elemento lógico, por ejemplo si un flip-flop se va usar o no, y en caso de usarlo, si se configura como D o T. La celda de configuración también determina la configuración de los elementos de ruteo y de las interconexiones.

Existen en la actualidad cuatro tipos de celdas de configuración de una FPGA y son:

3.8.1 CELDA SRAM

La celda de configuración tipo SRAM, es una pequeña celda que se usa para mantener la configuración de cada parte configurable del FPGA. La gran ventaja de los FPGA basados en celdas SRAM es que utilizan un proceso de fabricación estándar. La fábrica de FPGA-SRAM tiene un procedimiento de fabricación muy conocido, que es el usado en la fabricación de memorias SRAM, y la enorme cantidad de memorias SRAM que se producen para el mercado digital, permite lograr costos de producción muy bajos, muy alta performance y trabajar con un proceso de fabricación muy amortizado y de gran rendimiento. Como es sabido las celdas de memoria tipo SRAM pueden ser reprogramadas un sinnúmero de veces, del mismo modo una FPGA basado en celdas de configuración SRAM puede ser reprogramado un infinito número de veces, aun cuando la FPGAs y esté montado y soldado en un circuito impreso (PCB). Esta reprogramación en PCB se denomina Programable En Circuito (In-Circuit Programmable, ISP). Esta tecnología SRAM es muy útil también para llevar a cabo una actualización rápida del sistema digital dentro del FPGA. Por ejemplo, algunos sintonizadores de televisión satelital usan FPGA en su sistema para sus diversas funciones. Cuando por alguno motivo se necesita actualizar el

sistema dentro de la FPGA, se envía por satélite un nuevo archivo de configuración. La lógica del sintonizador, que interpreta el comando de actualización, procede a actualizar el archivo de configuración de la FPGA (auto-actualización), evitando de este modo tener que cambiar el sintonizador. La gran desventaja de las celdas SRAM es que son volátiles, lo que significa que aún un simple pulso en la tensión de alimentación borraría la configuración de la FPGA, quedando prácticamente sin ninguna funcionalidad hasta que se lo configure de nuevo. Otra desventaja es que, debido a que la selección del camino de conexión entre los diferentes bloques lógicos (llamado ruteo), se basa en celdas SRAM, se provocan grandes retardos de ruteo, lo que es un problema en diseños que requieren un rendimiento muy alto. Una desventaja más de estos dispositivos es que, en el producto final, una vez que la FPGA está en el PCB y listo para ser comercializado, necesita de una memoria externa pequeña (tipo Flash, serie o paralela) que mantiene el archivo de configuración de la FPGA. La FPGA tiene una pequeña MEF que, cuando se le da tensión de alimentación, le indica a la FPGA que tiene que ir a buscar la configuración del mismo a una memoria externa. Luego lee la información de la memoria y configura las celdas SRAM. Para algunas aplicaciones, como sistemas médicos de emergencia, este tiempo de lectura de la memoria y configuración es muy largo (50 ~ 500ms). Para otras aplicaciones, como interfaces series, este tiempo pasa desapercibido.

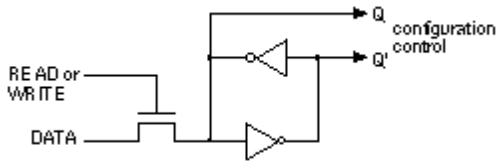


FIGURA III.14: CELDA BÁSICA SRAM DE CONFIGURACIÓN DE LOS FPGAS DE XILINX

Fuente:*FPGA – Cristian Sisterna*

La Figura III.14 muestra un ejemplo de la tecnología SRAM de configuración de los FPGAs. Esta celda básica está construida por dos inversores cruzados. La salida de la celda de configuración es conectada al transistor de paso, para conectarlo o desconectarlo. La celda se programa usando las líneas Write y Data.

3.8.2 CELDA ANTI-FUSE (ANTI-FUSIBLE)

Otro tipo de FPGA usa una celda llamada anti-fuse (anti-fusible) como celda básica de configuración. Esta celda consiste en una estructura microscópica que, a diferencia de un fusible regular, está normalmente abierta. Está compuesta de tres capas, las conductoras arriba y abajo, y la aisladora en el medio (Metal-Insulator-Metal, MIM). Para configurar el dispositivo se hace circular una cierta cantidad de corriente (~5mA), lo que causa una gran potencia de disipación en una área muy pequeña, provocando el derretimiento del aislante dieléctrico (tipo Oxide-Nitride-Oxide, ONO) entre los dos electrodos conductores (SiO2, Dióxido de Silicio) formando una unión permanente muy fina de unos 20nm. La estructura anti-fuse fue creada y se usa habitualmente en ciertas familias de los FPGAs de la empresa ACTEL. Esta estructura es conocida como Programmable Logic Interconnect Circuit Element, PLICE.

La estructura de la FPGA Anti-fuse de ACTEL, se muestra en la figura II.15:

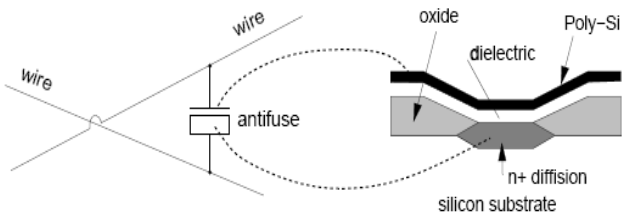


FIGURA III.15: ESTRUCTURA DEL ANTI-FUSE DE ACTEL

Fuente: *FPGA – Cristian Sisterna*

La estructura interna de la FPGA Anti-fuse de ACTEL en vista microscópica se puede ver en la figura II.16:

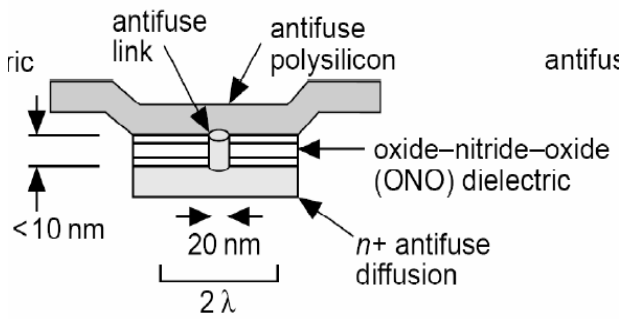


FIGURA III.16: VISTA MICROSCÓPICA DEL ANTI-FUSE DE ACTEL

Fuente: *FPGA – Cristian Sisterna*

La principal ventaja de los FPGA-Anti-Fuse es que no son volátiles, lo que para algunas aplicaciones es sumamente crítico, por ejemplo aplicaciones espaciales y aplicaciones médicas. También, debido a esta tecnología, los retardos de conexión entra los bloques lógicos son muy reducidos, por lo que el rendimiento de estos dispositivos es bastante elevado.

Como desventajas se tienen: primero, que requieren un proceso de fabricación específico, bastante complejo, lo que lleva a que el costo de los mismos sea bastante elevado comparado con los FPGA-SRAM (como mínimo 200 veces más caros). También requieren un programador especial para poder programar el anti-fusible, y la mayor desventaja es que una vez que se han configurado con cierta lógica, ésta no se puede cambiar, lo que es conocido técnicamente como One Time Programmable, OTP. El hecho de que estas FPGA sean OTP crea un proceso de verificación muy meticuloso de la lógica a ser programada, a fin de no tener que descartar este dispositivo tan caro por errores de diseño. Este proceso de verificación tan largo, implica también incrementar los costos de desarrollo del diseño, al

tener que disponer de más horas-ingeniero para tener listo y sin problemas el producto final. Una de las principales aplicaciones de este tipo de FPGA es para sistemas espaciales, ya que estas FPGA son tolerantes a las radiaciones de partículas de alta energía (los bits de configuración no pueden cambiar si son golpeados por una partícula).

Como para tener una idea, un FPGA anti-fusible de término medio cuesta cerca de U$S2.000, mientras que uno de similares características tipo SRAM puede rondar los U$S 100. La diferencia de costos es abrumadora, debido a que está en juego la estabilidad, seguridad y confiabilidad como por ejemplo un satélite que puesto en órbita cuesta cerca de U$S270.000.000. en una aplicación como estas no muchas opciones para elegir la tecnología a usar.

3.8.3 CELDAS TIPO FLASH

A diferencia de las SRAM, permiten mantener la configuración aún después de desconectada la alimentación del dispositivo. Hay también FPGA que tienen en el mismo dispositivo celdas Flash y celdas SRAM. Las celdas Flash que usan las FPGA-Flash, tienen como ventaja lo mejor del FPGA-SRAM y del FGPA-Anti-Fuse, son reprogramables y no son volátiles. Sin embargo todavía son caros, ya que usan una tecnología más cara que la SRAM, y las celdas FLASH se usan no solo para guardar la configuración en si del FPGA, sino que también para todo lo que es ruteo, lo que hace que la cantidad de celdas FLASH por FPGA sea un gran número. Los procesos de fabricación de celdas FLASH recién ahora son más comunes. Actualmente en el mercado están apareciendo más opciones de estos dispositivos, sobre todo en los tamaños de FPGA medianos-chicos, pero como la competencia es muy grande y a veces centavos marcan la diferencia, la demanda todavía no

es considerable. Otra desventaja para los FPGA FLASH es que el proceso de reconfiguración toma varios segundos.

La Figura III.17 detalla la estructura de una celda FLASH de la empresa ACTEL, que le llama FLASH Switch. Usa dos transistores que comparten la compuerta flotante, la que almacena la información de configuración. Uno es el transistor de sensado, el cual sólo se usa para escribir y verificar la tensión de compuerta flotante. El otro es el transistor de conexión (switching). Esta celda puede ser usada en el FPGA para conectar/separar rutas de conexiones o para configurar la lógica.

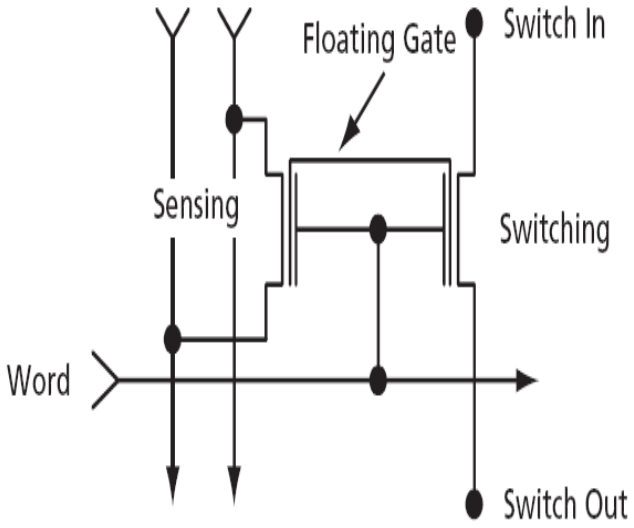


FIGURA III.17: ACTEL FLASH SWITCH

Fuente: *FPGA – Cristian Sisterna*

3.8.4 FPGAS BASADOS EN CELDAS FLASH Y SRAM

Finalmente hay algunos FPGA que tienen celdas Flash y SRAM en el mismo dispositivo. Las celdas Flash se utilizan para guardar los datos de configuración de la FPGA, mientras que las celdas SRAM para la configuración de la lógica de la FPGA. Cuando se da tensión de alimentación, las celdas SRAM se configuran en forma casi instantánea desde la Flash, resultando una configuración de la FPGA en menos de 1ms, a diferencia de una FPGA-

SRAM cuyo tiempo de configuración típicamente va de los 50 hasta los 500 ms (dependiendo del tamaño del dispositivo). Esta disponibilidad casi instantánea del FPGA lo hace muy útil para aplicaciones críticas en tiempo por ejemplo.Estas FPGA también permiten configurar solo las celdas SRAM, por ejemplo, durante el proceso de construcción del prototipo, sin tener que programar la Flash. Una gran ventaja, y que a veces es decisiva para el diseñador al elegir entre este tipo de FPGA o las FPGA-FLASH, es que, al no tener que acceder a un chip de memoria Flash externo, no hay una conexión física entre el FPGA y la memoria Flash que permita que la configuración del FPGA pueda ser expuesta, y de este modo vulnerada, para una posterior re-ingeniería (o ingeniería inversa) sobre el producto final. Se han descubierto muchos casos de productos copiados a través de la lectura de los datos de configuración (bitstream) disponibles en las rutas del circuito impreso que hay entre el FPGA y la memoria FLASH (éste problema también está presente en las FPGA-SRAM). Por supuesto que estos dispositivos son un poco más caros que los FPGA-SRAM, pero más baratos que los FPGA-FLASH, ya que en este caso las celdas FLASH solo se usan para guardar la configuración, por lo que la cantidad de estas celdas es mucho menor que en el caso de FPGA-FLASH. [17]

3.9 TABLA DE RESUMEN DE LAS CARACTERISTICAS DE LA FPGA

Las FPGAs son dispositivos orientados a para realizar una muy amplia gama de aplicaciones, desde un simple diseño digital básico combinacional, al igual a sistemas con microprocesador embebido, transmisiones tipo Ethernet, transmisiones de datos series a 3.5Gb/s, todo con la misma tarjeta.

Las FPGAs tienen características diversas, entre las que se mencionan en la siguiente tabla III.II:

TABLA III.II: CARACTERISTICA DE LA FPGA

Descripción	Características
Terminales de entrada y salida	Poseen una gran cantidad de terminales aproximadamente desde 100 hasta 1400.
Buffers de Entrada y salida programables	Control de sesgo, control de corriente, configuración del estándar de E/S , pull-up y pull-down configurables
Flips-Flops	Los dispositivos más grandes tienen unos 40.000 FFs
Tablas de Búsqueda	Aproximadamente 100.000
Memorias	Memoria (BRAM) de doble puerto, puerto simple, de hasta 18Mbits, configurables como RAM, ROM, FIFO y otras configuraciones
Bloques	Bloques dedicados de Multiplicación
Transceptores de transmisión	Transceptores para transmisión serie de muy alta velocidad , entre 1.5 a- 10.0Gb/s
Procesador en hardware embebido	Power-PC, ARM9
Procesadores descriptos en software HDL	8051, ARM3
Controladores	De reloj tipo Delay Lock Loop (DLL) y Phase Lock Loop (PLLs) de hasta 550MHz. De 2 a 8 controladores por dispositivo
Control	De impedancia programable por cada terminal de E/S
Interface	DDR/DDR2 SDRAM soportando interfaces de hasta 800 Mb/s
Interfaz	Con estándares de E/S tipo diferencial tales como LVDS, SSTL diferencial

3.10 FORMAS DE RECONFIGURACION LAS FPGA

a) Configuración la FPGA

Se descargan bits de configuración (todos están contenidos en un bitstream) para modificar la funcionalidad de los bloques lógicos y las interconexiones entre ellos. Generalmente los bits se almacenan en una memoria SRAM incluida en cada bloque

lógico, que es quien controla la interconexión y funcionalidad de los elementos lógicos de cada bloque lógico.

b) Tiempo que tarda reconfigurar una FPGA

El tiempo de configuración es proporcional al tamaño del bitstream. Las FPGA de grano fino tienen tiempos de configuración mayores, por la mayor cantidad de bloques funcionales que deben configurarse.

c) Formas que se pueden configurarse un FPGA

La tecnología permite diferentes formas de cambiar la configuración de los bloques y conexiones.

TABLA III.III: FORMAS DE RECONFIGURACIÓN DE LA FPGA

FORMA DE CONFIGURACIÓN	DESCRIPCION
Reconfiguración total, en tiempo de compilación o reconfiguración estática:	Cada vez que se realiza una nueva configuración, todo el FPGA se actualiza. Se tiene que detener la operación del FPGA, configurarlo todo y volver a ponerlo en marcha. Es para dispositivos con acceso secuencial a la memoria de Configuración. Hay penalizaciones de tiempo de configuración
Reconfiguración dinámica: **Único contexto**	Partes del FPGA se configuran nuevamente sin detener la ejecución del resto del FPGA. Existen tres variantes: **Único contexto:** La configuración entrante sustituye completamente a la anterior. Aunque solo una pequeña parte del bloque vaya a cambiar, todo el bloque cambia. **Multicontexto:** Existen varios bits de

Multicontexto	memoria de reconfiguración para cada bit de los elementos configurables. Los bits de memoria pueden considerarse como **múltiples planos** de información de configuración. Cada plano se configura totalmente como en el caso del contexto único. Se puede cargar una nueva configuración en un plano no activo mientras otro está activo. La reconfiguración implica un cambio de contexto, que Se realiza de forma rápida.
Reconfiguración parcial:	Algunos dispositivos la soportan. Se puede modificar una parte de la configuración mientras la otra sigue realizando la computación de forma no interrumpida. El plano de configuración funciona como una memoria RAM. Las direcciones especifican una localización para que se reconfigure. Con esto se puede cargar nuevas configuraciones en áreas del FPGA sin necesitar cambiar el contexto.
Reconfiguración pipeline:	Técnica para reducir el tiempo de descarga de los bitstreams parciales. (18)

3.11 PARAMETROS PARA ESCOGER UNA TARJETA FPGA

Al revisar las especificaciones de una tarjeta FPGA, generalmente están divididos en bloques lógicos configurables como segmentos, funciones fijas de lógica como multiplicadores, y recursos de memoria RAM en bloque embebidos. La tarjeta FPGA tiene otros componentes, pero éstos son generalmente los más importantes cuando se seleccionan y comparan FPGA para una aplicación en particular o depende del diseño a realizar.

La Tabla III.IV muestras especificaciones de recursos más usados para comparar las tarjetas FPGA dentro de varias familias de Xilinx. Como el número de compuertas ha sido una forma típica de comparar el tamaño de las tarjetas FPGA contra la tecnología ASIC, pero no describe realmente el número de componentes individuales dentro de la tarjeta FPGA. Ésta es una razón por las cuales Xilinx no especifica el número de compuertas de sistema equivalentes en la nueva familia Virtex-5. (19)

TABLA III.IV: ESPECIFICACIONES DE RECURSOS PARA ELEGIR FPGA

COMPONENTES	Virtex-II 1000	Virtex-II 3000	Spartan-3 1000	Spartan-3 2000	Virtex-5 LX30	Virtex-5 LX50	Virtex-5 LX85	Virtex-5 LX110
Compuertas	1 millón	3 millones	1 millón	2 millones	-----	-----	-----	-----
Flip – Flops	10,240	28,672	15,360	40,960	19,200	28,800	51,840	69,120
LUTs	10,240	28,672	15,360	40,960	19,200	28,800	51,840	69,120
Multiplicadores	40	96	24	40	32	48	48	64
RAM en Bloque (Kb)	720	1,728	432	720	1,152	1,728	3,456	4,608

3.12 APLICACIONES

En la Actualidad las FPGA están presentes en campos tan diversos como la automoción, la electrónica de consumo, o la investigación espacial. La tecnología FPGA tiene una aplicación horizontal en todas las industrias que requieren computación a alta velocidad. Tiene cabida en empresas que realizan las actividades indicadas en el listado siguiente:

- Alarmas.
- Arcos de seguridad de bancos.
- Climatización de autobuses.
- Comunicaciones por fibra óptica.

- Conducción Automática de Trenes.
- Control industrial.
- Control de instalaciones eléctricas.
- Electrónica de potencia.
- Electrónica espacial.
- Electrónica submarina.
- Electrónica aplicada a hoteles.
- Enclavadores Eléctricos.
- Ensayo de materiales.
- Equipos de medicina y radiología.
- Equipos de medidas de potencia.
- Equipos de medidas en audio.
- Equipos de transmisión de TV.
- Equipos ferroviarios.
- Equipos para parking públicos.
- Estaciones terrenas.
- Fabricación de azulejos y cerámicos.
- Herramientas EDA.
- Identificación dactilar.
- Ingeniería Nuclear.
- Internet.
- Tarjeta inteligente.
- Telemandos y Automatismos para Puertas.
- Maquinaria de empaquetamiento de frutas.
- Maquinaria para panaderías.
- Navegación GPS.
- Regulación de Tráfico.
- Seguridad Electrónica.
- Simuladores de vuelo.
- Sistemas de Emergencia tipo 112.
- Sistemas de Información a Viajeros.
- Sistemas de radiofrecuencia y microondas.
- Tarifación telefónica.
- Telefonía móvil.
- Telemedicina.
- Visión nocturna, etc. [20]

3.13 VENTAJAS Y DESVENTAJAS DE LAS FPGAs

TABLA III.5: VENTAJAS Y DESVENTAJAS de LAS FPGAs

VENTAJAS	DESVENTAJAS
❖ Son dispositivos reconfigurables. ❖ Bajo costo respecto a los ASIC. ❖ Los circuitos se "ejecutan" más rápido que en otros dispositivos reprogramables. ❖ Al ser circuitos digitales, la "ejecución" de cada bloque es en paralelo, no así en un micro controlador. ❖ Son útiles para realizar prototipos que luego serán llevados a ASIC si es necesario.	❖ Al estar basadas en RAM, pierden su configuración al suprimir la energía (hay soluciones a ello). ❖ Poseen retardos de propagación mayores a los existentes en ASIC o standard cells. ❖ Un procesador de alta velocidad (~GHz) se ejecuta mucho más rápido en ASIC que en una FPGA. ❖ Baja velocidad de operación y baja densidad lógica (poca Lógica implementable en un solo chip). Su baja velocidad se debe a los retardos introducidos por los conmutadores y las largas pistas de conexión.

CAPITULO IV

DESARROLLO EXPERIMENTAL DE GUIAS PRACTICAS PARA DISEÑO DIGITAL AVANZADO

En esta parte de la investigación se realiza el desarrollo de guías prácticas a través del CAD ISE DESIGN SUITE 14.2, Ver Anexo 2, manual CAD ISE DESIGN SUITE 14.2.

El cual empezaremos con guías prácticas de programación de algebra booleana como compuertas and, or, xor, etc. Luego con guías prácticas de sistemas secuenciales como decodificadores, multiflexores, etc. después sistemas combinacionales y finalmente la programación de la aplicación demostrativa que es parte de la investigación.

Esta parte de la investigación está desarrollada en base a la guía de sistemas digitales básicos de la escuela EIE-CRI de la ESPOCH, realizada por un profesor que dicta la asignatura el ingeniero José Guerra.

Los materiales comunes que se van utilizar en este capítulo es igual para todas las guías prácticas.

Materiales

- Computadora de mesa o portátil.
- CAD ISE DESIGN SUITE 14.2
- Tarjeta FPGA

Las guías prácticas a desarrollar son de los siguientes temas:

- Algebra Booleana.
- Sistemas Combinacionales.
- Sistemas Secuenciales.

4.1 GUIAS PRACTICAS DE ALGEBRA BOOLEANA

En este literal de la investigación se realizan codificaciones de programación de compuertas lógicas usas en sistemas digitales básicos.

4.1.1 COMPUERTA DE IGUALDAD

Marco Teórico

Compuerta de Igualdad

Este dispositivo electrónico posee tanto entradas como salidas la cual el valor que entrega a su salida (S) es exactamente el mismo valor que tiene en su entrada (A) pero amplificada.

TABLA IV.I: Tabla de Verdad de Igualdad

A	S
0	0
1	1

Representación gráfica de la compuerta de igualdad:

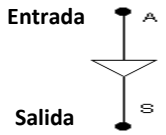


FIGURA IV.1. Compuerta de Igualdad

Fuente: *LOS AUTORES*

Como se puede observar en la tabla IV.I podemos decir que si el valor de ingreso (A) es 0 su valor de salida es cero pero si su valor de ingreso es 1 su valor de salida es uno.

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.II: Programación de Igualdad

VHDL	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 is
Port (a: **in** STD_LOGIC;
s : **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral of inicio2 **is**
begin
s<= a;
end Behavioral;
	mudule compigu (**input** a, **output** r);
	assign r= a;
	end module

4.1.2 COMPUERTA DE NEGACION (INVERSOR)

Marco Teórico

Compuerta de Negación

Esta compuerta de negación permite realizar una operación booleana, invirtiendo la señal de entrada (A) a la salida (S) como por ejemplo: Si la entrada es un 1 lógico en su salida tendremos un 0 lógico o puede tener un 0 a la entrada y de salida tenemos un 1 lógico.

(A = A')

La tabla de verdad de la compuerta inversora es como se muestra en la siguiente tabla.

TABLA IV.III: Tabla de Verdad Negación

A	S
1	0
0	1

Representación gráfica de la compuerta inversora:

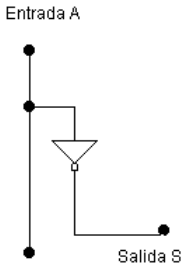


FIGURA IV.2. Compuerta Inversora

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.IV: Programación de Negación

VHDL	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
Port (a: **in** STD_LOGIC;
s : **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral **of** inicio2 **is**
 	
begin
s<= not a;
end Behavioral;
	mudule compneg (input a, output r);
	
	assign r= ~ a;
	
	endmodule

4.1.3 COMPUERTA DE UNION (OR)

Marco Teórico

La compuerta (OR) es un dispositivo electrónico la cual entrega a su salida (S) un valor de 1 si uno de sus valores de entrada (A) o (B) es 1 o ambos son 1 y un valor de 0 si sus valores de entrada de (A) y (B) son 0 en la tabla de verdad IV.V se puede observar de mejor manera.

TABLA IV.V: Tabla de Verdad OR

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

Representación gráfica de la compuerta de OR:

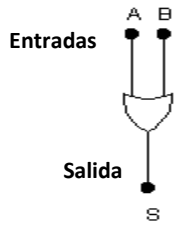


FIGURA IV.3: Compuerta de unión OR

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.VI: PROGRAMACION OR

VHDL Estilo flujo de datos- E.B	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164**.ALL**;
entity inicio2 **is**
Port (a,b: **in** STD_LOGIC;
s : **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral of inicio2 **is**
begin
s<= a **or** b;
end Behavioral;
	mudule compor (r,a,b);
	input a,b;
	output r;
	assign r= a | b;
	endmodule

VHDL Estilo funcional	
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (a,b: **in** STD_LOGIC; s : **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral **of** inicio2 **is**	

```
begin
comportamiento: process(a,b)
begin
        if (a='0' and b='0')then
            s<= '0';
        else
            s<='1';
         end if;
end process comportamiento;
end  Behavioral;
```

4.1.4 COMPUERTA DE NO UNION (NOR)

Marco Teórico

Compuerta de NOR

La compuerta NOR tiene todas sus entradas (A,B) un valor de 0 y como salida (S) nos da un 1 lógico, es decir que cuando hay un valor en 1 lógico siempre va tener un 0 lógico en sus salidas.

Como se muestra en la siguiente tabla de verdad de la compuerta:

TABLA IV.VII: Tabla de Verdad NOR

A	B	S
0	0	1
0	1	0
1	0	0
1	1	0

Representación gráfica de la compuerta NOR:

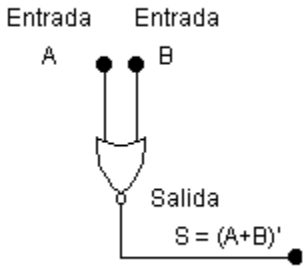


FIGURA IV.4: Compuerta de unión NOR

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.VIII: Programación de NOR

VHDL Estilo flujo de datos- E.B	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
Port (a,b: **in** STD_LOGIC;
s : **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral **of** inicio2 **is**
begin
s<= a **nor** b;
end Behavioral;
	mudule comnor (r,a,b);
	input a,b;
	output r;
	assign r=~ (a | b);
	endmodule

Estilo Flujo de datos utilizando **when - else**	
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (a,b: **in** STD_LOGIC; s : **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral **of** inicio2 **is** **begin**	

```
    s<= '1' when(a='0' and b='0')else
        '0';
end  Behavioral;
```

4.1.5 COMPUERTA DE INTERSECCION (AND)

Marco Teórico

La compuerta de inserción (AND) envía un valor de 1 a su salida (S) cuando los valores de entrada de (A) y (B) son 1 en caso uno de los valores de (A) o (B) es cero su valor a la salida 'S' es cero. Para mejor explicación mostramos la tabla de verdad en la siguiente tabla:

TABLA IV.IX: Tabla de Verdad AND

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

Representación gráfica de compuerta AND:

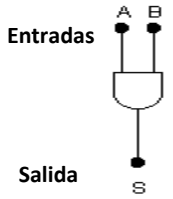


FIGURA IV.5: Compuerta AND

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.X: Programación de AND

VHDL Estilo flujo de datos- E.B	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
Port (a,b: **in** STD_LOGIC;
s : **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral **of** inicio2 **is**
begin
s<= a **and** b;
end Behavioral;
	module compand (r,a,b);
	input a,b;
	output r;
	assign r= a **&** b;
	endmodule

VHDL utilizando selector **with**	
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (a: **in** STD_LOGIC_vector(0 **to** 1); s : **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral **of** inicio2 **is** **begin** **with** a **select** s<= '1' **when** "00", '0' **when** "01", '0' **when** "10", '0' **when others**; **end** Behavioral;	

4.1.6 COMPUERTA DE NO INTERSECCION (NAND)

Marco Teórico

Compuerta de NAND

La compuerta NAND debe tener en sus entradas (A,B) un valor de 0 y como salida (S) nos da un 1 lógico, es decir que cuando todas sus entradas tienen igual un 1 lógico siempre va tener un 0 lógico en sus salidas.

Como se muestra en la siguiente tabla de verdad de la compuerta:

TABLA IV.XI: Tabla de Verdad

A	B	S
0	0	1
0	1	1
1	0	1
1	1	0

Representación gráfica de la compuerta NAND:

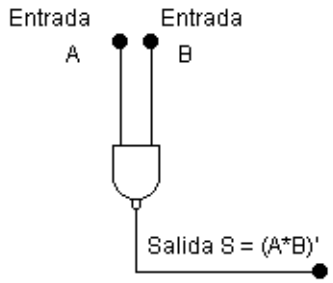


FIGURA IV.6: Compuerta NAND

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XII: Programación de NAND

VHDL Estilo flujo de datos- E.B	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
Port (a,b: **in** STD_LOGIC;
s : **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral **of** inicio2 **is**
begin
s<= a **nand** b;
end Behavioral;
	module compnand (r,a,b);
	input a,b;
	output r;
	assign r= ~ (a **&** b);
	endmodule

Estilo Flujo de datos utilizando **when - else**	
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (a,b: **in** STD_LOGIC; s : **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral **of** inicio2 **is** **begin** s<= '0' **when**(a='1' **and** b='1')**else** '1'; **end** Behavioral;	

4.1.7 COMPUERTA OR EXCLUSIVO (XOR = ⊕)

Marco Teórico

La compuerta conocida como OR Exclusiva (XOR) o disyunción exclusiva envía a su salida (S) un valor de 1 si exactamente uno de los valores de (A) o (B) es uno (pero no ambos) de dos condiciones.

Si lo representaríamos a nivel de las compuertas ya conocidas su salida (S) seria:

S = A'B + AB' o también S = A $\oplus$ B

TABLA IV.XIII: Tabla de Verdad XOR

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

Representación gráfica de la compuerta XOR

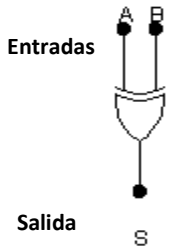


FIGURA IV.7: Compuerta XOR

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XIV: Programación de XOR

VHDL Estilo flujo de datos- E.B	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
  Port (a,b: **in** STD_LOGIC;
  s : **out** STD_LOGIC);
end inicio2;
	module compxor (r,a,b);
	input a,b;
	output r;
	assign r= a ^ b;
	endmodule

architecture Behavioral **of** inicio2 **is**
begin
s<= a **xor** b;
end Behavioral;	

VHDL por medio de operadores lógicos	

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
Port (a,b: **in** STD_LOGIC;
s : **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral **of** inicio2 **is**
begin
s<= (**not** a **and** b) **or** (a **and** not **b**);
end Behavioral;	

Para la resolver el ejercicio de una compuerta **XOR** se lo realizo por medio de operadores lógicos la cual consiste en tomar la ecuación e ir detallando termino a término de acuerdo a como este planteada la ecuación.

4.1.8 COMPUERTA DE NO OR EXCLUSIVO (NO XOR)

Marco Teórico

Compuerta de NO XOR

La compuerta NO XOR debe tener en todas sus entradas (A,B) un igual valor sea de 1 o 0 y como salida (S) nos da un 1 lógico, es decir que cuando todas sus entradas son diferentes de 1 o 0 lógico siempre va tener un 0 lógico en sus salidas.

Como se muestra en la siguiente tabla de verdad de la compuerta:

TABLA IV.XV: Tabla de Verdad NO XOR

A	B	S
0	0	1
0	1	0
1	0	0
1	1	1

Representación gráfica de la compuerta NOXOR:

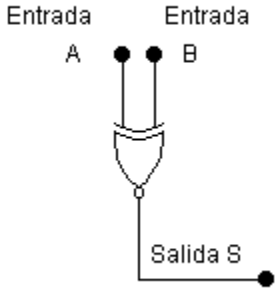


FIGURA IV.8: Compuerta NO XOR

Fuente: *LOS AUTORES*

Si lo representaríamos a nivel de las compuertas ya conocidas su salida (S) seria:

S = A'B' + AB o también S = (A $\oplus$ B)'

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XVI: Programación de NO XOR

VHDL Estilo flujo de datos- E.B	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
  Port (a,b: **in** STD_LOGIC;
  s : **out** STD_LOGIC);
end inicio2;
	module compnoxor (r,a,b);
	input a,b;
	output r;
	assign r= ~ (a ^ b);
	endmodule

architecture Behavioral **of** inicio2 **is**
begin
s<= **not**(a **xor** b);
end Behavioral;	

VHDL por medio de operadores lógicos	

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
Port (a,b: **in** STD_LOGIC;
s : **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral **of** inicio2 **is**
begin
s<= (**not** a **and not** b) **or** (a **and b**);
end Behavioral;	

A continuación se han realizado dos ejemplos del uso de compuertas lógicas múltiples:

EJEMPLO 1:

4.1.9 DISEÑO DE CIRCUITOS COMPUERTAS LOGICAS

Marco Teórico

Ejemplo de un cuarto de revelado modificado

Se ha acondicionado una habitación para revelar fotografías. Consiste en un circuito que tiene dos pulsadores para abrir la puerta: uno dentro de la habitación (P1) y otro fuera (P2). La puerta se abrirá cuando se pulse uno cualquiera de los dos pulsadores, o los dos a la vez.

Además, para asegurarse de que nadie entre cuando está revelando las fotografías, ha montado un interruptor que acciona una luz roja fuera de la habitación. Por seguridad, cuando la luz roja esté encendida, la puerta no se podrá abrir ni desde fuera ni desde dentro

1º: Identificación de entradas y salidas.

- Entradas: pulsadores P1, P2 e interruptor I
- Salidas: Luz roja en el exterior de la habitación y motor que abre y cierra la puerta

2º: Tabla de verdad

P1 = A

P2 = B

I = C

Luz roja = L

Motor = M

TABLA IV.XVII: Tabla de verdad ejemplo 1

A	B	C	L	M
0	0	0	0	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

Podemos observar que para que la puerta se abra (motor activado) tiene que ocurrir dos cosas:

- Que esté la luz roja apagada
- Que se activen cualquiera de los pulsadores, o los dos a la vez.

3º: Obtención de la expresión lógica

Para cada una de las salidas, sumamos las combinaciones de entradas que las activan. Así, tendremos que:

- L = a'b'c + a'bc + ab'c' + abc
- M = a'bc' + ab'c' + abc'

4º: simplificación por Mapa de Karnaugh

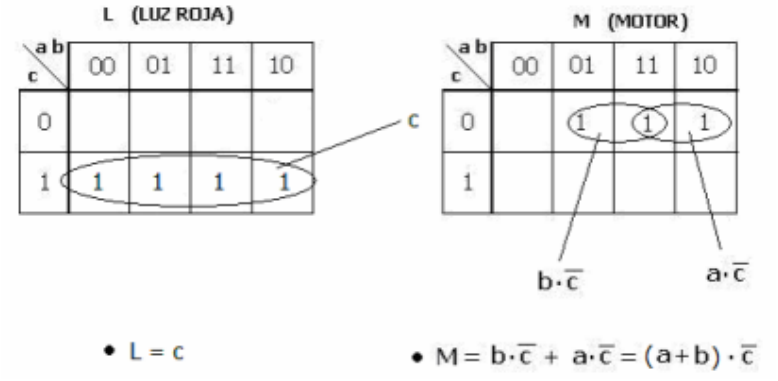


FIGURA IV.9: Simplificación de tablas

Fuente: *Practicas electrónicas. JOSE ORMENO. pag. 12*

5º: Circuito Simplificado

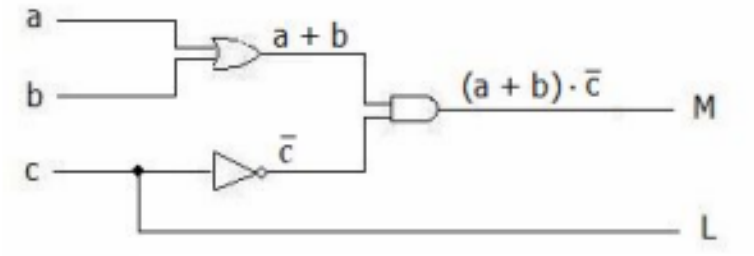


FIGURA IV.10: Circuito Simplificado

Fuente: *Practicas electrónicas. JOSE ORMENO. pag 12*

6º Montaje en la tarjeta digital

Para simplificar el montaje, vamos a sustituir el motor por un LED de color VERDE y mantendremos el LED ROJO para la luz roja. (21)

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XVIII: Programación del Ejemplo 1

VHDL por medio de operadores lógicos	VERILOG
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (a,b,c: **in** STD_LOGIC; M,L : **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral **of** inicio2 **is** **begin** M<= ((a **or** b)**and**(**not** c)) ; L<= c ; **end** Behavioral;	**module** inicio2 (**input** a,b,c, **output** M,L); **assign** M= (a | b) &(~c); **assign** L=c; **endmodule**

EJEMPLO 2:

4.1.10 DISEÑO DE CIRCUITOS COMPUERTAS LOGICAS

Objetivo General

Diseñar un circuito lógico que controle el encendido de la luz de carretera (larga) de un automóvil, de acuerdo con las siguientes especificaciones:

La luz debe encenderse cuando la luminosidad ambiental esté por debajo de un

Determinado nivel, a menos que exista niebla o se detecte un cruce con otro vehículo.

Igualmente debe encenderse, incluso con luminosidad ambiental elevada, si existe un obstáculo en la trayectoria, aunque exista niebla, pero no, si se detecta un cruce con otro vehículo.

1º: Identificación de entradas y salidas

S0 = Encendido de Luces

A = Luminosidad ambiental (1 = mayor, 0 = menor)

B = Niebla (1= existe; 0 = existe)

C = Cruce de vehículo (1= existe; 0 = no existe)

D = Obstáculo trayectoria (1= existe; 0 = no existe)

2º: Tabla de verdad

TABLA IV.XIX: Tabla de Verdad del ejemplo 2

A	B	C	D	S
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

3º: simplificación por Mapa de Karnaugh

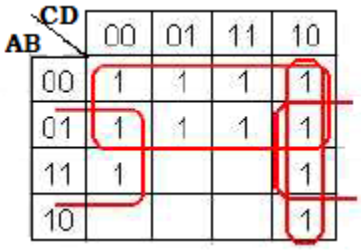


FIGURA IV.11: MAPA DE KARNAUGH

Fuente: *JÓSE GUERRA – PROBLEMAS RESUELTOS SISTEMAS DIGITALES Pag. 78*

S(A, B, C) = A’ + CD’ + BD’

4º: Circuito Simplificado

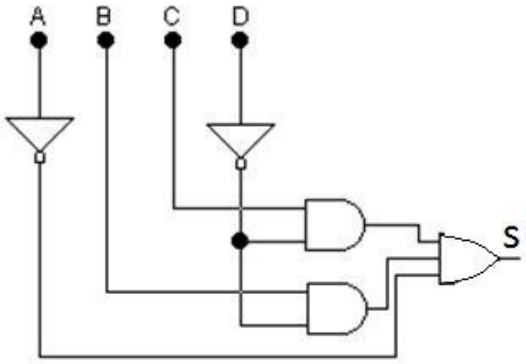


FIGURA IV.12: CIRCUITO SIMPLIFICADO

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XX: Programación del ejemplo 2

VHDL por medio de operadores lógicos	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
Port (a,b,c,d: **in** STD_LOGIC;
s: **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral of inicio2 **is**
begin
s<= ((**not** a) **or** (c **and not** d) **or** (b **and not** d));
end Behavioral;
	module inicio2 (a,b,c,d,s);
	input a,b,c,d;
	output s;
	assign s=((~a) |(c&(~d)) |(b&(~d)));
	endmodule

VHDL utilizando selector **with**	

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity inicio2 is
      Port ( e: in STD_LOGIC_vector(0 to 3);
        s : out  STD_LOGIC);
end inicio2;

architecture Behavioral of inicio2 is
begin
      with e select
      s<=     '0' when "1000",
              '0' when "1001",
              '0' when "1011",
              '0' when "1101",
              '0' when "1111",
              '1' when others;
end  Behavioral;
```

4.2 GUIAS PRACTICAS DE SISTEMAS COMBINACIONALES

4.2.1 MEDIO SUMADOR

Marco teórico

Medio sumador

El medio sumador es útil para sumar las cantidades de dos bits, de manera que el circuito de esta suma es el medio sumador. En el medio sumador existen dos entradas (A,B), son los dígitos a sumar, dos salidas (R, C) que son la respuesta (R) y el acarreo (C), donde lo último tiene mayor peso, la representación de circuito y su tabla de verdad se muestra a continuación.

TABLA IV.XXI: TABLA DE VERDAD MEDIO SUMADOR

	A	B	R	C
0	0	0	0	0
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1

Reducción de la expresión:

$$R(A, B) = A'B + AB'$$
$$= A \oplus B$$
$$C(A, B) = AB$$

Representación gráfica de la compuerta XOR

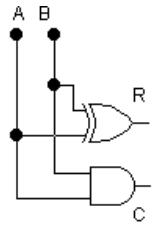


FIGURA IV.13: MEDIO SUMADOR

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XXII: Programación del medio Sumador

VHDL por medio de operadores lógicos	VERILOG
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (A,B: **in** STD_LOGIC; R,C: **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral **of** inicio2 **is** **begin** R<= A **XOR** B; C<= A **AND** B; **end** Behavioral;	**module** MedioS(**input** a,b, **output** r,c); **assign** r= a ^ b; **assign** c= a&b; **endmodule**

4.2.2 SUMADOR COMPLETO

Marco teórico

Sumador Completo

El sumador completo es la unión de los dos medios sumadores aquí se suman 3 bits, los dos bits de entrada (A, B) se suman en un medio sumador, la respuesta (R) se vuelve a sumar en un medio sumador con la tercera entrada (C), los acarreos de los dos medios sumadores se los suma y se tiene un acarreo total, ah toda esta operación se le denomina sumador completo. En la siguiente tabla se muestra la tabla de verdad del sumador completo.

TABLA IV.XXIII: TABLA DE VERDAD SUMADOR COMPLETO

	A	B	C	R	C0
0	0	0	0	0	0
1	0	0	1	1	0

2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

Simplificación del circuito

$R = A \oplus B \oplus C$

$C0 = (A \oplus B)C + AB$

Representación gráfica del Sumador Completo

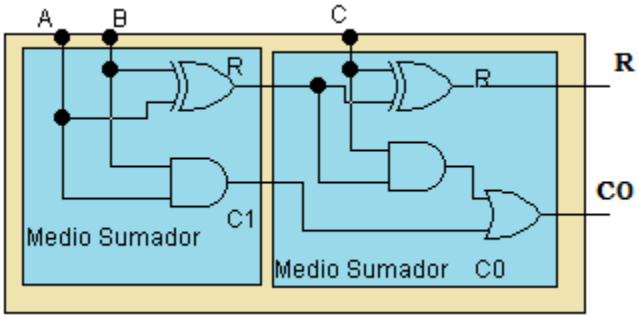


FIGURA IV.14: SUMADOR COMPLETO

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XXIV: Programación del Sumador Completo

VHDL por medio de operadores lógicos	VERILOG
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (A,B,C: **in** STD_LOGIC; R,C0: **out** STD_LOGIC); **end** inicio2;	**module** SumaC (**input** a,b,c, **output** r, c0); **assign** r= (a ^ b ^ c); **assign** c0= ((a ^ b) **&** c) | (a **&** b); **endmodule**

```
architecture Behavioral of inicio2 is
begin
        R<= (A XOR B XOR C);
       C0<= (((A XOR B)AND C) OR
(A AND B));

end  Behavioral;
```

4.2.3 MEDIO RESTADOR

Marco Teórico

Un medio retador es un circuito combinacional que sustrae dos bits y produce su diferencia e indica si ha tomado un préstamo.

TABLA IV.XXV: Tabla de Verdad medio Restador

M	S	R	P
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

M: Representa el minuendo

S: Representa al sustraendo

R: Representa el resultado de la resta.

P: Representa si ha habido algún préstamo

El resultado de las salidas quedaría de la siguiente manera:

$R = M'S$

$$P = M \oplus S$$

Representación gráfica para un 1/2 restador

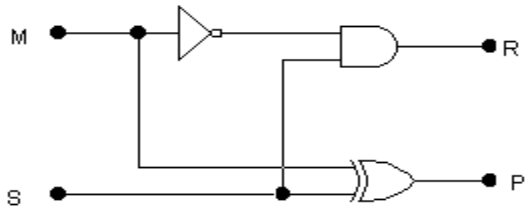


FIGURA IV.15: De un 1/2 Restador

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XXVI: Programación medio restador

VHDL por medio de operadores lógicos	VERILOG
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (M,S: **in** STD_LOGIC; R,P: **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral **of** inicio2 **is** **begin** **R<= (NOT** M **AND** S**) ;** **P<= (**M **XOR** S**);** **end** Behavioral;	**module** MedioR(**input** m,s, **output** r,p); **assign** r= (~ m) **&** s; **assign** p= m ^ s; **endmodule**

4.2.4 RESTADOR COMPLETO

Marco Teórico

Intervienen tres bits en esta operación de resta, esto es cuando consideramos a más de los bits a restar un bit de préstamo, es decir, se ha efectuado una operación de resta en una columna anterior que ha generado dicho bit, el circuito se convierte en un restador completo.

TABLA IV.XXVII: Tabla de Verdad Restador Completo

A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$S= A'B'C_{IN} + A'BC_{IN}' + ABC_{IN} + AB'C_{IN}'$

$S = B \oplus A \oplus C_{IN}$

$C_{OUT=} A'C_{IN} + A'B + BC_{IN}$

Gráfica de un restador completo

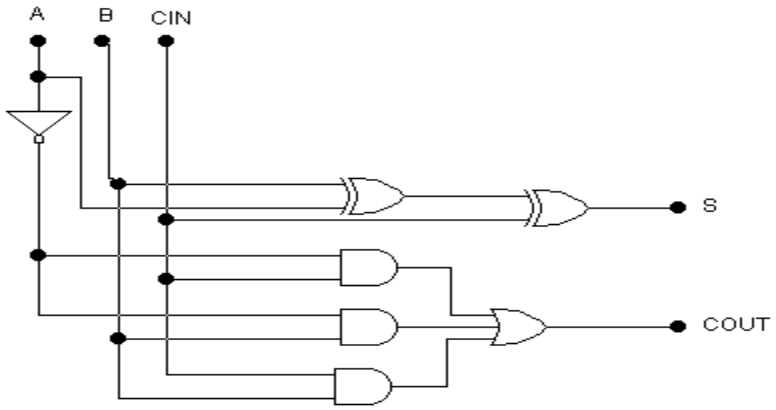


FIGURA IV.16. Restador completo

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XXVIII: Programación Restador Completo

VHDL por medio de operadores lógicos	VERILOG
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (A,B,C_{IN}: **in** STD_LOGIC; S,C_{0UT}: **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral **of** inicio2 **is** **begin** S<= (A **XOR** B **XOR** C_{IN}); C_{0UT}<=((**NOT** A **AND** C_{IN}) **OR** (**NOT** A **AND** B) **OR** (B **AND** C_{IN})); **end** Behavioral;	**module** ResctaC(**input** a,b,cin , **output** s, cout); **assign** s = b ^ a ^ cin; **assign** cout = ((~a)**&**cin) | ((~a)**&**b)| (b **&** cin); **endmodule**

4.2.5 CODIFICADOR

Marco teórico

Codificador

Un codificador es un circuito combinacional que permite pasar un conjunto de información sin codificar (dígitos decimales, octales, hexadecimales, etc) en un conjunto que responda a la estructura de un código (BCD, exs3, binario, etc).

El circuito que actué como codificador tendrá N salidas cuando se requiera codificar M entradas, teniendo un relación de 2*N entradas lo que será igual a M salidas, de tal manera podemos decir que M informaciones diferentes se pueden representar mediante grupos de N bits.

Mediante un ejemplo podemos ver mejor los conceptos dichos antes. Diseñar un codificador para los dígitos en base ocho. Ver la siguiente tabla de verdad:

TABLA IV.XXIX: Tabla de un codificador

	0	1	2	3	4	5	6	7	S0	S1	S2
0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	1
2	0	0	1	0	0	0	0	0	0	1	0
3	0	0	0	1	0	0	0	0	0	1	1
4	0	0	0	0	1	0	0	0	1	0	0
5	0	0	0	0		1	0	0	1	0	1
6	0	0	0	0	0	0	1	0	1	1	0
7	0	0	0	0	0	0	0	1	1	1	1

Simplificación del circuito

S0 = 4+5+6+7

S1 = 2+3+6+7

S2 = 1+3+5+7

Representación gráfica del codificador:

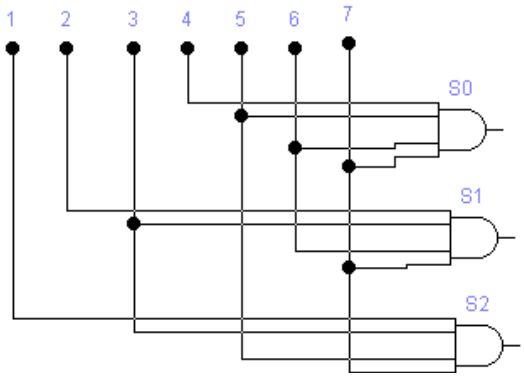


FIGURA IV.17: CODIFICADOR

Fuente: LOS AUTORES

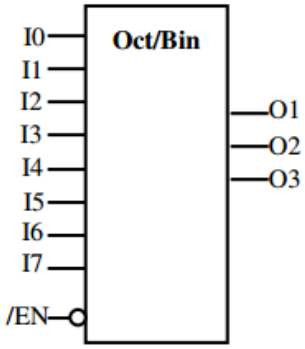


FIGURA IV.18: CODIFICADOR

Fuente: AUTOR*: http://1.bp.blogspot.com/-8KGiLO64Wmw/UMjmH1IOZDI/AAAAAAAAAB8/CZnf_QHm-eE/s1600/codificador.png*

El codificador diseñado funcionaria únicamente cuando una sola salida en toda la combinación presente un valor de uno pues si se activan simultáneamente dos o más entradas la salida generada seria incorrecta.

Pasa solucionar este problema se diseñan circuitos que aseguran la codificación de una sola entrada dado a estas prioridades, donde la línea de mayor prioridad seria la que se codifique indistintamente de cuantas estén activas, a este tipo de circuito se le conoce como codificador de prioridad.

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XXX: Programación de un codificador

VHDL por medio de operadores lógicos	VERILOG
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (A,B,C,D,E,F,G: **in** STD_LOGIC; S0,S1,S2: **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral of inicio2 **is** **begin** S0<=(D **AND** E **AND** F **AND** G); S1<=(B **AND** C **AND** F **AND** G); S2<=(A **AND** C **AND** E **AND** G); **end** Behavioral;	**module** Codificador(**input** a,b,c,d,e,f,g,h, **output** s0,s1,s2); **assign** s0 = e |f | g | h; **assign** s1 = c | d | g | h; **assign** s2 = b | d | f | h; **endmodule**

Para resolver la programación de este ejercicio le dimos a cada número una letra desde la "A hasta la F" en orden ascendente y se aplicaron las técnicas que ya hemos estudiado para programar el diseño digital.

4.2.6 DECODIFICADOR

Marco teórico

Decodificador

Los decodificadores son circuitos que realizan la función inversa a los circuitos codificadores, es decir, toman la información codificada en n bits y generan la información original, para ellos usaremos el ejemplo hecho en la anterior guía de codificador, para que dé binario decodifique a octal. Se muestra en la siguiente tabla de verdad:

TABLA IV.XXXI: Tabla de un Decodificador

	A	B	C	0	1	2	3	4	5	6	7
0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
2	0	1	0	0	0	1	0	0	0	0	0
3	0	1	1	0	0	0	1	0	0	0	0
4	1	0	0	0	0	0	0	1	0	0	0
5	1	0	1	0	0	0	0		1	0	0
6	1	1	0	0	0	0	0	0	0	1	0
7	1	1	1	0	0	0	0	0	0	0	1

Simplificación de circuito

0 = A’B’C’

1 = A’B’C

2 = A’BC’

3 = A’BC

4= AB’C’

5= AB’C

6= ABC’

7= ABC

Representación gráfica del decodificador del ejemplo:

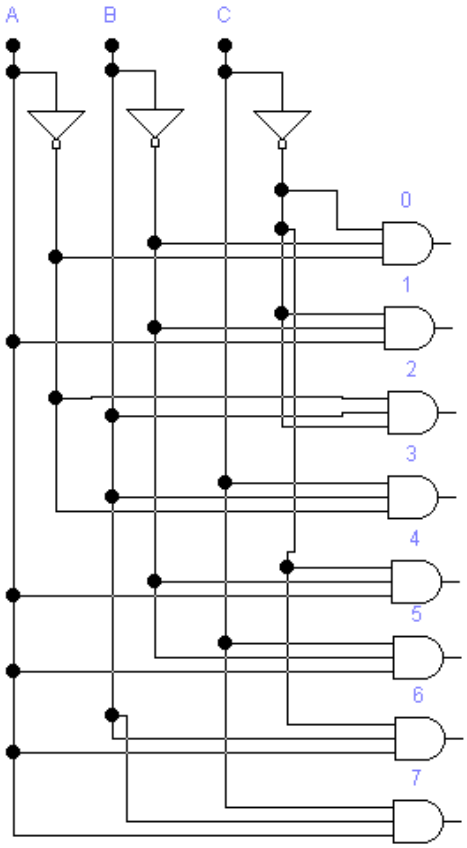


FIGURA IV.19: DECODIFICADOR

Fuente: *LOS AUTORES*

Simbología de un decodificador:

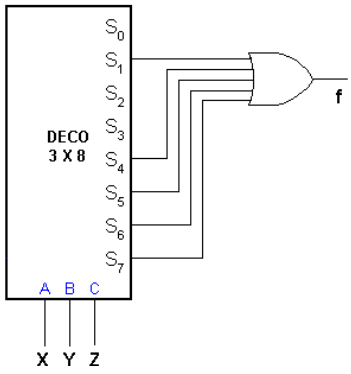


FIGURA IV.20: DECODIFICADOR 3 x 8

Fuente: *LOS AUTORES*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XXXII: Programación de un decodificador

VHDL por medio de operadores lógicos	VERILOG
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 is **Port** (A,B,C: in STD_LOGIC; S0,S1,S2,S3,S4,S5,S6,S7: **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral of inicio2 **is** **begin** S0<=(**NOT** A **AND NOT** B **AND NOT** C); S1<=(**NOT** A **AND NOT** B **AND** C); S2<=(**NOT** A **AND** B **AND NOT** C); S3<=(**NOT** A **AND** B **AND** C); S4<=(A **AND NOT** B **AND NOT** C); S5<=(A **AND NOT** B **AND** C); S6<=(A **AND** B **AND NOT** C); S7<=(A **AND** B **AND** C); **end** Behavioral;	**module** Decodificador(**input** a,b,c, **output** s0,s1,s2,s3,s4,s5,s6,s7); **assign** s0=(~a)**&**(~b)**&**(~c); **assign** s1=(~a)**&**(~b)**&**c; **assign** s2=**(**~a)**&**b**&**(~c); **assign** s3=(a)**&**b**&**c; **assign** s4=a**&**(~b)**&**(~c); **assign** s5=a**&**(~b)**&**c; **assign** s6=a**&**b(~c); **assign** s7=a**&**b**&**c; **endmodule**
VHDL por medio de IF –THEN	

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
 	
entity apren3 **is**
Port (selece : **in** STD_LOGIC_vector(2 **downto** 0);
selecs : **out** STD_LOGIC_vector(7 **downto** 0));
end apren3;
 	
architecture Behavioral **of** apren3 **is**
begin
proceso:**process** (selece)	

```
begin
                if selece ="000" then
                        selecs<= "10000000";
                elsif selece="001" then
                        selecs<= "01000000";
    elsif selece="010" then
                        selecs<= "00100000";
    elsif selece="011" then
                        selecs<= "00010000";
    elsif selece="100" then
                        selecs<= "00001000";
    elsif selece="101" then
                        selecs<= "00000100";
    elsif selece="110" then
                        selecs<= "00000010";
                else
                        selecs<= "00000001";
                end if;
end process proceso;
end Behavioral;
```

4.2.7 MULTIPLEXOR

Marco Teórico

Es un circuito combinacional el cual posee un conjunto de entradas tanto de datos como de control y un solo pin de salida. El trabajo que debe de realizar el multiplexor es el de pasar la información que se encuentra en la entrada de datos, donde los pines se irán activando de acuerdo al dato que se le envié en sus pines de control para que así la información pueda pasar desde su entrada a la salida única que posee este dispositivo.

En la siguiente grafica se mostrara de qué manera se representa un multiplexor de 4 x 1 con línea de activación.

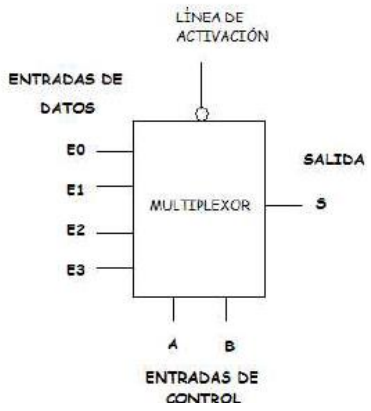


FIGURA IV.21: Multiplexor

Fuente: *Sistema digitales I- José Guerra-pag. 102*

TABLA IV.XXXIII: Tabla de Verdad Multiplexor

A	B	S
0	0	E0
0	1	E1
1	0	E2
1	1	E3

Cada vez que se vayan cumpliendo cada una de las combinaciones se ira activando cada una de las salidas y empezara a enviar la información cuando se envié una señal a través de la línea de activación.

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XXXIV: Programación de un multiplexor

VHDL utilizando selector **with**	VHDL
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** inicio2 **is** **Port** (E0,E1,E2,E3: **in** STD_LOGIC SELEC: **IN** STD_LOGIC_VECTOR(1 **DOWNTO** 0); s : **out** STD_LOGIC); **end** inicio2; **architecture** Behavioral **of** inicio2 **is** **begin** **with** SELEC **select** s<= E0 **when** "00", E1 **WHEN** "01", E2 **WHEN** "10", E3 **when others;** **end** Behavioral;	**module** multiflexor(a,b,c,d,r,s); **input** [1:0] r ; **input** a,b,c,d; **output** s; **reg** s; **always @**(r **or** a **or** b **or** c **or** d) **case** (r) 2'b00: s= a; 2'b01: s= b; 2'b10: s= c; 2'b11: s= d; **default**: s=0; **endcase** **endmodule**
VHDL utilizando IF	

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
Port (E0,E1,E2,E3: **in** STD_LOGIC;
SELEC: **IN** STD_LOGIC_VECTOR(1
DOWNTO 0);
s : **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral **of** inicio2 **is**
begin
proceso:**process** (E0,E1,E2,E3,SELEC)
BEGIN
if selec ="00" **then**
s<= E0;
elsif selec ="01" **then**
s<= E1;
elsif selec ="10" **then**	

```
            s<= E2;
        else
            s<= E3;
        end if;
end process proceso;
end  Behavioral;
```

4.2.8 DEMULTIPLEXOR

Marco Teórico

Los demultiplexores son circuitos combinacionales los cuales me permiten realizar una acción inversa a la realizada en los multiplexores debido a que me permiten enviar el valor de la entrada de datos (E) a alguna de sus salidas (S0, S1, S2, s3) la cual estas salidas se activaran de acuerdo al valor que se encuentre en las líneas de control (A, B) que al igual que los multiplexores darán el paso cuando la línea de activación lo indique.

En la siguiente grafica mostraremos la forma que tiene un Demultiplexor 1x4.

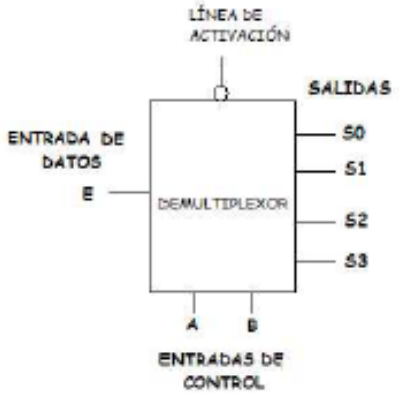


FIGURA IV.22: Demultiplexor

Fuente: *Sistema digitales I- José Guerra-pag. 104*

TABLA IV.XXXV: Tabla de Verdad Demultiplexor 1x4

A	B	S0	S1	S2	S3
0	0	E	0	0	0
0	1	0	E	0	0
1	0	0	0	E	0
1	1	0	0	0	E

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

Tabla IV.XXXVI: Programación de un Demultiplexor 1X4

VHDL resuelto con IF	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
 	
entity DEMUX **is**
Port (A : in STD_LOGIC_VECTOR (1 downto 0);
LA : **in** STD_LOGIC;
E : **in** STD_LOGIC;
S : **OUT** STD_LOGIC_VECTOR (3 **downto** 0));
end DEMUX;
 	
architecture Behavioral **of** DEMUX **is**
begin
PROCESS (A,LA)
begin
IF LA='1' **THEN**
if e ='0' **then**
S <= "0000";
else
	module demuxor (
	demin,e,s0,s1,s2,s3);
	input [1:0] demin;
	input e;
	output s0,s1,s2,s3 ;
	reg s0,s1,s2,s3;
	always @ (demin)
	case (demin)
	2'b00: **if** (e==1) {s0,s1,s2,s3}= 4'b1000;
	else {s0,s1,s2,s3}=4'b0000;
	2'b01: **if** (e==1) {s0,s1,s2,s3}= 4'b0100;
	else {s0,s1,s2,s3}=4'b0000;
	2'b10: **if** (e==1) {s0,s1,s2,s3}= 4'b0010;
	else {s0,s1,s2,s3}=4'b0000;
	2'b11: **if** (e==1) {s0,s1,s2,s3}= 4'b0001;
	else {s0,s1,s2,s3}=4'b0000;

```
            S <= "1000";
            end if;
elsif a = "01" then
    if e ='0' then
     S <= "0000";
  else
            S <= "0001";
            end if;
elsif a = "10" then
            if e ='0' then
     S <= "0000";
  else
            S <= "0001";
            end if;
elsif a = "11" then
            if e ='0' then
     S <= "0000";
  else
            S <= "0001";
            end if;
END IF;
END PROCESS;
end Behavioral;
```

```
endcase
endmodule
```

Se han realizado dos ejemplos de los sistemas combinacionales :

EJEMPLO 1:

4.2.9 MULTIPLEXOR

Se desea diseñar un sumador completo con multiplexores de 2x1 y el menor número de compuertas.

TABLA IV.XXXVII. Tabla de Verdad Ejercicio Multiplexor

A	B	C	R	CA
0	0	0	0	0

0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Reducimos la tabla eliminando la entrada (C) que la relacionamos con la salida (R, C) en cada combinación.

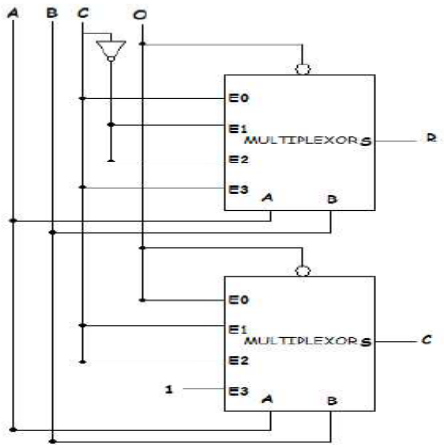


FIGURA IV.23: Multiplexor-EJEMPLO

Fuente: *Sistema digitales I- José Guerra-pag. 104*

TABLA IV.XXXVIII: Tabla de Verdad Multiplexor REDUCIDA

A	B	R	CA
0	0	C	0
0	1	C’	C
1	0	C’	C
1	1	C	1

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XXXIX: Programación de un Sumador C multiplexor

VHDL resuelto con IF	VERILOG

l**ibrary** IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
entity inicio2 **is**
Port (C,O: in STD_LOGIC;
SELEC: IN STD_LOGIC_VECTOR(1 **DOWNTO** 0);
R,CA: **out** STD_LOGIC);
end inicio2;
 	
architecture Behavioral **of** inicio2 **is**
begin
proceso:**process** (C,O,SELEC)
BEGIN
IF O= '1' **THEN**
if selec ="00" **then**
R<=C;
CA<='0';
elsif selec ="01" **then**
R<=**NOT** C;
CA<=C;
elsif selec ="10" then
R<=**NOT** C;
CA<=C;
else
R<=C;
CA<='1';
end if;
END IF;
end process proceso;
end Behavioral;
	module Sucmux(a,b,c,r,s,c0);
	input [1:0] r ;
	input a,b,c;
	output s,c0;
	reg s,c0;
	always @(r or a or b or c)
	case (r)
	2'b00: **begin** s= c; c0=0; **end**
	2'b01: **begin** s=~c; c0=c; **end**
	2'b10: **begin** s=~c; c0=c; **end**
	2'b11: **begin** s=c; c0=1; **end**
	endcase;
	Endmodule

4.2.10 SUMADOR COMPLETO CON DECODIFICADOR DE 4 A 16

Diseñar un sumador completo para números de dos bitss, con un decodificador de 4 x 16 y el menor número de compuertas. Tabla de valores del ejercicio ver en la siguiente tabla:

Tabla IV.XL: Tabla de verdad del ejemplo 3

	A	B	C	D	S0	S1	S2
0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
2	0	0	1	0	0	1	0
3	0	0	1	1	0	1	1
4	0	1	0	0	0	0	1
5	0	1	0	1	0	1	0
6	0	1	1	0	0	1	0
7	0	1	1	1	1	0	0
8	1	0	0	0	0	1	0
9	1	0	0	1	0	1	1
10	1	0	1	0	1	0	0
11	1	0	1	1	1	0	1
12	1	1	0	0	0	1	1
13	1	1	0	1	1	0	0
14	1	1	1	0	1	0	1
15	1	1	1	1	1	1	1

Representación del circuito se muestra en la siguiente figura:

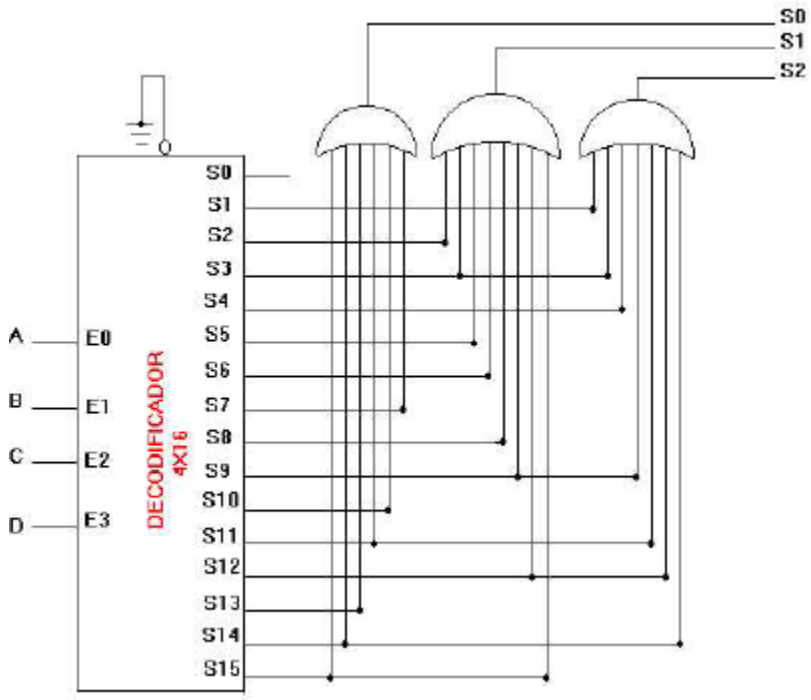


FIGURA IV.24: Decodificador 4x16

Fuente: *Sistema digitales I- José Guerra-pag. 126*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XLI: Programación del ejemplo 3

VHDL RESOLUCION CON IF y Case –is	VERILOG
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **entity** DEC416 **is** **Port** (A : **in** STD_LOGIC_VECTOR (3 **downto** 0); E : **in** STD_LOGIC; S : **OUT** STD_LOGIC_VECTOR (2 **downto** 0));	**module** Scdeco416(r,s0,s1,s2); **input**[3:0]r; **output** s0,s1,s2; **reg** s0, s1, s2; **always** @ (*) **case** (r) 4'b0000:{s0,s1,s2}=3'b000; 4'b0001:{s0,s1,s2}=3'b001; 4'b0010:{s0,s1,s2}=3'b010; 4'b0011:{s0,s1,s2}=3'b011; 4'b0100:{s0,s1,s2}=3'b001;

```
end DEC416;

architecture Behavioral of DEC416
is
begin
PROCESS (A,E)
BEGIN
        IF E='0' THEN
        S <= "000";
        ELSE
CASE A IS
WHEN "0000" => S <= "000";
WHEN "0001" => S <= "001";
WHEN "0010" => S <= "010";
WHEN "0011" => S <= "011";
WHEN "0100" => S <= "001";
WHEN "0101" => S <= "010";
WHEN "0110" => S <= "010";
WHEN "0111" => S <= "100";
WHEN "1000" => S <= "010";
WHEN "1001" => S <= "011";
WHEN "1010" => S <= "100";
WHEN "1011" => S <= "101";
WHEN "1100" => S <= "011";
WHEN "1101" => S <= "100";
WHEN "1110" => S <= "101";
WHEN "1111" => S <= "111";
WHEN OTHERS => S <= "000";
END CASE;
END IF;
END PROCESS;
end Behavioral;
```

```
4'b0101:{s0,s1,s2}=3'b010;
4'b0110:{s0,s1,s2}=3'b010;
4'b0111:{s0,s1,s2}=3'b100;
4'b1000:{s0,s1,s2}=3'b010;
4'b1001:{s0,s1,s2}=3'b011;
4'b1010:{s0,s1,s2}=3'b100;
4'b1011:{s0,s1,s2}=3'b101;
4'b1100:{s0,s1,s2}=3'b011;
4'b1101:{s0,s1,s2}=3'b100;
4'b1110:{s0,s1,s2}=3'b101;
4'b1111:{s0,s1,s2}=3'b111;
endcase
endmodule
```

4.3 GUIAS PRACTICAS DE SISTEMAS SECUENCIALES

En esta sección trabajaremos con sistemas secuenciales, estos sistemas son aquellos donde sus salidas dependen no solo de sus entradas actuales además también dependen de sus entradas generadas anteriormente.

Para poder realizar los diseños de los distintos tipos de sistemas secuenciales también podemos tener en cuenta que la mayoría de elementos para este tipo de circuitos tienen elementos de memoria los cuales se tienen que describir en términos de lógica secuencial.

Se desarrollaran algunos circuitos secuenciales en los cuales encontraremos flip – flop, contadores etc.

4.3.1 FLIP – FLOP TIPO 'D'

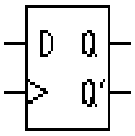


FIGURA IV.25: FLIP-FLOP D

Fuente: *Los autores*

Este dispositivo electrónico posee la característica de preservar su valor (D) de entrada en la salida (Q) esto quiere decir que si (D=1) la salida (Q=1) pero si su valor de entrada (D=0) asi mismo su valor de salida (Q=0.) , donde los valores de entrada van a ser enviados a la salida cada vez que exista una transición de reloj.

TABLA IV.XLII: Tabla de Verdad FLIP-FLOP ‘D’

A	B
0	0
0	1

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

VHDL

Para resolver este ejercicio encontramos un término nuevo el cual es el atributo **(EVEN).** Este es un evento del lenguaje que sirve para definir características que se pueden asociar con cualquier tipo de datos, objeto o entidades. Este atributo se utiliza para describir un hecho u ocurrencia de una señal en particular. En el código que se realizó para este ejercicio podemos observar que la condición **if** clk’ **event** es cierta solo cuando ocurre un cambio de valor es decir un suceso de cambio de valor del reloj.

VERILOG

Para la programación que se está realizando en verilog nos aparece un término nuevo que al igual que el evento que se produce en VHDL existe también un evento similar en Verilog pero llamado **posedge** este bloque siempre se ejecuta, cuando se produce el evento de disparo, el código dentro comienza y se ejecuta al final y el una vez más espera siempre el bloque para el próximo **posedge** de reloj, este PROCESO de esperar y ejecutar en el evento se repite hasta que se detiene la simulación

TABLA IV.XLIII: Programación de un flip – flop 'D'

VHDL RESOLUCION CON if y **event**	VERILOG

library IEEE;
use IEEE.STD_LOGIC_1164.**ALL**;
 	
entity apren3 **is**
Port (D,CLK : IN STD_LOGIC;
Q : **out** STD_LOGIC);
end apren3;
 	
architecture Behavioral of apren3 **is**
begin
proceso:**process** (CLK)
begin
IF (CLK' **EVENT AND** CLK='1')
THEN
Q<=D;
END IF;
end process proceso;
end Behavioral;
	module ffposid(**input** d,clk,
	output reg q);
	always @(**posedge** clk)
	q<=d;
	endmodule

4.3.2 DISEÑE UN CONTADOR BINARIO PARA DIGITOS EN BASE 4 UTILIZANDO FLIP – FLOP TIPO "D".

DIAGRAMA DE ESTADO

Es una representación gráfica que indica la secuencia de los estados que se presentan en un circuito secuencial en forma de círculos y líneas, teniendo en cuenta las entradas salidas. Los círculos representan los estados del circuito secuencial y cada uno de ellos contiene un número que identifica su estado. Las líneas indican las transiciones entre estados y se marcan con dos números separados por un (/), estos dos números corresponden a la entrada y salida en la transición.

TABLA IV.XLIV: Tabla de Verdad Contador Binario Base 4

	A	B	QA(N+1)	QB(N+1)	DA	DB
0	0	0	0	1	0	1
1	0	1	1	0	1	0
2	1	0	1	1	1	1
3	1	1	0	0	0	0

Representación del diagrama de estados del ejemplo se muestra en la figura IV.26:

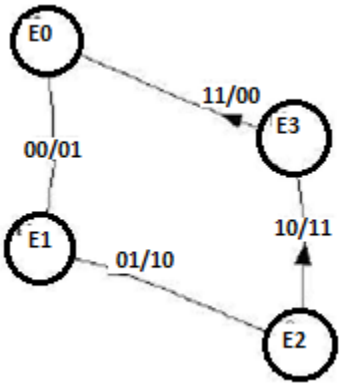


FIGURA IV.26 Estados Del Sisitema

Fuente: *Sistema digitales I- José Guerra-pag. 86*

Como se puede observar en la tabla y diagrama de estado este ejercicio tendrá un sistema secuencial del cero al tres y así indefinidamente mientras se envié los pulsos de reloj.

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XLV: Programación de un contador binario base 4

VHDL RESOLUCION CON if y **event**	VERILOG

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.**ALL**;
ENTITY FFCONT4 **IS**
PORT(CLK, D0, D1 : **IN** STD_LOGIC;
RESET: **IN** STD_LOGIC; Q0,Q1: **OUT**
STD_LOGIC);
END FFCONT4;
ARCHITECTURE BEHAVIORAL **OF**
FFCONT4 **IS**
BEGIN
PROCESS (CLK)
BEGIN
IF (CLK' **EVENT** AND CLK='1') **THEN**
IF RESET='1' **THEN**
Q0<=((NOT D0) AND D1) **OR** (D0
AND(**NOT** D1));
Q1<=**NOT** D1;
ELSE
Q0<=D0;
Q1<=D1;
END IF;
END IF;
END PROCESS;
END BEHAVIORAL;
	module contador4b(input
	a,b,clk,reset,
	output reg da,db);
	always @ (**posedge** clk **or**
	posedge reset)
	if (reset) **begin**
	da<= a^b;
	db<= ~b;
	end
	else
	begin
	da<= a;
	db<= b;
	end
	endmodule

EJEMPLO 2:

4.3.3 DISEÑE UN CONTADOR BINARIO ASCENDENTE DE 0 AL7 USE

PARA EL EFECTO FLIP – FLOP TIPO “D”.

TABLA IV.XLVI: Tabla de verdad de contador binario base 7

	A	B	C	QA(N+1)	QB(N+1)	QC(N+1)	DA	DB	DC
0	0	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	0	1	0
2	0	1	0	0	1	1	0	1	1
3	0	1	1	1	0	0	1	0	0
4	1	0	0	1	0	1	1	0	1
5	1	0	1	1	1	0	1	1	0
6	1	1	0	1	1	1	1	1	1
7	1	1	1	0	0	0	0	0	0

Representación del diagrama de estados del ejemplo se muestra en la figura IV.27:

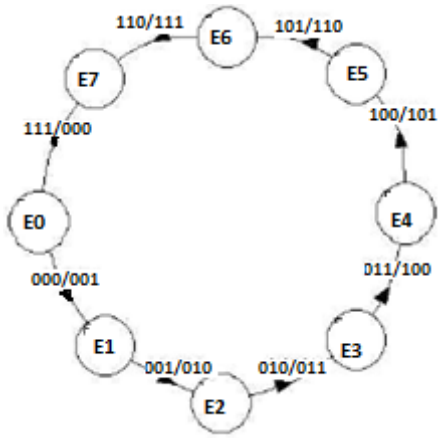


FIGURA IV.27: Estados del sistema

Fuente: *Sistema digitales I- José Guerra-pg.a 86*

En este diagrama observamos de qué manera se tiene que ir generando el diagrama de estado del contador binario.

En esta figura IV.28, se puede observar la simplificación de cada una de las tablas aplicando el método de Karnaugh. Y su respectiva implementación de acuerdo a las salidas deseadas.

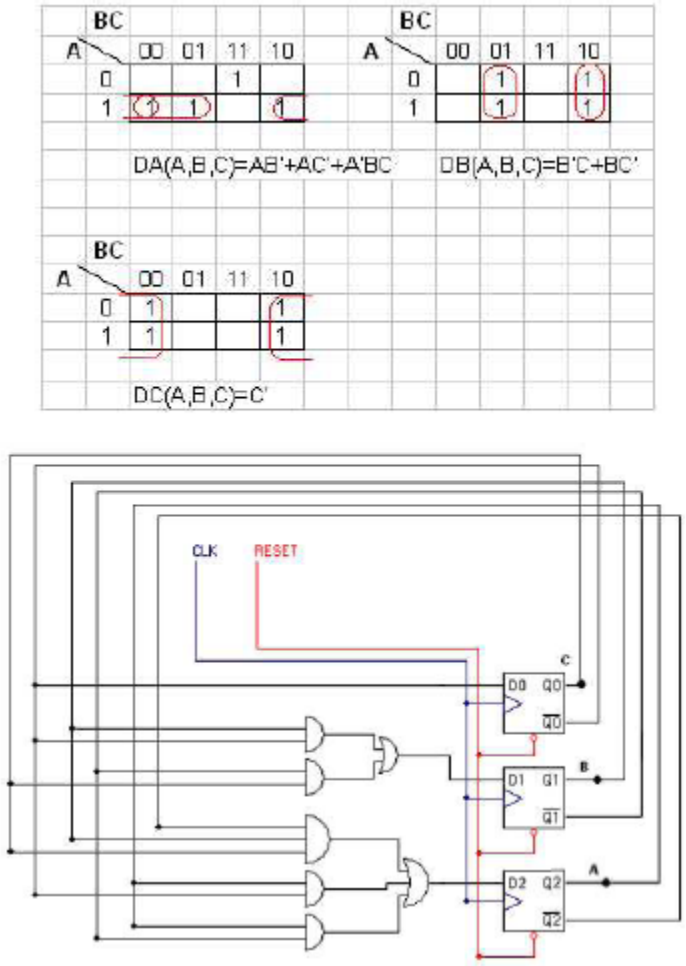


FIGURA IV.28 Estados del sistema

Autor*: Sistema digitales I- José Guerra-pag. 86*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XLVII: Programación de un contador binario base 7

VHDL RESOLUCION CON if y **event**	VERILOG

USE IEEE.STD_LOGIC_1164.**ALL**;
ENTITY FFCONTASCD **IS**
PORT(CLK,A,B,C : **IN** STD_LOGIC;
RESET: **IN** STD_LOGIC;
DA,DB,DC: **OUT** STD_LOGIC);
END FFCONTASCD;
ARCHITECTURE BEHAVIORAL **OF** FFCONTASCD **IS**
BEGIN
PROCESS (CLK)
BEGIN
IF (CLK'EVENT AND CLK='1') **THEN**
IF RESET='1' **THEN**
DA<=(A **AND** (**NOT** B)) **OR** (A **AND** (**NOT** C)) **OR** ((**NOT** A) **AND** B **AND** C);
DB<=(B **XOR** C);
DC<=**NOT** C;
ELSE
DA<=A;
DB<=B;
DC<=C;
END IF;
END IF;
END PROCESS;
END BEHAVIORAL;
	module Contador07(**input** a,b,c,clk,reset,
	output reg da,db,dc);
	always @ (**posedge** clk **or** **posedge** reset)
	if (reset) **begin**
	da<=(a**&~**b**)**|(a**&~**c)|(~a&b&c);
	db<= b^c;
	dc<= ~c;
	end
	else
	begin da<= a;
	db<= b;
	dc<= c;
	end
	endmodule

4.3.4 FLIP –FLOP TIPO "J-K"

Este tipo de dispositivo como el flip – flop (J K) se puede decir que este es el más versátil de los flip – flop básicos. Posee características similares al del flip – flop tipo "D" debido a que tiene el carácter de seguimiento de entradas sincronizadas.

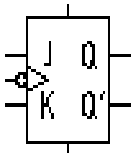


FIGURA IV.29: FLIP-FLOP (J;K)

Fuente: *Los autores*

TABLA IV.XLVIII: Tabla de verdad Flip-Flop "J K"

J	K	CLK	Q
0	0	-	Q0 SIN CAMBIO
1	0	-	1
0	1	-	0
1	1	-	Q0' CAMBIO

Este dispositivo tiene dos entradas por la cual se las conoces como (j, k).

Observamos que cuando (J) es diferente de (K) la salida (Q) toma el valor de (J) durante la subida del siguiente pulso de sincronismo. Si (J) y (Q) poseen ambos un valor de cero observamos que no se produce un cambio alguno pero si (J) y (K) ambas tienen un valor de uno en la siguiente subida de reloj la salida hara un cambio de estado.

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV.XLIX: Programación de flip – flop "J K"

VHDL RESOLUCION CON if y **event**	VERILOG
library IEEE; **use** IEEE.STD_LOGIC_1164.**ALL**; **use** IEEE.STD_LOGIC_ARITH.**ALL**; **use** IEEE.STD_LOGIC_UNSIGNED.**ALL**; **entity** JK_FF_VHDL **is** **port**(J,K: **in** std_logic;	**module** ffjk(input j,k, **input** clk, **output** reg q); **always @(negedge** clk) **case**({j,k}) 2'b11 : q<= ~q; 2'b01 : q<= 1'b0; 2'b10 : q<= 1'b1;

```
 Reset: in std_logic;
 Clock_enable: in std_logic;
 Clock: in std_logic;
 Output: out std_logic);
end JK_FF_VHDL;
 architecture Behavioral of JK_FF_VHDL
is signal temp: std_logic;
begin process (Clock)
 begin
 if (Clock'event and Clock='1') then
 if Reset='0' then
 temp <= '0';
 elsif Clock_enable ='0' then
 if (J='0' and K='0') then
temp <= temp;
 elsif (J='0' and K='1') then
 temp <= '0';
 elsif (J='1' and K='0') then
temp <= '1';
elsif (J='1' and K='1') then
 temp <= not (temp);
 end if;
 end if;
 end if;
 end process;
 Output <= temp;
 end Behavioral;
```

```
2'b00 : q<= q;
Endcase
endmodule
```

4.3.5 DISEÑE UN CONTADOR BINARIO DE TRES BITS DE NUMEROS IMPARES, USE PARA EL EFECTO CUALQUIER TIPO DE FLIP – FLOP.

TABLA IV.L: Tabla de verdad contador binario 3 bit

	A	B	C	QA(N+1)	QB(N+1)	QC(N+1)	DA	DB	DC
0	0	0	0	X	x	X	x	x	X
1	0	0	1	0	1	1	0	1	1
2	0	1	0	X	x	X	x	x	X

3	0	1	1	1	0	1	1	0	1
4	1	0	0	x	x	X	x	x	X
5	1	0	1	1	1	1	1	1	1
6	1	1	0	x	x	X	x	x	X
7	1	1	1	0	0	1	0	0	1

En este grafico se puede observar la simplificación de cada una de las tablas aplicando el método de Karnaugh. Y su respectiva implementación de acuerdo a las salidas deseadas

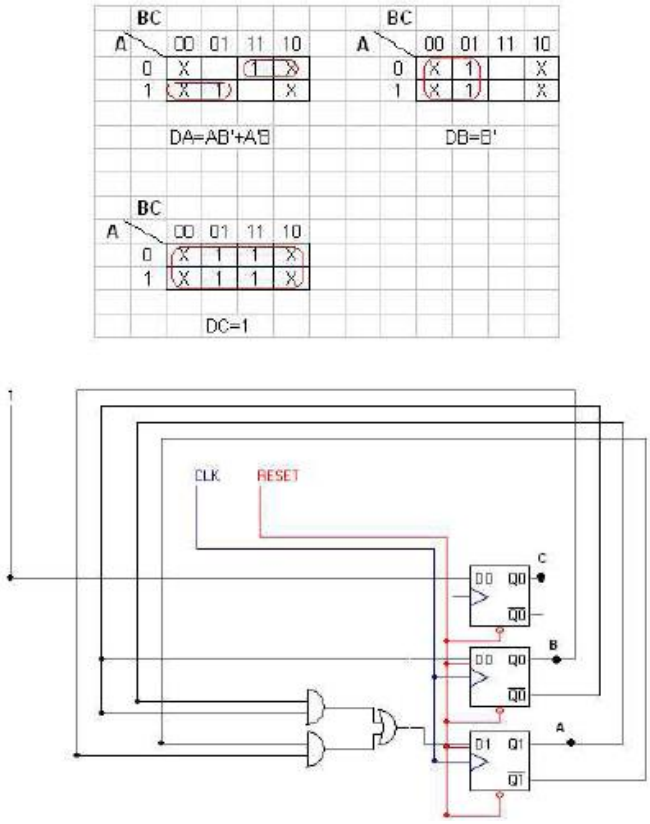


FIGURA IV.30. Estados del Sisitema

Fuente:*Sistema digitales I- José Guerra-pag. 86*

Programación en Vhdl y Verilog

La sentencia de codificación de cada uno de los lenguajes se describe en la siguiente tabla:

TABLA IV:LI: Programación del contador binario 3 Bit

VHDL RESOLUCION CON if y **event**	VERILOG

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.**ALL**;
ENTITY FFIMPAR **IS**
PORT(CLK,A,B,C : **IN** STD_LOGIC;
RESET: **IN** STD_LOGIC;
DA,DB,DC: OUT STD_LOGIC);
END FFIMPAR;
ARCHITECTURE BEHAVIORAL **OF** FFIMPAR **IS**
BEGIN
PROCESS (CLK)
BEGIN
IF (CLK'EVENT **AND** CLK='1') **THEN**
IF RESET='1' **THEN**
DA<=(A **AND** (**NOT** B)) **OR** ((**NOT** A) **AND** B);
DB<=(**NOT** B);
DC<='1';
ELSE
DA<=A;
DB<=B;
DC<=C;
END IF;
END IF;
END PROCESS;
END BEHAVIORAL;
	module conta3bits(**input** a,b,c,clk,reset, **output** reg da,db,dc);
	always @ (**posedge** clk **or posedge** reset)
	if (reset) **begin**
	da<= a^b;
	db<= ~b;
	dc<= 1;
	end else begin
	da<= a;
	db<= b;
	dc<= c;
	end
	endmodule

4.4 PROCESO DE DESCARGA DE LA PROGRAMACION HDL PARA LAS PRUEBAS FÍSICAS EN LA TARJETA FPGA

4.4.1 INTRODUCCIÓN

En esta parte de la investigación, se explicará paso a paso a utilizar la tarjeta FPGA para descargar el código HDL (VERILOG o VHDL), para realizar las pruebas físicas de la programación en los HDL.

Previo a descargar el programa en la tarjeta ya se debe tener realizado la programación en cualquiera de los lenguajes ya sea en VHDL o en Verilog.

4.4.2 PASOS PARA DESCARGAR EL PROGRAMA HDL.

Después de haber hecho la respectiva programación o codificación de nuestro diseño debemos de proceder a compilar para ver si se encuentra correcto el código escrito, guardamos los cambios efectuados (ctrl + s), para ello debemos de seguir los siguientes tres pasos:

1. Dirigirse al recuadro "**SYNTHESIZE – XTS"**
2. Escoger **"CHECK SYNTAX" y** damos un click derecho.
3. Dar click en "**RUN"**

Las opciones se dan en la figura IV.31:

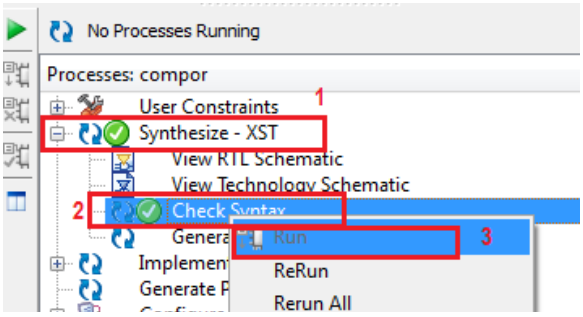


FIGURA IV.31: Compilación del código HDL

Fuente: *Los Autores*

La compilación se demora un tiempo hasta que el programa compile, una vez compilado la sintaxis del programa y que este bien escrito debe nos aparecerá el siguiente mensaje: figura IV.32, "**PROCESS "CHECK SYNTAX" completed successfully**. La cual me indica que todo está sin ningún error. En caso de haber un error corregirlo.

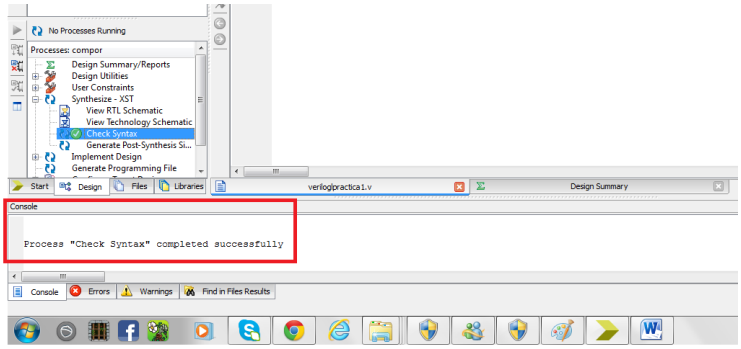


FIGURA IV.32: Mensaje de compilación

FUENTE: *Los Autores*

Una vez comprobada la sintaxis, se debe crear un archivo de extensión .ucf, para realizar esta extensión, se procede a asignar de los pines de entrada y salida en la FPGA de acuerdo a la programación del ejercicio a implementar.

Para ello debemos de dirigirnos a la opción (1) **USER CONSTRAINTS** y luego a la opción (2) **I/O Pin Planning (Plan Ahead) – POST Synthesis,** hacer click derecho con el mouse y se elige la opción **Run** como se muestra en la figura IV.33 y la figura IV.34:

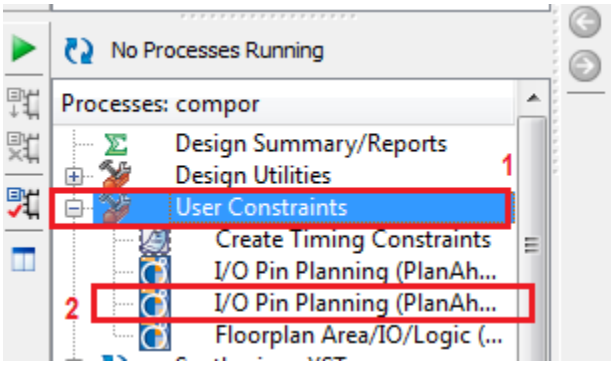


FIGURA IV.33: Ingreso al I/O Planning

Fuente: *Los Autores*

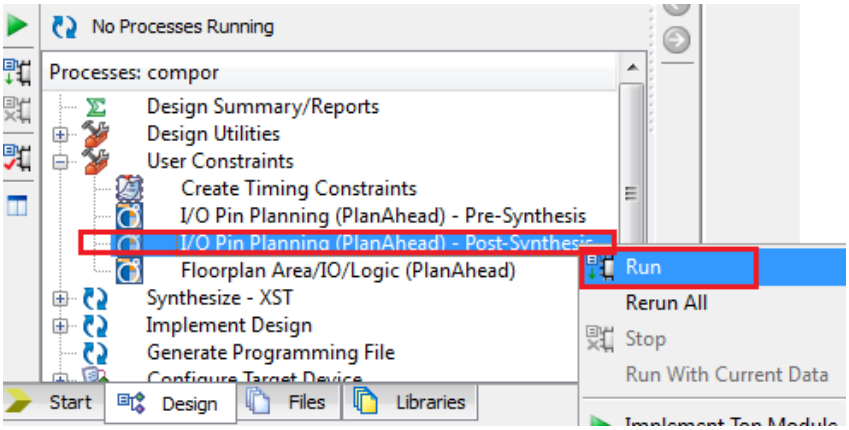


FIGURA IV.34: I/O Pin Planning al programa PlanAhead

Fuente: *Los Autores*

Esperamos un tiempo determinado y luego aparece el siguiente cuadro de dialogo donde indica si se desea crear la extensión **UCF** damos click en (**YES**) figura IV.35:

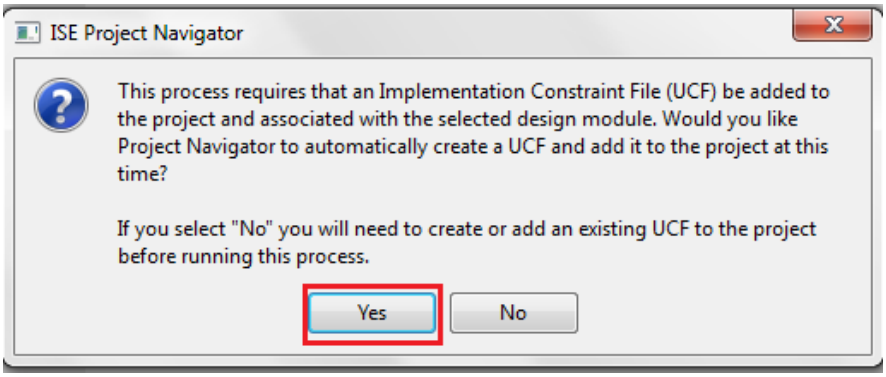


FIGURA IV.35: Cuadro de dialogo para añadir fichero UCF

Fuente: *Los Autores*

Esperamos un tiempo determinado y podremos observar que se abre el programa **Plan Ahead.**

En este programa se instalara automáticamente cuando se está realizando la instalación del programa **ISE DESING SUITE 14.2,** se realizara la respectiva asignación de los pines a utilizar de la FPGA, en la figura IV.36 se muestra la FPGA esquemática de las entradas y salidas.

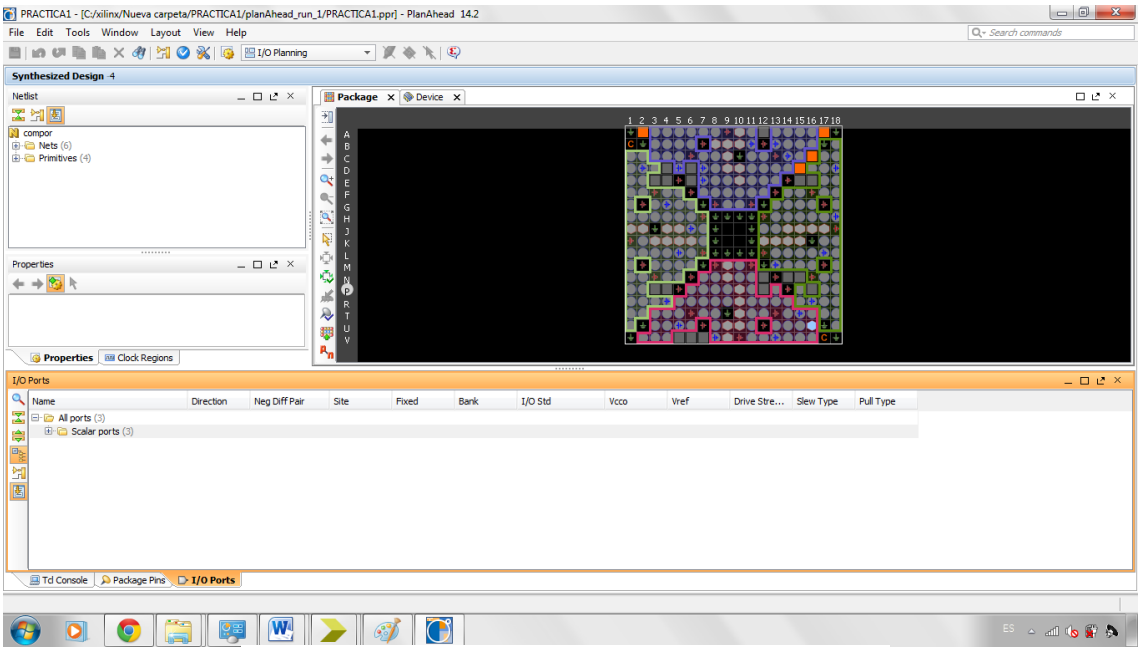


FIGURA IV.36: Programa Plan Ahead

FUENTE: *Los Autores*

En la figura IV.37 se muestra la asignación de pines para la FPGA donde se hará las respectivas asignaciones de la siguiente manera:

- (1) Dar un click en **SCALAR PORTS** (puertos escalares) Al realizar esta paso podemos observar que se desplegó un submenú donde me indica todas las entradas y salidas que tengo declaradas en el programa del diseño digital.

  Recordemos que para este ejemplo se escogió la compuerta **OR** donde (a, b) son entradas y (r) la salida.

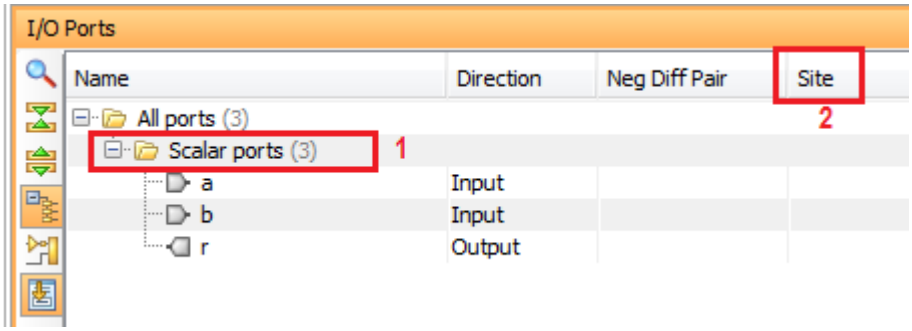


FIGURA IV.37: Selección de Scalar Ports

Fuente: *Los Autores*

- (2) Configurar el **SITE (**sitio, lugar), En este paso se realiza la respectiva asignación del pin de la FPGA a la entrada o salida deseada. Para este ejemplo se le asignara los siguientes pines que se enumeran en la tabla IV.LII:

TABLA IV.LII: Asignación de pines de la FPGA

Entrada de datos	**PINES**
A	L13
B	L14
Salida de datos	**PINES**
r	F12

El código de los pines se los encuentra en la tarjeta FPGA SPARTAN-3E, la numeración de los pines puede variar dependiendo del tipo de tarjeta FPGA o del fabricante de la tarjera FPGA.

Una vez ya designado los pines en la opción **FIXED** se marcó con visto indicando que se asignó correctamente el pin de la FPGA, se muestra en la figura IV.38:

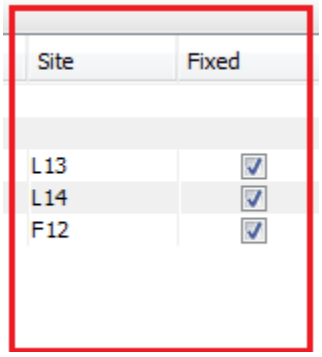


FIGURA IV.38: Asignación de Pines FPGA

Fuente: *Los Autores*

Después de la asignación en las respectivas configuraciones de los pines de la FPGA el paso siguiente es guardar (1) lo realizado y salir del programa **Plan Ahead (2),** mirar la figura IV.39:

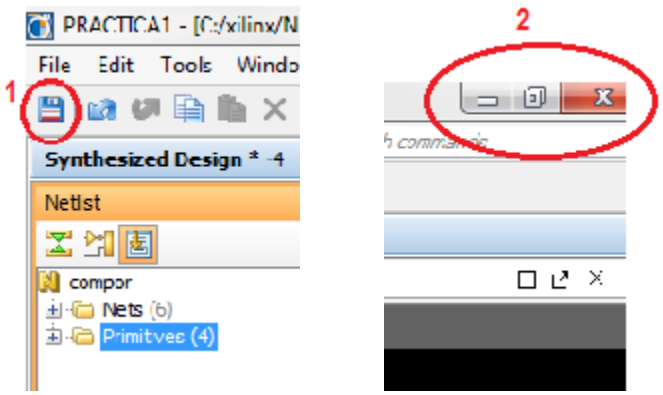


FIGURA IV.39: Guardar y salir del programa Plan ahead

Fuente: *Los Autores*

El archivo con extensión **UCF** ya está creado y se muestra en la figura IV.40:

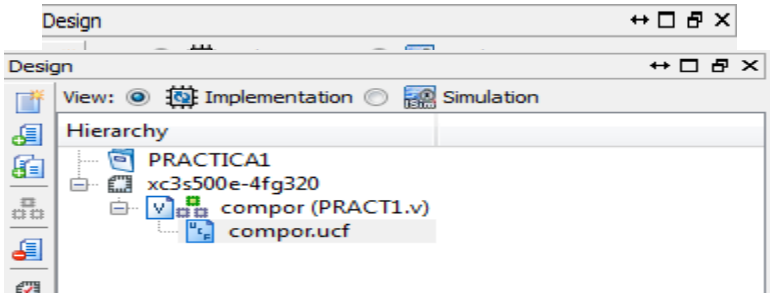


FIGURA IV.40: Archivo UCF

FUENTE: *Los Autores*

Si se desea ver o modificar el contenido del código con la extensión UCF se realizarán los siguientes pasos:

- (1) Dar un clik derecho sobre el fichero con extensión UCF
- (2) Escoger la opción **OPEN** (abrir)

Como se muestra en la figura IV.41:

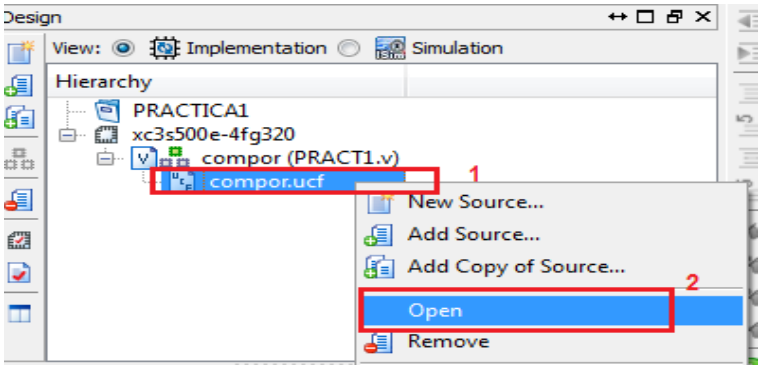


FIGURA IV.41: Abrir Archivo UCF

Fuente: *Los Autores*

En el siguiente cuadro se observara un resumen de cada una de las entradas y salidas configuradas para la tarjeta FPGA, en la cual el programador puede realizar cualquier tipo de cambio o conservar las configuraciones ya realizadas según su necesidad, se muestra en la figura IV.42:

```
1
2  # PlanAhead Generated physical constraints
3
4  NET "a" LOC = L13;
5  NET "b" LOC = L14;
6  NET "r" LOC = F12;
7
```

FIGURA IV.42: Resumen de pines configurados de la FPGA

Fuente: *Los Autores*

❖ El siguiente paso a realizar es el siguiente:

- (1) Dar un click en el fichero con la extensión ".V"
- Dar un click derecho en la opción **IMPLEMENT DESIGN** (2) y luego escoger la opción **RUN (3),** como se muestra en la figura IV.43:

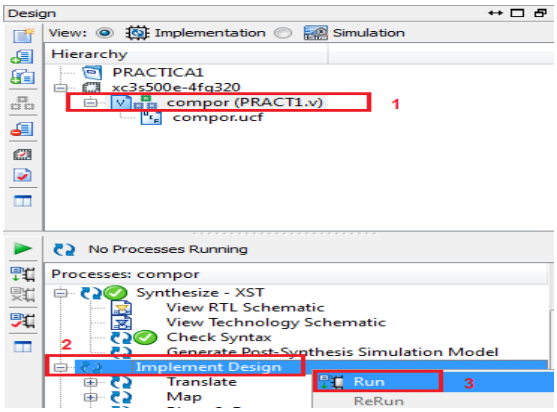


FIGURA IV.43: Implementar el Diseño

Fuente: *Los Autores*

Esperar un tiempo hasta que el programa ejecute la implementación del diseño la cual una vez ejecutara tendremos el siguiente figura IV.44, indicando que se ha ejecutado las opciones de (TRANSLATE, MAP, PLACE&ROUTE)

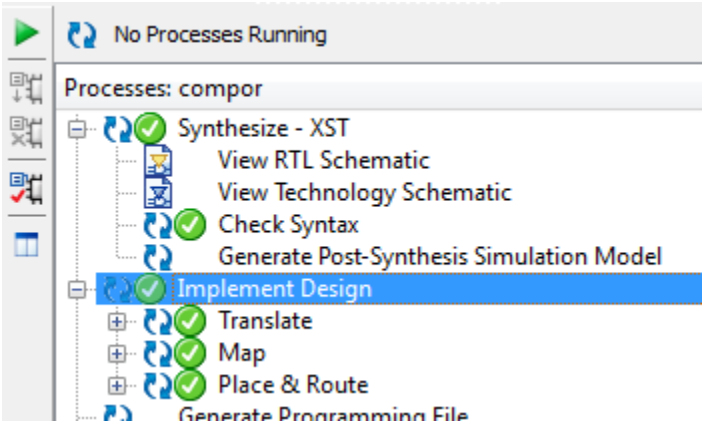


FIGURA IV.44: Ejecución Implementar Diseño

Fuente: *Los Autores*

- En esta sección ya está listo el código HDL listo y configurado para poder ser cargado a la tarjeta FPGA para la cual se necesita seguir los siguientes pasos:

a) Conectar la tarjeta FPGA a la fuente de alimentación y al computador con su respectivo cable USB.

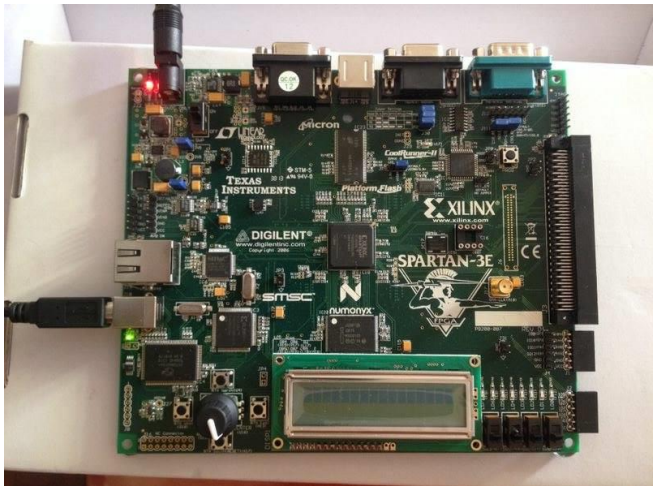


FIGURA IV.45: Conexión a la fuente y el cable USB

Fuente: *Los Autores*

b) Led indicador que existe conexión entre el computador y la tarjeta FPGA

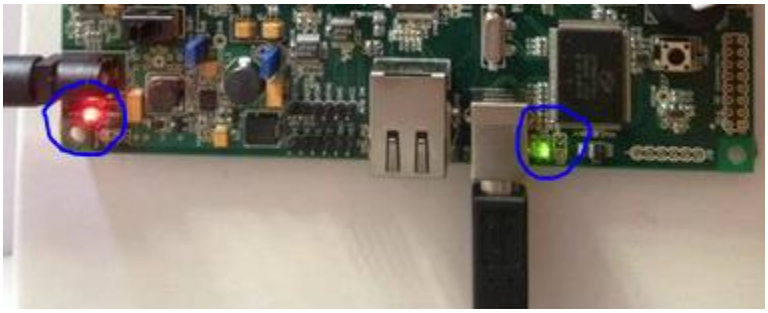


FIGURA IV.46: Led Indicadores

Fuente: *Los Autores*

c) Escogemos la opción (1) **CONFIGURE TARGET DEVICE** (configurar dispositivo tarjeta) , dar un click derecho en la opción (2) **MANAGE CONFIGURATION PROJECT (IMPACT)** (GESTIONAR PROYECTO DE CONFIGURACION (IMPAC)) Y escogemos la opción (3) **RUN** , según se muestra en la figura IV.

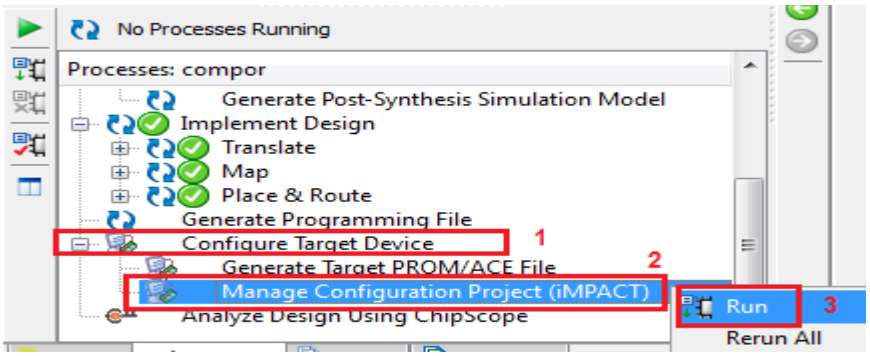


FIGURA IV.47: Configurar dispositivo de la tarjeta

Fuente: *Los Autores*

Se abrirá la siguiente ventana con el nombre **ISE Impact** dar click en la opción **LAUNCH WIZARD**, que se encuentra señalada en la figura IV.48:

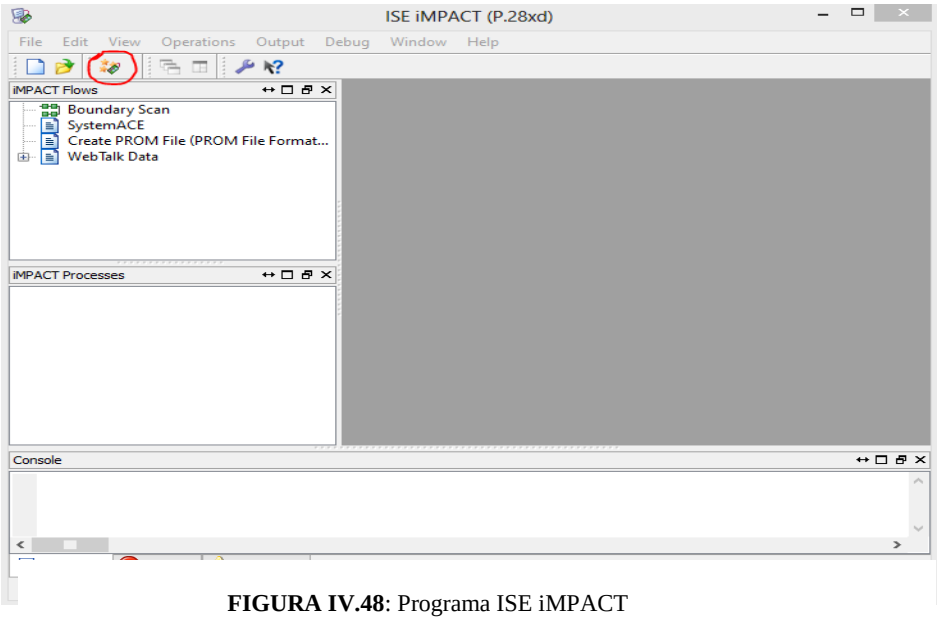


FIGURA IV.48: Programa ISE iMPACT

FUENTE: *Los Autores*

Se abrirá la siguiente ventana y debemos escoger los siguientes ítems marcados en la figura IV.49 ya que son los mismos que nos dio el programa al hacer su ingreso a esta ventana

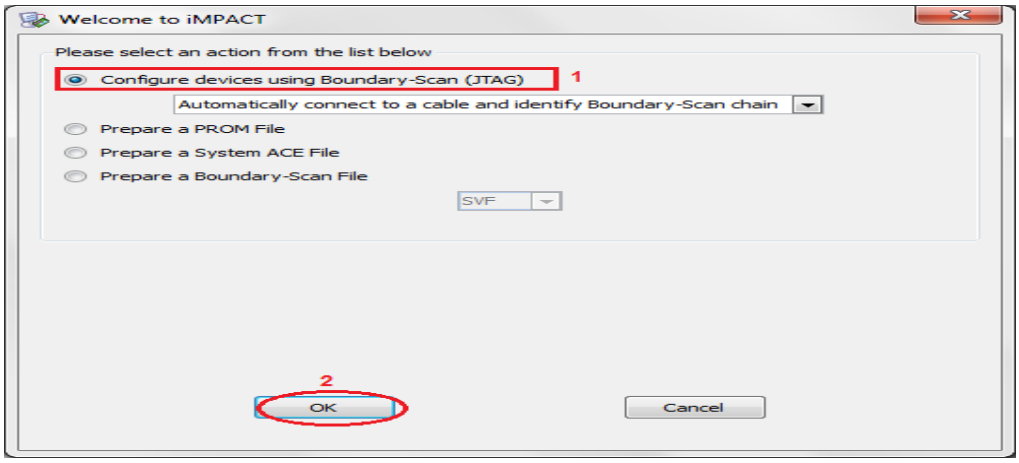


FIGURA IV.49: Programa Impact configuración

Fuente: *Los Autores*

En el siguiente cuadro de dialogo preguntara si desea realizar otras configuraciones al archivo para la cual se escogerá la opción **YES**, como se muestran en la figura IV.50:

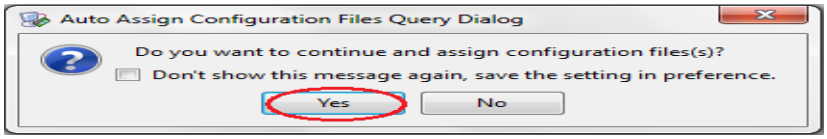


FIGURA IV.50: Asignar configuracion

Fuente: *Los Autores*

En el siguiente figura IV.51 escogemos el archivo con extensión "**.BIT**" (1) que es el archivo que se va a descargar a la tarjeta FPGA y luego escogemos la opción (2) **OPEN** (ABRIR).

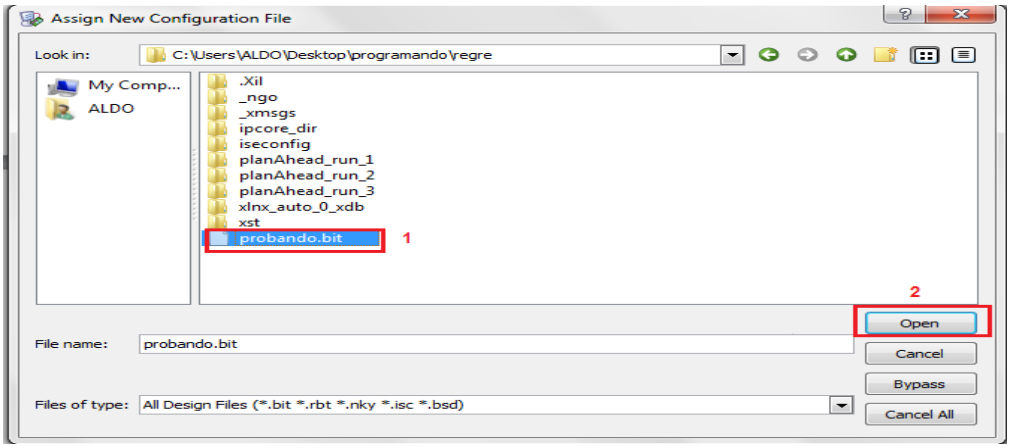


FIGURA IV.51: Archivo extensión BIT

Fuente: *Los Autores*

Después aparecerá 3 cuadros de asignación de configuración los cuales son para pasar el código a los otros dispositivos que existen en la tarjeta los cuales para este ejemplo no se los utilizara y podemos cerrar esas ventanas

En el presente cuadro me indica las propiedades del dispositivo en el cual me muestra todos los dispositivos que posee la tarjeta y como se va a trabajar sobre el dispositivo FPGA escogemos la opción **OK**, como se muestra en la figura IV.52:

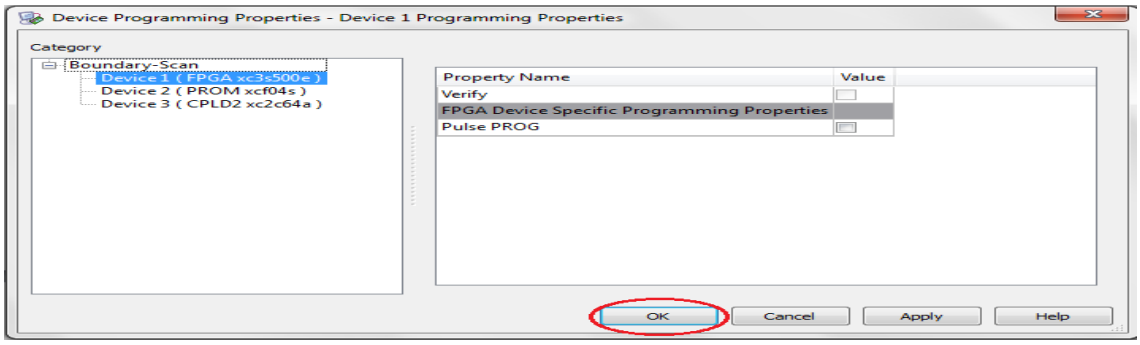


FIGURA IV.52: Escogiendo la FPGA

Fuente: *Los Autores*

En la siguiente figura IV.53 indica el dispositivo FPGA y las demás memorias (FLASH, PROM) que posee la tarjeta.

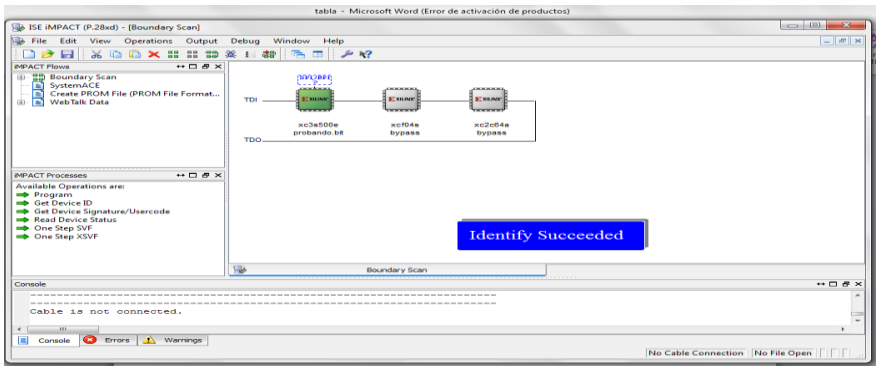


FIGURA IV.53: Memorias en la FPGA

Fuente: *Los Autores*

❖ Para programar el dispositivo FPGA realizar el siguiente pasó:

d) Dar click derecho sobre la gráfica que representa al dispositivo FPGA y escoger la opción **PROGRAM**, como se muestra en la figura IV.54:

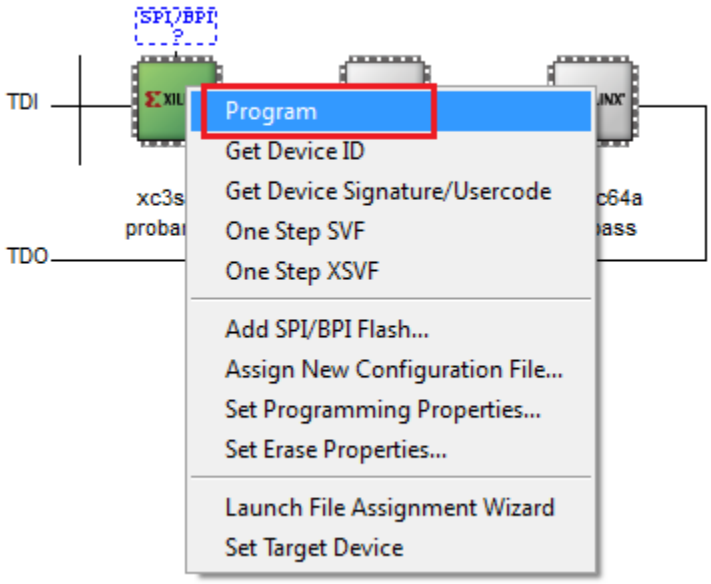


FIGURA IV.54: Programando FPGA

Fuente: *Los Autores*

Después de un tiempo deberá aparecer el siguiente mensaje que se presenta en la figura IV.55:

FIGURA IV.55: Programa cargado en la FPGA

Fuente: *Los Autores*

Este me indica que mi tarjeta FPGA ha sido correctamente programada la cual ya podemos empezar a realizar las pruebas físicas en la tarjeta del ejercicio planteado.

4.4.3 PRUEBAS FISICAS EN LA TARJETA FPGA

Los pines que se le asignaron a la tarjeta tanto de entrada como de salida de datos son los que podemos ver en la figura.

Por la cual ya se puede empezar a realizar las respectivas combinaciones de la tabla de verdad para la compuerta lógica **OR.** Si Recordamos la tabla de verdad de la compuerta sabemos que si ambas entradas (a,b) son 0 nuestra salida es 0 lógico por la cual podemos observar que nuestra salida (F12) está apagada

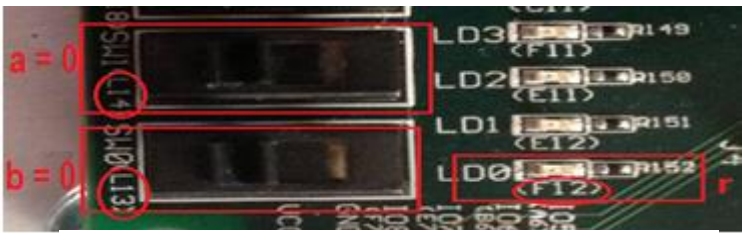


FIGURA IV.56: Prueba física 1 de la tarjeta

LOS AUTORES

Si (a=0 y b= l), la salida (r=1), entonces se encenderá el led que se observa en la gráfica.

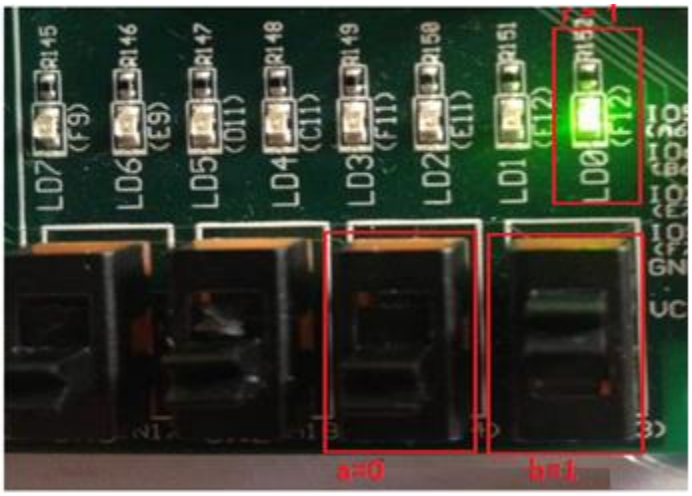


FIGURA IV.57: Prueba física 2 de la tarjeta

LOS AUTORES

Si (a=1 y b= 0) la salida (r=1), lo encenderá el led que se observa en la gráfica.

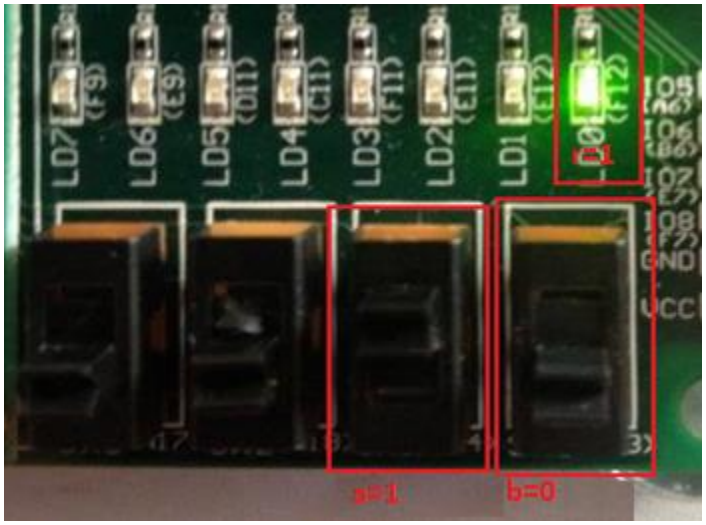


FIGURA IV.58: Prueba física 3 de la tarjeta

LOS AUTORES

Si (a=1 y b= 1) la salida (r) encenderá el led que se observa en la gráfica.

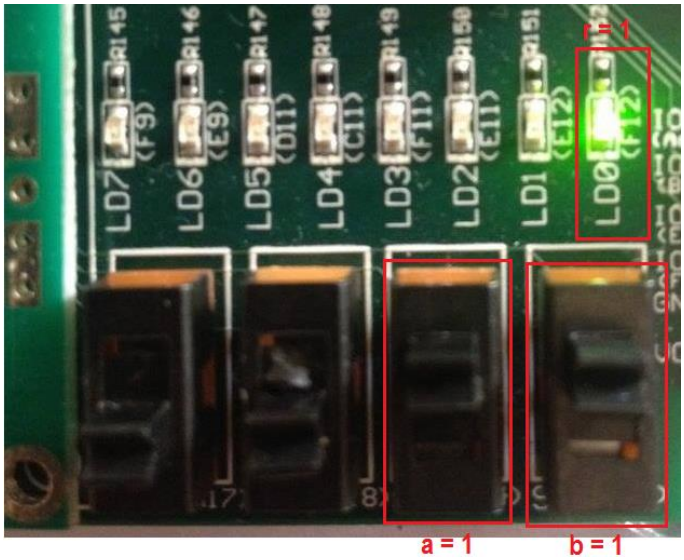


FIGURA IV.59: prueba física 4 de la tarjeta

LOS AUTORES

Aquí se terminan todas las pruebas físicas para ver que si está funcionando el diseño digital realizado antes por medio de la programación los lenguajes HDL.

CAPITULO V

ESTUDIO COMPARATIVO DE VHDL Y VERILOG

5.1 INTRODUCCION

En este capítulo se realizara la recopilación de varios artículos científicos publicados por personas que han realizado una comparación de los dos lenguajes de descripción de hardware VHDL y Verilog. Se estudiara cada artículo científico para encontrar parámetros de comparación referente a la sintaxis de programación que será la base para el estudio comparativo del trabajo de investigación.

Se incluirá nuestro propio criterio de cada uno de los lenguajes, según las guías prácticas que se han desarrollado en el capítulo IV, para concretar un mejor resultado del estudio comparativo a realizarse. De cada artículo se tomaran las cosas más importantes referentes a la comparación realizada por cada uno de los autores en base a la sintaxis de programación.

Después de obtener los parámetros de la sintaxis de programación se realizarán, cuadros estadísticos de los artículos científicos y de nuestra investigación basada en las guías prácticas de programación, para luego sacar un solo resultado y establecer cuál de los dos lenguajes es el más óptimo entre VHDL y Verilog.

Con la finalidad de demostrar la hipótesis, planteada en este trabajo de investigación.

5.2 ESTUDIO COMPARATIVO DE ARTICULOS CIENTIFICOS

En esta parte de la investigación se ha recopilado diez artículos científicos referentes a los estudios realizado en VHDL y Verilog, para determinar parámetros de sintaxis de programación de los lenguajes, tomando en cuenta sus experiencias con cada uno de estos lenguajes, para luego sacar conclusiones finales en el siguiente literal 5.4.

5.2.1 VERILOG VHDL VS

INTRODUCCION

Si quieres ser un programador FPGA, ¿cuál de los dos lenguajes de programación de FPGA dominantes Qué se aprende? Esta pregunta tan a menudo por los ingenieros nuevos en el campo del diseño digital, se podría pensar que habría una respuesta definitiva. En este artículo voy a cubrir las principales diferencias, pros y contras de cada idioma, pero primero voy a decirte que estás haciendo la pregunta equivocada.

TABLA V.I: Comparación de lenguajes
Articulo Cientifico#1

VHDL	VERILOG
Se presta en HD (Descripción de hardware) en más niveles abstractos (como declaraciones, si/entonces, etc.).	Es bueno en HD (Descripción de hardware) hasta el nivel de la puerta (nand, xor, and, etc.).

Es un lenguaje fuertemente tipado, por lo que los errores de sintaxis se encuentran con mayor facilidad por el compilador en lugar de registrar de forma manual a través del código en busca de un uso incorrecto que está causando problemas. Tratar con números con signo y sin signo es natural, y hay menos posibilidades de cometer un error de precisión o la asignación de una señal de 16 bits para una señal de 4 bits.	Lenguaje débilmente tipado. El código es más propenso a errores debido a la combinación accidental de diferentes tipos de señal. Facilidad para crear la señal con un error tipográfico.
Capacidad para definir tipos personalizados. Una máquina de estados VHDL se puede codificar de forma natural utilizando los nombres reales del estado (por ejemplo, esperar, reconocer, transmitir, recibir, etc), no números binarios estatales (por ejemplo, 00, 01, 10, 11).	No hay soporte de tipos personalizados. Codificación de la máquina del Estado es más difícil porque los estados deben ser valores numéricos con un ancho de bits estáticos (existen técnicas para aliviar este problema).
Los tipos de registro. Definir varias señales en un tipo. Estilo de codificación natural para resets asíncronos. Revertir fácilmente orden de los bits de una palabra. Declaración lógica (caso similar y si / entonces) las terminaciones están claramente marcados.	Operadores de reducción. Realizar pruebas lógicas en toda una serie de bits con un solo operador.
Extremadamente detallado de codificación. Módulos VHDL deben ser definidas por un prototipo y declaró antes de ser utilizados, lo que le permite cambiar el código en al menos 3 lugares si usted quiere hacer un cambio en la interfaz. El uso de la palabra clave "downto" en cada definición poco vector es tedioso.	Descripciones de bajo nivel más cerca de hardware real. Crear una instancia de puertas directamente. Declarar explícitamente cables y registros directamente. Directivas del compilador

Listas de sensibilidad. Falta una sola señal en la lista de sensibilidad puede causar diferencias catastróficas entre la simulación y la síntesis. Cada proceso debe tener una lista de sensibilidad que a veces puede ser muy largo.	No hay listas de sensibilidad. Menos de una posibilidad de problemas de síntesis de diseño y simulación derivados de la codificación.
Escriba conversiones. Tipos de señales que están claramente relacionados (por ejemplo std_logic y STD_LOGIC_VECTOR) no pueden ser utilizados simplemente juntos y se deben convertir a otro tipo.	Lenguaje compacto, huella de código pequeño. Convenciones del lenguaje familiar similares a C. Mezclar y combinar las señales es muy fácil.
	Declaraciones de señales confusas. No siempre se requieren declaraciones de señal. La diferencia entre un cable y reg no siempre es obvia para los principiantes, y un reg a veces puede ser un alambre.
	Soporte reducido para las señales asíncronas. El uso de señales asíncronas se ralentiza software de simulación. Las construcciones de lenguaje no son compatibles con el uso natural de las señales asíncronas.

De cualquier idioma es completamente capaz de implementar programas de FPGA exigentes. Prefiero Verilog porque es más rápido al código, pero vuelvo a VHDL para hacer la mayoría de mis diseños de máquinas de estado. He contratado a una serie de ingenieros para trabajos de FPGA, y yo nunca he considerado saber un idioma o el otro rompe el acuerdo. [(22)]

5.2.2 IMPLEMENTACION DE MODELOS DE FOTODIODOS EN LENGUAJES DE DESCRIPCIÓN DE HARDWARE DE SEÑAL MIXTA

INTRODUCCION

Originalmente estos lenguajes se usaban para describir bloques digitales, en la actualidad se ha ampliado a la extensión AMS (Analog and Mixed Signals), con la que se pueden describir no solo bloques analógicos sino cualquier otro sistema conservativo, lo que amplía las posibilidades de descripción de sistemas naturales más allá de sistemas puramente eléctricos, además de permitir la simulación del comportamiento de bloques combinados que incluyan módulos digitales, analógicos, mecánicos, térmicos, etc.

TABLA V.II: Comparación de lenguajes
Articulo Cientifico#2

VHDL	VERILOG
Extensión AMS (Analog and Mixed Signals).	Extensión AMS (Analog and Mixed Signals).
Su extensión para bloques analógicos y de señal mixta, es ligeramnte menos utilizada para el diseño de circuitos.	Verilog-AMS es un lenguaje que fue desarrollado para ser similar en su sintaxis a C, con el propósito de acelerar su aceptación, además de que las extensiones del mismo fueron desarrolladas en partes, así la primera extensión Verilog-A daba soporte a la descripción de bloques analógicos y posteriormente se incluyó la extensión AMS para el soporte a circuitos de señal mixta;

Si bien ambos lenguajes son comparables en la descripción de los modelos de fotodiodos se optó por Verilog -AMS, por su mayor utilización, en los entornos de diseño.

Verilog-AMS constituye un lenguaje unificado con estructuras semánticas para bloques tanto digitales como analógicos, separando la descripción de cada módulo de la descripción de su comportamiento y de la interconexión de éstos, teniendo cuidado de solo asociar las señales analógicas y mixtas a los nodos, no así las señales puramente digitales. (23)

5.2.3 HDL PROGRAMMING FUNDAMENTALS: VHDL AND VERILOG

INTRODUCCION

Los avances en la tecnología de semiconductores continúan para aumentar la potencia y la complejidad de los sistemas digitales. Para diseñar estos sistemas requiere de un gran conocimiento de los circuitos integrados de aplicación específica (ASIC) y Field Programmable Gate Arrays (FPGA), así como las herramientas CAD requeridas. Lenguaje de Descripción del hardware (HDL) es una herramienta esencial de CAD que ofrece a los diseñadores una forma eficaz de implementar y sintetizar el diseño en un chip. La programación fundamental de HDL: VHDL y Verilog enseña a los estudiantes los elementos esenciales de HDL y la funcionalidad de los componentes digitales de un sistema. A diferencia de otros textos, este libro cubre ambos idiomas HDL estandarizadas IEEE: VHDL y Verilog. Estos dos idiomas son ampliamente utilizados en la industria y el mundo académico y tienen una lógica similar, pero son diferentes en estilo y sintaxis. Al aprender los dos idiomas los estudiantes serán capaces de adaptarse a cualquiera de ellos, o implementar entornos de lenguaje mixto, que están ganando impulso a medida que se

combinan las mejores características de las dos lenguas en el mismo proyecto. El texto comienza con los conceptos básicos de HDL, y cubre los temas clave, tales como el modelado de flujo de datos, modelado de comportamiento, modelado a nivel de entrada, y

TABLA V.III: Comparación de lenguajes
Articulo Cientifico#3

VHDL	**VERILOG**
VHDL es el mayor de los dos, y se basa en Ada y Pascal, heredando lo tanto las características de los dos idiomas.	Verilog es relativamente reciente, y sigue los métodos de codificación del lenguaje de programación C.
Un lenguaje fuertemente tipado como VHDL no permite que se mezclen, o el funcionamiento de las variables, con diferentes clases.	Verilog utiliza tipado débiles, que es lo contrario de un lenguaje fuertemente tipado.
VHDL no distingue entre mayúsculas y minúsculas, y los usuarios pueden cambiar libremente el caso, siempre y cuando los caracteres en el nombre, y el orden, siendo el mismo.	Verilog es sensibilidad a las mayúsculas. Diferencia entre mayúsculas y minúsculas.
VHDL tiene la ventaja de tener una gran cantidad más construcciones que ayudan en el modelado de alto nivel, y que refleja la operación real del dispositivo que está siendo programada. Complejo de tipos de datos y paquetes son muy deseables en la programación de sistemas grandes y complejos, que podrían tener una gran cantidad de piezas funcionales.	Verilog no tiene concepto de paquetes, y toda la programación se debe hacer con los tipos de datos simples que son proporcionados por el programador. Los sistemas complejos, que pueden tener una gran cantidad de piezas funcionales.
Posee gestión de bibliotecas.	Carece de la gestión de biblioteca

Verilog es más fácil de aprender que VHDL. Esto se debe, en parte, a la popularidad del lenguaje de programación C, por lo que la mayoría de los programadores familiarizados con las convenciones que se utilizan en Verilog. VHDL es un poco más difícil de aprender y programar. [(24)]

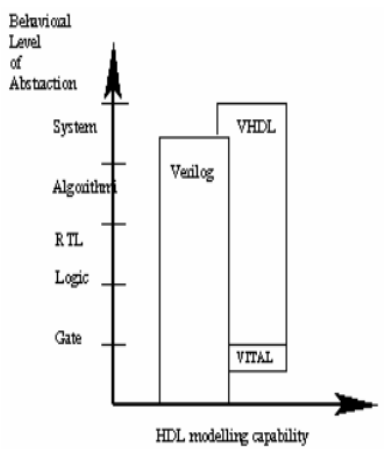
5.2.4 PROGRAMMABLE LOGIC DESING – LECTURE 12 VHDL vs VERILOG

INTRODUCCION

En la actualidad hay dos lenguajes de descripción de hardware estándar en la industria, VHDL y Verilog. La complejidad de diseños ASIC y FPGA ha significado un aumento en el número de herramientas específicas y bibliotecas de células macro y mega-escritas en VHDL o Verilog. Como resultado de ello, es importante que los diseñadores conozcan tanto VHDL y Verilog, y que los proveedores de herramientas EDA proporcionen herramientas que ofrezcan un entorno que permita ambas lenguas que se utilizarán al unísono.

TABLA V.IV: Comparación de lenguajes
Articulo Cientifico#4

VARIABLES	VHDL	VERILOG
COMPATIBILIDAD	Es parecido al lenguaje de programación ADA o Pascal.	Es parecido al lenguaje de programación C.
	Estructura del hardware puede ser modelada con la misma eficacia tanto en VHDL.	Estructura del hardware puede ser modelada con la misma eficacia tanto en Verilog.

	Las construcciones de modelado de VHDL y Verilog cubren un espectro ligeramente diferente a través de los niveles de abstracción del comportamiento.	
COMPILACION	Múltiples diseños de unidades (pares entidad / arquitectura), que residen en el mismo archivo de sistema, pueden ser compilados por separado si así lo desea.	El lenguaje Verilog todavía tiene sus raíces en el mismo de modo interpretativo nativo. La compilación es un medio de acelerar la simulación, pero no ha cambiado la naturaleza original de la lengua.
	Es una buena práctica de diseño para mantener cada unidad de diseño en su propio archivo de sistema en cuyo caso la compilación separada no debería ser un problema.	El cuidado debe ser tomado con tanto el orden de compilación de código escrito en un solo archivo y el orden de compilación de varios archivos. Resultados de la simulación pueden cambiar con sólo cambiar el orden de compilación.
	Una multitud de idiomas o tipos de datos definidos por el usuario se puede utilizar. Esto puede significar funciones de conversión dedicados que son necesarios para convertir objetos de un tipo a otro.	Verilog tiene tipos de datos muy simple, fácil de usar y muy orientada a la estructura de hardware de modelado en lugar de modelado hardware abstracto. A diferencia de VHDL, todos los tipos de datos utilizados

TIPOS DE DATOS	La elección de qué tipo de datos a utilizar debe ser considerada con prudencia, sobre todo tipos de datos enumerados. VHDL puede ser preferible porque permite una multitud de idioma o los tipos de datos definidos por el usuario para ser utilizado.	en un modelo de Verilog se definen por el lenguaje Verilog y no por el usuario. Un modelo con una señal cuyo tipo es uno de los tipos de datos de red tiene un cable eléctrico correspondiente en el circuito de modelado implícito. Verilog puede ser preferible debido a su simplicidad.
DISEÑO DE REUTILIZACION	Los procedimientos y funciones pueden ser colocados en un paquete de modo que están disponibles para cualquier unidad de diseño que los desea utilizar.	Funciones y procedimientos que se utilizan dentro de un modelo hay que definir en el módulo. Para realizar las funciones y los procedimientos de acceso general, de diferentes estados del módulo de las funciones y los procedimientos deben ser colocados en un archivo de sistema separado e incluyó el uso del `incluir la directiva del compilador.
CONSTRUCCIONES DE ALTO NIVEL	Hay más construcciones y características para el modelado de alto nivel en VHDL que hay en Verilog. Tipos abstractos de datos se pueden usar junto con las siguientes afirmaciones: ❖ Sentencias de paquete para el modelo de reutilización, ❖ Sentencias de configuración para la configuración de	A excepción de la posibilidad de parametrizar los modelos por la sobrecarga de las constantes de los parámetros, no hay un equivalente a los estados de modelización VHDL de alto nivel en Verilog.

	diseño de estructura. ❖ Generar instrucciones para la estructura de la replicación. ❖ Declaraciones genéricas para modelos genéricos que se pueden caracterizar de forma individual, por ejemplo, un poco ancho. Todas estas declaraciones de idiomas son útiles en modelos sintetizables.	
LIBRERIAS	Una biblioteca es un almacén para las entidades compiladas, arquitecturas, paquetes y configuraciones. Útil para la gestión de múltiples proyectos de diseño.	No existe el concepto de una biblioteca en Verilog. Esto se debe a sus orígenes como un lenguaje interpretado.
CONSTRUCCIONES DE BAJO NIVEL	Dos operadores lógicos de entrada simples se construyen en el lenguaje, que son: NOT, AND, OR, NAND, NOR, XOR y XNOR. Cualquier momento se debe especificar por separado utilizando la cláusula siguiente. Construcciones separadas definidas a bajo lenguaje VITAL deben ser utilizadas para definir las células primitivas de bibliotecas ASIC y FPGA.	El lenguaje Verilog fue desarrollado originalmente con el modelado de nivel de la puerta en la mente, y lo ha hecho muy buenas construcciones para el modelado de este nivel y para el modelado de las primitivas células de bibliotecas ASIC y FPGA. Los ejemplos incluyen los primitivos definidos por el usuario (UDP), tablas de verdad y el bloque de especificar para especificar temporización de retraso a través de un módulo.

LA GESTIÓN DE GRANDES DISEÑOS	Configuración, generar, declaraciones genéricas y empaquetar todos ayudan a manejar grandes estructuras de diseño.	No hay declaraciones en Verilog que ayudan a administrar grandes diseños.
PROCEDIMIENTOS Y TAREAS	Se permite llamar procedimientos concurrentes.	No se le permite llamar procedimientos concurrentes.
REPLICACION ESTRUCTURAL	La declaración generar replica varias instancias del mismo diseño de unidad o con alguna subparte de un diseño, y se conecta de manera apropiada.	No existe un equivalente a la declaración de generar en Verilog.
VERBOSENESS (VERBOSIDAD).	Debido VHDL es un modelos de lenguaje muy inflexible de tipos debe ser codificado con precisión con los tipos de datos coincidentes definidos. Esto puede considerarse como una ventaja o desventaja. No modelos de medias son a menudo más detallado, y el código a menudo más largo, que es Verilog equivalente.	Las señales que representan objetos de diferentes anchuras de bits pueden ser asignados a cada uno otro. La señal que representa el número menor de bits se rellena automáticamente a la de la mayor cantidad de bits, y es independiente de si se trata de la señal o no asignado. Los bits no utilizados se optimizarán automáticamente fuera durante el proceso de síntesis. Esto tiene la ventaja de no tener que modelar tan explícita como en VHDL, pero sí significa errores de modelado no intencionales no se identificarán con un analizador.

Esto es más una cuestión de estilo de codificación y experiencia de las características del lenguaje. VHDL es un lenguaje conciso y detallado; Verilog es más parecido a C porque es construcciones se basan aproximadamente el 50% en C.

Por esta razón, un programador de C existente puede preferir Verilog sobre VHDL.

Sea cual sea el HDL se utiliza, al escribir o leer un modelo de HDL a sintetizar, es importante pensar en la intención de hardware. (25)

5.2.5 ESTUDIO DE CASO - VERILOG VHDL VS

Introducción:

La batalla entre Verilog y VHDL se ha desatado en la industria de automatización de diseño electrónico (EDA) desde mediados de los años 80. La batalla entre estos dos lenguajes de software que compiten es un caso particularmente interesante debido a los problemas económicos, competitivos y gubernamentales que están involucrados. Lo que también hace que este caso interesante es el de la lucha entre estas dos normas de la competencia se encuentra actualmente en pleno apogeo y se entiende sin un vencedor claro.

La estandarización de VHDL y Verilog

Hay un mercado contraste entre el proceso de normalización de VHDL y Verilog. Es interesante señalar aquí que, si bien los métodos de normalización eran muy diferentes, los productos resultantes son a la vez muy fuerte y los expertos no se ponen de acuerdo sobre cuál es mejor. (26)

TABLA V.V: Comparación de lenguajes
Articulo Cientifico#5

VHDL	VERILOG
❖ Se puede describir el comportamiento y la estructura de los sistemas electrónicos, pero es particularmente adecuado como un lenguaje para describir la estructura y el comportamiento de los diseños de hardware electrónicos digitales, tales como los ASICs y FPGAs, así como circuitos MSI convencionales	❖ Rápido en la simulación a nivel de puertas, y la capacidad de modelar en los niveles superiores de abstracción

5.2.6 COMPARACION DE VHDL, VERILOG

INTRODUCCION:

A medida que el número de mejoras a diversos Hardware Lenguajes de descripción (HDL) tiene aumentado en los últimos años, también lo ha hecho la complejidad de determinar qué idioma es mejor para un diseño particular.

Muchos diseñadores y organizaciones están considerando si deben cambiar de una a otra de HDL. [27]

TABLA V.VI: Comparación de lenguajes
Articulo Cientifico#6

VHDL	VERILOG
❖ sus requisitos de idioma son más detallado que Verilog. ❖ VHDL no define ningún tipo de control de simulación o capacidades de monitoreo dentro del lenguaje. Estas capacidades son herramienta dependientes	❖ Su requisito de idioma es más corto que VHDL. ❖ Todos los tipos de datos están predefinidos en el Idioma. ❖ Reconoce que todos los tipos de datos tienen una representación a nivel de bit ❖ Define un conjunto de control básico de simulación capacidades (tareas del sistema) en el lenguaje.

5.2.7 VERILOG HDL VS VHDL PARA LA PRIMERA VEZ DEL USUARIO

DESARROLLO

La búsqueda de la HDL perfecto es como la búsqueda del coche perfecto, la perfecta casa o tal vez incluso la relación perfecta. Probablemente no existe. La decisión de qué idioma a elegir se basa en un serie de requisitos importantes en particular para la primera vez de usuario. Entrevisté a los equipos de diseño para recopilar algunas estadísticas interesantes.

Las conclusiones fueron:

- ❖ Diseños realizados en Verilog fueron, sin falta, completado más rápido que los hechos en VHDL. (En términos de puertas).

- Añadiendo los diseñadores diseños basados VHDL y Verilog basados en retardo a nivel de puertas, pero la adición de los diseñadores para diseños basados en VHDL tenía un mayor impacto negativo. (Probablemente debido a problemas de datos de escritura.)
- 80 diseñadores de diferentes niveles de experiencia, trabajando en grupos de 2 a 10. Medir desde el final de (la) especificación (ciclo) para el visto bueno final, la de mayor rendimiento fue un equipo Verilog en 1500 puertas, la más baja (Realización) fue un equipo VHDL con 8 puertas.
- Para VHDL las deficiencias fueron más evidentes en los niveles de puerta y de transistores. Además, no había instalaciones para el manejo de información de temporización. Debido a estos cruciales, limitaciones, VHDL no hizo un gran impacto en la comunidad del diseño, a pesar de que fue muy promocionado en las revistas de comercio y por las empresas.
- Verilog HDL viene del mundo comercial y fue desarrollado como parte de un sistema de simulación completa. También se ha desarrollado para ser utilizado para describir sistemas de hardware que utilizan medios digitales. Verilog HDL dejó permitir a los diseñadores para representar sus diseños en el método familiar (nivel de la puerta y el interruptor descripciones), así como abstracta o comportamiento. Al permitir que está arriba hacia abajo o metodología de abajo hacia arriba que se contemplaban los diseñadores de un período de aprendizaje para obtener cómodo y al mismo tiempo lograr significativas ganancias de productividad. [28]

5.2.8 VHDL-AMS Y VERILOG AMS COMO ALTERNATIVA HDL PARA EL EFICIENTE MODELADO DE MULTI DISCIPLINA DEL SISTEMA.

Introducción

Este documento se centra en los aspectos comunes y diferencias entre la descripción en dos hardware de señal mixta idiomas VHDL -AMS y Verilog -AMS en el caso de modelar el sistema en papel heterogénea o multi- disciplina tiene dos objetivos. La primera de ellas consiste en modelar la estructura y el comportamiento de un sistema de airbag utilizando tanto el VHDL - AMS y Verilog -AMS.

Dicho sistema comprende varias abstracciones de tiempo (es decir, en tiempo discreto y en tiempo continuo) , varias disciplinas o dominios de energía (es decir , eléctricas, térmicas , ópticas , mecánica , y química) , y varios de tiempo continuo. Descripción formalismos (es decir, la ley conservadora y flujo de señal descripciones). El segundo objetivo es analizar los resultados de los procesos de modelado propuesto en términos de lo descriptivo capacidades de los lenguajes VHDL -AMS y Verilog -AMS y de las herramientas de simulación generados. En el documento se muestra que los dos idiomas ofrecen medios eficaces para describir y simular sistemas multi-disciplina, aunque el uso de diferentes enfoques descriptivos también pone de relieve las limitaciones actuales de la herramienta ya que las definiciones de lenguaje total aún no son compatibles. [29]

TABLA V.VII: Comparación de lenguajes
Articulo Cientifico#9

característica s de su clase	Características	vhdl - AMS	Verilog - AMS
Aspectos del Lenguaje	Definición	IEEE Std 1076.1 – 1999 Extensión estricta a la norma IEEE 1076	versión estándar 2.1 (Enero 2003) Extension IEEE Std 1364(verilog HDL)

		(VHDL)	
	Sucesiones	Ada Caso insensible	C Caso sensible
	Modularidad	Separación de puntos de vistas externos (entidades) y vistas internas (arquitecturas)	Los módulos incluyen ambos externos/interfaz
		paquetes configuración	
	Generalidad	Parámetros, generar sentencias	
	Gestión de la Biblioteca	si (pre compilado de las unidades de diseño)	No
	Subconjunto Analógico	No	Si (limitados soportes de sus tipos
expresiones de Estructura	Puertos	Por eventos y continuos	
		Conservador y no conservadora (flujo de señal)	
		Continuos puertos son modelados	Continuos puertos son de entrada salida
	Composición	Instanciación jerárquica de los componentes	
	Conservación	Definidos dominios de energía naturales,	Disciplina dominios de energía definidos,
	Semántica	subtipo de atributos Naturaleza	naturaleza de los atributos de disciplina definidos,
		no predefinida	Conjunto predefinido de disciplina.
		terminal y sucursal de cantidades	Funciones de acceso y sucursales nombradas
Expresiones de	Objetos	terminales, cantidades, señales, variables,	variables, registros, alambres

Comportamiento		constantes	
	declaraciones	Concurrente, secuencial, continuo (simultánea y de procedimiento) Declaraciones continuas se pueden mezclar	Estados continuos tienen que ser agrupados en un
		libremente con las declaraciones concurrente	Bloque analógico. A solo un bloque analógico por modulo
	Expresión de DAE	Forma explícita e implícita de soporte de ecuaciones Formulaciones Simultaneas y procesal Atributos derivados no es posible en cantidad Los atributos se pueden encadenar para	explicita forma de soporte de ecuaciones apoyo limitado de forma implícita de ecuaciones Formulación de procedimiento El operador no puede ser encadenado
		derivadas en orden superior Las funciones matemáticas definidas por separado el estándar IEEE 1076.2	para la derivada de orden superior Se debe declarar los nodos locales pre definidas funciones matemáticas
	discontinuidad	Las discontinuidades se deben anunciar explícitamente en el modelo.	
	Manejo	definidos por el usuario re - inicialización	El orden de la discontinuidad se puede especificar

		después del soporte discontinuidad	
Semántica conservación	Dominio Energía	naturaleza de los dominios de energía definido y subtipos atributos naturales definidos Pode de cantidades	Disciplina definida por lo dominios de energía y Naturaleza definida atributos de la disciplina. Pre conjunto definido de disciplina
			Acceso a funciones y derivación de objetos
	Formulación	Formulación de ecuación orientada con declaraciones simultáneas No especifica circuitos gráficos ni representaciones forzadas	Formulación de circuitos orientados con exploración y derivación de origen y aditivas declaraciones de contribución Formulación nodal forzada
semántica de la señal de flujo	Interfaz del Modelo	Interfaz direccional (gratis) cantidades	Puertos módulo asociado a una disciplina con una sola naturaleza (flujo de señal disciplina) Puertos de flujo de señales se interpretan como
			Bien fundado. Exploración o fuentes potenciales.
	Bloques funcionales	Laplace y transformada z	
Aspectos de señal Mixta	modelo Interfaz	A/D y D/A atributos de la interfaz del lenguaje ('ramp, 'slew, 'above) No se encontró asociación puerto directo	A / D y D / A de filtros de idiomas de interfaz y Los detectores de eventos Inserción automática de los módulos de conexión
			(infra-o inter-

			disciplinas)
	Interacción del comportamiento	Acceso a discretas señales en continuos Contextos Acceso a continuas cantidades en discretos	El acceso de redes discretas y variables en contexto continuo El acceso de las redes continuas y variables en
		Contextos	contexto discreta
Controles de Simulación	Solubilidad	Chequeo solvencia realizada en una unidad de nivel de diseño	No solvencia de chequeo
	Tiempo de paso	El tamaño del tiempo de paso puede ser delimitado	
	Tolerancia	La cadena de anotaciones genéricas no están vinculado al simulador formalmente	Tolerancias definidas en naturaleza

5.2.9 VERILOG VHDL VS

Desarrollo

- La capacidad de resistencia de Verilog.
- Verilog es más fácil de aprender que VHDL.
- Se puede discutir todo lo que quieras acerca de los méritos técnicos de las dos lenguas, y la "comprensibilidad" de cada uno. Sé que aprendido personalmente Verilog en un período muy corto de tiempo. Más tarde, cuando Decidí que realmente debería aprender VHDL con el fin de ser capaz de comercializar mi

producto en contra de ella, me di cuenta de que realmente era aprender VHDL más difícil. Estoy seguro de que he pasado más de esfuerzo tratando de aprender VHDL que yo hice en los primeros días de mi uso Verilog, y yo no soy más que apenas sabe leer y escribir en VHDL

- Al par de esta barrera inferior para usar Verilog con el hecho de que en realidad no hay una buena razón para cambiar de Verilog VHDL y hasta VITAL había buenas razones para cambiar de VHDL Verilog. [30]

5.10 DISEÑO DE PROCESADORES CON VHDL

VHDL vs VERILOG

Llevamos casi una década debatiendo 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 de peso para claramente decantarse por uno o por otro. 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 de un lenguaje simple y flexible que es fácil de aprender aunque tiene ciertas limitaciones. Por otro, VHDL nace como un lenguaje que resuelve las necesidades actuales de diseño, y otras futuras, es decir con mayor amplitud miras. El resultado es un lenguaje más rígido y con una sistematización en el código que lo hace más legible, aunque las reglas de diseño obligan a proceder de forma, a veces poco ágil. Ante la duda de cuál de ellos enseñar en este curso la respuesta está en la propia comunidad científica: lo ideal es conocer ambos. En este sentido se coincide que el

conocedor de Verilog le cuenta tanto aprender VHDL, sin embargo el proceso contrario es en general, más aceptado.

El programador de VHDL aprender Verilog en poco tiempo La experiencia de los autores de estos capítulos es precisamente esa, y que conceptualmente es más didáctico el VHDL. (31)

5.3 ESTUDIO COMPARATIVO DE LOS AUTORES DE LA INVESTIGACION

TABLA V.VIII: Comparación de lenguajes de los autores

PARAMETRO	VHDL	VERILOG
CODIGO	❖ Se pueden desarrollar diseños digitales de distintos estilos como el funcional, flujo de datos, flujo de datos con ecuaciones booleanas y de estilo estructural este lenguaje permite que el programador diseñe de acuerdo a la necesidad o pericia que tenga en programación de este lenguaje para poder llegar a la resolución de cualquier diseño digital. Se necesita más líneas de código para la resolución del diseño digital a resolver. ❖ Si se desarrolla la programación del diseño digital estudiando su estructura, comportamiento y	❖ 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. ❖ Es de fácil aprendizaje debido a que es una derivación del lenguaje C y todo ingeniero lo

	dividiéndolos en pequeños procesos se obtendrá la resolución de este diseño de manera fácil y rápida. ❖ Este lenguaje es un poco complicado de aprender debido a que está basado en derivaciones de los lenguajes ADA y Pascal. ❖ Se necesita declarar la entidad y la arquitectura del diseño digital a desarrollar de manera obligatoria para que se pueda ejecutar el programa. ❖ Presenta una sola forma de asignar valores. ❖ La declaración de las compuertas básicas conocidas se las realizas llamándolas por su debido nombre. ❖ Las estructuras de selección como if, case with etc en su declaración tiene forma similar a las de un lenguaje de programación como en pascal. ❖ Posee una forma de generar eventos o pulsos de reloj a través de la palabra reservada del programa "Event". ❖ Se pueden declarar	conoce. ❖ 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 compuertas básicas conocidas se las realiza llamándolas por su nombre o a través de símbolos. ❖ Las estructuras de selección como if, case, with etc en su declaración es muy semejante a la declaración del lenguaje C. ❖ Posee dos formas de generar eventos o pulsos de reloj a través de la palabra reservada "Posedge y Negedge" ❖ Se pueden declarar módulos y sus módulos.

	procesos y funciones.	
SIMULACIÓN	❖ 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 CAD para la simulación en 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 CAD para la simulación en ambos lenguajes.
TIEMPO DE SIMULACIÓN	❖ 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.	❖ 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.
LIBRERÍAS	❖ Se necesita declararlas de manera obligatoria para que pueda ser ejecutadas las líneas de código.	❖ No necesitan declararse que ya se ejecutan automáticamente.
SOFTWARE	❖ Se utilizó el mismo software ISE Design suite 14.7 para la programación y para la simulación se usa una herramienta llamada isim que está incluida en el software. ❖ Fácil instalación, manipulación y aprendizaje. ❖ Gratis de descargar.	
HARDWARE	❖ Entrenador de Xilinx FPGAs Spartan 3E Starter Board compatible con ambos lenguajes. ❖ Fácil de utilizar en ambos lenguajes.	

5.4 RESULTADOS DEL ESTUDIO COMPARATIVO

En esta parte de la investigación se obtendrán los resultados de los lenguajes HDL estudiados y analizados en los artículos científicos mencionados en literal (5.2) y el estudio

comparativo de los lenguajes realizado por los autores del documento de investigación que realizo por medio de las guías prácticas unidad (4) de esta investigación.

Realizando cuadros estadísticos de cada uno de los parámetros que sea han considerado como son: Líneas de codificación, Simulación, Librerías, Tipos de Datos, Sensibilidad, Preferencia.

Se eligió un total de 10 artículos científicos el cual va a representar el 100%.

Para los autores de esta investigación existen 50 prácticas, lo que representa el 100% para nuestro cuadro estadístico en la representación de los resultados.

Finalmente se realizará un promedio entre los estudios científicos y la investigación de los autores, para determinar valores estadísticos que permitan determinar cuál es el lenguaje más óptimo entre VHDL y Verilog.

Los cuadros estadísticos se describen a continuación:

- **LÍNEAS DE CODIFICACIÓN**

Cuando hablamos de líneas de codificación nos referimos a la cantidad de líneas de código que se utilizan para desarrollar un diseño

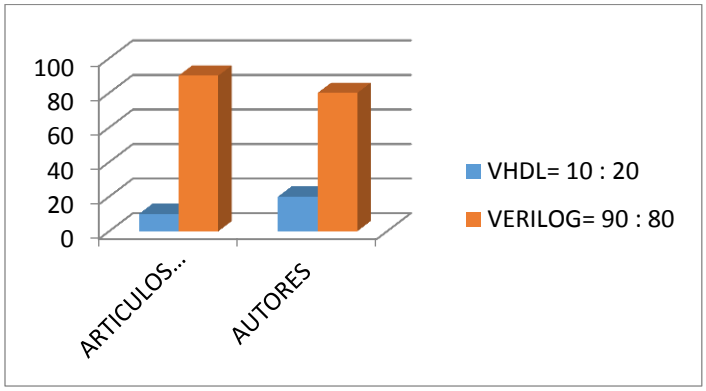


FIGURA V.1: Cuadro Estadístico Líneas de Codificación

Fuente: *Los Autores*

Los autores le dimos una calificación del 80% a **VERILOG** debido a que tiene similitud a lenguaje "C" por lo que se torna fácil su aprendizaje y el 20% a **VHDL** debido a que tiene similitud a lenguaje pascal por lo que se torna más complicado su aprendizaje. De los artículos científicos el 90% de artículos prefiere Verilog debido que conocen la programación en C y el 10% prefiere VHDL ya que empezó programando en este lenguaje.

- **SIMULACIÓN**

Cuando hablamos de simulación nos referimos a las simulaciones realizadas en nuestro sistema CAD.

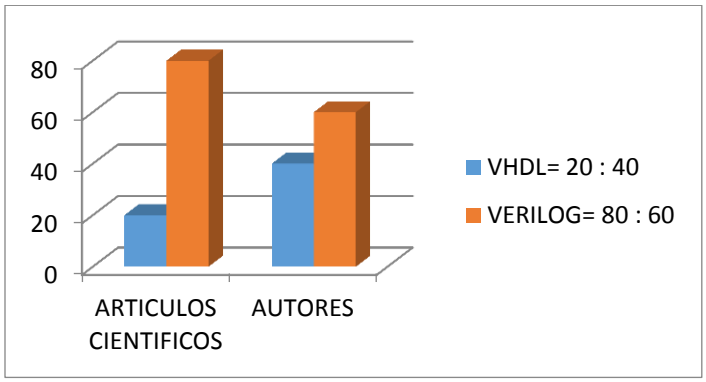


FIGURA V.2: Cuadro Estadístico de Simulación

Autor: *Los Autores*

Los autores le dimos una calificación del 60% a VERILOG debido a que se utilizó el mismo sistema CAD para ambos lenguajes pero aun así verilog ejecuto mucho más rápido sus líneas de código y del 40% a VHDL debido a que a pesar de utilizar la misma plataforma CAD ejecutaba un mismo programa en mucho más tiempo. De los artículos científicos el 70% prefiere verilog porque compila más rápido por menos líneas de codificación y el 30% prefiere VHDL ya que dice que la compilación depende de las características de la tarjeta.

❖ LIBRERÍAS

Cuando hablamos de librerías nos referimos a los recursos o paquetes que se ponen en la descripción de la sintaxis de programación.

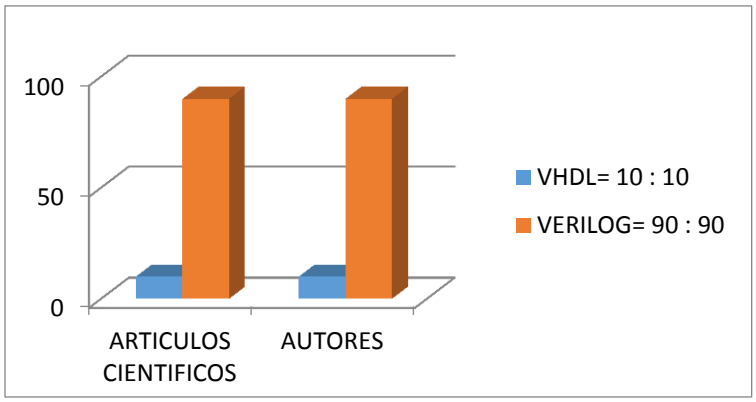


FIGURA V.3: Cuadro Estadístico de Librerías

Autor: *Los Autores*

Los autores le dimos una calificación del 90% a VERILOG porque este lenguaje nos evita la tarea de estar declarando librerías y ahorra tiempo al diseñador y del 10% a VHDL porque hay que estar realizando las respectivas declaraciones de sintaxis de las librerías para que nuestras líneas de código puedan ser ejecutadas correctamente.

❖ TIPO DE DATOS

Cuando hablamos de tipo de datos nos referimos a la utilización de los datos de tipo array, registros, vectores, señales externas (analógicas, digitales) etc.

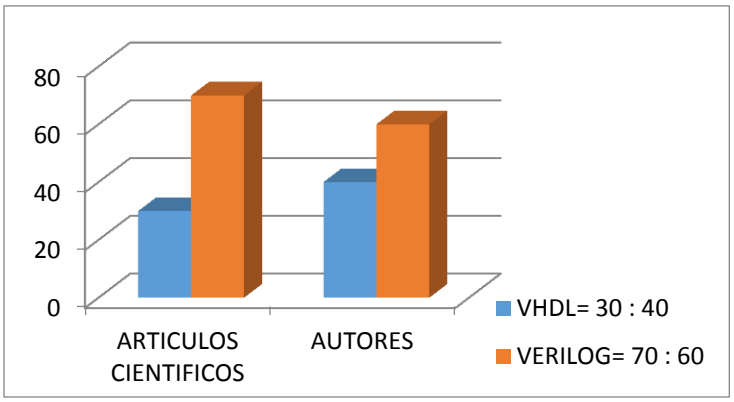


FIGURA V.4: Cuadro Estadístico Tipos de Datos

Autor: *Los Autores*

Los autores le dimos una calificación del 60% a VERILOG porque para poder hacer uso de este tipo de datos se debe de realizar su respectiva declaración de sintaxis que se la representa de una manera muy abstracta y entendible para cualquier persona que desee utilizarlos y del 40% a VHDL porque se necesita de una declaración de sintaxis extensa y si se es principiante en este lenguaje se tiene dificultad para poder hacer uso de estos datos.

- **SENSIBILIDAD**

Cuando hablamos de sensibilidad nos referimos a la facilidad de poder cometer errores en los momentos de codificación de sintaxis, además nos referimos a la sensibilidad que tienen en distinguir entre mayúsculas y minúsculas.

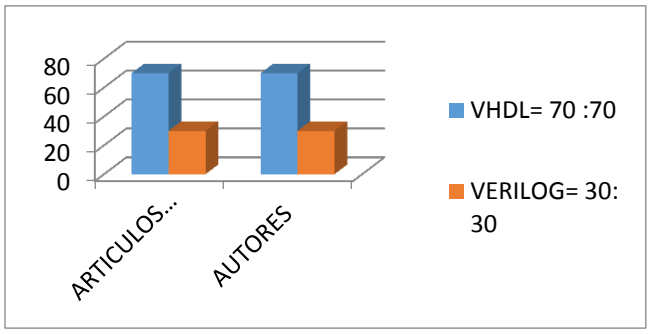


FIGURA V.6: Cuadro Estadístico Sensibilidad

Autor: *Los Autores*

Los autores le dimos una calificación del 30% a VERILOG porque este programa presenta una sensibilidad muy grande debido a que distingue entre mayúsculas y minúsculas, y hay ciertas palabras reservadas que obligatoriamente deben escribirse en mayúscula o en minúscula en caso de no hacerlo al compilar el diseño existirá un error de sintaxis. Para VHDL una calificación del 70% debido a que este programa no diferencia entre mayúsculas y minúsculas simplemente deben de estar correctamente declaradas.

- **PREFERENCIA**

Cuando hablamos de preferencia nos referimos a la aceptación que tienen cada uno de los lenguajes.

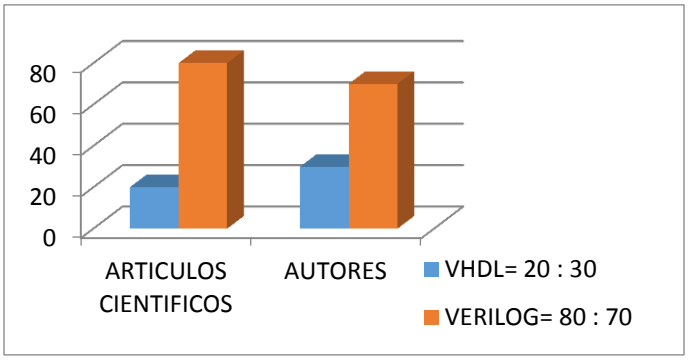


FIGURA V.7: Cuadro Estadístico de Preferencia

Autor: *Los Autores*

Los autores le dimos una calificación del 30% a VHDL, como podemos observar en la gráfica tiene un 10% más de calificación de los artículos científicos, debido a que se nos hizo fácil aprender VHDL porque teníamos conocimientos en lenguaje de programación en Pascal. Para VERILOG le asignamos una calificación del 70% por todas las ventajas que hemos mencionado anteriormente y el potencial que tiene este lenguaje para los presentes y futuros diseños digitales.

5.5 DEMOSTRACION DE LA HIPOTESIS

❖ CUADRO DE RESUMEN DE LOS ARTÍCULOS CIENTÍFICOS

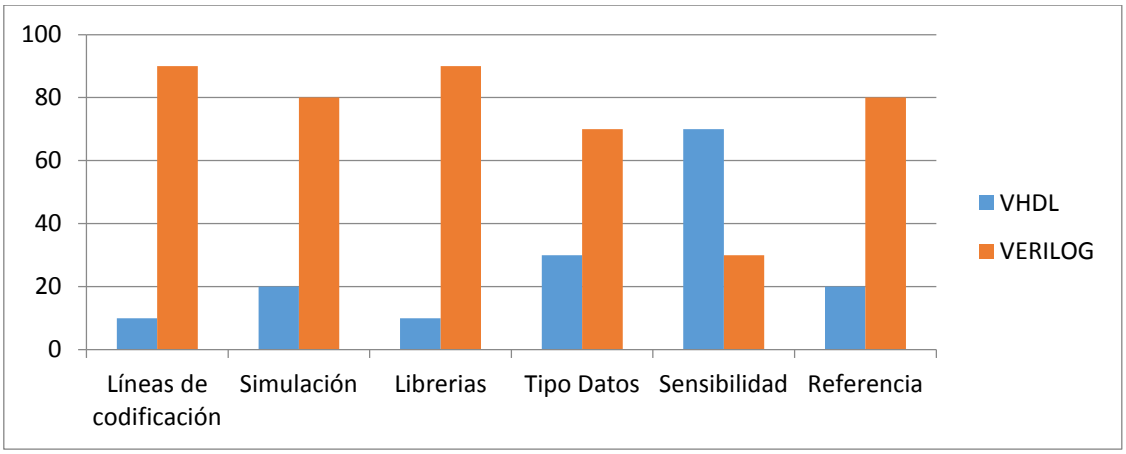


FIGURA V.8: Cuadro Estadístico del Resumen de Artículos Científicos

Autor: *Los Autores*

Media Aritmética

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} ai$$

$$\overline{x} = \frac{a_1 + a_2 + \cdots . a_n}{n}$$

❖ VHDL

$$\overline{x} = \frac{10 + 20 + 10 + 30 + 70 + 20}{6}$$

$$\overline{x} = 26.67$$

❖ **VERILOG**

$$\bar{x} = \frac{90 + 80 + 90 + 70 + 30 + 80}{6}$$

$$\bar{x} = 73.33$$

Después de realizar un promedio entre todos los parámetros de la sintaxis de programación de cada uno de los lenguajes HDL, tenemos como respuesta para Verilog 73.33% y el VHDL 26.67%, teniendo como resultado el lenguaje más óptimo para descripción de hardware a Verilog, estos resultados luego se promediarán con los promedios de los autores.

❖ **CUADRO DE RESUMEN DE LA MEDIA ARITMÉTICA DE LOS ARTICULO CIENTÍFICOS**

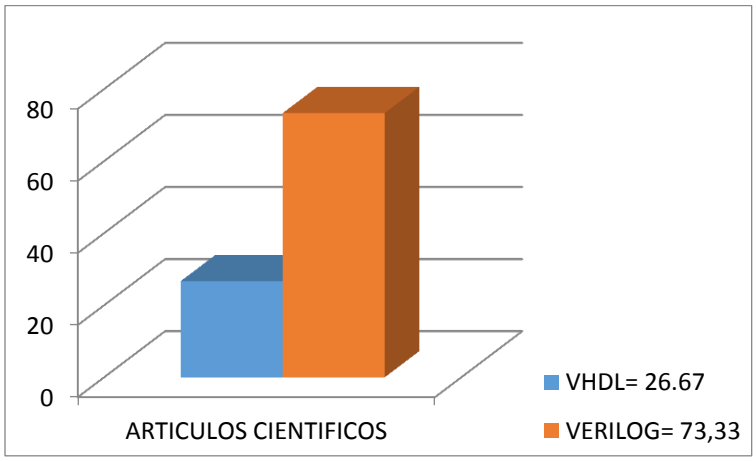


FIGURA V.9: Cuadro Estadístico Media Aritmética de los Artículos Científicos

Autor: *Los Autores*

Es la representación gráfica de los promedios de los artículos científicos, con los resultados antes mencionados.

❖ **CUADRO DE RESUMEN DE LOS AUTORES**

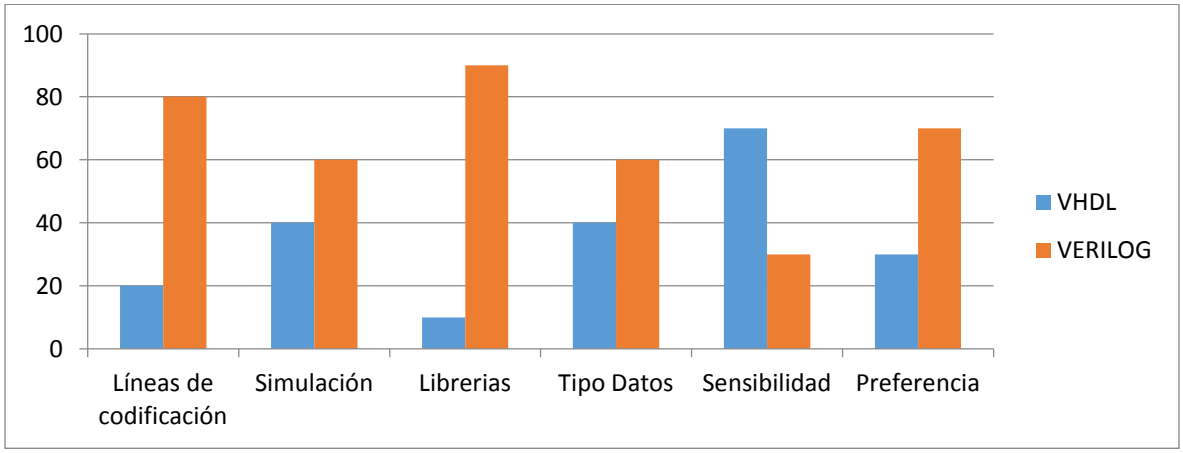


FIGURA V.10: Cuadro Estadístico de los Autores

Autor: *Los Autores*

Media Aritmética

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} ai$$

$$\overline{x} = \frac{a_1 + a_2 + \cdots . a_n}{n}$$

❖ **VHDL**

$$\overline{x} = \frac{20 + 40 + 10 + 40 + 70 + 30}{6}$$

$$\overline{x} = 35$$

❖ **VERILOG**

$$\overline{x} = \frac{80 + 60 + 90 + 60 + 30 + 70}{6}$$

$$\overline{x} = 65$$

Promedio tomado en cuenta los parámetros de sintaxis de programación para cada lenguaje HDL, en Verilog con el 65% y VHDL con 35%, lo que determina que el lenguaje más óptimo es Verilog.

- **CUADRO DE RESUMEN DE LA MEDIA ARITMÉTICA DE LOS AUTORES.**

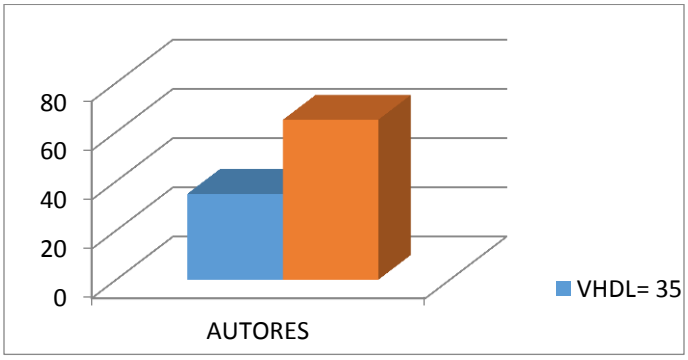


FIGURA V.11: Cuadro Estadístico Media Aritmética de los Autores

Autor: *Los Autores*

Representación gráfica de los promedios entre VHDL y Verilog, lo que a simple vista el lenguaje más óptimo es Verilog.

❖ CUADRO DE RESUMEN DE LAS MEDIAS ARITMETICAS

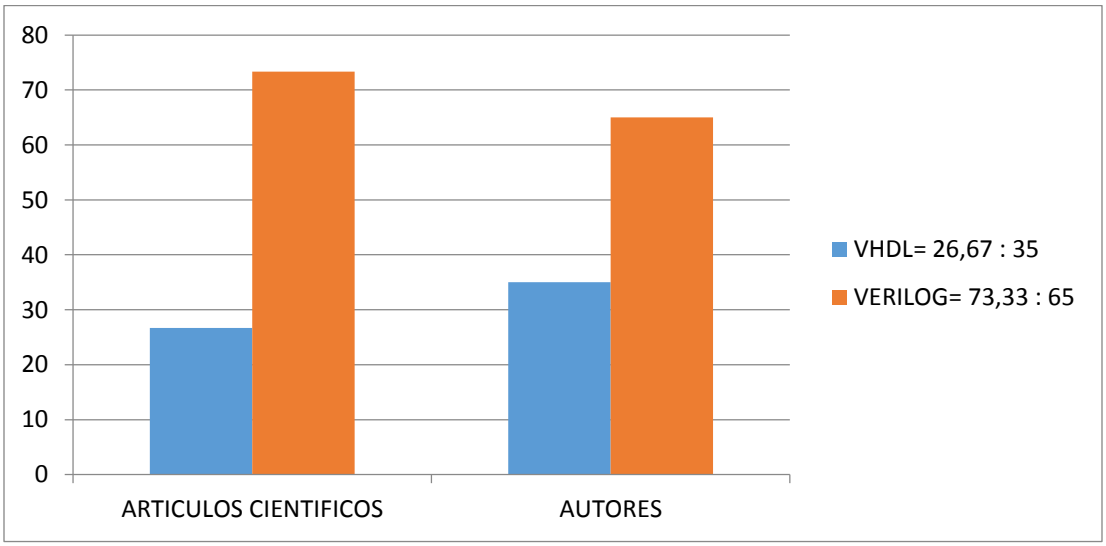


FIGURA V.12: Cuadro Estadístico de Medias Aritméticas de los Artículos y los Autores

Autor: *Los Autores*

Media Aritmética

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} ai$$

$$\overline{x} = \frac{a_1 + a_2 + \cdots . a_n}{n}$$

❖ VHDL

$$\overline{x} = \frac{26.67 + 35}{2}$$

$$\overline{x} = 30.84$$

❖ VERILOG

$$\overline{x} = \frac{73.33 + 65}{2}$$

$$\overline{x} = 69.16$$

CUADRO FINAL DE RESULTADOS

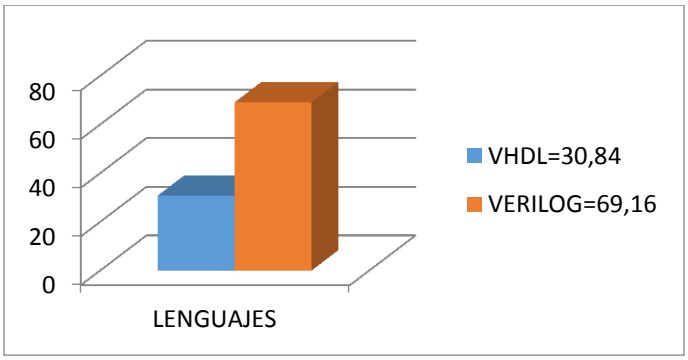


FIGURA V.13: Cuadro Estadístico Final de VHDL vs VERILOG

Autor: *Los Autores*

Se ha llegado al final del estudio de los HDL VERILOG y VHDL donde se puede observar en el gráfico que VERILOG tiene un 69.16 % frente al 30.84% de VHDL.

VERILOG supera a VHDL en la mayoría de los parámetros escogidos para el estudio comparativo, convirtiéndolo a VERILOG en un lenguaje óptimo para su estudio.

Verilog tiene una diferencia del 38.32% mayor que VHDL.

CONCLUSIONES

- El Estudio comparativo de los lenguajes de descripción de hardware, nos permitió conocer que hay muchas nuevas tecnologías que en varios países están en pleno auge en el diseño digital, en nuestra investigación analizamos el uso de las tarjetas FPGA y su programación para configurarlas son los lenguajes VHDL y Verilog. Esta nueva tecnología se está usando en muchas aplicaciones en donde se necesita que trabajen a una mejor velocidad como por ejemplo: en aeronáutica, biomédica, sistemas de control, etc...

- Con el desarrollo de las guías prácticas de programación de los lenguajes VHDL y Verilog permitieron tener un mejor conocimiento en diseño digital con la ayuda de la tarjeta FPGA, servirán para contribuir al aprendizaje en la parte teórica y práctica de los estudiantes de la EIE-CRI de la ESPOCH.

- Mediante los parámetros de sintaxis de programación de las guías prácticas, ayudo a determinar cuál de los lenguajes de programación es el más óptimo para aprender a programar debido que mientras se realizaban las programaciones en cada uno de los lenguajes se veían diferencias notables al momento de diseñar un circuito.

- Las guías prácticas permitirán a los estudiantes de EIE-CRI tener conocimientos básicos de esta nueva tecnología de las tarjetas FPGA, desarrollando sus propios diseños digitales que les ayudara para aplicar en el ámbito profesional y así generar nuevas aptitudes en la tendencia tecnológica que apunta en esos rumbos. Esta tecnología se utiliza en otros países y en varias universidades de nuestro país.

- Mediante los fundamentos teóricos de programación en VHDL y Verilog se pudo determinar cuál de los dos lenguajes es más fácil de programar y cual tiene menor código de programación, como resultado a la investigación es Verilog que tiene mucha semejanza a la base de programación es C, que permiten entender de una mejor manera el diseño digital en los estudiantes de la EIE-CRI.

RECOMENDACIÓNES

- Las tarjetas de FPGA es una nueva tecnología que debe ser explotada en varios campos aplicativos de la Ingeniería Electrónica y la EIE-CRI de la ESPOCH aún no se ha involucrado en los nuevos avances tecnológicos que se están dando en el mundo, por lo cual la escuela EIE-CRI debería contribuir al desarrollo investigativo de aplicaciones, tales como los sistemas digitales, sistemas de control, robótica, biomédica, etc., con el fin de mejorar en el ámbito profesional de las nuevas tecnologías.

- La EIE-CRI debería incluir la asignatura de sistemas digitales avanzado, como materia propia de la malla de estudio, ya que todo ingeniero Electrónico debe conocer por lo menos un lenguaje de descripción de hardware con la finalidad de encaminarse a los nuevos avances tecnológicos en el uso de las tarjetas FPGA.

- Los Estudiantes que deseen aprender HDL debe tener conocimientos básicos, de programación en C y sistemas digitales, recomendándoles que empiecen programado en Verilog, en esta investigación se le enseña modelos de sistemas digitales básicos y para profundizar a diseños más avanzados les recomiendo revisar la bibliografía de este documento.

- La EIE-CRI debería adquirir más tarjetas FPGA, para implementar un laboratorio de sistemas digitales avanzados con el fin de que los estudiantes puedan encaminarse al avance tecnológico actual.

RESUMEN

El estudio comparativo de los lenguajes de descripción de hardware (HDL) y su aplicación en la implementación del laboratorio de sistemas digitales avanzados mediante FPGA (Arreglo de Compuertas Programable en el Campo) en la Escuela de Ingeniería Electrónica en Control y Redes Industriales (EIE-CRI).
El estudio comparativo determina cual lenguaje es el más óptimo entre VHDL (Circuitos Integrados de muy Alta Velocidad) y Verilog, tomando en cuenta la sintaxis de programación, realizando guías prácticas de programación.

Utilizando el método científico experimental, se desarrolló diferentes códigos de programación para establecer diferencias entre VHDL o Verilog, mediante herramientas de software ISE Design Suite 14.7 y tarjeta FPGA necesarias para diseño digital avanzado. Además se aplicó el método Delphi, esta técnica permite recopilar consensos de opiniones de expertos en el tema, en esta investigación analizamos 10 artículos científicos referentes a VHDL vs Verilog.

Mediante el estudio comparativo y analizando parámetros de sintaxis en programación como: líneas de codificación, simulación, librerías, tipos de datos, sensibilidad y preferencia del lenguaje, igual analizando artículos científicos publicados referentes al tema de Verilog vs VHDL, dando como resultado en la investigación de los autores y los artículos científicos en VHDL un 30,84% y en Verilog un 69.16% estableciendo una diferencia de aproximadamente un 38,32% entre los HDL.

Se concluye que Verilog es el HDL más óptimo para diseño digital, debido a que está basado en el lenguaje C y de fácil aprendizaje.

Se recomienda el uso de Verilog para los diseños digitales avanzados, para los estudiantes de la EIE-CRI interesados a incursionar en esta nueva tecnología de tarjetas FPGA.

ABSTRACT

The comparative study of hardware description languages (HDL) and its application in the implementation of advanced digital system labs through FPGA (Field Programable Gate Array) in the Control and Industrial Networks Engineering School (EIE-CRI).

The Comparative study determines what language is the best between VHDL (Integrated circuit of very high speed) and Verilog, taking into account the programming syntax, perfoming practical guides of programming.

Different programming was developed codes by using the experimental scientific method in order to establish differences between VHDL and Verilog through software tools ISE Desing Suite 14.7 and card FPGA needed to advance digital desing. Besides, the Delphi method was applied, this technique let to collect expert consensus opinions in the theme, in this research 10 scientific articles related to VHDL against Verilog were analyzed.

By the means of a comparative study and analyzing syntax parameters on programming like: lines of codification, simulation, bookstores, data type, sensitivity, language preference, analyzing published scientific article related VHDL against Verilog as well. As a result VHDL 30,84% and Verilog 69,16%, establishing a difference about 38,32% between HDL.

It is concluded that Verilog is the best HDL for digital desing due to it is base don the language C and of easy learning.

It is recommended the use of Verilog for advance digital desing for students of the EIE-CRI interested to make an incursion in these new technological cards FPGA.

BIBLIOGRAFIA

[1]. **DSPACE.UPS.,** Introducción al diseño digital utilizando Lenguaje Descriptivo de Hardware Verilog.,UPS – Ecuador., Pp 1-5
http://dspace.ups.edu.ec/bitstream/123456789/40/7/Capitulo1.pdf
05/11/2013

[2]. **JAQUENOD GUILLERMO.,** Lenguajes de Descripción de Hardware. Una breve visión sobre los HDLs en general y AHDL en particular., Colombia., Proenergia.,1999., Pp 4-5, 9.
http://www.proenergia.net/ftp/colarte/Cursos%20Academicos/LOGICA%20DIGITAL/CLASES/AHDL.pdf
05/11/2013

[3]. **ORENATO.,**Lenguaje de Descripción de Hardware: Vhld., La Mancha-España., Orenato., Pp 2, 9-10.
http://oretano.iele-ab.uclm.es/~miniesta/intro%20hdl.pdf
06/11/2013

[4]. **BROWN STEPHEN y otros.,** Fundamentos de lógica digital con diseño VHDL., 2da.ed., Mexico., McGraw-Hill Interamericana., 2006.,Pp 58-62

[5]. **SANCHEZ MARCO.,** Introducción a la programacion en VHDL., 1er.ed., Madrid-España., Faculdad de Informatica-Universidad Complutense de Madrid., 09/2011., Pp 8-10.

[6]. **CAMACHO LOZANO PELEGRÍN.,** VHDL orientado a sintesis en FPGAs., 1er.ed., Malaga-España., Departamento de Tecnología Electrónica-Universidad de Málaga, 2012., Pp 17-31.

[7]. **IEEE,** Verilog Grupo Verilog Normalización., 2012.
http://www.verilog.com
12/12/2013

[8]. **NUÑEZ.** Tutorial Verilog., Las Palmas-España., Pp 4-5, 15-17.
http://www.iuma.ulpgc.es/~nunez/clases-FdC/verilog/Verilog%20Tutorial%20v1.pdf
13/12/2013

[9]. **Dr-Ing. VEGA PAOLA.,** Introducción a Verilog., Costa Rica,. 2007., Pp 3-4.
http://www.ie.itcr.ac.cr/pvega/Lab.%20Dise%F1o%20Sist.%20Digitales/Introducci%F3n%20a%20Verilog.pdf
16/12/2013

[10]. **WIKIBOOKS.,** Programación en Verilog- Elementos básicos del lenguaje., 30/10/2013.
http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Verilog/M%C3%B3dulos.
16/12/2013

[11]. **MATPIC.,** Tutorial Verilog., 2013.
http://www.matpic.com/esp/vhdl/verilog.html
17/12/2013

[12]. **WIKIPEDIA.,** Field Programmable Gate Array., 22/07/2013.
http://es.wikipedia.org/wiki/Field_Programmable_Gate_Array#Historia.
20/10/2013

[13]. **RIVERA LUIS.,** Taximetro Digital en VHDL.,Colombia., Universidad de Ibagué, 2004-2005., Pp4.
http://www.monografias.com/trabajos-pdf2/taximetro-digital-vhdl/taximetro-digital-vhdl.pdf.
20/12/2013

[14]. **REINALDO NOELIA.,** FPGA., 06/12/2007.
http://reinaldo-noelia-fpga.blogspot.com
20/12/2013

[15]. **LÓPEZ VALLEJO M. L. y otros.,** FPGA: Nociones básicas e Implementación., Madrid-España., Departamento de Ingeniería Electrónica., 2004. P4 - P6.
http://www.lsi.die.upm.es/~marisa/docencia/fpga_a2_2004.pdf
20/12/2013

[16]. **GONZÁLEZ JUAN y otros.,** Tarjeta entrenadora para FPGA., Madrid-España., Universidad Autonoma de Madrid,. 2003., Pp 3-5.
http://www.iearobotics.com/personal/juan/publicaciones/art1/html/jps.html
20/12/2013

[17]. **SISTERNA CRISTIAN.,** FPGAs., San Juan-Argentina., P7 - P11; 35-37.
http://dea.unsj.edu.ar/sisdig2/Field%20Programmable%20Gate%20Arrays_A.pdf
20/12/2013

[18]. **MORALES SANDOVAL MIGUEL.,** Introducción a los FPGAs y el Cómputo., Mexico.,INAOE., 2006., Pp1-3.
http://www.tamps.cinvestav.mx/~mmorales/documents/FPGAsyReconfig.pdf.
20/12/2013

[19]. **CONSORCIO MEXICANO DE MICROSISTEMAS.,** Tecnología FPGA., Mexico., 2012.
http://www.cmm.org.mx/index.php/microsistemas/tecnologia-fpga
20/12/2013

[20]. **BOEMO SCALVINONI EDUARDO.,** Estado del Arte de la Tecnología FPGA., 1ra.ed.,Argentina., Inti., 2005., Pp5.
http://www.inti.gob.ar/electronicaeinformatica/instrumentacion/utic/publicaciones/cuadernilloUE/CT_Microelectronica17_FPGA.pdf
05/01/2014

[21]. **ORMENO JOSE.,** Prácticas de Electrónica Digital., 26/08/2013., Pp12.14
http://es.scribd.com/doc/163262581/Practicas-Electronica-Digital#download.
05/01/2014

[22]. **HILBERT SHANNON.,** VERILOG VHDL VS., 1ra.ed., 04/02/2013.
http://www.bitweenie.com/listings/verilog-vs-vhdl/
06/01/2014

[23]. **BRAN CARLOS GUILLERMO Y OTROS.,** Implementación de modelos de fotodiodos en lenguajes de descripción de hardware de señal mixta., 1ra.ed., El Salvador., 11/2011., Pp1-2.

[24]. **BOTROS NAZEIH M..,** HDL programming fundamentals: VHDL and Vrilog., lra.ed., : DaVinci Engineering Press., 18/11/2011.

[25]. **BUDZYN GRZEGORZ.,** Programmable Logic Desing - Lecture 12 VHDL vs Verilog., Polonia : Wroclaw University of Technology., Pp37-58.

[26]. **MAGDANZ ERIK y otros.,** Estudio de caso - Verilog Vhdl Vs., California-Estados Unidos., Instituto de Berkeley.
http://inst.eecs.berkeley.edu/~eecsba1/sp97/reports/eecsba1h/verilog.html
07/01/2014

[27]. **BAILEY STEPHEN.,** Comparación de VHDL , Verilog y systemVerilog., Estados Unidos., Model tecnhology., 2003.Pp 27.
http://www.fpga.com.cn/advance/vhdl_14919.pdf
07/01/2014

[28]. **GOLSON STEVE.,** Verilog HDL vs VHDL para la primera vez del usuario.,

1ra.ed., Estados Unidos., College of Information Sciences and Technology Penn State University 1994., Pp1-2.

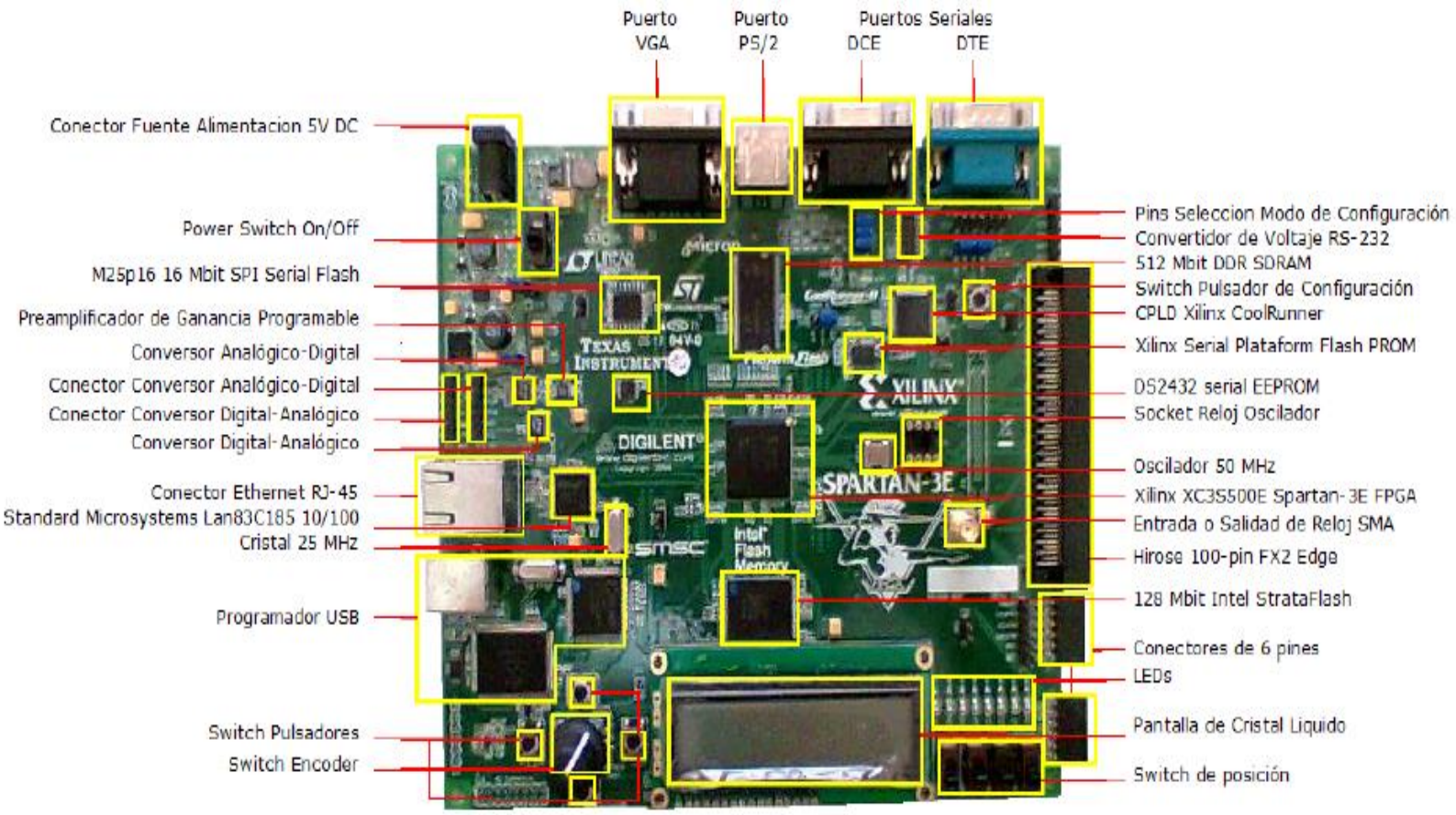
[29]. **F. PECHEUX Y OTROS.,** VHDL-Ams y Verilog Ams como alternativa Hdl para el eficiente modelado de multi disciplina del sistema., IEEE., Pp1,20-21.
http://infoscience.epfl.ch/record/53637/files/TCAD-final_3.pdf
11/01/2014

[30]. **SANGUINETTI JUAN.,** Verilog VHDL vs., p1.
http://www.angelfire.com/in/rajesh52/verilogvhdl.html
15/12/2013

[31]. **OLIVO GREDIAGA ÁNGEL Y OTROS.,** Diseño de Procesadores con VHDL., 1ra.ed., Alicante-España., Textos Docentes., 2006., Pp49.

ANEXOS

ANEXO 1



ANEXO 2

ISE DESIGN SUITE 14.2 DISEÑO DE SOFTWARE

ISE DESIGN SUITE 14.2 es un software de diseño (CAD) y una solución importante para la industria, diseñado para Linux, Windows XP y Windows 7. ISE DESIGN SUITE 14.2 es la solución ideal para programar FPGA y CPLD pudiendo realizar la síntesis de los diseños de HDL y la simulación, implementación, instalación de dispositivos, ISE DESIGN SUITE 14.2 ofrece un completo flujo de diseño proporcionando acceso instantáneo a las características y funcionalidad ISE. Xilinx ha creado una solución que permite la productividad conveniente, proporcionando una solución de diseño.

3.2.1 CARACTERÍSTICAS DEL ISE DESIGN SUITE 14.2

- Un ambiente libre, descargable PDL diseñado para Microsoft Windows y Linux
- Cierre de sincronización más rápida de la industria con la tecnología Xilinx.
- Verificación HDL Integrado con la versión Lite del ISE Simulator (ISIM)
- La forma más fácil, más bajo costo para empezar a trabajar con el líder de la industria para la productividad, el rendimiento y el poder
- Fácilmente capaz de actualizar ninguna de las ediciones ISE Design Suite desde el Xilinx Tienda Online

3.2.2 PROCEDIMIENTO DE DISEÑO EN ISE WEBPACK

Con la herramienta ISE DESIGN SUITE 14.2, estas son las etapas que deben seguirse para diseñar un sistema digital lógico:

a) Definición del sistema lógico, que en función de su complejidad se subdividirá en subsistemas formando una estructura jerárquica. Cada uno de estos subsistemas se definirá de alguna de las siguientes maneras:

- ❖ Editando gráficamente su esquema, esto es, dibujando su esquema a partir de bibliotecas de elementos básicos y símbolos que representan otros subdiseños.
- ❖ Editando en forma de texto o escribiendo su descripción en un lenguaje de descripción de hardware como VHDL y Verilog.

b) Compilación: Después de definir el circuito, éste debe compilarse. Se detectan errores introducidos en la definición. Tras corregirlos se crea un fichero que contiene la lógica del sistema y que permitirá simular su funcionamiento. Este es el momento de elegir una PLD concreta en la que programar el sistema completo. El compilador necesitará esta información para realizar simplificaciones y minimizar la lógica del sistema adecuándola de la mejor manera a la PLD.

c) Definición de entradas todo sistema lógico procesa señales externas y genera resultados. Por ello, antes de simular el funcionamiento del mismo, hay que definir sus entradas, es decir los valores aplicados a todas las entradas del sistema durante el tiempo que dure la simulación.

d) Simulación lógica: A continuación, ya puede realizarse la simulación lógica del sistema, verificando su comportamiento. Se pueden realizar dos tipos de simulación: simulación funcional o lógica y simulación 82 temporal o física. En la primera no se tienen en cuenta los tiempos de respuesta del circuito ni los retardos. En la segunda, en cambio, sí.

e) Implementación en PLD: Una vez comprobado que el sistema funciona como se desea, puede programarse en un dispositivo PLD, si ese es el objetivo. Después, en el laboratorio, se realizarán las fases de montaje y prueba.

f) El programa funciona mediante menús. A continuación se va a describir de manera muy breve las pautas a seguir para diseñar con esta herramienta.

Todo diseño digital tiene un diagrama de flujo típico que se muestra en la figura:

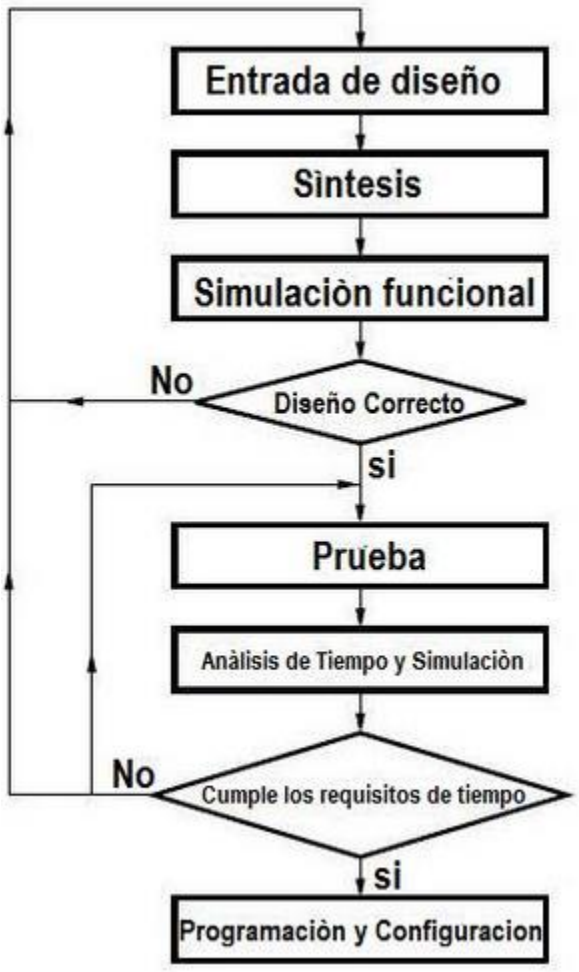


DIAGRAMA DE FLUJO TIPICO CAD

Fuente: *Los Autores*

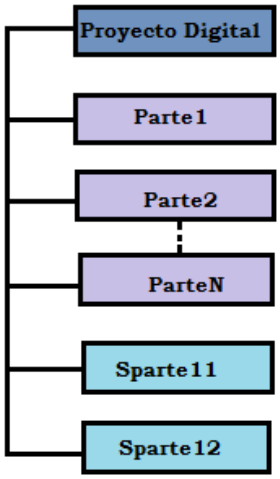
3.2.3 PASOS PARA EL DISEÑO DE DIGITAL

Para realizar un diseño digital en VHDL o en Verilog se usan los mismo pasos la única diferencia será al rato de escoger el tipo de lenguaje que se va a utilizar, se explicara cuando llegue el momento.

1. **Directorio para el diseño digital**

   Antes de abrir el programa ISE DESIGN SUITE 14.2, primero creamos una carpeta en algún lugar del directorio donde queramos almacenar lo que vamos a diseñar. Es preferible usar el mismo nombre en la carpeta y el proyecto que vamos a crear, para poder identificar más rápido en el caso que necesitemos buscarlo en nuestra pc. Cuando empezamos a definir nuestro diseño, lo realizaremos en forma jerárquica, pero en definición grafica o textual seguiremos la sintaxis del Bottomup (esto es comenzando el diseño de los submodulos desde el nivel más bajo hasta completar el diseño completo, siendo el diseño más alto de la jerarquía un módulo completo del diseño), suponiendo que el proyecto que vamos a crear se llama **Proyecto Digital** y dentro de este existan muchas partes como: parte1, parte2, etc., y dentro de este se encuentren varios submodulos sparte11 y sparte12 ahí estamos definiendo los submodulos de un proyecto total.

Donde la estructura de directorios a crear seria como se muestra en la figura:

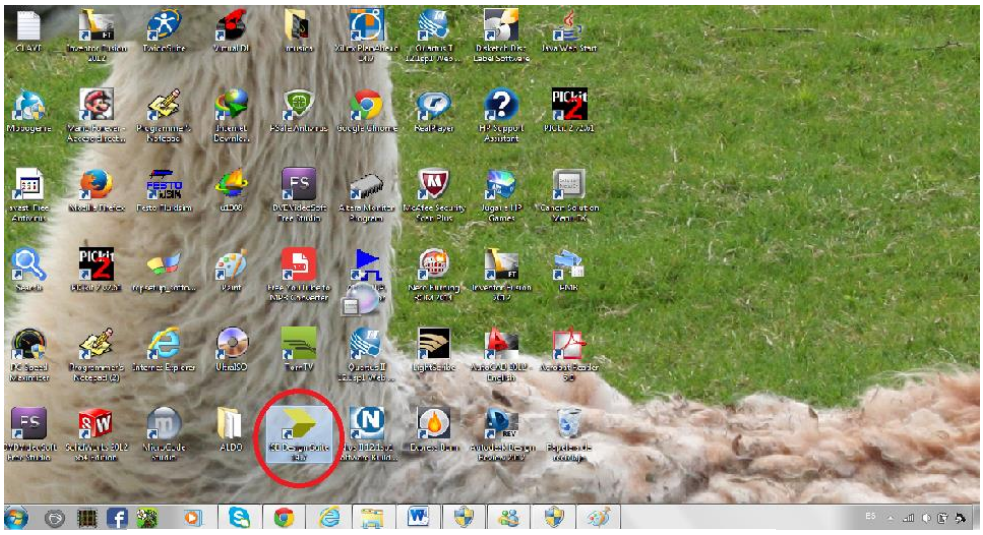


ESTRUCTURA DE DIRECTORIOS EN ISE DESIGN SUITE 14.2

Fuente: *Los Autores*

2. Abrir ISE Desing Suite 14.2 de Xilinx y crear un proyecto de diseño.

Damos doble click sobre el icono del programa que se puede observar en el grafico encerrado en un círculo rojo para poder ingresar al programa.



ISE DESIGN SUITE 14.2

Fuente: *Los Autores*

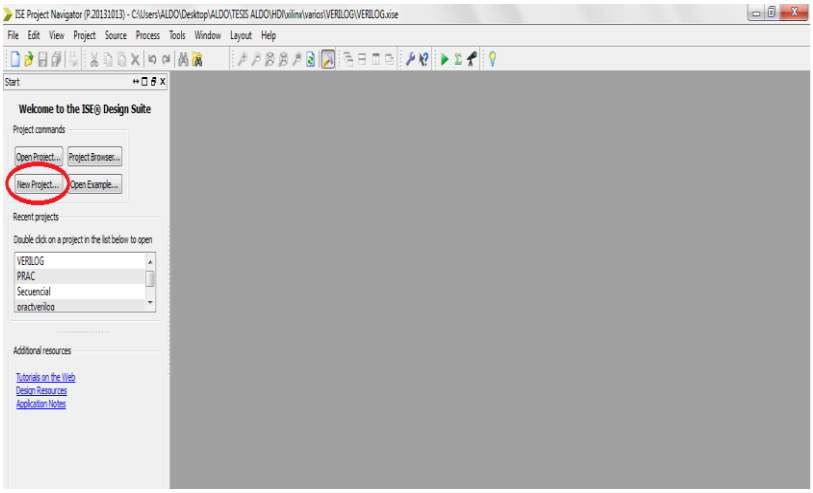
Aparecerá la siguiente pantalla de presentación del programa:

Abriendo ISE

Fuente: *Los Autores*

Luego nos aparecerá la pantalla donde podemos empezar a crear un nuevo proyecto, abrir nuevo proyecto o abrir algún ejemplo entre otras opciones.

Como vamos a crear un proyecto nuevo escogemos la opción (NEW PROJECT).

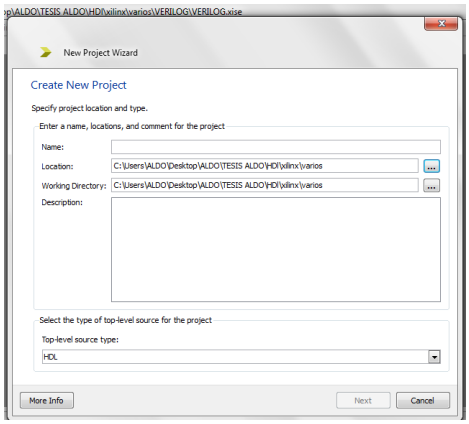


Crear Nuevo Proyecto

Fuente: *Los Autores*

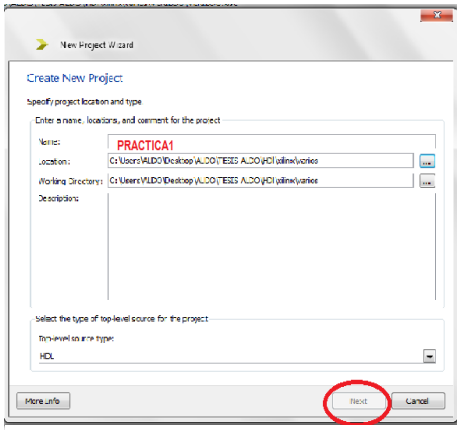
Luego nos aparecerá una nueva pantalla llamada “NEW PROJECT WIZAR”

En esta pantalla le daremos un nombre a nuestro proyecto y la ubicación en la que se va a guardar. Es importante crear una carpeta exclusiva para guardar nuestros proyectos a realizar. Para nuestro ejemplo le llamaremos "**PRACTICA1**" y luego presionamos "**NEXT**".



Agregar Dirección ISE

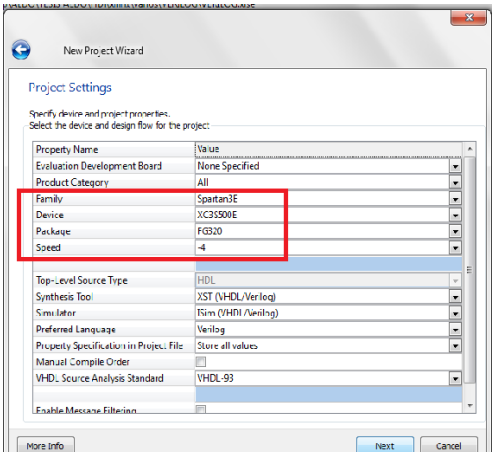
Fuente: *Los Autores*



Nombre del Proyecto

Fuente: *Los Autores*

Luego nos aparecerá la siguiente pantalla donde se deberá configurar el dispositivo FPGA a utilizar. Se utilizara la FPGA de marca Xilinx Spartan 3E, donde siempre debemos de configurar lo siguiente:



FAMILY: SPARTAN3E

DEVICE: XC3S5OOE

PACKAGE: FG320

SPEDD: -4

Configuración de FPGA

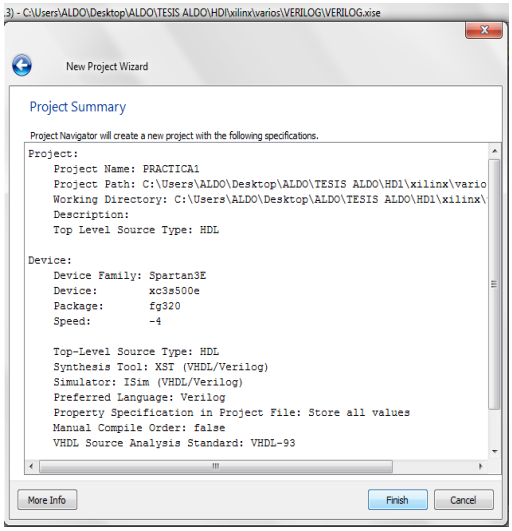
Fuente: *Los Autores*

Estos datos mencionados son los que se deben de configurar para una tarjeta spartan 3E pero en caso de tener otra tarjeta de diferente característica se debe de revisar el manual de la tarjeta FPGA que vayas a utilizar.

El resto de opciones que presenta esta pantalla de configuración quedan igual como nos da el programa.

Presionamos la opción "NEXT" para continuar a la siguiente pantalla.

Esta pantalla nos presentara un resumen de los datos ya configurados anteriormente de nuestra tarjeta por la cual podríamos presionar la opción "**FINISH"** para continuar en la creación de nuestro primer proyecto.

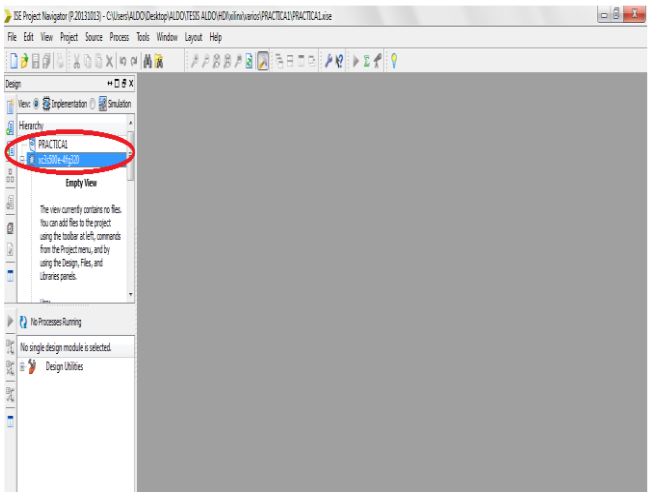


Resumen de Características

Fuente: *Los Autores*

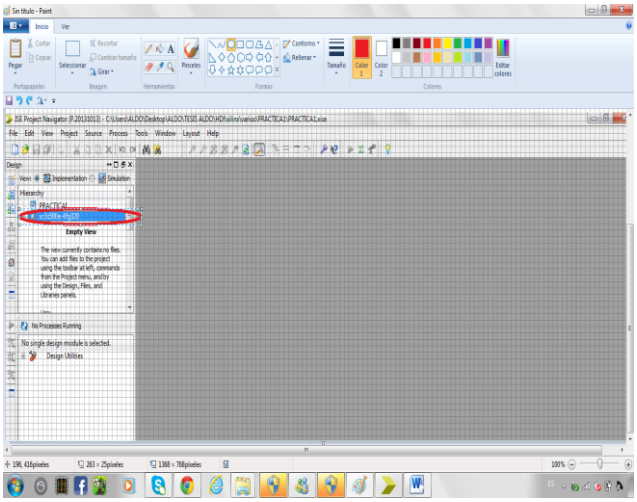
En esta pantalla podemos observar que ya nos aparece el nombre que le dimos al proyecto y la tarjeta a utilizar con sus respectivos datos ya configurados (imagen izquierda).

El siguiente paso a realizar es colocar el puntero en donde se encuentra el nombre de la tarjeta con sus datos y hacer click derecho con el mouse (ratón) (imagen derecha).



Tarjeta FPGA

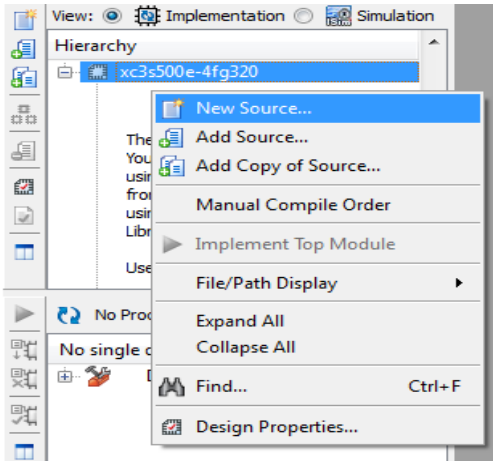
Fuente: *Los Autores*



Selección para Nuevo Modulo

Fuente: *Los Autores*

Nos aparece la siguiente pantalla donde escogemos la opción "**NEW SOURCE**"

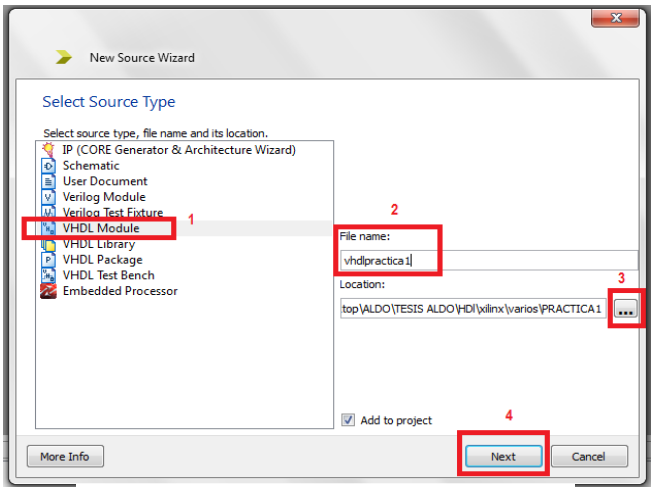


Crear nueva Fuente

Fuente: *Los Autores*

Si se va a realizar una programación en VHDL (1) Para este ejercicio le dimos el nombre (2) de **“vhdlpractica1”.** La localización del archivo ya nos da el programa (3)

(4) presionamos NEXT para seguir a la siguiente pantalla.

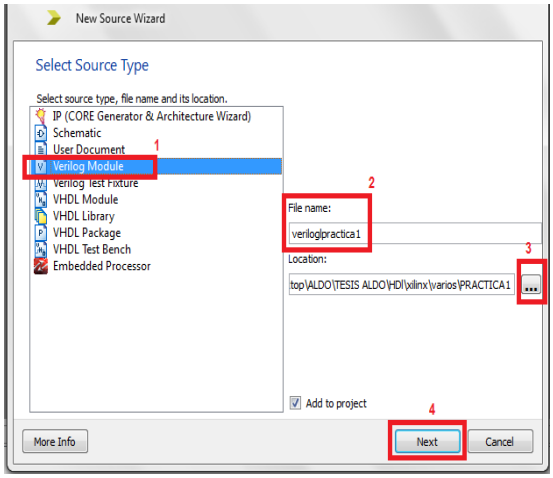


Elegir fuente a programar

Fuente: *Los Autores*

Si se va a realizar una programación en VERILOG (1).Para este ejercicio le dimos el nombre (2) de **“verilogpractica1”.** La localización del archivo ya nos da el programa (3)

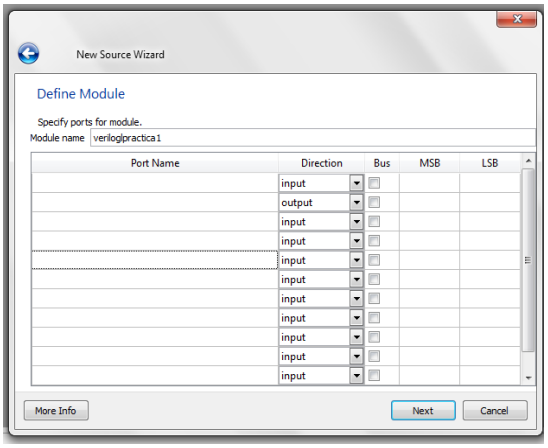
(4) presionamos NEXT para seguir a la siguiente pantalla.



Nombre de la Fuente

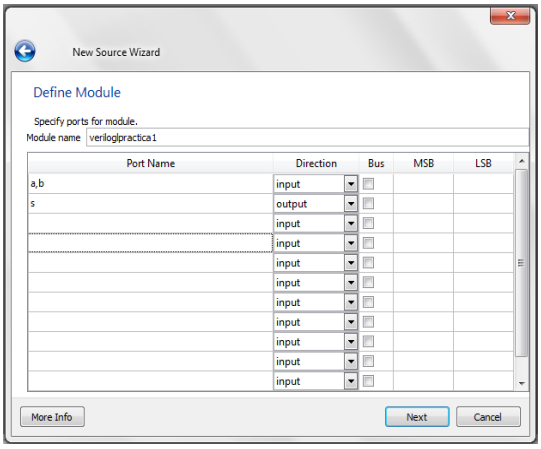
Fuente: *Los Autores*

En esta pantalla podremos configurara todos los datos de entrada como de salida de acuerdo al ejercicio a resolver. Para nuestra primera practica realizaremos la configuración de una compuerta **Or** donde tendremos como datos de entrada "**INPUT"** (a,b) y de salida "**OUTPUT**" (r).(imagen derecha)



Ventana de Entradas y Salidas

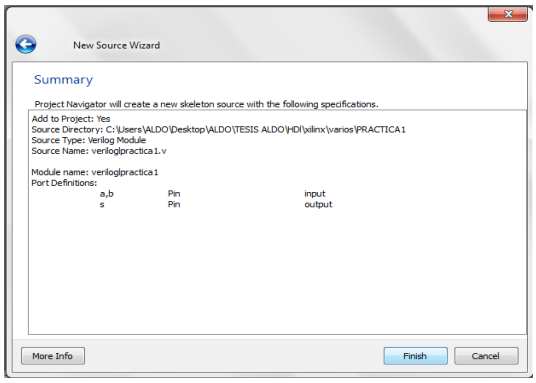
Fuente: *Los Autores*



Definiendo In y Out

Fuente: *Los Autores*

Ya ingresados los datos de entrada como de salida presionamos en la opción "**NEXT"** para continuar a la siguiente pantalla. Esta pantalla nos mostrara un resumen de los datos ingresados de las entradas y salidas de datos configurados en el paso anterior. Presionamos la opción "**FINISH**" Para continuar a la siguiente pantalla.

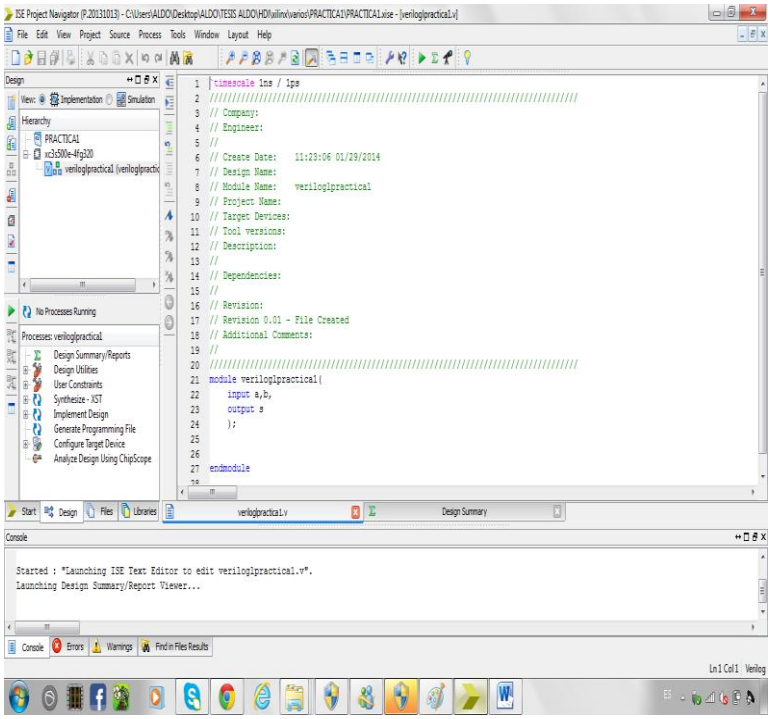


Cuadro de resumen de la Fuente

Fuente: *Los Autores*

En esta pantalla ya podemos empezar a programar. Las líneas que están de color verde y la del (`timescale 1ns / 1ps) se pueden borrar.

El ISE DESIGN SUITE 14.2 ya me da escribiendo la codificación de la entrada y salida de datos declarados anteriormente la cual solo deberíamos de centrarnos en escribir el proceso que se desee que haga nuestro diseño.



Ventana de Programación de la Fuente

Fuente: *Los Autores*

Como vamos a realizar la configuración de una compuerta **OR** deberíamos de tener el siguiente código para cada lenguaje:

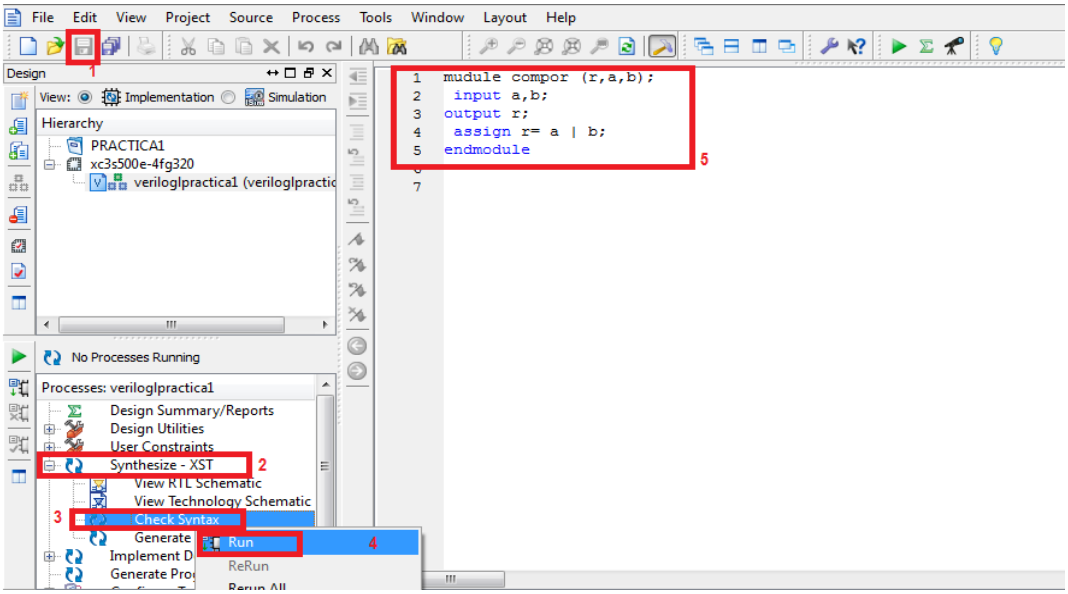
PROGRAMACION COMPUERTA OR

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity inicio2 is
   Port ( a,b: in  STD_LOGIC;
          s : out  STD_LOGIC);
end inicio2;
architecture Behavioral of inicio2 is
begin
       s<= a or b;
end  Behavioral;
```

VERILOG

```
modulo compor (r,a,b);
        input a,b;
        output r;
        assign r= a | b;
endmodule
```



Programación de la Compuerta OR

Fuente: *Los Autores*

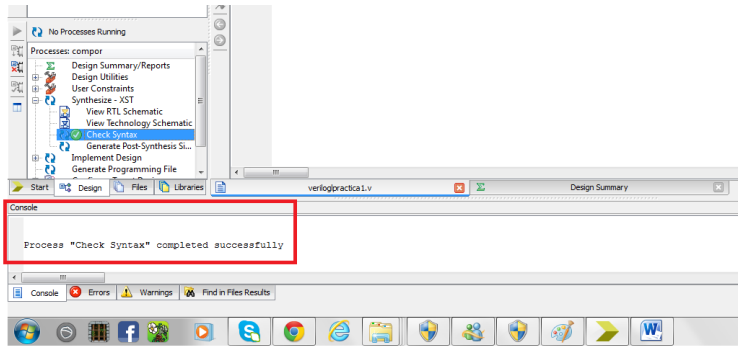
Después de haber hecho la respectiva programación o codificación (5) de nuestro diseño debemos de proceder a chequear que se encuentre correctamente el código escrito para ello debemos de seguir los siguientes cuatro pasos:

1. Guardamos los cambios efectuados (ctrl + s)
2. Nos vamos a la opción de "**SYNTHESIZE – XTS"**

3. Nos vamos a la opción de **“CHECK SYNTAX” y** damos un click derecho.
4. Nos vamos a la opción de “**RUN”**

Después de haber hecho estos pasos debemos de esperar un tiempo hasta que el programa terminar de compilar y chequear que la sintaxis del programa este bien escrito y si todo está bien nos aparecerá el siguiente mensaje:

“**PROCESS “CHECK SYNTAX” completed successfully**. La cual me indica que todo está sin ningún error.

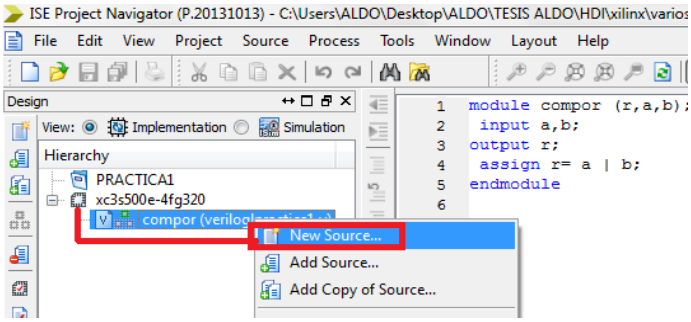


Compilación del Programa

Fuente: *Los Autores*

Una vez comprobada la sintaxis el código se encuentra listo para poder realizar la **SIMULACIÓN** del ejercicio para ello debemos de seguir los siguientes pasos:

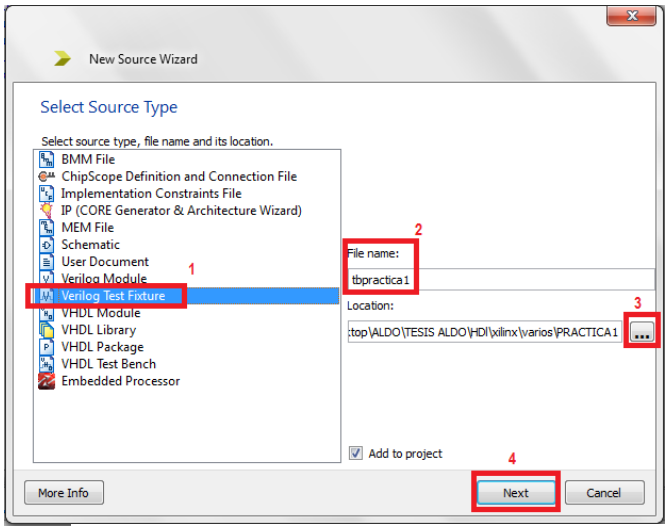
Si recordamos cuando iniciamos a realizar las configuraciones habían solo 2 iconos (practica1, xc3s500…) ahora tenemos un tercer icono con el nombre que ya le habíamos dado de (verilogpractica1) sobre ese icono vamos a dar “**click derecho**” en la opción “**NEW SOURCE (**nueva fuente**)**” la cual me llevara a la siguiente ventana de configuración:



Nueva Fuente para el Testbench

Fuente: *Los Autores*

Como estamos programando en **VERILOG** la opción (1) debe de escogerse "**VERILOG TEST FIXTURE**". E n el paso (2) le dimos el nombre "**tbpractica1**" para reconocer que es el testbench del ejercicio. La opción (3) ya me da el programa y me dirijo a la opción (4) "**NEXT**" para ir a la siguiente configuración para poder realizar la respectiva simulación.

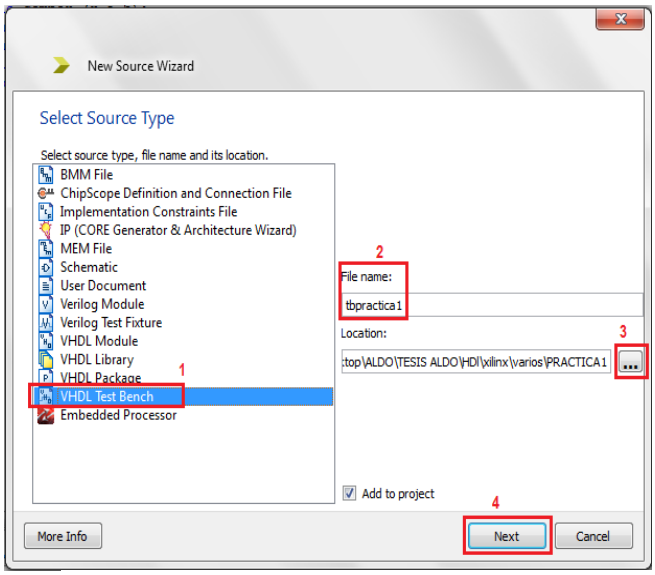


Testbench en Verilog

Fuente: *Los Autores*

Si se estuviere programando en "**VHDL"** se escogería las siguientes opciones de configuración: (1) debe de escogerse "**VHDL TESTBENCH**". E n el paso (2) le dimos el nombre "**tbpractica1**" para reconocer que es el testbench del ejercicio. (3) me da el

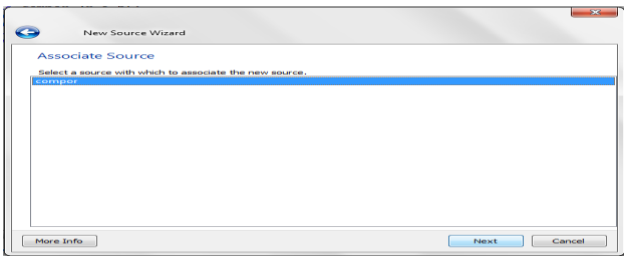
programa y me dirijo a la opción (4) "**NEXT**" para ir a la siguiente configuración para poder realizar la respectiva simulación.



Testbench en VHDL

Fuente: *Los Autores*

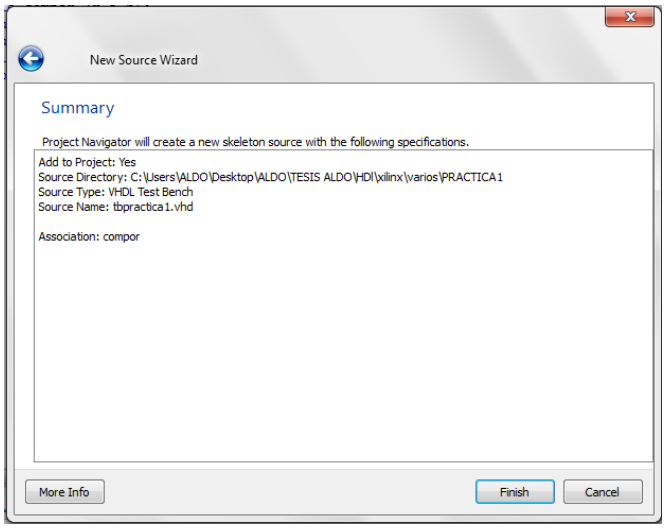
Nos aparecerá un cuadro de con las opciones a vincular nuestro testbench escogemos la opción a la que le vamos a vincular, luego damos en "**NEXT**" para seguir a la siguiente pantalla.



Elección de la Fuente para el Testbench

Fuente: *Los Autores*

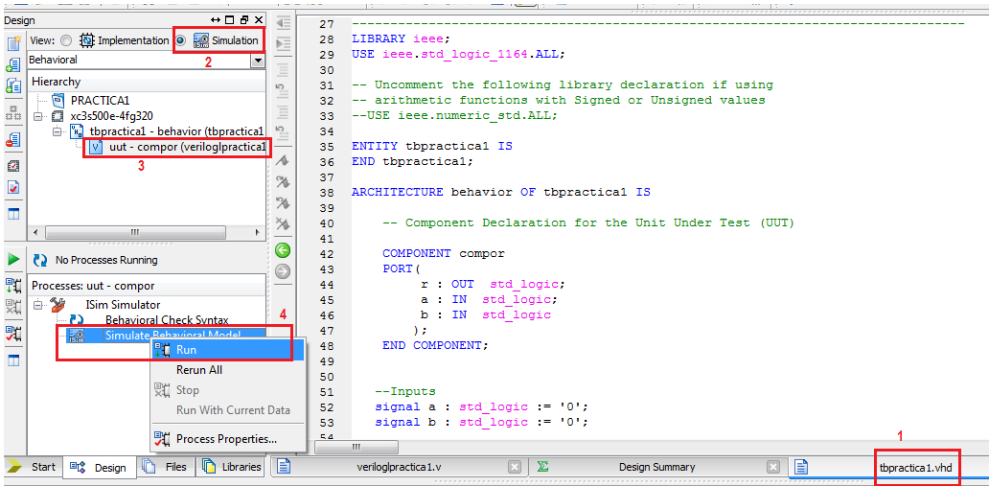
Nos aparece un cuadro de resumen. Escogemos la opción "**FINISH**" para seguir a la siguiente pantalla.



Descripción de Testbench

Fuente: *Los Autores*

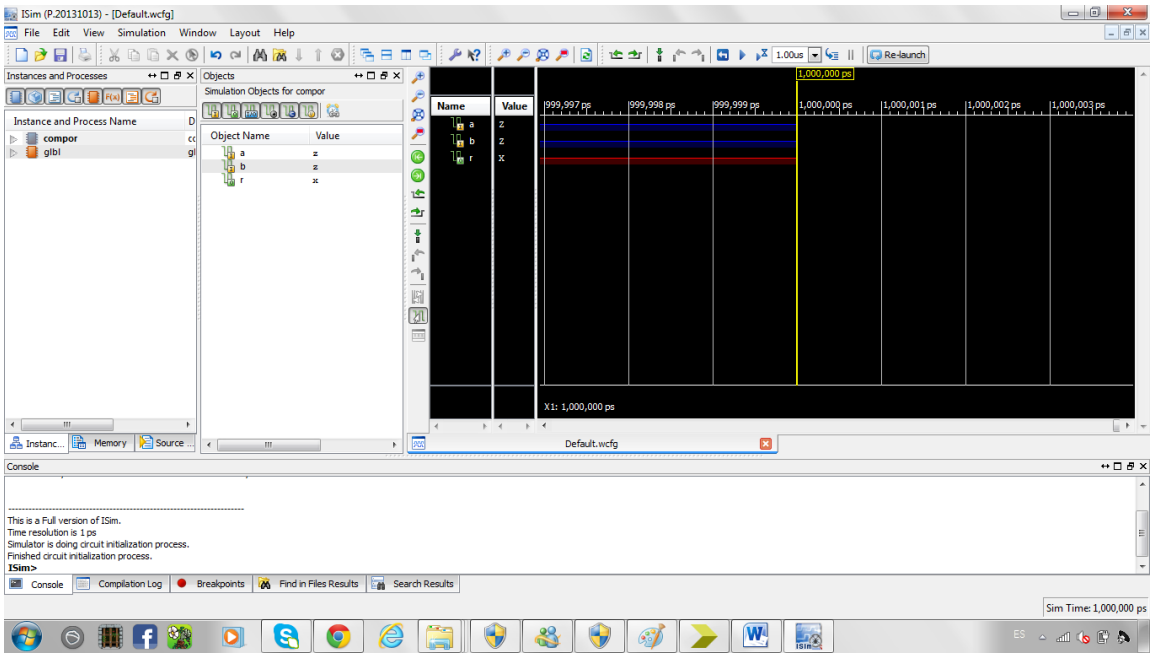
Una vez realizado los pasos ya mencionados y configurados anteriormente se creara una nueva ventana con el nombre "**tbpractica1**" (1) que si recordamos es el nombre que dimos para el testbench.



Archivo Testbench y Simulación

Fuente: *Los Autores*

Después escogemos la opción (2) "**SIMULATION**" se nos ha creado un cuarto icono (**uut – compor(veriloglpractica1**) el cual lo seleccionamos es la opción (3) después nos dirigimos y seleccionamos la opción (4) "**SIMULATION BEHAVIORAL MODEL**" le damos click derecho y escogemos la opción "**RUN**". Esperamos un tiempo hasta que se compile y nos aparecerá la siguiente pantalla:



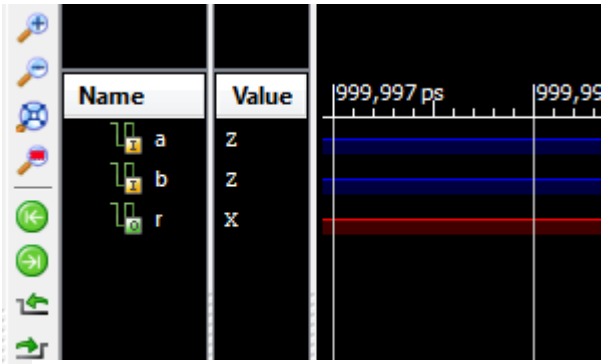
Pantalla ISIM

Fuente: *Los Autores*

En esta pantalla podremos realizar la respectiva simulación del ejercicio donde podremos realizar el ingreso y manipulación de datos para ver de qué manera se comporta en distintos instantes de tiempo.

Para poder empezar la simulación debemos de ingresar los datos recordemos que estamos haciendo el ejercicio para una compuerta "**OR"** la cual tenemos como (a,b) entrada de datos y (r) salida de datos. Debemos de realizar el siguiente paso para poder ingresar los respectivos valores:

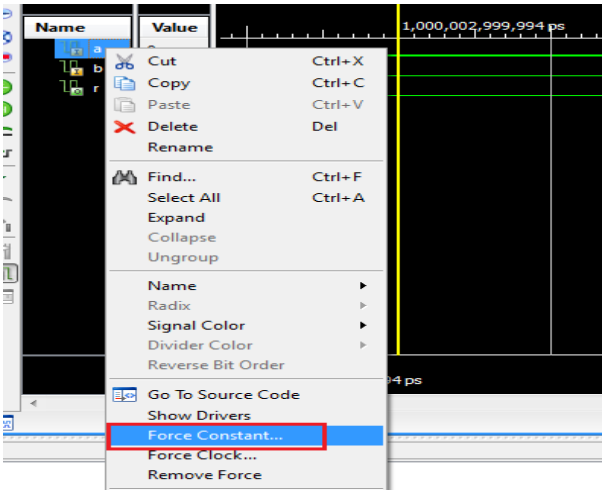
Aquí podemos observar las letras que fueron declaradas para el ingreso y salida de datos.



Entradas y Salidas en Simulador

Fuente: *Los Autores*

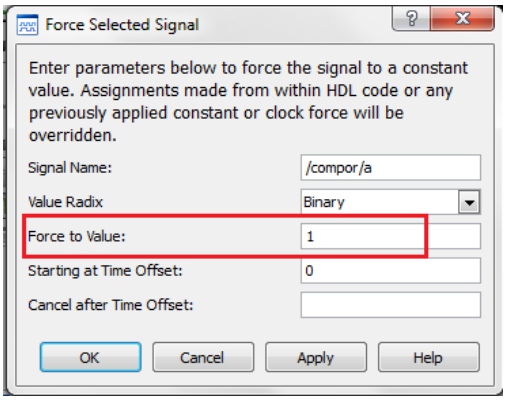
Debemos de colocarnos sobre las letras de entrada de datos dar click derecho y escogemos la opción "**FORCE CONSTANT.**"



Selección de Parámetros

Fuente: *Los Autores*

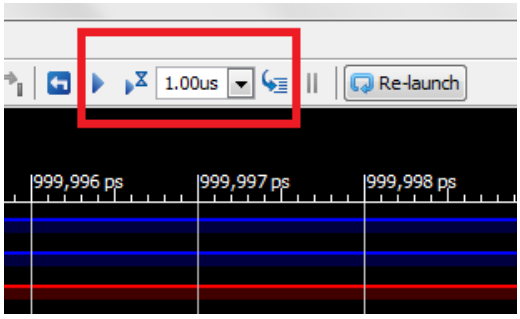
En esta pantalla podemos hacer el ingreso de dato deseado para este caso debe de ser un dato binario de 0 o 1.



Ingreso de Parámetros

Fuente: *Los Autores*

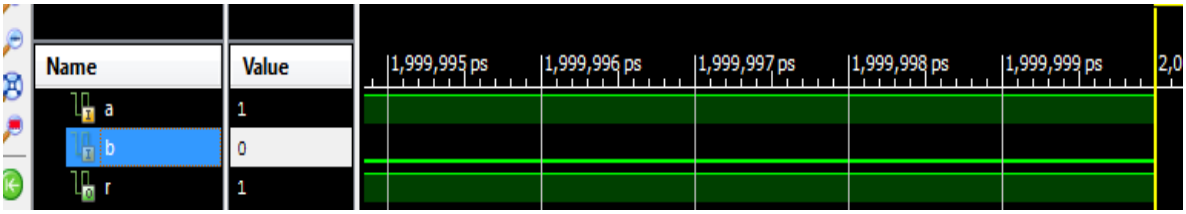
Presionamos esta opción para que nuestro programa empiece la simulación de los datos ingresados.



Velocidad de Ejecución

Fuente: *Los Autores*

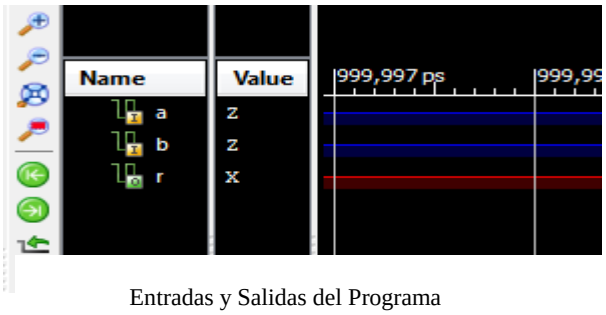
En esta imagen podemos observar que los datos ingresado (a=1) y (b=0) y su repuesta (r=1) lo que nos indicaría que nuestra repuesta es correcta si revisamos la tabla de verdad para una compuerta "OR".



Simulación General

Fuente: *Los Autores*

Si deseas enviar datos continuamente se debe de realizar la siguiente configuración:

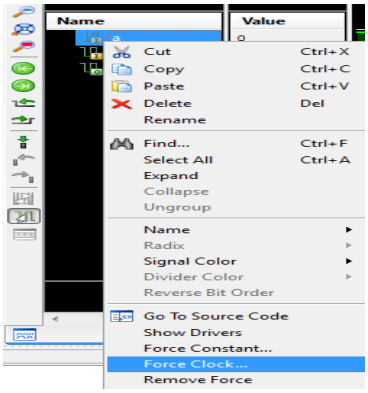


Entradas y Salidas del Programa

Fuente: *Los Autores*

Aquí podemos observar las letras que fueron declaradas para el ingreso y salida de datos.

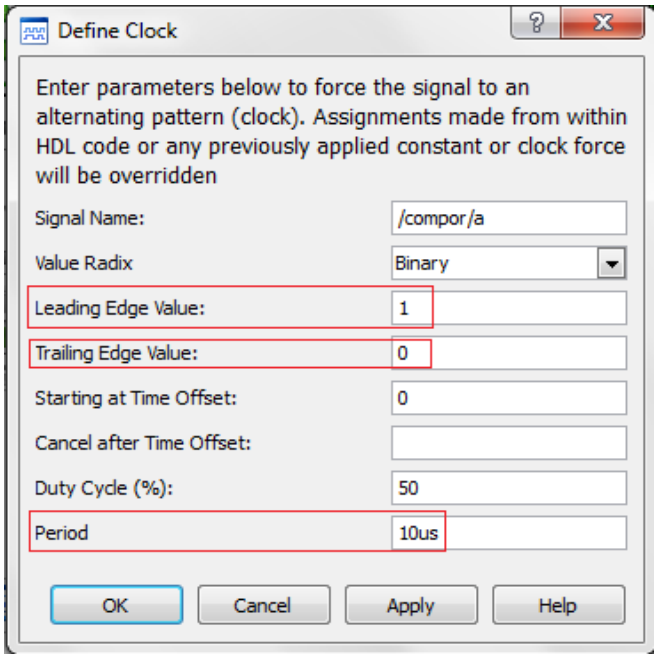
Debemos de colocarnos sobre las letras de entrada de datos dar click derecho y escogemos la opción "**FORCE CLOCK.**"



Parámetro Force Clock

Fuente: *Los Autores*

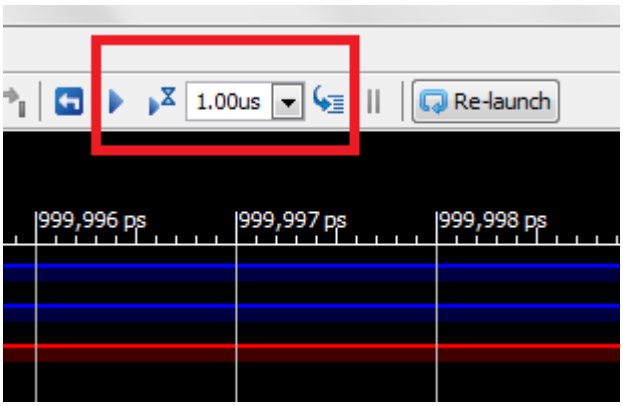
Realizamos el ingreso de datos en los recuadros que están marcados el cual los podemos ir variando para ver de qué manera se va comportando las gráficas. Y luego presionamos **"Ok"**.



Ingresando Parámetros en Force Clock

Fuente: *Los Autores*

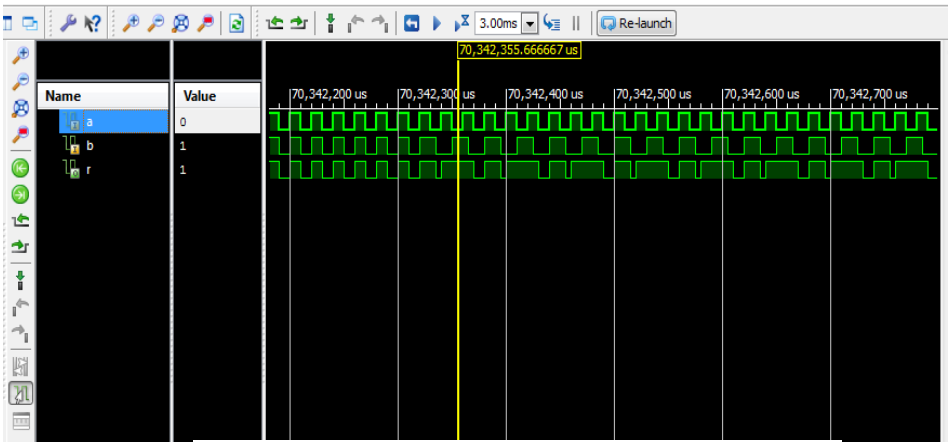
Presionamos esta opción para que nuestro programa empiece la simulación de los datos ingresados, podemos variar el tiempo a unos (3.00ms) para poder observar de mejor manera las gráficas.



Velocidad de Simulación

Fuente: *Los Autores*

Una vez enviado a simular tendremos la siguiente gráfica: Podremos ir observando los cambios que se van presentando.



Ondas de Entrada y Salida

Fuente: *Los Autores*

En este manual hemos realizado con la finalidad de tener idea de cómo crear y hasta como simular los programas de diseño digital, para estudiantes que quieran incursionar en el campo de las tarjetas FPGA y programando en HDL, y no tenga conocimiento de cómo crear los archivos para desarrollar sus diseños digitales.