Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería Mecánica Eléctrica

DESARROLLO DE APLICACIONES SINTETIZANDO HARDWARE EN FPGA UTILIZANDO EL LENGUAJE VHDL A NIVEL MEDIO

Haroldo José López de los Rios
Asesorado por el Ing. Byron Odilio Arrivillaga Méndez

Guatemala, febrero de 2020

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

DESARROLLO DE APLICACIONES SINTETIZANDO HARDWARE EN FPGA UTILIZANDO EL LENGUAJE VHDL A NIVEL MEDIO

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

HAROLDO JOSÉ LÓPEZ DE LOS RIOS
ASESORADO POR EL ING. BYRON ODILIO ARRIVILLAGA MÉNDEZ

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN ELECTRÓNICA

GUATEMALA, FEBRERO DE 2020

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANA	Inga. Aurelia Anabela Cordova Estrada
VOCAL I	Ing. José Francisco Gómez Rivera
VOCAL II	Ing. Mario Renato Escobedo Martínez
VOCAL III	Ing. José Milton de León Bran
VOCAL IV	Br. Christian Moisés de la Cruz Leal
VOCAL V	Br. Kevin Armando Cruz Lorente
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Pedro Antonio Aguilar Polanco
EXAMINADOR	Ing. Byron Odilio Arrivillaga Méndez
EXAMINADORA	Inga. María Magdalena Puente Romero
EXAMINADOR	Ing. Carlos Eduardo Guzmán Salazar
SECRETARIA	Inga. Lesbia Magalí Herrera López

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

DESARROLLO DE APLICACIONES SINTETIZANDO HARDWARE EN FPGA UTILIZANDO EL LENGUAJE VHDL A NIVEL MEDIO

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería Mecánica Eléctrica, con fecha de enero de 2017.

Haroldo José López de los Rios

Guatemala 19 de marzo, 2019

Ingeniero Otto Fernando Andrino González
Director de Escuela de Ingeniería Mecánica Eléctrica
Facultad de Ingeniería
Universidad de San Carlos de Guatemala.

Por medio de la presente me permito informarle que he procedido a revisar el trabajo de graduación titulado **"DESARROLLO DE APLICACIONES SINTETIZANDO HARDWARE EN FPGA UTILIZANDO EL LENGUAJE VHDL A NIVEL MEDIO"** elaborado por el estudiante Haroldo José López de los Ríos quien se identifica con el número de DPI 2244 59732 0101 y carnet 201114661. A su vez, quiero mencionar que el mismo cumple los objetivos trazados de acuerdo con el protocolo presentado, por lo que le doy por APROBADA. De tal manera, se solicita darle trámite correspondiente.

Byron Arrivillaga Méndez
Ingeniero Electrónico
Colegiado 5217

Atentamente
Ing. Byron Odilio Arrivillaga Méndez
Colegiado No. 5217

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERIA

Guatemala, 14 de octubre de 2019

Señor Director
Armando Alonso Rivera Carrillo
Escuela de Ingeniería Mecánica Eléctrica
Facultad de Ingeniería, USAC

Estimado Señor Director:

Por este medio me permito dar aprobación al Trabajo de Graduación titulado **DESARROLLO DE APLICACIONES SINTETIZANDO HARDWARE EN FPGA UTILIZANDO EL LENGUAJE VHDL A NIVEL MEDIO,** desarrollado por el estudiante **Haroldo José López de los Ríos**, ya que considero que cumple con los requisitos establecidos.

Sin otro particular, aprovecho la oportunidad para saludarlo.

Atentamente,

ID Y ENSEÑAD A TODOS

Ing. Julio César Solares Peñate
Coordinador de Electrónica

FACULTAD DE INGENIERIA · UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCION ESCUELA DE INGENIERIA MECANICA ELECTRICA

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÍA

REF. EIME 71. 2019.

El Director de la Escuela de Ingeniería Mecánica Eléctrica, después de conocer el dictamen del Asesor, con el Visto bueno del Coordinador de Área, al trabajo de Graduación del estudiante: HAROLDO JOSÉ LÓPEZ DE LOS RÍOS titulado: DESARROLLO DE APLICACIONES SINTETIZANDO HARDWARE EN FPGA UTILIZANDO EL LENGUAJE VHDL A NIVEL MEDIO, procede a la autorización del mismo.

Ing. Armando Alonso Rivera Carrillo

FACULTAD DE INGENIERÍA · UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCION ESCUELA DE INGENIERIA MECANICA ELECTRICA

GUATEMALA, 31 DE OCTUBRE 2019.

Universidad de San Carlos
De Guatemala

Facultad de Ingeniería
Decanato

Ref. DTG.079-2020

La Decana de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería Mecánica Eléctrica, al trabajo de graduación titulado: **DESARROLLO DE APLICACIONES SINTETIZANDO HARDWARE EN FPGA UTILIZANDO EL LENGUAJE VHDL A NIVEL MEDIO,** presentado por el estudiante universitario: **Haroldo José López de los Rios,** y después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, se autoriza la impresión del mismo.

IMPRÍMASE.

Inga. Aurelia Anabela Cordova Estrada
Decana

Guatemala, febrero de 2020

AACE/asga

ACTO QUE DEDICO A:

Dios	Por darme la oportunidad de estudiar esta carrera y ser una importante influencia en mi vida.
Mis padres	Haroldo López y Vania de López. Por todos sus esfuerzos, consejos y apoyo para que pudiera llegar a cumplir esta meta en mi vida.
Mis hermanas	Vania López y Paola López por estar siempre apoyándome y ayudándome a lo largo de mi vida.
Mi novia	Linda Estrada. Por todo su amor y apoyo en la realización del presente trabajo de graduación.
Mis amigos de la carrera	Gabriel Cabrera, Juan de Dios Mérida y Esther Pineda. Por todos los momentos que pasamos durante la carrera y apoyarme.
Mi primo	Juan Diego Quevedo. Siempre estarás presente en todo lo que logre.

Mis abuelos	Alma vda. de López, Haroldo López (q. e. p. d.), Cesar de los Ríos y Clara de de los Ríos, por su apoyo, cariño, sabio consejo e interés por mi futuro.
Mis padrinos	Alma de Guzmán y Braulio Guzmán, por su apoyo incondicional y su ejemplo de vida personal y profesional.
Tíos y primos	Personas incondicionales con las que siempre he contado, con mucho cariño.

AGRADECIMIENTOS A:

Dios	Porque siempre ha estado junto a mí y es quien me fortalece y me da la sabiduría para alcanzar mis metas.
Universidad de San Carlos de Guatemala	Casa de estudios que me brindó la oportunidad de realizar esta meta profesional.
Mis padres	Haroldo López y Vania de López. Por todos sus consejos y apoyo en conocimientos para realizar este trabajo.
Mis amigos de la carrera	Gabriel Cabrera, Juan Mérida y Esther Pineda, por todos los momentos que sufrimos y logramos superar para alcanzar nuestra meta.
Ing. Byron Arrivillaga	Por asesorarme durante todo el proceso que ha conllevado la realización del presente trabajo, compartir sus conocimientos y motivarme a seguir adelante.
Linda Estrada	Por brindarme su amor, apoyarme con sus conocimientos en distintos temas de ingeniería y motivarme a concluir este trabajo de graduación.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

Figuras

TABLAS

LISTA DE SÍMBOLOS

Símbolo	Significado
A	Amperios
$\boldsymbol{W}$	Ancho de banda
C	Capacitancia
Hz	Ciclo por segundo hercio
I	Corriente
$\boldsymbol{x_N}$	Enésima entrada muestreada del filtro digital
$\boldsymbol{y_N}$	Enésima salida muestreada del filtro digital
$\boldsymbol{h_N}$	Enésimo coeficiente del filtro digital
F	Faradios
$\boldsymbol{f_s}$	Frecuencia de muestreo
f	Frecuencia
$\boldsymbol{w_o}$	Frecuencia fundamental
kHz	Kilohercio
k	kilo
MHz	Megahercio
ns	Nano segundos
Ω	Ohm
$\boldsymbol{T_s}$	Período de muestreo
s	Segundos
$\boldsymbol{X(f)}$	Señal en el dominio de la frecuencia
<=	Símbolo de asignación de señales
t	Tiempo
$\boldsymbol{V_{dd}}$	Voltaje de alimentación

Vcco	Voltaje de corriente continua
$\boldsymbol{V_{in}}$	Voltaje de entrada
$\boldsymbol{V_{ref}}$	Voltaje de referencia
-V	Voltaje negativo
+V	Voltaje positivo
V	Voltios

GLOSARIO

AND	Compuerta lógica que implementa el y.
Assembler	Lenguaje de bajo nivel para programación de microprocesadores.
Bank	Banco de memoria.
Carry	Acarreo generado en operaciones de suma.
Clock skew	Indica que existe un sesgo en el reloj.
Data-	Dato de entrada negativo.
Data+	Dato de entrada positivo**.**
DCM	Lógica implementada en un bloque.
Ethernet	Estándar de redes para comunicación entre dispositivos.
Flags	Instrucción que valida estados en un microprocesador.
Flip Flops	Circuito secuencial multivibrador capaz de almacenar estados.

FPGA	Dispositivo compuesto por compuertas programables.
Hardware	Conjunto de elementos físicos que componen un circuito.
Logic	Lógica implementada en un bloque.
Look up tables	Tablas de búsqueda, donde se indican entradas y salidas.
Máster	Módulo maestro encargado de la sincronización.
MUX	Circuito combinacional con varias entradas y una única salida, implementando una función de multiplexor.
NAND	Compuerta lógica que implementa la negación de y.
Neumónico	Palabra que sustituye instrucciones binarias de lenguaje de máquina.
NOR	Compuerta lógica que implementa la negación de o.
NOT	Compuerta lógica que implementa la negación.
OR	Compuerta lógica que implementa el o.

RAM	Memoria de acceso aleatorio.
Rx	Receptor**.**
Sample	Muestra tomada de una señal.
Slave	Módulo esclavo.
Software	Conjunto de programas que definen comportamientos.
String	Tipo de dato que define cadena de caracteres.
TFilter	Aplicación web para el cálculo de coeficientes para un filtro digital.
TOP	Módulo general que incluye otros módulos más pequeños.
Tx	Transmisor**.**
UART	Comunicación serial asíncrona para transmisión de datos.
VHDL	Lenguaje de descripción de hardware.
XOR	Compuerta lógica que implementa el o exclusivo.

RESUMEN

El diseño de circuitos digitales en la actualidad tiene muchas aplicaciones debido a que existe la necesidad de digitalizar todos los procesos existentes. En las aplicaciones reales algunas veces se requieren diseños de hardware complejos y contratar a una empresa para que fabrique un circuito integrado.Cumplir esta función es muy costoso, por lo tanto, la utilización de FPGA reduce los costos y entrega una muy buena eficiencia. En el trabajo de tesis *Desarrollo de aplicaciones sintetizando hardware en FPGA utilizando el lenguaje VHDL a nivel medio* se expone la manera de diseñar circuitos utilizando el lenguaje de descripción de hardware VHDL y así explicar de una mejor manera cómo funciona este dispositivo y algunas aplicaciones en las cuales se puede emplear.

A manera de dar una mejor explicación de la tecnología utilizada con FPGA se expone la forma de utilizar el lenguaje de descripción de hardware VHDL explicando primeramente la definición del lenguaje y todos los componentes necesarios para construir diseños digitales.

Se hace una descripción de cómo es internamente el dispositivo FPGA para que se comprenda de una mejor manera cada uno de los bloques que comprende este dispositivo. Después de comprender todos los conceptos de la descripción de hardware se describe la implementación de aplicaciones en las cuales es ideal la utilización de FPGA.

OBJETIVOS

General

Describir e implementar diseños de hardware para crear material específico y disponible como apoyo al laboratorio de electrónica.

Específicos

1. Tomando como base la teoría de electrónica digital del pensum de la carrera ingeniería electrónica, comprender la síntesis de circuitos digitales y sus aplicaciones.

2. Crear destrezas para la descripción de hardware y de circuitos para tratamiento analógico.

3. Exponer los conocimientos necesarios para la implementación hardware en aplicaciones competitivas en el mercado de la electrónica digital.

4. Promover en la carrera de electrónica la tecnología de síntesis de hardware utilizada en diversos países del mundo.

INTRODUCCIÓN

La evolución del diseño de circuitos digitales ha llevado a la creación de nuevos dispositivos semiconductores en base al mismo principio. La síntesis de hardware es ejemplo de esta evolución, tomando simples compuertas digitales y lógica secuencial se interconectan de forma matricial para la implementación de un hardware. Actualmente en la carrera de ingeniería electrónica se ha promovido esta tecnología que cada vez es más utilizada en dispositivos sofisticados en diferentes ámbitos de la electrónica digital.

El dispositivo principal para implementar los diseños descritos es la denominada FPGA, la cual tiene la capacidad de ejecutar tareas en paralelo a diferencia de otros dispositivos como los microprocesadores que tienen una ejecución secuencial, es decir, realiza una instrucción a la vez. Este procesamiento en paralelo se debe a que este dispositivo permite la descripción de hardware físico el cual se implementa y se comporta tal y como si se hubiese creado el hardware digital.

Esta guía contiene una descripción detallada de las bases necesarias para que el estudiante pueda desarrollar aplicaciones más allá de lo que se ha estado proponiendo actualmente, ya que los estudiantes toman más tiempo en desarrollar elementos básicos, que en implementar aplicaciones reales en el área de la electrónica digital actual. Esta descripción de hardware se describe tanto para el tratamiento de señales analógicas como digitales.

1. LENGUAJE DE DESCRIPCIÓN DE HARDWARE VHDL

1.1. Definición del lenguaje VHDL y sus alternativas

VHDL es un lenguaje utilizado para describir circuitos digitales definido por los estándares de *Institute of Electrical and Electronics Engineers*. VHDL en su origen surgió del trabajo realizado la década de 1970 y principios de la de 1980 por el Departamento de Defensa de Estados Unidos. Este lenguaje se basa en otro llamado ADA. VHDL comparte su estructura general con ADA, así como otras instrucciones de VHDL. La utilización de VHDL se ha incrementado rápidamente desde su creación y es utilizado por decenas de miles de ingenieros de todo el mundo para crear productos electrónicos sofisticados.

VHDL es un lenguaje de gran alcance con numerosas implementaciones de hardware los cuales son capaces de describir un comportamiento muy complejo. Aprender todas las características de VHDL no es una tarea sencilla, pero se tratará de exponer de una manera más sencilla.

1.2. Generadores de VHDL en lenguajes convencionales

Usualmente los conceptos para el diseño de hardware son dificultosos en lenguaje como VHDL, es por esto, que se ha planteado una forma para comprender mejor la lógica de la descripción de hardware. Para una persona que está acostumbrada a los lenguajes convencionales es difícil adaptarse a la descripción de hardware con VHDL. Los lenguajes convencionales son tales como los modulares, orientados a objetos o ya sea funcionales y son ampliamente utilizados en el desarrollo de sistemas de software.

Como una manera de ir introduciendo a la implementación de hardware se sugiere hacer el cambio de manera gradual, de tal forma que utilizando la lógica de programación convencional se comprendan conceptos básicos e importantes de la lógica utilizada en descripción de hardware con VHDL. A estos conceptos es a los que se les llama generadores de VHDL en lenguajes convencionales.

A continuación, se presentan algunos de los generadores que son importantes para obtener la lógica de la descripción de hardware utilizando VHDL. Lo básico en la electrónica digital se refiere a la creación de circuitos compuestos por compuertas digitales. En VHDL es realmente muy simple describir este tipo de circuitos.

Tabla I. **Sintaxis de símbolos propuestos**

VHDL	Análogo propuesto
OR	|
AND	&
NOT	@
XOR	%
NAND	!
NOR	$
<=	:

Fuente: elaboración propia.

1.3. Elementos básicos del lenguaje VHDL

Como se describió anteriormente, VHDL es un lenguaje que se utiliza para describir hardware y con esto generar un circuito digital real, es por esto que no se trata de un lenguaje de programación convencional utilizado principalmente para programar algún controlado o procesador.

En VHDL existen elementos que dan la estructura a los diseños de circuitos digitales deseados, los cuales se describirán a continuación.

Entity: todos los diseños por implementar, en VHDL, se expresan en términos de entidades. Una entidad es el bloque de construcción básico de la mayoría de los diseños, es decir, es el nivel que incluye los componentes de hardware deseado. Es en el *entity* donde se define cómo va a ser el componente por implementar, es decir que el *Entity* es como una caja negra donde solo se describen las entradas y salidas del diseño. VHDL permite crear *entity* dentro de otro *entity*, creando así una jerarquía de *entity*s. Las descripciones de los niveles más bajos serán entidades de menor nivel contenida en la descripción entidad de nivel superior.

Architecture: todas las *Entitys* que pueden ser implementadas van a tener un comportamiento, este es llamado *architecture*, que describe el comportamiento de la entidad. Una sola entidad puede tener múltiples arquitecturas. Una arquitectura podría ser conductual como se describió anteriormente pero también puede ser estructural, por lo tanto, se crean más de una *architecture* una puede ser de comportamiento y la otra podría ser la estructura descripción de otro diseño.

Package: es una colección de tipos de datos y subprogramas utilizados en diseños de VHDL. Un paquete o *package* es como una caja que contiene las herramientas utilizadas para crear diseños.

Driver: es la fuente de una señal. Si una señal es alimentada por dos fuentes, entonces cuando ambas están activas, la señal tendrá dos *drivers.*

Bus: es usualmente dado a un grupo de señales para la comunicación entre distintos módulos. En VHDL un bus es especialmente un tipo de señal que puede estar conectado a múltiples *drivers*.

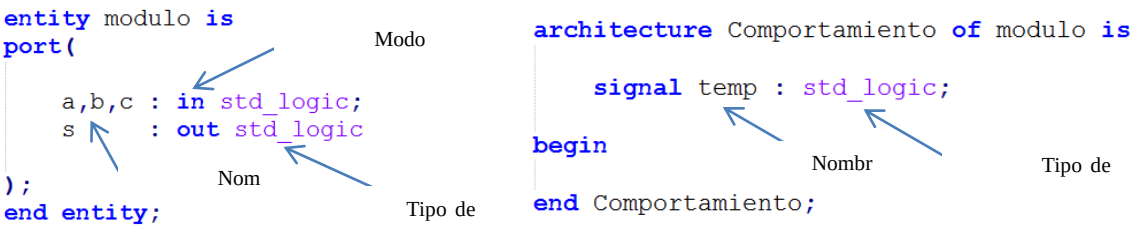
Generic: este término es utilizado como un parámetro, permite definir información importante en los módulos, es constante una vez definida. Entre los usos más frecuentes está la definición de anchos de bus y límite para contadores.

1.4. Señales en VHDL

Las señales son elementos muy importantes en VHDL, ya que son los que se encargan de guardar los estados actuales o futuros de los componentes del diseño. Las memorias en hardware son análogas a las señales, también una forma muy común de referirse a ellas es como cables o alambres, ya que conectan distintos elementos generados en la descripción de hardware, pero si se piensa detenidamente este concepto es lo mismo a una memoria de una posición.

Las señales pueden declararse en distintos puntos en el diseño pero siempre representarán lo mismo y estarán bajo las mismas reglas. Los puertos de entrada y salida que se declaran en un diseño son señales y tienen su respectiva forma de ser declarados. Las formas de declarar señales son las siguientes:

Figura 1. **Partes de módulo en VHDL**



Fuente: elaboración propia.

Una característica muy importante de las señales es que estas no pueden ser inicializadas, ya que al ser análogas a cables en un circuito combinacional no pueden tener un estado inicial. Por ejemplo, si se realiza un circuito con una compuerta *and*, dos señales de entrada y una señal salida, si se inserta un voltaje en una entrada, pero en la otra no, la salida debería ser cero voltios, pero si se hubiese inicializado la salida con un voltaje alto el circuito no tendría sentido.

Existen varios tipos de señales dependiendo del uso que se les desee dar, estas pueden ser:

- BIT
  - Puede tener dos posibles valores: ‘0’,’1’
  - *Signal sbit:* bit

- BIT_VECTOR
  - Es un arreglo de bits
  - *Signal vbit: bit_vector*

- INTEGER
  - Es un tipo de dato entero de 32 bits y puede contener signo, sus posibles valores van desde -2147483648 hasta 2147483648.
  - Signal entero : integer range 0 to 10.
  - Para declarar este tipo de señales se debe incluir un rango con la palabra reservada *range*, indicando los posibles valores que tendrá la señal.

- NATURAL
  - Es un tipo de señal como el integer, solo que este incluye números positivos, por lo tanto, puede tomar valores de 0 hasta 2147483648.

- STD_LOGIC
  - Este es un tipo de señal el cual se incluye en una librería, es el tipo de señal que se utiliza en vez del nativo bit, ya que incluye más que un estado de uno y cero, permite utilizar más estados en un circuito, como por ejemplo la alta impedancia que debería tener una entrada.
  - Este tipo de señal incluye nueve estado que pueden ser utilizado en la señal.
    - '1' : es el que representa un estado lógico alto.
    - '0' : este es un estado lógico bajo.
    - 'X' : se utiliza cuando no se conoce exactamente el valor lógico de la señal.
    - 'Z' : este representa un valor de alta impedancia muy utilizado en entradas a circuitos.
    - 'U' : es el valor para representar un estado indefinido para la señal.
    - 'W' : es llamado indefinido débil.
    - 'H' : es referido a un estado de lógico débil.

- ‘L’ : se utiliza cuando se tiene un estado lógico bajo débil.
- ‘-’ : define que el valor de la señal no importa.

- STD_LOGIC_VECTOR
  - Este tipo de señal surge al crearse un bus de *std_logic*, es decir es un arreglo unidimensional de *std_logic*.
  - *Signal* tipoBus : std_logic_vector(7 downto 0)
  - Se debe especificar la longitud de bits por utilizar, en el caso anterior se tiene que el bit del bus del lado izquierdo será el más significativo debido a la palabra *downto,* si se desea al revés de como se definió, se debe utilizar la palabra reservada to.

- CHARACTER
  - Se utiliza para definir caracteres con código ASCII.

- STRING
  - Es un arreglo de caracteres definidos por el tipo anterior.

1.5. Lógica concurrente, el paralelismo de una FPGA

Al desarrollar circuitos se evidencia que las señales se distribuyen a todos los componentes al mismo tiempo, lo mismo sucede dentro de este tipo de lógica.

1.5.1. Ejecución en paralelo dentro de una FPGA

Cuando se diseñan circuitos digitales básicamente se utilizan elementos como compuertas lógicas conectadas entre sí para alcanzar una salida,

independiente del tiempo de ejecución del resultado final. Por cada uno de los componentes pasan niveles de voltaje representando unos y ceros.

Por lo tanto, si hay un cambio de nivel de voltaje automáticamente cambiaría en cada uno de los componentes del circuito, es decir, no por estar conectada primera una compuerta tiene prioridad en ejecución, a esto se le llama lógica concurrente o ejecución en paralelo. Una FPGA tiene una ejecución concurrente, ya que al estar conformado por un arreglo de compuertas los componentes van a estar conectados tal y como se haría en un circuito combinacional de compuertas.

1.5.2. Diferencia entre lógica secuencial y lógica concurrente

Normalmente un circuito digital no siempre se ejecuta de forma concurrente ya que en muchas ocasiones se necesita realizar acciones de forma secuencial, como es el caso de las transmisiones seriales, por ejemplo, si se realizara de forma concurrente o paralela se requería de más líneas de transmisión y esto puede ser problemático en muchos casos.

Debido a esto en el diseño digital de circuitos se crea un área para tratar estos casos, llamada lógica secuencial, lo componen siempre circuitos y combinaciones solamente que, trabajando en conjunto con otras señales como los relojes, para conformar otros componentes, por ejemplo los llamados *flip-flops.*

1.5.3. Compuertas digitales

Algo fundamental en VHDL es describir circuitos combinacionales construidos con compuertas digitales. Antes de explicar cómo es el

funcionamiento de este tipo de circuitos en VHDL se debe conocer dónde se declaran los dos primeros elementos que construyen un módulo, los cuales se describieron anteriormente. El primero es el llamado *entity* en los cuales se definen los puertos de entrada y salida del módulo. Se declara en el principio del módulo VHDL de la siguiente manera:

Figura 2. **Estructura de entidad en VHDL**

```
entity Compuertas is
    port(
    a,b : in std_logic;
    c   : out std_logic
);
end entity;
```

Fuente: elaboración propia.

De esta forma se pueden declaran dos entradas y una salida de un tipo especial que en forma simple es equivalente a un bit.

El segundo elemento es llamado *architecture* el cual es utilizado para describir el comportamiento del circuito diseñado. Este se declara después del *entity* de la siguiente manera:

Figura 3. **Estructura de arquitectura en VHDL**

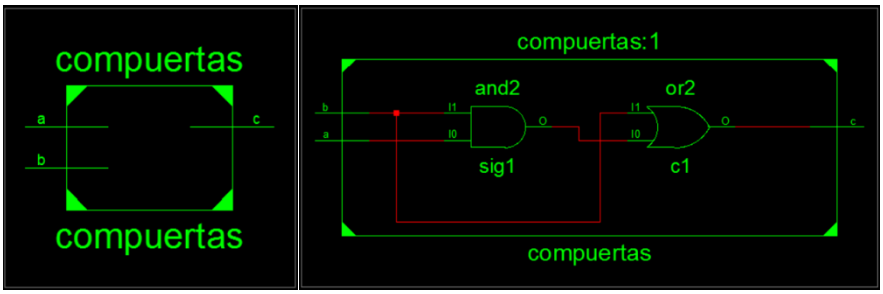
```
architecture Behavioral of compuertas is
signal sig : std_logic;
begin
    sig <= a and b;
    c <= sig or b
end Behavioral;
```

Fuente: elaboración propia.

Esta es la forma de describir un comportamiento, dentro del *architecture* se hacen las declaraciones de cualquier comportamiento del circuito como simples asignaciones o declaración de compuertas lógicas. Una señal como se describió anteriormente se debe colocar entre el *is* y el *begin* del *architecture* como se muestra en la figura.

La manera de hacer que una entrada o una señal tenga un valor es utilizando el operador de asignación simple, el cual asemeja una flecha apuntando en dirección a la señal de salida. En la figura se muestra que se hace una asignación simple de una compuerta *and* a una señal llamada sig, el resultado de esta operación lógica se utiliza para realizar una operación *or* con una señal de entrada para finalmente ser asignada a la salida. El diagrama combinacional equivalente a la descripción de hardware es la siguiente:

Figura 4. **Visualización de módulo general**



Fuente: elaboración propia.

1.6. Tipos de asignación concurrente

Los tipos de asignación concurrente se utilizan dependiendo el tipo de componente que se necesite.

1.6.1. Asignación simple

La asignación simple es la que se utiliza para dar valor a las señales directamente. Cuando se trata de una señal su operador es '<=' lo que hace referencia a una flecha, pero cuando se refiere a una variable su operador es ':=' el cual es parecido al anterior.

Figura 5. **Tipos de asignación**

```
s <= a and b;
v := a and b;
```

Fuente: elaboración propia.

La asignación simple incluye distintos tipos de valor ya que dependiendo del tipo de señal casi será la sintaxis asignada.

Cuando se necesita hacer una asignación de un valor binario a una señal tipo *std_logic_vector* se debe hacer con comillas dobles, como se muestra a continuación:

Figura 6. **Asignación tipo vector**

```
signal senial : std_logic_vector(7 downto 0);
senial <= "10101010";
```

Fuente: elaboración propia.

Si se desea hacer una asignación simple de un número hexadecimal se debe hacer de la siguiente forma:

Figura 7. **Asignación hexadecimal**

```
senial <= x"AA";
```

Fuente: elaboración propia.

En el caso de hacer una asignación de un segmento de un bus se debe colocar entre paréntesis el rango al cual se quiere acceder utilizando la palabra reservada *downto* o *to*.

Figura 8. **Asignación vectorial parcial**

```
senial(3 downto 0) <= "1111";
```

Fuente: elaboración propia.

Para asignar solo un bit del bus se debe utilizar la posición del bus entre paréntesis y colocar la posición, como la señal tipo bus es de un bit será de tipo std_logic, por lo tanto, se debe utilizar comillas simples.

Figura 9. **Asignación de un bit**

```
senial(7) <= '1';
```

Fuente: elaboración propia.

Algunas veces se necesita poner todos los bits de un bus de la misma forma, por ejemplo, si se desea inicializar con cero todo el bus se debe realizar de la siguiente manera.

Figura 10. **Asignación por defecto en un vector**

```
senial <= (others => '0');
```

Fuente: elaboración propia.

Cuando una señal es de tipo entero debe asignarle un rango para restringir sus posibles valores. La asignación simple de este tipo de señal es solamente el número deseado.

Figura 11. **Asignación a enteros**

```
signal entero : integer range 0 to 255;
entero <= 1;
entero <= entero + 1;
```

Fuente: elaboración propia.

Como se puede notar también se puede asignar una señal a otra señal haciendo alguna operación o no. Esto es muy útil para generar contadores.

1.6.2. Asignación condicional

La asignación condicional es un tipo de asignación concurrente que tiene un análogo directo en hardware, por lo tanto, genera acciones en tiempo real. Este tipo de estructura se parece a una estructura de condicional *if* en lenguajes

convencionales. Es una estructura análoga también a multiplexores construidos con compuertas lógicas, en el análisis digital se estudian temas como tablas de verdad y simplificación de circuitos digitales, esta estructura ayuda a describir este tipo de circuitos.

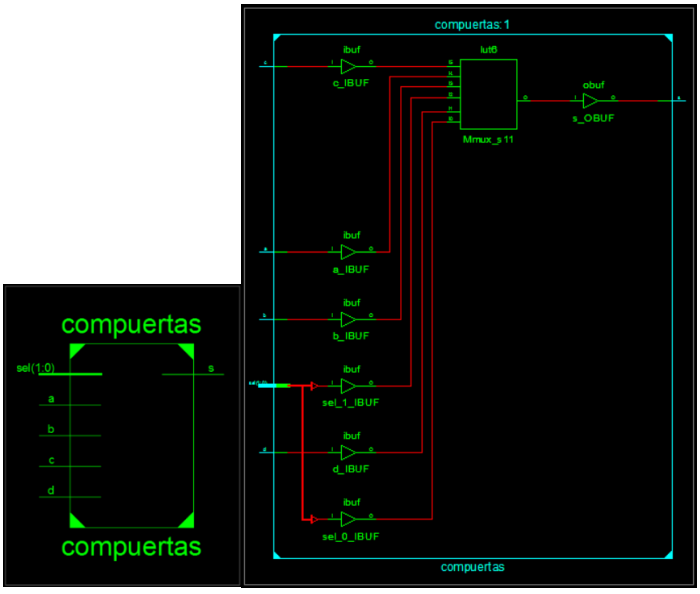
Figura 12. **Asignación condicional**

```
<valor> <= <señal/valor> when <condición 1> else
           <señal/valor> when <condición 2> else
           <señal/valor> when <condición 3> else
                  ..................
           <señal/valor> when <condición n> else
           <señal/valor>;
```

Fuente: elaboración propia.

Si se analiza el diseño como un bloque se puede notar lo siguiente:

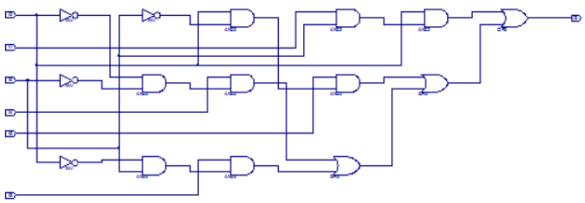
Figura 13. **Módulo general de asignación condicional**



Fuente: elaboración propia.

El diagrama de compuertas digitales y la ecuación correspondiente del multiplexor Mmux_s11, es el siguiente:

Figura 14. **Diagrama digital de compuertas de una asignación condicional**



O = ((I0 * !I2 * I3) + (!I0 * !I2 * I4) + (!I0 * I2 * I5) + (I0 * I1 * I2));

Fuente: elaboración propia.

El circuito descrito es muy simple y ejecuta todas las acciones en paralelo al ser un circuito digital combinacional.

Figura 15. **Descripción VHDL de asignación condicional**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity compuertas is
port (
a,b,c,d: in std_logic;
sel : in std_logic_vector(1 downto 0);
s: out std_logic
);
end compuertas;

architecture Behavioral of compuertas is
signal sig : std_logic;
begin

s <= a when sel = "00" else
     b when sel = "01" else
      c when sel = "10" else
      d when sel = "11" else
      '0';


end Behavioral;
```

Fuente: elaboración propia.

1.6.3. Asignación selectiva

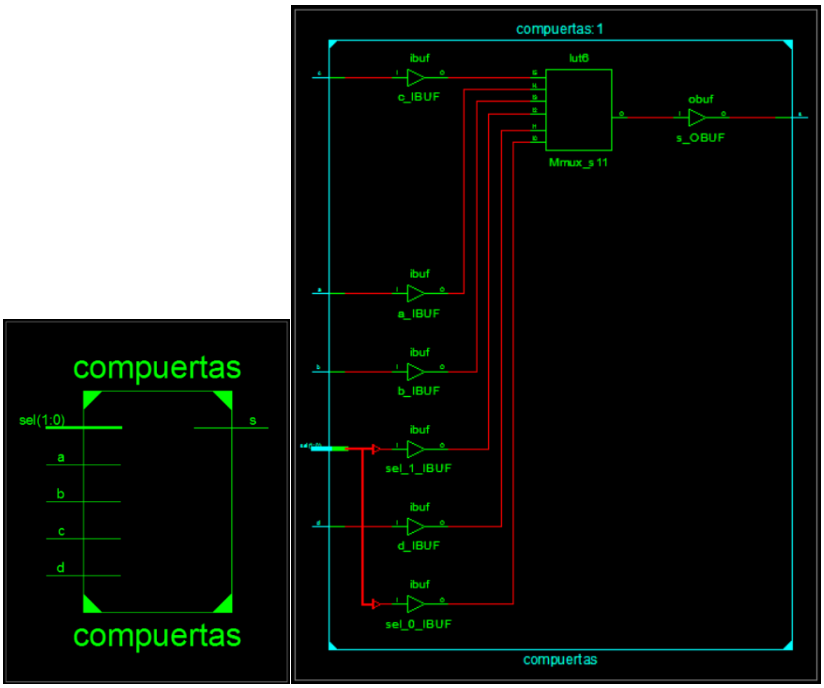
Este tipo de asignación se utiliza para describir selectores, dependiendo el valor de una señal se asignará otra señal o valor. Si se hace una analogía con lenguajes convencionales de programación esta sería con la estructura de selección *switch.* Esta estructura de asignación se utiliza principalmente para facilitar la selección de señales.

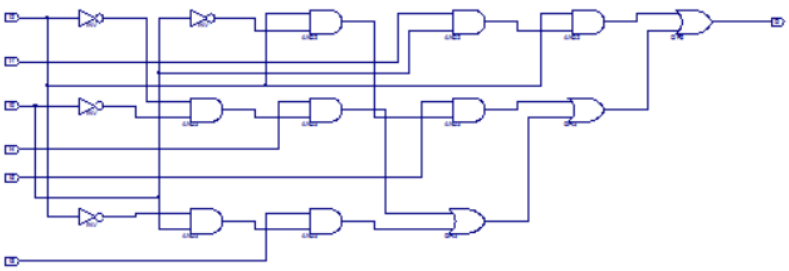
Figura 16. **Estructura de asignación selectiva**

```
with <señal/expresión> select
    <señal destino>  <= <señal/valor> when <condición 1>,
                        <señal/valor> when <condición 2>,
                        <señal/valor> when <condición 3>,
                                 ..................
                        <señal/valor> when <condición n>,
                        <señal/valor> when others;
```

Fuente: elaboración propia.

Figura 17. **Modelo de compuertas para asignación selectiva**





```
O = ((I0 * !I2 * I3) + (!I0 * !I2 * I4) + (!I0 * I2 * I5) + (I0 * I1 * I2));
```

Fuente: elaboración propia.

Figura 18. **Descripción VHDL para asignación selectiva**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity compuertas is
port (
a,b,c,d: in std_logic;
sel : in std_logic_vector(1 downto 0);
s: out std_logic
);
end compuertas;

architecture Behavioral of compuertas is
signal sig : std_logic;
begin

with sel select
s <= a  when "00",
     b  when "01",
     c  when "10",
     d  when "11",
    '0' when others;


end Behavioral;
```

Fuente: elaboración propia.

Este diseño se realizó para que tuviese la misma funcionalidad que se realizó en el diseño generado en la asignación condicional. Como se puede demostrar, el circuito es el mismo ya que VHDL solo es un lenguaje de descripción, por lo tanto, si se describe la misma acción deberá generarse el mismo circuito sin importar si el código es distinto.

1.7. Generador de lógica concurrente en lenguaje convencional

Cuando se necesitan componentes que funcionen de forma secuencial se necesita un tipo de lógica especial, para implementarlos.

1.7.1. Los record

Los record en VHDL se pueden usar para simplificar el código del diseño de hardware. Los records son similares a los llamados *structs* en C. Estos se usan con mayor frecuencia para definir un nuevo tipo de dato en VHDL. Este nuevo tipo contiene cualquier grupo de señales que el usuario desee. Usualmente se utiliza para simplificar las interfaces.

Esto es muy útil con interfaces que tienen una gran lista de señales que siempre es la misma. Pueden usarse para reducir el tamaño del código en la definición de puertos, es decir, definir menos señales en el *entity*. Para esto el diseñador simplemente necesita definir el tipo de record en un paquete y luego usar el archivo de paquete para cualquier entidad que haga uso de ese tipo de registro.

Usualmente un record se declara dentro del módulo VHDL entre el *is* y el *begin* del *architecture* de la siguiente manera:

Figura 19. **Estructura de un record**

```
type recordPuertos is record
    wr    : std_logic;
    data  : std_logic_vector(7 downto 0);
    rd    : std_logic;
    cont  : integer range 0 to 10;
end record t_TO_FIFO;

signal entradas : recordPuertos;
```

Fuente: elaboración propia.

1.8. Lógica secuencial

Como se ha mostrado en una arquitectura todas las declaraciones son concurrentes. Los enunciados secuenciales en VHDL se refieren a las sentencias que se ejecutan con un predefinido, una detrás de otra. Para que esto sea posible existe una sentencia llamada *process,* en el cual toda la lógica definida dentro de esta se ejecuta de forma secuencial.

La declaración del *process* es en sí misma una declaración concurrente, quiere decir que si se declara esta al mismo nivel que un componente concurrente ambos se ejecutarán al mismo tiempo pero la lógica dentro del *process* tendrá otro tipo de ejecución. Un *process* tiene una sección de declarativa y una parte de ejecución. En la sección declarativa, se incluyen elementos como variables, constantes y subprogramas. La otra parte contiene solo declaraciones secuenciales. Los enunciados secuenciales consisten en sentencias *case*, *then, if,* else, *loop*, y otras.

1.8.1. El process

Un *process* describe la ejecución secuencial en un diseño de hardware en VHDL. Es la unidad básica para tratamiento de señales secuenciales.

1.8.1.1. Descripción y forma correcta de utilizarlo

Este tipo de estructura es activada mediante señales que se declaran en una lista sensitiva. Varios *process* pueden coexistir dentro la misma arquitectura ejecutándose en paralelo. El *process* puede estar en dos estados:

- Ejecución: las sentencias que define el *process* es ejecutado.

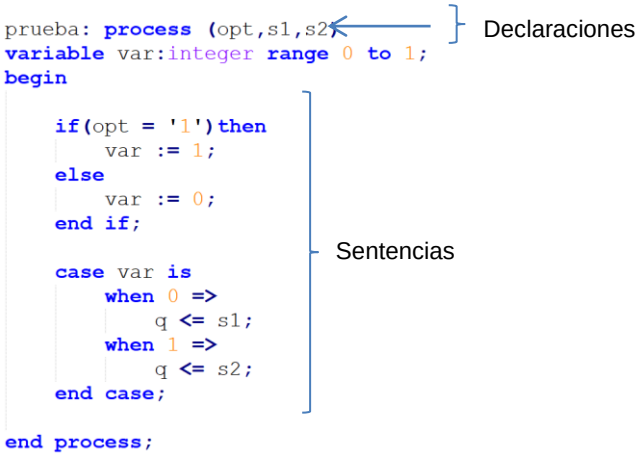
- Espera: el *process* se mantiene a la espera que alguna señal definida en su lista sensitiva cambie su valor y pasará al estado de ejecución una vez eso suceda.

Las declaraciones locales que se pueden definir en un *process* incluyen definición de tipos, funciones, procedimientos y variables.

La lista sensitiva es una serie de señales las cuales son las encargadas de activar un *process*. Un *process* puede tener una lista sensitiva explícita. Esta lista se define junto a la palabra *process* dentro de paréntesis. Un *process* se ejecuta cada vez que uno o más elementos de la lista cambian su valor. Las herramientas de síntesis consideran que la instrucción *process* describe la lógica secuencial.

Una lista sensitiva puede ser útil para comunicar *process,* ya sea con otros o con la lógica combinacional. La sintaxis de un *process* se muestra a continuación:

Figura 20. **Estructura de un *process***



Fuente: elaboración propia.

1.8.1.2. Comportamiento de señales y variables dentro de un *process*

Algo importante sobre los *process* es que todas las señales se mantienen sin cambio mientras este se ejecuta, por lo tanto, si se describe dentro de un *process* algún cambio de una señal, esta no actualizará su valor hasta que se termine de ejecutar ya que las señales tienen un análogo directo en hardware. Las señales son las que comunican la lógica secuencial y la lógica concurrente, es debido a esto que cuando se necesita que el hardware funcione en algunos momentos de manera secuencial y en otros momentos de manera concurrente, las señales son fundamentales.

Una variable se define localmente dentro de un *process* utilizándose principalmente para validaciones ya que el valor de una variable puede cambiar durante la ejecución de un *process* a diferencia de las señales. Una variable no

tiene un equivalente en hardware es por esto que no pueden existir fuera de un *process* y funcionar de manera concurrente.

Algo muy importante es que todas las señales que están dentro de la lista sensitiva y son evaluadas en las estructuras condiciones son señales de entrada. Sí las señales de entrada no están sicronizadas con un reloj, es decir, se colocan fuera del condicional de sincronización, deben ser parte de la lista sensitiva ya que de lo contrario el comportamiento será el indeseado. Todas las señales a las cuales se les asigna un valor dentro de un *process* se les llaman señales de salida, que deben tener un valor en todos los posibles casos.

1.8.1.3. Ventajas y desventajas de utilizar *process*

Las ventajas incluyen el poder utilizar una lógica de una programación convencional, es decir el tipo de programación modular que comparte ciertas instrucciones similares entre sí. Un diseño de hardware necesitará mayor cantidad de componentes al generar esta lógica convirtiéndose esto en la mayor desventaja de la utilización de *process.* Lo recomendable es hacer uso de ambas e intercomunicar toda esta parte concurrente con la parte secuencial mediante señales.

1.8.1.4. El reloj del sistema digital y su aplicación a las iteraciones

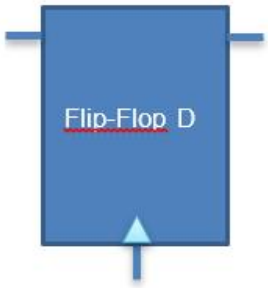
Como es bien sabido un reloj en un sistema digital es el que permite la generación de procesos sincronizados. Los *flip-flops* se utilizan para generar lógica secuencial, por lo tanto utilizarlos es la base de esta lógica. La activación de un *process* por medio de un reloj crea un ciclo de repeticiones esto es equivalente a un ciclo *for* en lenguajes convencionales, por esto que el

diseñador del hardware puede utilizarlo para generar iteraciones como contadores o tiempos de espera.

1.8.1.5. Relación entre los *flip-flops* y los *process*

Un *process* en sí funciona como un *flip-flop* donde las salidas pueden estar afectadas por un reloj y por las entradas.

Figura 21. **Estructura de *flip flop***




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

entity FF is
    port(
        d   : in std_logic;
        clk : in std_logic;
        q   : out std_logic
    );
end entity;

architecture behavioral of ff is
begin

    process(clk)
    begin
        if(rising_edge(clk)) then
            q <= d;
        end if;
    end process;
end behavioral;
```

Fuente: elaboración propia.

Como se puede observar en la figura, un simple *process* es análogo a un *flip-flop* la instrucción, dentro de la condición se puede definir si se desea hacer la validación para un flanco de subida o un flanco de bajada.

1.8.2. Asignación condicional en lógica secuencial

La asignación condicional en VHDL es muy parecida a la que se utiliza en lenguajes modulares como el lenguaje C. A continuación se describirá cuáles son las que se permite utilizar.

1.8.2.1. Sentencia if, else if y else

Esta sentencia puede ser utilizada únicamente dentro de los *process,* debido a esto es que al empezar muchos diseñadores de hardware prefieren construir toda su lógica utilizando *process,* pero esta no es la mejor práctica debido a la cantidad de hardware generado. Un *if* en un lenguaje de descripción de hardware se refiere a crear lógica selectiva dependiendo de alguna condición dada. Los *if* pueden ser anidados, lo cual significa que si no se cumple la primera condición pase a alguna siguiente y así sucesivamente. La recomendación que se da es anidar la menor cantidad de *if* debido a la cantidad de hardware generado.

En la parte condicional se puede tener más de una condición por evaluar, utilizando los conectores lógicos *AND, OR, XOR,* y otros*.* Algo muy importante que se debe tener en cuenta es que estas sentencias pueden tener más de una rama y puede existir más de una instrucción secuencial en cada rama de *If,* pero estas se ejecutarán concurrentemente, es decir todas al mismo tiempo.

Una recomendación para cuando se desea trabajar con ramas de *if* es colocar siempre un *else* para definir todos los casos posibles ya que si no se realiza esto se generará un *latch*, un *latch* es un circuito biestable asíncrono que no necesita una sincronización de reloj y tiene un estado indefinido. Un *latch* puede hacer que el circuito tome comportamientos inesperados y sí se tienen

flip-flops sincronizados con reloj puede causar incompatibilidades en el diseño de hardware.

La sintaxis de un *if,* como se podrá notar en la siguiente figura, es muy parecida a un lenguaje modular.

Figura 22. **Estructura condicional en *process***

```
if(rst='1')then
    s <= '1';
elsif(rst='0')then
    s <= '0';
else then
    s <= 'U';
end if;
```

Fuente: elaboración propia.

1.8.2.2. When – Case

Esta sentencia es similar al *if* pero la diferencia es que depende solamente de una expresión. Es muy utilizada cuando se necesita evaluar una señal o variable como si fuese un menú con diferentes opciones. Se recomienda mucho siempre colocar la sentencia *when others* para definir todos los escenarios posibles.

Figura 23. **Estructura selectiva en *process***

```
case sel is
    when "001" =>
        x <= "0001";
    when "010" =>
        x <= "0010";
    when "011" =>
        x <= "0011";
    when "100" =>
        x <= "0101";
    when "101" =>
        x <= "0110";
    when "110" =>
        x <= "0111";
    when "111" =>
        x <= "1000";
    when others =>
        x <= "0000";
end case;
```

Fuente: elaboración propia.

1.8.2.3. Cuándo utilizar asignación secuencial y cuándo utilizar asignación concurrente

La cantidad de hardware generado en un diseño de un circuito es muy importante, ya que determinará si una FPGA es viable o no. Si se crea un diseño óptimo el circuito podrá sintetizarse en una FPGA de gama menor siempre y cuando las características del sistema lo permitan. También si se tiene una FPGA de mayor gama se podrá sintetizar más hardware y así ampliar las funcionalidades del diseño creado.

Una forma de optimizar los diseños de hardware es saber mezclar la lógica secuencial con la lógica concurrente. La lógica concurrente que se genera al describir un circuito digital está compuesta por un conjunto de compuertas lógicas, las cuales son muy elementales en el diseño de circuitos y, por lo tanto, ocuparán menor cantidad de hardware a diferencia de la lógica

secuencial, ya que este genera una serie de *flip-flops.* Estos están compuestos por una serie de lógica concurrente dependiente de cambios de flancos, además de esta lógica en un proceso secuencial se genera lógica concurrente adicional para generar de manera correcta las acciones y salidas del sistema. Como se puede observar al sintetizar diseños secuenciales la cantidad de hardware necesario es mayor que la cantidad requerida en un diseño concurrente.

Hay momentos en la descripción de hardware que pareciera que utilizar la lógica secuencial facilitaría el diseño, pero realmente no es necesario ni óptimo emplear dicha lógica. Cualquier diseño que no requiera *process* consecutivos puede ser descrito con lógica concurrente. Puede pensar en el proceso necesario para hacer circuitos concurrentes, primero pensar en las entradas, luego colocar para cada una la salida correspondiente, si esto es posible el diseño que se está esperando puede crearse con lógica concurrente.

Todo proceso dependiente del tiempo o del cambio en un momento específico de una señal puede describirse con lo lógica secuencial. Para optimizar completamente el diseño puede construir la lógica secuencial cambiando el valor de una señal sin utilizar tantas sentencias como *if o when* y generar los resultados con lógica concurrente utilizando las estructuras de selección.

1.8.2.4. Generador de la asignación condicional en lógica secuencial en lenguaje convencional

En esta sección se recomienda hacer un programa utilizando un lenguaje convencional para conseguir un mejor concepto de la lógica secuencial. Los

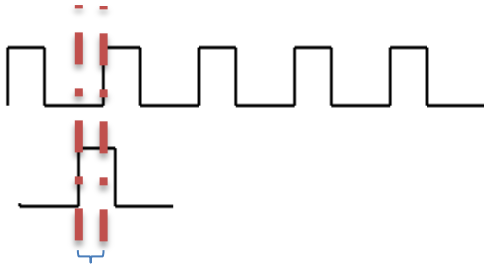
generadores de código son muy útiles para obtener los conceptos de la descripción de hardware ya que si se tiene una base conocida se puede construir de mejor manera el conocimiento.

1.8.3. Tipos de reset

Imagine que el circuito descrito inicie y dentro de un *process* se evalué una señal con una estructura *if,* ¿qué valor tiene al activarse el *process*?, la respuesta a esta pregunta es que el valor de la señal es un estado indefinido. Al momento de diseñar un circuito digital con VHDL la recomendación principal es escribir el diseño lo más detallado posible, por lo tanto, lo ideal es describir también con qué valor inicia esta señal antes de ser evaluada. Una solución es colocar una señal que inicialice las señales de entrada para que tengan un valor diferente a indefinido. Se definirán tres tipos de *reset:*

- Síncrono: utiliza una sincronización con un reloj, esto hace que no se creen estados de metaestabilidad, es decir, que no se permanece en un estado de equilibrio débilmente estable durante un considerable período de tiempo. Cuando se crea este tipo de *reset* no suceden indeseados y se asegura la sincronización total del sistema. Este tipo de reset es relativamente lento, ya que tiene que esperar a que se sincronice con el reloj del sistema. Ocupa más espacio que el *reset* asíncrono, ya que utiliza *if* anidados para describir su funcionamiento.

Figura 24. **Tipo de reset síncrono**



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

entity FlipFlop is
    port (
        d, clk, rst: in std_logic;
        q: out std_logic
    );
end FlipFlop;

architecture Behavioral of FlipFlop is
begin

    process(clk) then

        if(rising_edge(clk))then

            if rst = '1' then
                q <= '0';
            else
                q <= d;
            end if;

        end if;

    end process;

end Behavioral;
```

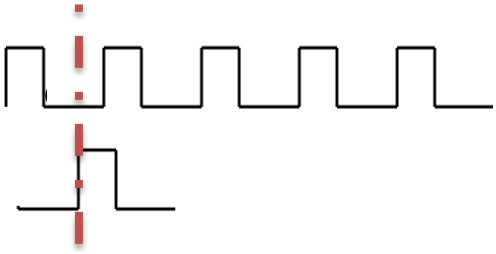
Fuente: elaboración propia.

Como se observa en la figura, al activarse el *reset* se tendrá el efecto deseado únicamente al pasar el tiempo t, pudiendo tener un retraso máximo de casi un ciclo de reloj, ya que el *reset* puede activarse un poco después de que suceda el próximo cambio de flanco de reloj correspondiente a la sincronización.

- Asíncrono: este tipo no necesita un reloj para funcionar, simplemente cuando se detecta ejecuta las sentencias. Se genera una ruta de *reset* reservada para todos los *flip-flops.* Es más rápido que un *reset* síncrono. Ocupa menos espacio en la síntesis de hardware. Este tipo de *reset* sí genera un estado de metaestabilidad, debido a que no se conoce el momento en el que se activará y no se tiene un reloj de sincronización

que pueda causar *reset* indeseados causados por *glitches* o ruido indeseado.

Figura 25. **Tipo de reset asíncrono**



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

entity FlipFlop is
    port (
        d, clk, rst: in std_logic;
        q: out std_logic
    );
end FlipFlop;

architecture Behavioral of FlipFlop is
begin

    process(clk, rst) then

        if rst = '1' then
            q <= '0';
        elsif(rising_edge(clk)) then
            q <= d;
        end if;

    end process;

end Behavioral;
```

Fuente: elaboración propia.

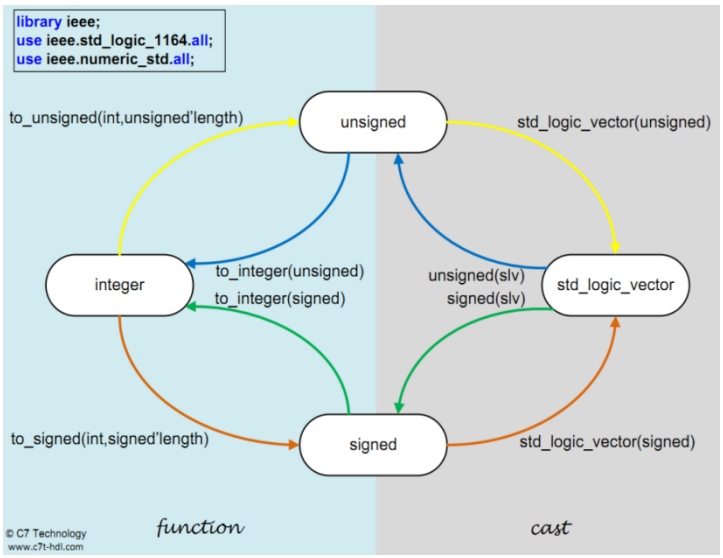
- Asíncrono con desactivación síncrona: l mayor problema que existe con el *reset* asíncrono es la desactivación o el finalizar el proceso de ejecución ya que si este durara mucho tiempo el sistema podría entrar en conflicto. Por esto es que, si se tienen los *process* sincronizados a un reloj, se podría pausar el sistema hasta que haya una desactivación síncrona, a esto se le llama *reset* con desactivación síncrona.

1.8.4. Conversión de tipos

Como se definió anteriormente, existen tipos de señales dependiendo del valor que se les desee asignar. Existen muchos casos en donde es necesario

utilizar vectores de bits llamados *std_logic_vector*, como por ejemplo en las interfaces de entrada. Pero en otros momentos se necesita utilizar cálculos con enteros. Si las señales son de diferente tipo al soportado, no se pueden realizar las operaciones. Es por esto que se necesita la conversión de tipos, que se muestra en el siguiente diagrama.

Figura 26. **Conversiones de tipos de datos**



Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

El diagrama define una serie de caminos para mostrar las funciones necesarias al convertir los tipos de datos. Por ejemplo, si se tiene una señal entera se empezará por el estado "*integer".* Si se desea convertir a "std_logic_vector" se seguirá la línea amarilla con flecha utilizando la función "to_unsigned (int,unsigned'length)" colocando el *integer* como primer parámetro y la longitud del vector como segundo parámetro pasando así al estado

"*unsigned".* Como se puede notar el tipo deseado no es ese, por lo tanto, se debe continuar utilizando la función "*std_logic_vector (unsigned)"* pasando al último estado que es el tipo "*std_logic_vector".* La función final se muestra en la siguiente figura.

Para cualquier otra conversión deseada se debe seguir el mismo procedimiento siguiendo las flechas correspondientes.

1.8.5. Tipos definidos por el usuario

VHDL permite al diseñador crear sus propios datos personalizados respaldados en usar tipos existentes. Esto se utiliza para que el diseñador pueda tener mayor comprensión del código, tanto para definir condicionales como para crear máquinas de estados.

La sintaxis para crear un nuevo tipo de señal es de la siguiente forma.

Figura 27. **Tipos definidos por el usuario**

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

entity Tipos is
    port (
        a,b: in std_logic;
        q:   out std_logic
    );
end Tipos;

architecture Behavioral of Tipos is

type tipoNuevo is (EstadoUno, EstadoDos, EstadoTres);
signal senial: tipoNuevo;

begin

    senial <= EstadoUno;

end Behavioral;
```

Fuente: elaboración propia.

1.8.6. Máquinas de estado con una FPGA

Se utiliza una máquina de estados finitos para modelar un sistema que transita entre un número finito de estados internos. Las transiciones dependen del estado actual y de la entrada externa. A diferencia de un circuito secuencial regular, las transiciones de estado de un FSM no muestran un patrón simple y repetitivo. Su lógica del siguiente estado generalmente se construye desde cero y a veces se la conoce como lógica aleatoria. Esto es diferente de la lógica del siguiente estado de un circuito secuencial regular, que está compuesto principalmente de componentes estructurados, como circuitos incrementadores o desplazadores.

Una máquina de estados finitos generalmente se especifica mediante un diagrama de estado abstracto, que representa la entrada, salida, estados y transiciones del FSM en una interpretación gráfica. La representación FSM es más compacta y mejor para aplicaciones simples. La representación del gráfico ASM (gráfico de máquina de estado algorítmico) es un diagrama de flujo y es más descriptiva para aplicaciones con condiciones y acciones de transición complejas.

Un diagrama de estado se compone de nodos, que representan estados y se dibujan como círculos y arcos de transición anotados. Una expresión lógica expresada en términos de señales de entrada se asocia con cada línea de transición y representa una condición específica. La línea se toma cuando la expresión correspondiente se evalúa como verdadera. Los valores de salida de Moore se colocan dentro del círculo, ya que dependen solo del estado actual. Los valores de salida de Mealy están asociados con las condiciones de las líneas de transición, ya que dependen del estado actual y de la entrada externa.

La señal de salida toma el valor predeterminado de lo contrario. una señal de salida de Moore (es decir, yl) y una señal de salida de Mealy (es decir, yo).

Un circuito secuencial que es implementado en un número finito de posibles estados es llamado máquina de estados finita Estas máquinas son críticas para la realización, el control y la decisión de la lógica de sistemas digitales. Las máquinas de estados finitas son una parte integral en el diseño de sistemas.

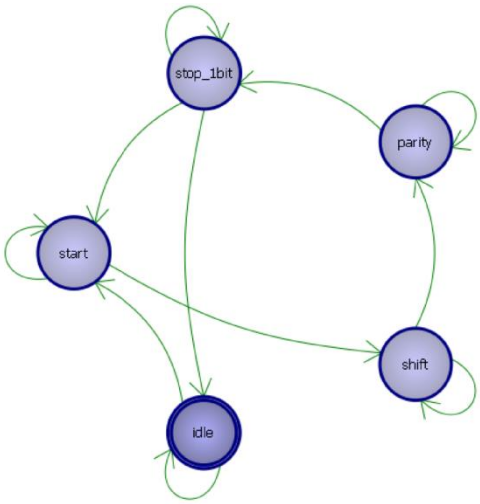
VHDL no tiene un formato formal para modelar las máquinas de estado finitas, pero para hacerlo se deben seguir lineamientos para tener la certeza de que estas van a funcionar.

Para ejemplificar mejor cómo funciona una máquina de estados finita se tomarán los estados de un transmisor serie tipo RS-232. El dato recibido y el dato por transmitir deben tener el siguiente formato:

- 1 bit de *start.*
- 8 bits de datos.
- Paridad programable, con paridad, sin paridad, paridad par o paridad impar.
- 1 bit de *stop.*
- Frecuencia de transmisión por defecto es de 9 600 bauds, pero se puede transmitir a otras frecuencias como: 4 800, 38 400, 115 200 bauds.

Por lo tanto, el diagrama de estados para este sistema digital se vería de la siguiente manera.

Figura 28. **Diagrama de estados de UART**



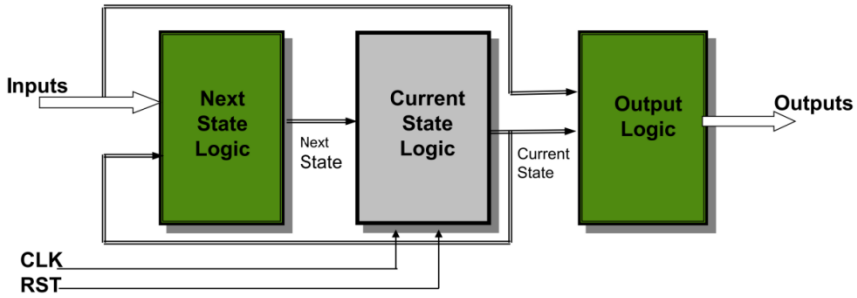
Fuente: elaboración propia.

El diagrama general de una máquina de estados se puede dividir en tres partes muy importantes:

- Lógica del estado siguiente (*Next State Logic)*: lógica que se encarga de velar cuál será el próximo estado dependiendo de las entradas que se tengan el sistema.
- Lógica del estado presente (*Current State Logic*): esta lógica se encarga de preguntar en qué estado me encuentro, esto siempre es dependiente del estado anterior.
- Lógica de salida (*Output Logic):* es en esta lógica en donde se elige el tipo de salida que tendrá el sistema.

El diagrama general de una máquina de estados finita se puede describir en el siguiente diagrama:

Figura 29. **Diagrama general de máquina de estados**



Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

Como es bien sabido, los estados en una máquina tienen tipos de codificaciones entre los cuales se puede mencionar *Automatic, Binary, Gray, One-hot, Compact State, Jhonson State, Sequential State, Speed1 State* y *User State.*

1.8.6.1. Estilos de codificación de máquinas de estado en VHDL

En VHDL las máquinas de estados también tienen estilos de codificación:

- Estilo A
- Estilo B
- Estilo C
- Estilo D
- Estilo E

Dependiendo del estilo de codificación de la máquina de estados se tendrán distintos lineamientos y lógica de diseño, también la dificultad de código se ve reflejada en esto. Algunos estilos son más fáciles de utilizar que otros, pero la cantidad de hardware generado varía mucho dependiendo del estilo de codificación de la máquina de estado por implementar. En esta sección se va a definir cada uno de los estilos con sus ventajas y desventajas junto con su forma de escribirlos en VHDL.

En la tabla se muestra los *process* necesarios en cada estilo de codificación de máquinas de estados en VHDL.

Tabla II. **Tipo de codificación de máquinas de estados**

Estilo	Lógica para estado presente	*Lógica* para siguiente estado	Lógica de salida
Estilo A			
Estilo B			
Estilo C			
Estilo D			
Estilo E			

Procesos *Procesos*

Fuente: elaboración propia.

La declaración de las máquinas de estados realmente es utilizar los tipos definidos por el diseñador descrito en el punto anterior. Por ejemplo, se podría escribir de la siguiente manera.

Figura 30. **Declaración de máquina de estados**

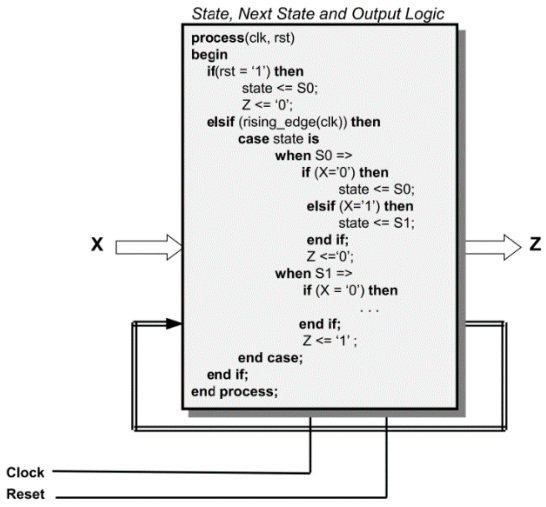
```
type FSM_states is (IDLE, START, STOP, STOP_1BIT, PARITY, SHIFT);
signal current_state: FSM_states;
```

Fuente: elaboración propia.

1.8.6.1.1. Estilo B

Se iniciará la descripción de las codificaciones con el estilo B ya que este es el tipo de codificación de máquina más simple de implementar y con la cual se puede comprender de mejor manera la idea de máquinas de estado finitas en VHDL. Básicamente, se dividen las lógicas en tres *process* diferentes. Se muestra a continuación el código de una máquina de estado.

Figura 31. **Estructura de máquina de estados codificación B**



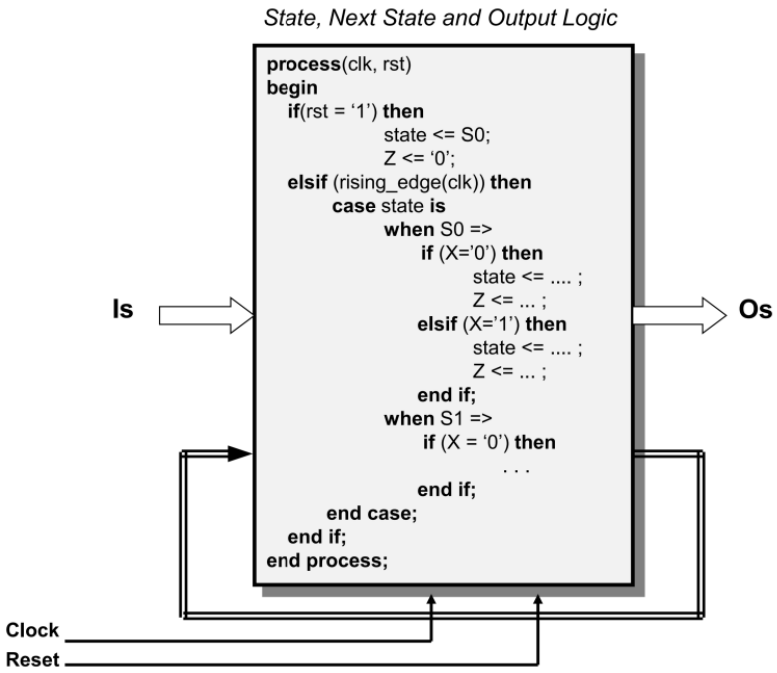
Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

Algo importante es identificar las partes de una máquina de estados finita, cuando se comprende esto, todas son iguales solamente que definidas de diferente manera. El estilo B tiene una estructura muy clara de estas partes, las cuales se definieron antes como lógicas. La lógica del estado presente se puede notar en que se coloca en la parte del "*case state is*", ya que al utilizar los *when* se está validando en qué estado está actualmente la máquina de estados.

La lógica del estado siguiente se puede notar que está adentro de estos *when,* es decir en los *if* y *elsif,* ya que es en esta parte donde se describe a qué estado se quiere pasar. La lógica del estado siguiente siempre va manejada por entradas del sistema, las cuales determinan si sigue al siguiente estado o se queda en el mismo.

Estas entradas pueden ser más de una validando con conectores lógicos todas las posibles combinaciones. Por último, se tiene la lógica de salida, estalógica se describe en este ejemplo como Z y cada vez que se asigna un valor a Z se está haciendo la lógica de salida. En este caso la lógica de salida solo depende del estado actual de la máquina de estados, siendo esta una máquina de estado con salida tipo Moore. Si se necesita una máquina de estados tipo Mealy se debe colocar las salidas dentro de los *if* de la lógica del estado siguiente, como se muestra a continuación.

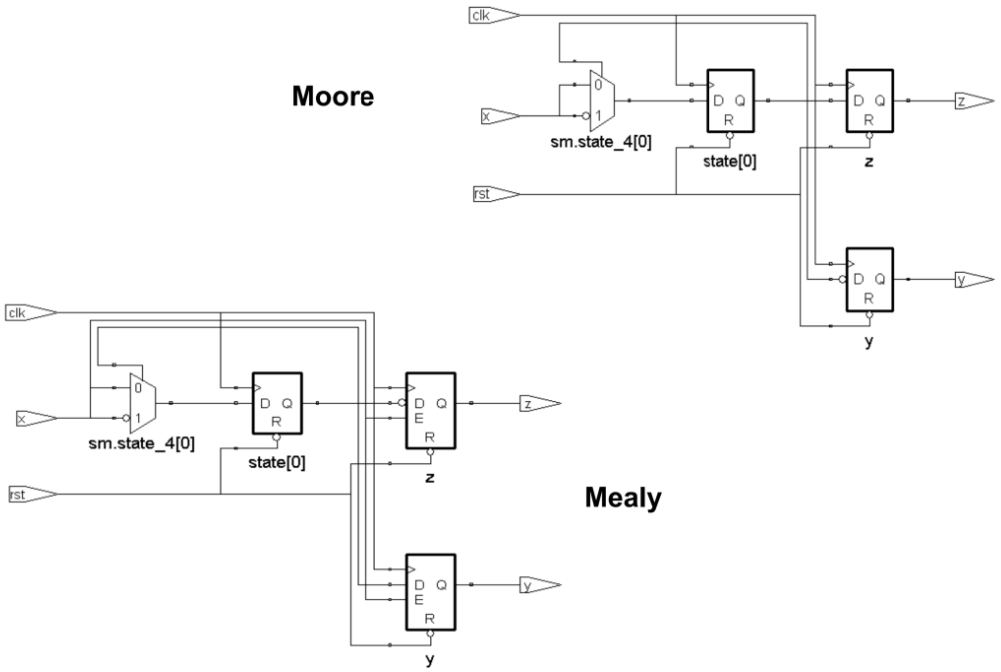
Figura 32. **Estructura máquina de estados con salida tipo Mealy**



Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

Una máquina de estados en síntesis lo que hace es llevar un proceso lógico donde, dependiendo de las entradas del sistema y señales de estado, así se manejarán los valores de señales de salida para determinar un estado de lógica concurrente y así realizar acciones específicas dependientes de este estado definido. El diagrama de hardware sintetizado se muestra a continuación.

Figura 33. **Diagrama de circuitos máquinas salida tipo Moore y Mealy**

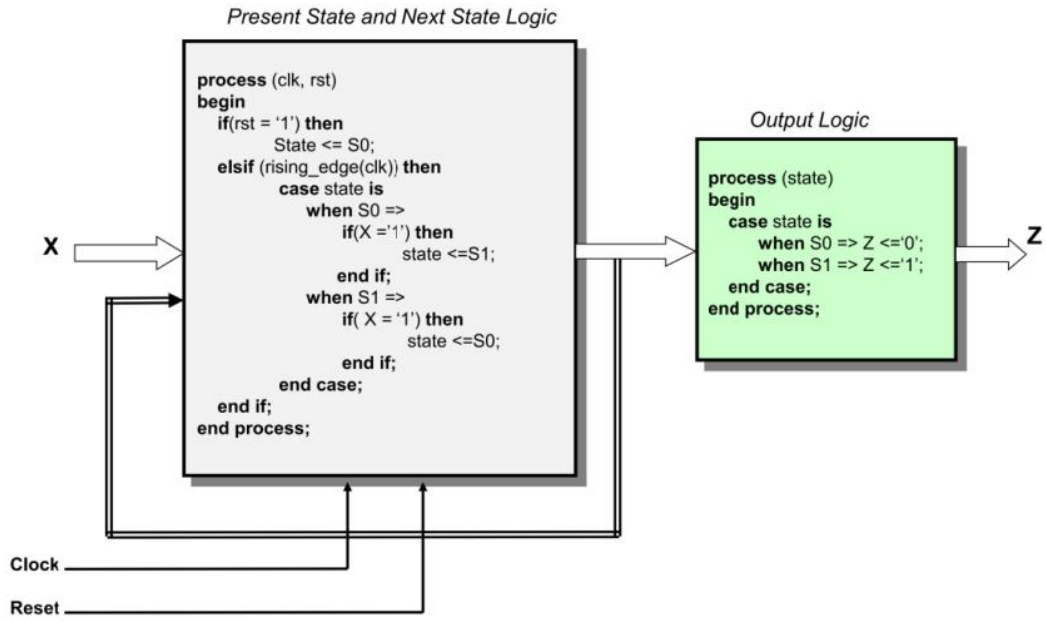


Fuente: MORALES, Iván. *Conversión de datos.*
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 10 de noviembre de 2018.

1.8.6.1.2. Estilo C

Este tipo de máquina de estados tiene la ventaja de que genera menos hardware sintetizado en una FPGA y se optimiza un ciclo de reloj para mostrar las salidas, a diferencia del estilo B. Ya no se utiliza solamente un *process,* sino que se trata que la lógica se divida en dos *process* distribuyendo la lógica del estado presente y el estado siguiente en un solo *process* y la lógica de salida en otro.

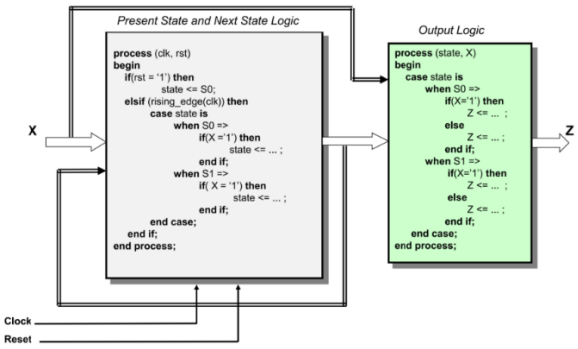
Figura 34. **Estructura de máquina de estado con codificación C**



Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

Como se puede notar, la lógica del primer *process* es la misma que la que se hizo en el estilo B con la diferencia de que no se coloca la lógica de salida. Se trata de mover la lógica de salida en otro *process,* logrando así una activación más eficiente de las salidas. En el código anterior se muestra una salida tipo Moore, ya que la salida solo depende del estado actual. Para que sea tipo Mealy se debe tomar en cuenta la misma validación que se realiza con *if* en la lógica del estado futuro. En la siguiente imagen se muestra una máquina de estados de tipo Mealy.

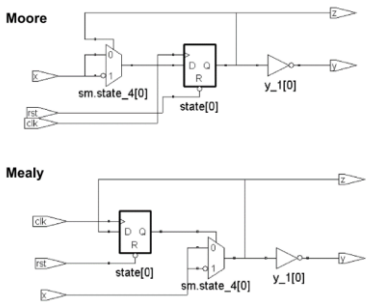
Figura 35. **Estructura de la máquina de estados con codificación y salida Mealy**



Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

El diagrama de hardware sintetizado de este tipo de máquina de estado se muestra a continuación,

Figura 36. **Diagrama de circuitos para máquina de estados con codificación C y salida Moore y Mealy**



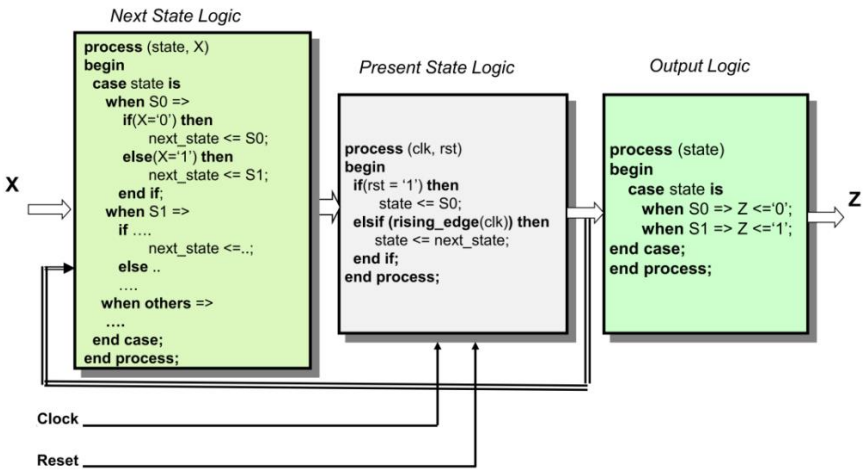
Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

Como se muestra la lógica generada en hardware es mejor que la generada con la codificación tipo B.

1.8.6.1.3. Estilo A

Esta es la codificación de máquina de estados más eficiente que puede utilizar, se optimiza cada una de las lógicas tanto en hardware generado como en tiempo de ejecución. Esta consta de tres distintos *process* uno para cada lógica, separando así todas las ejecuciones y colocando cada una de manera concurrente. Esta puede ser la mejor de las implementaciones para máquinas de estado pero de igual manera es más difícil de utilizar ya que emplea dos señales de estado para sincronizar los *process*; las entradas pueden causar conflicto al utilizar este tipo de implementación. La descripción de este tipo de máquina de estados se presenta a continuación.

Figura 37. **Estructura de máquina de estados con codificación A**



Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

Esta descripción, como se puede notar, es un tanto diferente a las anteriores, pero no deja de tener cierta similitud. El primer paso para comprender este tipo es comenzar por el *process* que se muestra en medio entre los otros dos *process*. Este es el *process* que está sincronizado con el reloj de entrada del sistema, dentro de este se tiene una estructura de reset asíncrono, cada vez que existe un flanco de subida de reloj se hace una asignación.

Como se puede recordar un *process* activa debido a un cambio de una señal, es decir, a una asignación de cualquier valor no importando si es el mismo valor u otro. Por lo tanto, al hacer esa asignación de *state_next* a *state,* se estaría activando cualquier *process* que tuviera dentro de su lista sensitiva esta señal.

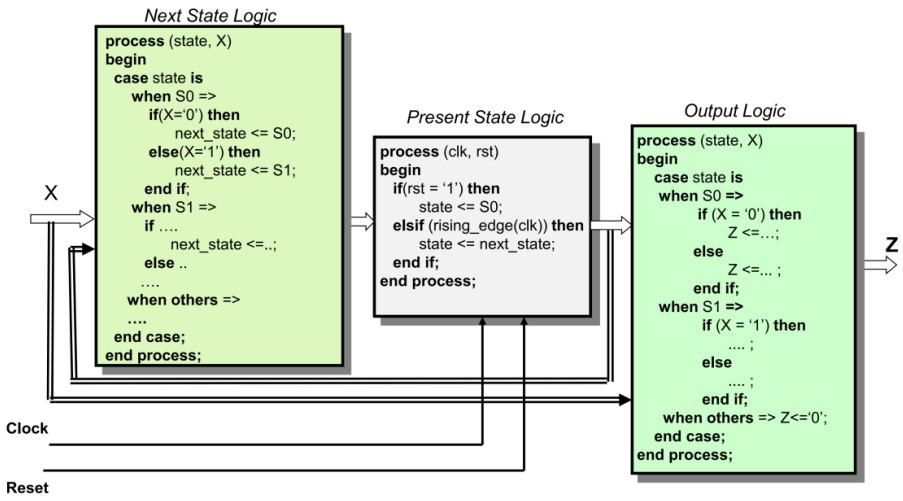
Los *process* que tienen dentro de su lista a esta señal "*state"* son los otros dos *process* que se ejecutan en paralelo al primer *process* ejecutandose al mismo tiempo. Si se centra ahora en el *process* de lado izquierdo se tiene la lógica de estado siguiente dependiente de la lógica del estado presente, el cual dependiendo de qué estado se tenga y del valor de señal de entrada va a asignar un nuevo valor a "*next_state".*

Ahora, si se centra en el *process* del lado derecho este se ejecutaría al mismo tiempo que el *process* del lado izquierdo, realizando una asignación dependiendo del valor actual de la señal "*state".* Cuando se finaliza el tiempo de ejecución de los *process* laterales y se cumple otro ciclo de reloj se vuelve a ejecutar el *process* central y se realiza la asignación del valor de "*next_state"* a "state" logrando así la finalización de la lógica del estado presente, ya que el valor temporal de "*next_state"* se asigna a "state" para volver a ser censado en

los otros dos *process.* Si las condiciones cambian el valor final de "*state"* será modificado, de lo contrario seguirá siendo el mismo.

Una gran ventaja que tiene este tipo de codificación de máquina de estado es que no se debe esperar otro ciclo de reloj para que se haga la asignación de la salida, sino que es inmediata. El ejemplo anterior muestra una salida de tipo Moore, si se desea una salida tipo Mealy se realiza la validación correspondiente resultando de la siguiente manera.

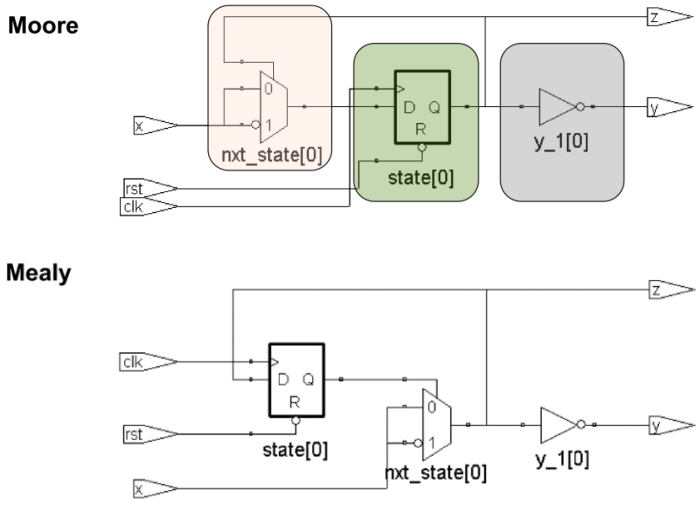
Figura 38. **Máquina de estados con codificación A tipo de salida Mealy**



Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

El diagrama de hardware sintetizado de este tipo de máquina de estado se muestra a continuación.

Figura 39. **Diagrama de circuitos de máquina de estados con codificación A salidas Moore y Mealy**



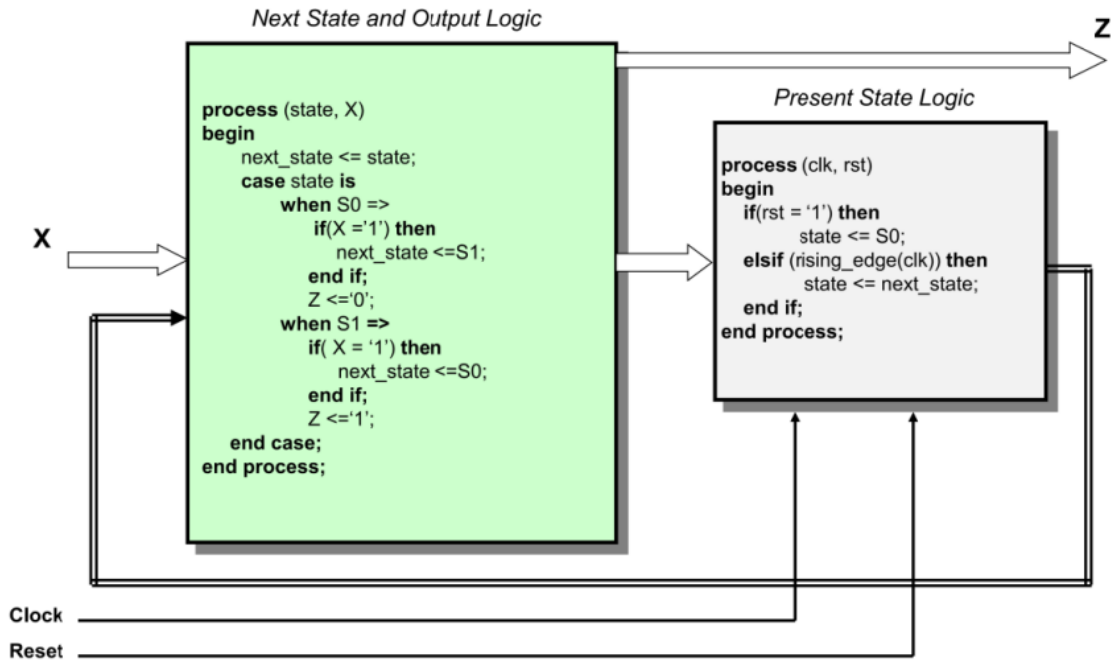
Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

Como se puede notar, este tipo de codificación simplemente genera un *flip-flop* con una lógica combinacional por ello es el tipo de máquina óptima que se puede implementar.

1.8.6.1.4. Estilo D

Este es un tipo de codificación de máquina de estado que utiliza un poco de la lógica del tipo A solamente que no genera un *process* independiente para la lógica de salida, sino que lo introduce dentro del *process* de la lógica del estado siguiente. La descripción de este tipo de máquina se muestra a continuación.

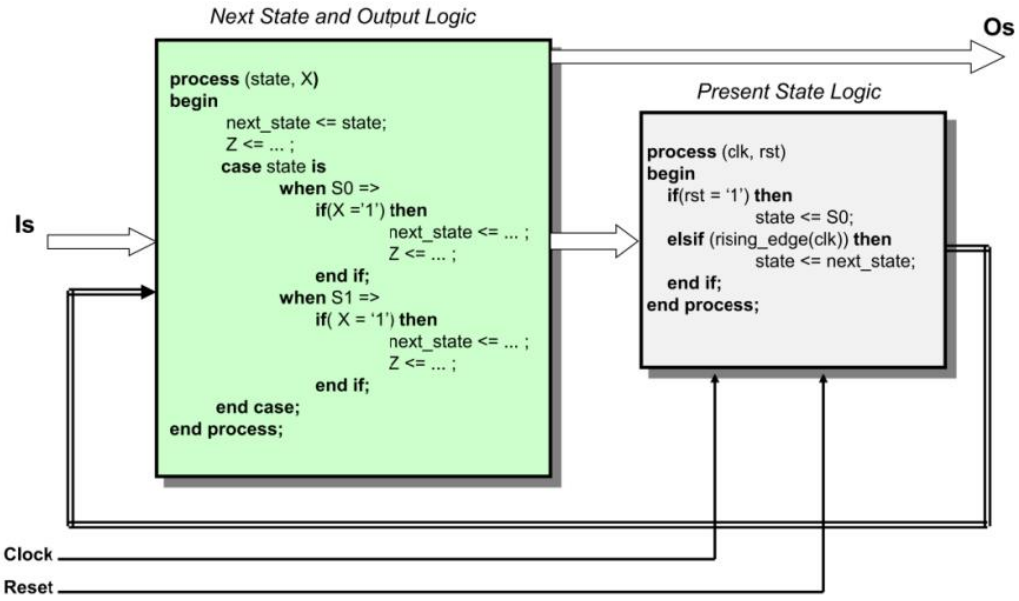
Figura 40. **Estructura máquina de estados con codificación D**



Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

Esta es la descripción para una salida de tipo Moore, haciendo las validaciones mencionadas se puede realizar una salida tipo Mealy.

Figura 41. **Estructura de máquina de estados con codificación D y salida Mealy**



Fuente: MORALES, Iván. *Conversión de datos.* http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf. Consulta: 10 de noviembre de 2018.

1.8.6.2. Tipos de salidas en una máquina de estado

Las salidas en una máquina de estado definen una parte muy importante del sistema, ya que dependiendo de estas la lógica podría funcionar de diferente manera. Los tipos de salidas pueden dividirse en dos Moore y Mealy.

1.8.6.2.1. Moore

Es un tipo de salida que depende del estado en el que se encuentra para dar valor a las salidas. La diferencia radica en que no se incluye en la parte

condicional de la lógica siguiente, solamente se incluye en la parte condicional de la lógica actual.

1.8.6.2.2. Mealy

Es el tipo de salida al que le importa tanto el estado en el que se encuentra como el estado al que se va a dirigir. Su diferencia es que sí se incluye en la parte condicional de la lógica siguiente, importando así a donde irá para dar un valor a su salida.

1.8.6.3. Generador de máquinas de estado en un lenguaje convencional

Normalmente realizar una máquina de estados con alguna codificación se vuelve una tarea muy larga y difícil de aprender, es por esto que se recomienda realizar un generador de máquina de estado en un lenguaje convencional como puede ser C, java, javascript, y otros. La idea es tener una herramienta funcional para cuando se necesite realizar una implementación en un tiempo corto, ya que en un diseño se tendrán varias máquinas de estados controlando diferentes procesos secuenciales. La herramienta debe constar de un programa que genere las máquinas de estado especificando qué tipo de codificación de máquina se requiere y también qué tipo de salida, ya sea Moore o Mealy.

1.9. Lógica modular

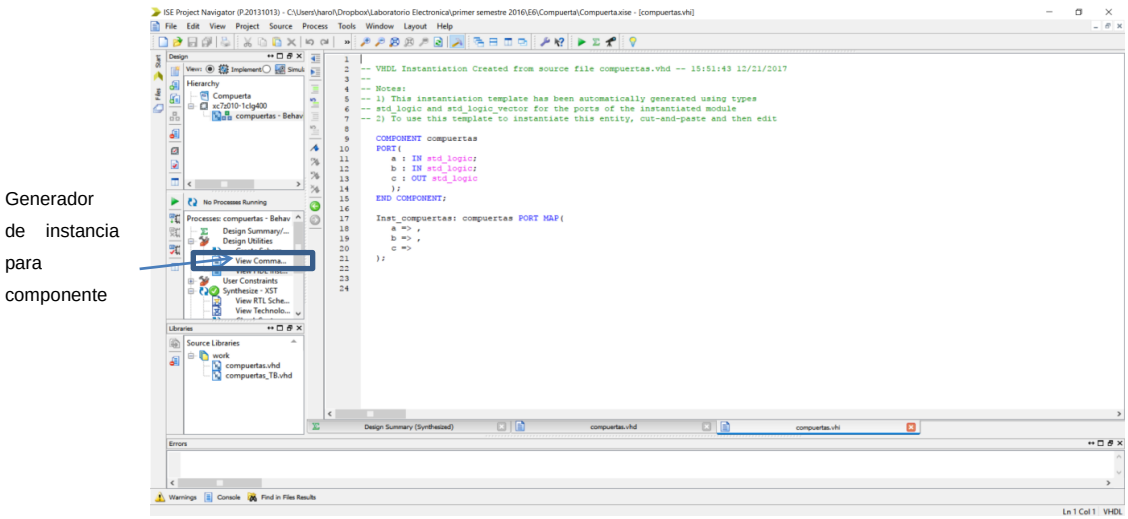
Un dicho dice “divide y vencerás”, una frase de dudoso origen atribuida al dictador y emperador romano Julio Cesar. Está idea se puede aplicar a muchos ámbitos como el diseño de circuitos. Siempre será más sencillo realizar pequeños módulos de hardware, intentando dividir el diseño para luego ser

unido y formar el diseño final que se tenía planteado. VHDL permite construir estos módulos de lógica sencilla permitiendo intercomunicarlos todos entre sí de la manera que se desee.

1.9.1. Módulo “top” y sus submódulos

A los módulos pequeños se les llamará submódulos y al módulo que contiene estos módulos se le llamará “top”. Un sub módulo se constituye de los mismos elementos de los que se compone un módulo ordinario que se ha estado tratando, solamente que se creará una instancia, es decir, una estructura de inicialización para que pueda ser definido en el módulo que lo contendrá. ¿Cómo generar una instancia de un módulo?, es una pregunta que debe surgir, una instancia siempre constará de la siguiente estructura.

Figura 42. **Indicación para la generación de instancias VHDL**



Fuente: elaboración propia.

Como se puede ver en la figura el programa ISE *Proyect Navigator* tiene una herramienta para generar la instancia de componentes.

Esta estructura consta de dos partes muy importantes, la primera es un *component* que se utiliza para definir cómo será el componente, esta es análoga al definir un nuevo tipo de señal cuando se utiliza *type*. La segunda parte es la instancia que está compuesta por un nombre de instancia luego dos puntos seguido por el nombre del módulo que se va a mapear. El *port map* define qué pines de entrada y salida se tendrán en el módulo, esta es la parte donde se declara qué se conectará con qué.

En un módulo top se pueden colocar todos los sub componente que se deseen, en la siguiente imagen se muestra cómo se vería un módulo top con algunas instancias.

Figura 43. **Estructura de la generación de módulo general**

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

entity TOP is
end TOP;

architecture Behavioral of TOP is

    COMPONENT compuertas
    PORT(
        a : IN std_logic;
        b : IN std_logic;
        c : OUT std_logic
        );
    END COMPONENT;

    COMPONENT compuertas2
    PORT(
        a : IN std_logic;
        b : IN std_logic;
        c : OUT std_logic
        );
    END COMPONENT;

begin

    Inst_compuertas: compuertas PORT MAP(
        a => ,
        b => ,
        c =>
    );


    Inst_compuertas2: compuertas2 PORT MAP(
        a => ,
        b => ,
        c =>
    );

end Behavioral;
```

Fuente: elaboración propia.

En este ejemplo no se han conectado los componentes solamente se agregaron al módulo que con tendrá todos los componentes y conexiones.

Como puede observar la parte del *component* es solamente declarativa para describir cómo será el submódulo y la parte de la instancia es ya el subcomponente por ser utilizado, con su respectiva descripción de puertos.

1.9.2. Conexiones

Hasta el momento se han realizado solo las declaraciones e instancias de los subcomponentes en el módulo *top*. A continuación, se describen los dos casos que van a ocurrir al tener conexiones en el módulo *top.*

1.9.2.1. Conexiones de un sub módulo a los puertos del módulo *TOP*

Esta es la forma más simple de conexión de componentes, simplemente para conectar un puerto del módulo *TOP* con un puerto del sub módulo se debe colocar el nombre del puerto después de la flecha que se colocó en la instancia del sub componente. Se irá completando el ejemplo de la conexión de dos compuertas.

Figura 44. **Estructura de la generación de módulo general con instancias**

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

entity TOP is
port(
    a_top,b_top : IN std_logic;
    c_top : OUT std_logic
);
end TOP;

architecture Behavioral of TOP is

    COMPONENT compuertas
    PORT(
        a : IN std_logic;
        b : IN std_logic;
        c : OUT std_logic
        );
    END COMPONENT;

    COMPONENT compuertas2
    PORT(
        a : IN std_logic;
        b : IN std_logic;
        c : OUT std_logic
        );
    END COMPONENT;

begin

    Inst_compuertas: compuertas PORT MAP(
        a => a_top,
        b => b_top,
        c =>
    );


    Inst_compuertas2: compuertas2 PORT MAP(
        a => a_top,
        b => ,
        c => c_top
    );

end Behavioral;
```

Fuente: elaboración propia.

Como se puede observar se conectaron las dos entradas del primer sub módulo. También se conectó solamente el primer puerto del segundo sub módulo al puerto del módulo *top* y puerto de salida del módulo *top* con el puerto de salida del segundo módulo. Esto se hizo así por elección propia, pueden conectarse los puertos que requiera el diseño.

1.9.2.2. Conexiones entre sub módulos

Para la conexión entre submódulos se utilizan las señales para utilizarlas como cables y así conectarlos. Ahora se procede a conectar los puertos faltantes de la instancia creando una señal que estará encargada de conectar la salida del primer sub módulo al segundo puerto del segundo sub módulo, como se muestra a continuación.

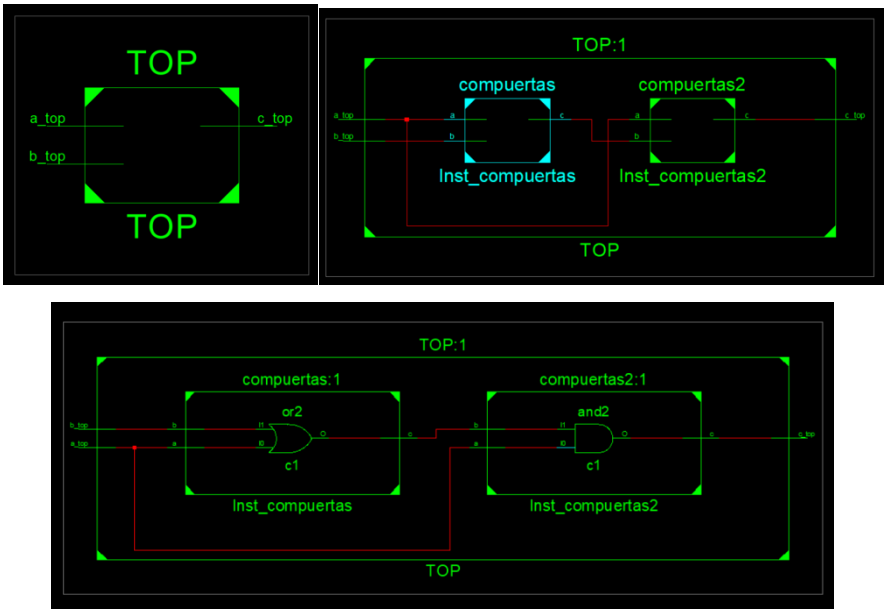
Figura 45. **Estructura de módulo general con la creación de señales**

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

entity TOP is
port(
    a_top,b_top : IN std_logic;
    c_top : OUT std_logic
);
end TOP;

architecture Behavioral of TOP is

    signal connect : std_logic;

    COMPONENT compuertas
    PORT(
        a : IN std_logic;
        b : IN std_logic;
        c : OUT std_logic
        );
    END COMPONENT;

    COMPONENT compuertas2
    PORT(
        a : IN std_logic;
        b : IN std_logic;
        c : OUT std_logic
        );
    END COMPONENT;

begin

    Inst_compuertas: compuertas PORT MAP(
        a => a_top,
        b => b_top,
        c => connect
    );


    Inst_compuertas2: compuertas2 PORT MAP(
        a => a_top,
        b => connect,
        c => c_top
    );

end Behavioral;
```

Fuente: elaboración propia.

Terminando así el diseño requerido, utilizando submódulos para generar un módulo TOP que los contiene.

Figura 46. **Diagrama de módulo general**



Fuente: elaboración propia.

1.10. Utilización de *for* en VHDL

En VHDL no existe una instrucción que genere iteraciones de procesos, es decir, el concepto de que exista una instrucción que repita ejecución de comportamientos no existe. La forma de hacer esto es la utilización de un reloj con los *process*. La instrucción *for* no hace el papel del *for* conocido por lenguajes convencionales como es C o Java, más bien es una herramienta que ayuda a generar fáciles instrucciones de declaración de hardware.

Esto quiere decir que realiza repeticiones para generar componentes o lógica de descripción que puedan ser iguales cambiando solamente en algún punto. Por ejemplo, hasta ahora se ha podido generar un componente que funcione como una compuerta *XOR.* En muchas aplicaciones es muy útil tomar esta compuerta que tiene puertos y hacer copias para hacer algún diseño de lógica de encriptación. Como se puede notar el *for* no se está utilizando para repetir el comportamiento, solamente para repetir la creación de componentes. Al final, lo único que se hace con VHDL es describir el hardware que dependiendo la lógica realizará o no algún comportamiento.

1.10.1. For-loop

Esta instrucción es utilizada para repetir secciones de bloques VHDL. Un *loop* se repetirá indefinidamente pero un *for-loop* se ejecutará un número determinado de veces a criterio del diseñador. Como se describió anteriormente el objetivo es replicar un número de veces bloques de hardware, no es en sí un ciclo como el de lenguajes de programación convencional. Algo importante es que el *for* es el único ciclo sintetizable y funciona solamente dentro de un *process* las demás sentencias sirven únicamente para simulación. La sintaxis se muestra a continuación.

Figura 47. **Estructura de la instrucción de repetición**

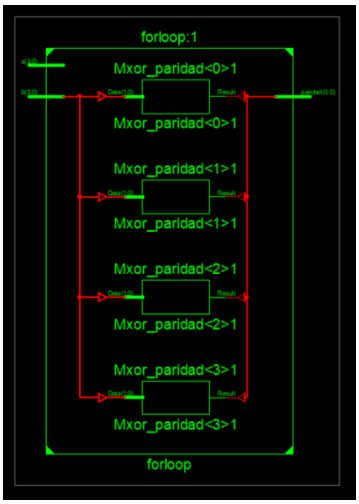
```
for <iterador> in <valor_incial> to <valor final> loop
    paridad(<iterador>) <= a(<iterador>) xor b(<iterador>);
end loop;
```

Fuente: elaboración propia.

A continuación, se presenta un ejemplo de la utilización de esta instrucción.

Figura 48. **Estructura de la generación de componentes con repeticiones**

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

entity forloop is
generic(
    bus_width : natural := 4
);
port(
    a,b     : in std_logic_vector(bus_width - 1 downto 0);
    paridad : out std_logic_vector(bus_width - 1 downto 0)


);
end forloop;

architecture Behavioral of forloop is

begin

    process(a,b)
    begin
        for i in 0 to bus_width - 1 loop
            paridad(i) <= a(i) xor b(i);
        end loop;
    end process;
end Behavioral;
```



Fuente: elaboración propia.

1.10.2. For-generate

Esta instrucción fue hecha para realizar múltiples instancias de un componente. A diferencia del *for-loop,* esta sí es una sentencia concurrente. La sintaxis se describe a continuación.

Figura 49. **Estructura de generación con iteraciones**

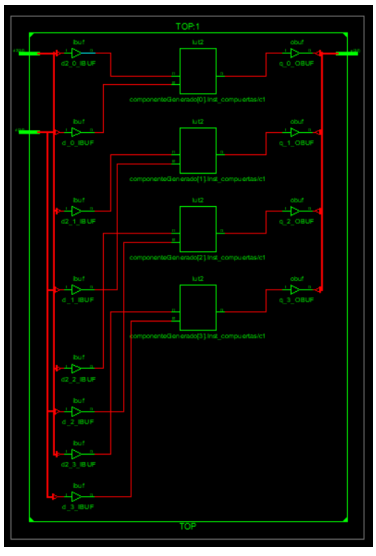
```
<etiqueta>: FOR <iterador> IN <rango> GENERATE
    BEGIN
        <setenciasConcurrentes>; --Instanciación
    END GENERATE <etiqueta>;
```

Fuente: elaboración propia.

Esta sentencia se puede utilizar ampliamente en la generación de componentes, ya que se pueden crear ciclos para crear subcomponentes.

Figura 50. **Estructura y generación de los componentes por iteraciones**

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

entity TOP is
generic(
    ancho : integer := 4
);
port(
    d  : in std_logic_vector(ancho-1 downto 0);
    d2 : in std_logic_vector(ancho-1 downto 0);
    q  : out std_logic_vector(ancho-1 downto 0)
    );
end TOP;

architecture Behavioral of TOP is

    COMPONENT compuertas
    PORT(
        a : IN std_logic;
        b : IN std_logic;
        c : OUT std_logic
        );
    END COMPONENT;

begin


    componenteGenerado: for i in 0 to ancho-1 generate
        Inst_compuertas: compuertas PORT MAP(
            a => d(i),
            b => d2(i),
            c => q(i)
        );
    end generate componenteGenerado;


end Behavioral;
```



Fuente: elaboración propia.

1.11. Estructuras extras en VHDL

En VHDL existen estructuras que normalmente no se toman en cuenta, pero el diseñador puede encontrar un uso interesante. Se describirá cada una de las estructuras que se puede utilizar.

1.11.1. Records

Los records son análogos a los *structs* del lenguaje C, su función en este lenguaje convencional es hacer una colección de tipo de variables para no solo tener un tipo asociado a una variable sino varios tipos a una variable. En VHDL existe este concepto solamente que se aplica en señales. Se puede crear una colección de tipos de señal asociados a una sola señal. La forma de hacer la declaración de un *record* se define a continuación.

Figura 51. **Estructura tipo de dato record**

```
type <nombre_tipo_record> is
   record
      <nombre_señal_tipo_1> : <tipo_1>;
      <nombre_señal_tipo_2> : <tipo_2>;
      <nombre_señal_tipo_3> : <tipo_3>;
      ...
      <nombre_señal_tipo_n> : <tipo_n>;
   end record;

signal <nombre_señal_tipo_record> : <nombre_tipo_record>;
```

Fuente: elaboración propia.

Los records como se ha dicho pueden tener una colección de muchos tipos de señal y luego implementarlos ya sea con conversiones de tipo o simplemente con los mismos tipos. A continuación, se muestra un ejemplo de cómo utilizarlo en VHDL.

Figura 52. **Descripción en VHDL de la señal tipo record**

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

entity records is
port(
    a : in std_logic_vector(7 downto 0);
    s1,s2 : out std_logic_vector(7 downto 0)

);
end records;

architecture Behavioral of records is

    type tipo_record is
        record
            slv : std_logic_vector(7 downto 0);
            int : integer range 0 to 255;
        end record;

    signal dat_record : tipo_record;

begin


    dat_record.slv <= a;
    dat_record.int <= to_integer(unsigned(a));


    s1 <= dat_record.slv;
    s2 <= std_logic_vector(to_unsigned(dat_record.int,8));


end Behavioral;
```

Fuente: elaboración propia.

1.12. Funciones

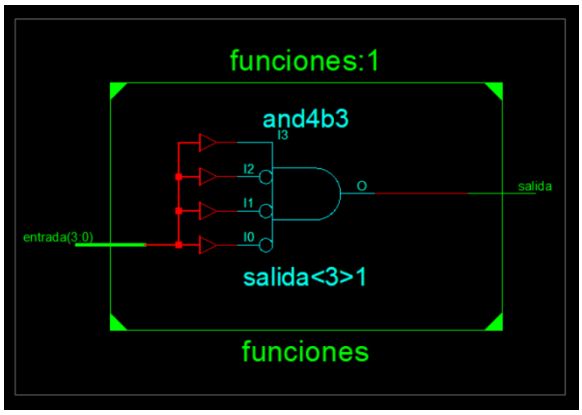
En los lenguajes de programación convencional, como es el caso de C, es posible separar porciones de código y colocarlas dentro de funciones. Esto permite evitar repeticiones innecesarias y obtener un código más simple de mantener.

Figura 53. **Descripción del uso de una función**

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

entity funciones is
port(
    entrada : in std_logic_vector(3 downto 0);
    salida  : out std_logic
);
end funciones;

architecture Behavioral of funciones is


    FUNCTION es_uno(din : std_logic_vector(3 downto 0)) RETURN std_logic IS
            VARIABLE val : std_logic;
        BEGIN
            IF din = "0001" THEN
                val := '1';
            ELSE
                val := '0';
            END IF;

            RETURN val;
    END es_uno;


begin


    salida <= es_uno(entrada);


end Behavioral;
```

Fuente: elaboración propia.

Figura 54. **Diagrama de módulo general de una función**



Fuente: elaboración propia.

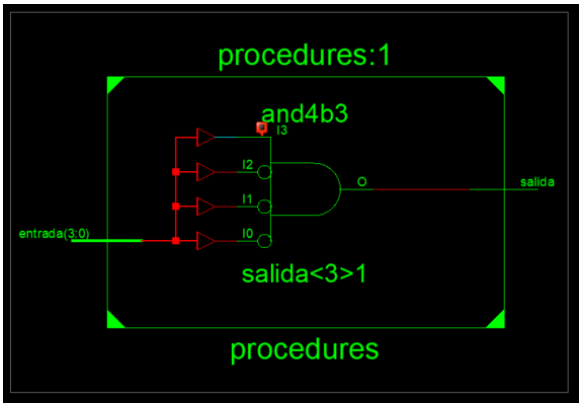
Como se puede observar la utilización de una función es solo segmentar la descripción de hardware retornando un valor. La lógica dentro de una función genera lógica concurrente es una gran ventaja para utilizar describir dicha lógica.

Algo importante es que para agregar más de un parámetro en una función se coloca un punto y coma entre ellos para separarlos.

1.12.1. Procedimientos

Un procedimiento básicamente tiene la misma funcionalidad del *function* solamente que una función retornará un valor que se define previamente mientras un procedimiento no retorna nada. La función de un procedimiento es recibir entradas y asignar salidas, solo hay que definir esto como parámetros del procedimiento. La forma de los parámetros es diferente a la de una función, ya que se debe colocar si es una señal de entrada o una de salida. De igual manera se pueden declarar variables o constantes como parámetros.

Figura 55. **Módulo general de la generación de un procedimiento**



Fuente: elaboración propia.

Figura 56. **Estructura de la generación de un procedimiento en VHDL**

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

entity procedures is
port(
    entrada : in std_logic_vector(3 downto 0);
    salida  : out std_logic
);
end procedures;

architecture Behavioral of procedures is

SIGNAL tmp : std_logic;

    PROCEDURE es_uno(signal din: in std_logic_vector(3 downto 0); signal tmp: out std_logic) IS
        VARIABLE val : std_logic;
    BEGIN
        IF din = "0001" THEN
            tmp <= '1';
        ELSE
            tmp <= '0';
        END IF;
    END es_uno;


begin


    es_uno(entrada,tmp);

    salida <= tmp;


end Behavioral;
```

Fuente: elaboración propia.

En el ejemplo se puede observar que se tiene una señal de entrada llamada “din” tipo *std_logic_vector* y una señal de salida llamada “tmp”. Utilizando la entrada se realiza una validación condicional para saber si el vector es equivalente a uno, si es así, se asigna a la salida un uno y si no es así se asigna un cero.

Es muy importante notar que, como la funcionalidad final de la descripción, tanto en la función como en el procedimiento es la misma, el diseño de hardware generado es el mismo. Esta es otra prueba de que no importa cómo sea el código de la descripción si su funcionalidad es el mismo, el diseño de hardware generado será equivalente, ya que VHDL es un lenguaje de descripción de hardware.

1.12.2. Paquetes

Un paquete consta de un conjunto de subprogramas, constantes, declaraciones, y otros, con el objetivo de implementar una colección de definiciones comunes en el diseño de hardware. Los paquetes se separan en dos partes, declaraciones y cuerpo, aunque esta última puede ser eliminada si no se definen funciones o procedimientos. A continuación, se muestra la estructura de un paquete.

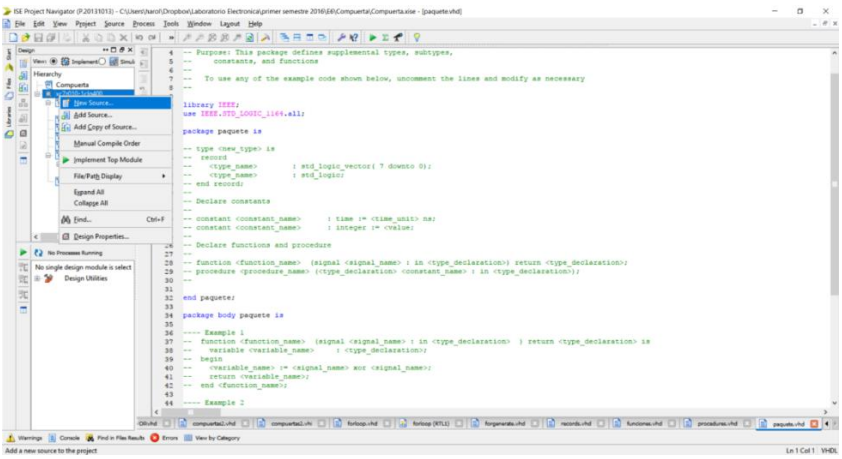
Figura 57. **Estructura de los paquetes en VHDL**

```
-- Declaración de paquete
PACKAGE nombre IS
    declaraciones
END [PACKAGE] [nombre];

-- Declaración del cuerpo
PACKAGE BODY nombre IS
    declaraciones
    subprogramas
        ...
END [PACKAGE BODY] [nombre];
```

Fuente: elaboración propia.

Para crear un paquete en el programa *ISE Proyect Navigator* debe seguir el siguiente procedimiento, hacer *click* derecho en el módulo en donde se desea implementarlo.

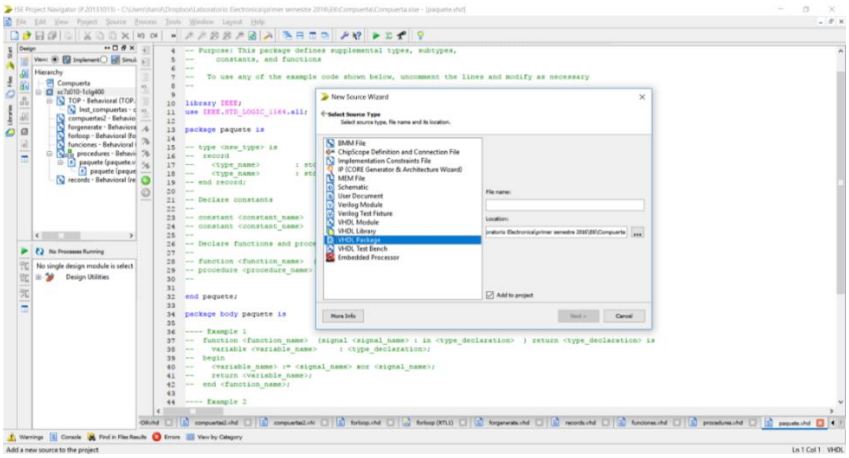
Figura 58. **Procedimiento de la generación de un paquete**



Fuente: elaboración propia.

Seleccionar *VHDL Package* y presionar el botón siguiente. Generando así el archivo de paquete con comentarios de cómo utilizar todos los elementos posibles.

Figura 59. **Procedimiento para elegir un paquete en el menú de diálogo**



Fuente: elaboración propia.

En el ejemplo anterior se describió un procedimiento para saber si un vector era equivalente a uno, este procedimiento se puede migrar a un módulo. Este módulo se crea en la librería *work,* que es la librería por defecto. El código del paquete se muestra a continuación.

Figura 60. **Estructura de un paquete en VHDL**

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

package paquete is

    procedure es_uno(signal din: in std_logic_vector(3 downto 0); signal tmp: out std_logic);

end paquete;

package body paquete is


    PROCEDURE es_uno(signal din: in std_logic_vector(3 downto 0); signal tmp: out std_logic) IS
            VARIABLE val : std_logic;
        BEGIN
            IF din = "0001" THEN
                tmp <= '1';
            ELSE
                tmp <= '0';
            END IF;
    END es_uno;

end paquete;
```

Fuente: elaboración propia.

Figura 61. **Estructura de la declaración de un procedimiento**

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

entity procedures is
port(
    entrada : in std_logic_vector(3 downto 0);
    salida  : out std_logic
);
end procedures;

architecture Behavioral of procedures is

SIGNAL tmp : std_logic;

begin


    es_uno(entrada,tmp);

    salida <= tmp;


end Behavioral;
```

Fuente: elaboración propia.

La forma para utilizar un paquete VHDL es agregarla hasta arriba utilizando la librería *work.*

Figura 62. **Incluir un paquete en otro módulo**

```
use work.paquete.ALL;
```

Fuente: elaboración propia.

1.12.3. Librerías

Hasta el momento se ha tratado con varios elementos del lenguaje, como las entidades, las arquitecturas, los paquetes, y otros. Cuando se realiza una descripción en VHDL se utilizan estos componentes, en uno o más ficheros denominados ficheros de diseño. Estos ficheros serán compilados para obtener una librería o biblioteca de diseño. La librería donde se guardan los resultados de la compilación se denomina *work*.

Una librería se compone de dos partes. La primera de las unidades primarias, que corresponderán a entidades, paquetes y archivos de configuración. La segunda parte se llama unidades secundarias, serán arquitecturas y cuerpos de paquetes. Por lo tanto, cada unidad secundaria deberá estar asociada con una unidad primaria.

Es importante establecer un orden lógico de las distintas unidades, ya que se deben declarar las dependencias necesarias para ejecutar las siguientes. La forma que toma la librería una vez compilada es muy diversa, esto se debe a que en VHDL no existe un estándar para crear bibliotecas.

1.12.3.1. Descripción de librerías

Para incluir una librería a un diseño será suficiente utilizar la palabra reservada *library* seguida del nombre de la biblioteca por utilizar. La forma de hacer visibles los elementos internos de la librería es con el uso de la sentencia *use*. En el caso de querer hacer visible todos los elementos de un paquete se puede utilizar la palabra reservada ALL.

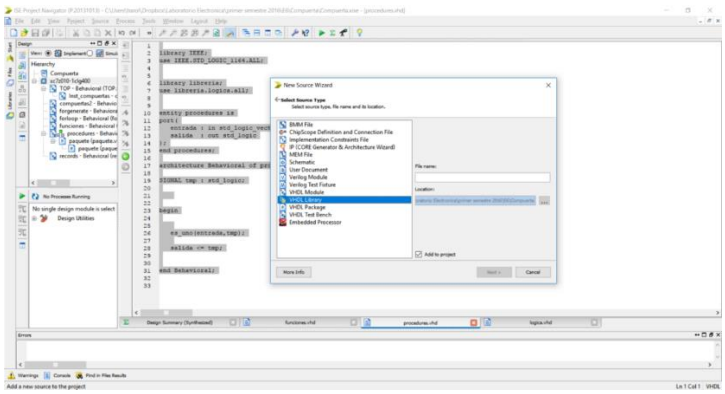
Figura 63. **Incluir librería en un módulo**

```
library libreria;
use libreria.logic.ALL;
```

Fuente: elaboración propia.

Para crear librería con el programa *ISE Proyect Navigator* se debe crear un nuevo archivo de la misma manera como se hizo un paquete solamente que se elegirá un VHDL *Library.*

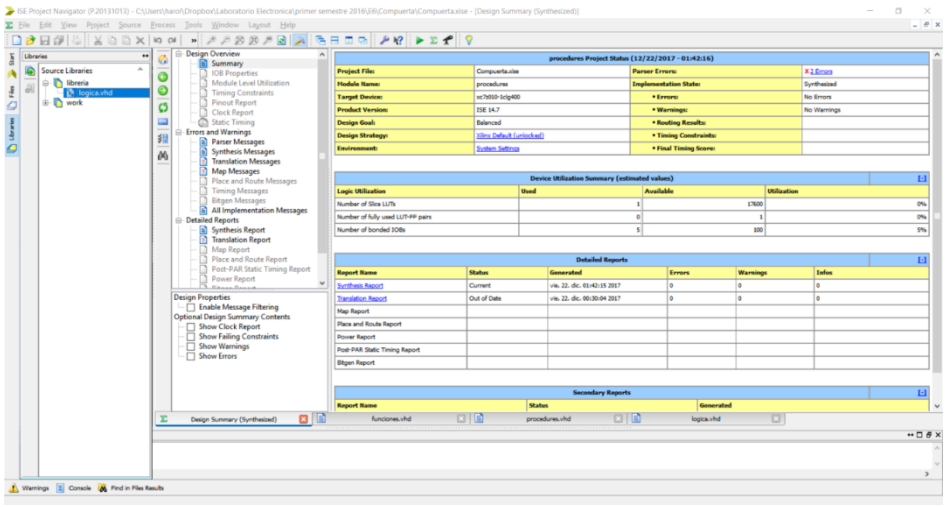
Figura 64. **Procedimiento para incluir librería**



Fuente: elaboración propia.

Luego se podrá administrar todas las librerías en la pestaña *Libray* permitiendo así agregar paquetes a las librerías de la misma forma que se hizo anteriormente. Esta es la forma de crear librerías o bibliotecas en VHDL.

Figura 65. **Indicación de la gestión de librerías**



Fuente: elaboración propia.

La forma de utilizar librerías y paquetes es como se describió anteriormente con la palabra reservada *use,* tal y como se muestra en la siguiente imagen.

Figura 66. **Estructura para la utilización de una librería**

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

library libreria;
use libreria.logica.all;

entity procedures is
port(
    entrada : in std_logic_vector(3 downto 0);
    salida  : out std_logic
);
end procedures;

architecture Behavioral of procedures is

SIGNAL tmp : std_logic;


begin

    es_uno(entrada,tmp);

    salida <= tmp;

end Behavioral;
```

Fuente: elaboración propia.

1.12.3.2. Aplicación de librerías a puertos de un módulo

Las librerías y los paquetes pueden ser útiles cuando se necesita crear puertos de un tipo diferente de los que habitualmente se utilizan. Imagine que se requiere compartir entre módulos una señal de tipo máquina de estado para censar en qué estado se está. VHDL permite crear tipos definidos por el usuario y asignarlo a un puerto. En el siguiente ejemplo se agregó en una librería un tipo definido por el usuario para colocarlo como tipo de dato en la salida otra.

Figura 67. **Descripción de la utilización de una librería como tipo de dato**

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

package logica is

    procedure es_uno(signal din: in std_logic_vector(3 downto 0); signal tmp: out std_logic);

    type MiTipo is (uno, dos, tres);

end logica;

package body logica is


    PROCEDURE es_uno(signal din: in std_logic_vector(3 downto 0); signal tmp: out std_logic) IS
            VARIABLE val : std_logic;
        BEGIN
            IF din = "0001" THEN
                tmp <= '1';
            ELSE
                tmp <= '0';
            END IF;
    END es_uno;

end logica;
```

Fuente: elaboración propia.

Figura 68. **Descripción de la utilización de una librería como tipo de dato en el módulo general**

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


library libreria;
use libreria.logica.all;


entity procedures is
port(
    entrada : in std_logic_vector(3 downto 0);
    salida  : out std_logic;
    otra    : out MiTipo
);
end procedures;

architecture Behavioral of procedures is

SIGNAL tmp : std_logic;

begin


    es_uno(entrada,tmp);

    salida <= tmp;

    otra <= uno;


end Behavioral;
```

Fuente: elaboración propia.

1.13. Definición de memorias en VHDL

Una memoria es un espacio en el cual se puede almacenar información y es análogo a un arreglo o *array* en programación convencional. En VHDL una memoria o *array* es un tipo definido por el usuario construido por tipos ya definidos, dependiendo el tipo de memoria que se declare así será su funcionamiento.

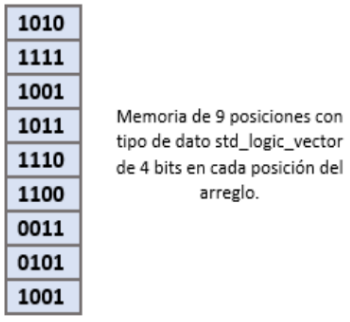
1.13.1. Tipos de memorias

En este documento se definirán tres tipos de memorias. Se nombrará a manera de tener una mejor definición.

1.13.1.1. Tipo n datos

Esta memoria se define para construir un arreglo n dimensional con un mismo tipo de dato en cada posición del arreglo. Se utiliza para almacenar tipos de datos en forma de pila. La forma de definirla se mostrará a continuación.

Figura 69. **Memoria tipo n datos**



Fuente: elaboración propia.

Figura 70. **Descripción de la memoria tipo n datos**

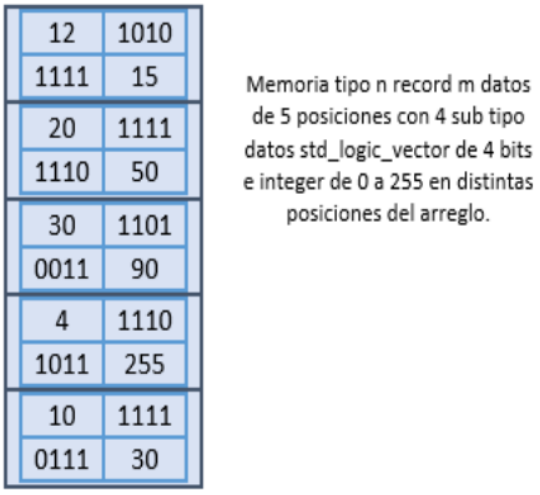
```
type tipo_mem_1 is array(0 to 5) of std_logic_vector(7 downto 0);
signal mem1 : tipo_mem_1;
```

Fuente: elaboración propia.

1.13.1.2. Tipo n records m datos

Es un arreglo de colecciones de datos. Se define para construir un arreglo n dimensional, pero con m tipos de datos en cada posición del arreglo, accediendo a cada uno por un punto. Es análogo al struct utilizado en lenguajes convencionales como el lenguaje C. La forma de definirla se muestra a continuación.

Figura 71. **Memoria tipo record**



Fuente: elaboración propia.

Figura 72. **Estructura de memoria tipo record**

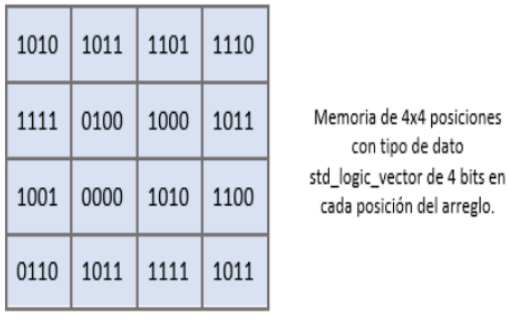
```
type record_array is
  record
     slv : std_logic_vector(7 downto 0);
     int : integer range 0 to 255;
  end record;

type tipo_mem_2 is array (integer range 0 to 5) of record_array ;
signal mem2 : tipo_mem_2;
```

Fuente: elaboración propia.

1.13.1.3. Tipo n por m datos

Esta es la declaración matricial. Se define para construir un arreglo de n por m posiciones con un mismo tipo de dato. La ventaja de este tipo a diferencia de los anteriores, es el modo de acceso matricial a un tipo de dato. A continuación se muestra la forma de definirla.

Figura 73. **Memoria matricial**



Fuente: elaboración propia.

Figura 74. **Estructura memoria tipo matricial**

```
type tipo_mem_3 is array(7 downto 0, 7 downto 0) of std_logic;
signal mem3 : tipo_mem_3;
-- 3.1) La misma declaracion que en el punto 3 solamente que tipo de dato integer en cada posición
type tipo_mem_3_1 is array(7 downto 0, 7 downto 0) of integer range 0 to 255;
signal mem3_1 : tipo_mem_3_1;
```

Fuente: elaboración propia.

1.14. Módulo VGA en VHDL

VGA es un tipo de estándar hecho para mostrar una imagen en una pantalla utilizando diferentes señales para sincronizar el tiempo y controlar el color de una imagen mostrada en la pantalla.

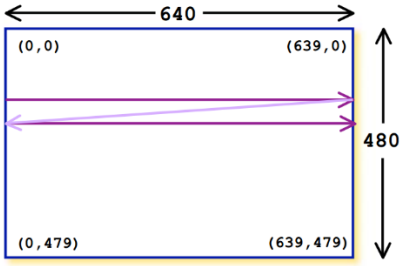
El estándar de video VGA contiene cinco señales:

- Sincronización horizontal: señal digital utilizada para la sincronización del video.
- Sincronización vertical: señal digital utilizada para la sincronización del video.
- Rojo (R): señal analógica utilizada para controlar el color.
- Verde (G): señal analógica utilizada para controlar el color.
- Azul (B): señal analógica utilizada para controlar el color.

Al cambiar los niveles analógicos de las tres señales RGB, todos los colores son producidos. El haz de electrones debe escanearse sobre la pantalla de visualización en una secuencia de líneas horizontales para generar una

imagen. La información de color RGB en la señal de video se usa para controlar la fuerza del haz de electrones.

Figura 75. **Funcionamiento de módulo VGA**



Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 12 noviembre de 2018.

El proceso de actualización de la pantalla comienza en la esquina superior izquierda y pinta 1 píxel a la vez de izquierda a derecha. Al final de la primera fila, la fila se incrementa y la dirección de la columna se restablece a la primera columna, una vez que se ha pintado toda la pantalla, el proceso de actualización comienza de nuevo.

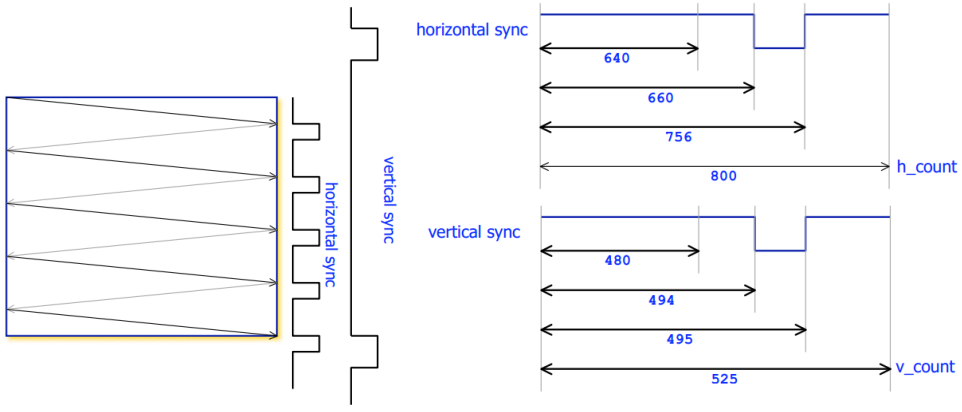
La señal de video debe redibujar toda la pantalla 60 veces por segundo para proporcionar movimiento en la imagen y reducir parpadeo, este período se llama frecuencia de actualización. La actualización de las tasas a más de 60 Hz se utiliza en monitores de PC.

Las pantallas de 640 por 480 píxeles, que tienen una frecuencia de actualización de 60 Hz, generan aproximadamente 40 ns por píxel. Un reloj de 25 MHz tiene un período aproximado de 40 ns.

La señal de sincronización vertical le dice al monitor que comience a mostrar la nueva imagen. El monitor comienza en la esquina superior izquierda con pixel (0,0). La señal de sincronización horizontal le dice al monitor que actualice otra fila al terminar de contar. Después de 480 filas de píxeles se actualiza la señal horizontal, una señal de sincronización vertical reinicia el monitor a la esquina superior izquierda esquina y el proceso continúa.

Durante el tiempo en el cual el contador se desborda y no se muestra nada, el haz está volviendo a la columna izquierda para iniciar otra sincronización horizontal, las señales RGB deberían configurarse en color negro. Lo mismo pasa cuando se termina de mostrar el último pixel, se necesita que el conteo de sincronización espere mientras el haz de electrones regrese a la posición inicial.

Figura 76. **Algoritmo de funcionamiento de VGA**



Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 12 noviembre de 2018.

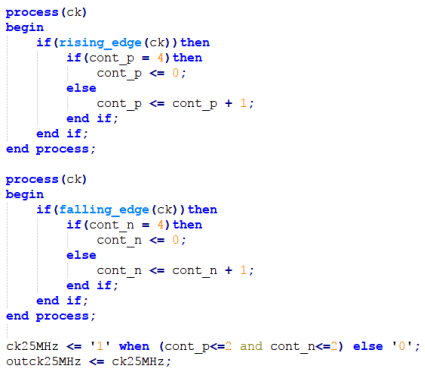
Sí la pantalla es de 640 por 480 pixeles la frecuencia necesaria para mostrar cada uno tendría que ser de 25MHz, por lo tanto, el primer paso para generar este módulo es crear un reloj a esta frecuencia. En una FPGA Zybo la frecuencia de reloj es de 125MHz si se hace la división de reloj para hacer un contador resulta en 5 ciclos de reloj para llegar a la frecuencia de 25MHz. Como se puede notar este es un número impar, por lo tanto, si se quiere crear un nuevo reloj la parte alta del ciclo quedaría diferente a la parte baja, por ejemplo, podría quedar 3 ciclos de reloj alto y 2 ciclos de reloj bajo.

Este es un problema que no se enfrenta en otras FPGA como Nexys 2, ya que su reloj interno es de 100MHz, por lo tanto, podría quedar con 5 ciclos de reloj para la parte alta y 5 para la parte baja.

Este problema se puede solucionar si no se piensa en ciclos de reloj completos sino en medio ciclo de reloj, esto haría que el conteo se multiplique por 2, resultando ya no 5 ciclos de reloj sino 10 medios ciclos de reloj siendo este ya un múltiplo de 2, por lo tanto, se necesita de 5 semiciclos para la parte alta y 5 semiciclos para la parte baja.

En una Zybo no es permitido utilizar un *if* para censar los flancos de subida y al mismo tiempo utilizar un *else* para censar los flancos de bajada en un mismo *process,* debido a esto se utilizarán dos *process* diferentes.

Figura 77. **Estructura de funcionamiento módulo de VGA**



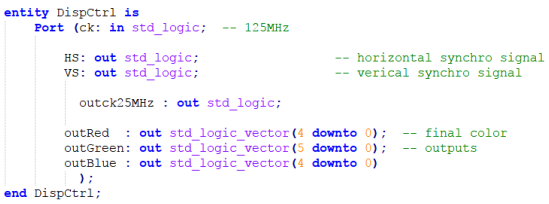
```
process(ck)
begin
    if(rising_edge(ck)) then
        if(cont_p = 4) then
            cont_p <= 0;
        else
            cont_p <= cont_p + 1;
        end if;
    end if;
end process;

process(ck)
begin
    if(falling_edge(ck)) then
        if(cont_n = 4) then
            cont_n <= 0;
        else
            cont_n <= cont_n + 1;
        end if;
    end if;
end process;

ck25MHz <= '1' when (cont_p<=2 and cont_n<=2) else '0';
outck25MHz <= ck25MHz;
```

Fuente: elaboración propia.

En la figura está la implementación de un reloj a 25MHz llevando el conteo de cada semiciclo de reloj con una estructura de selección concurrente.

La entidad correspondiente al módulo de VGA tiene una entrada de reloj y salidas como la de sincronización horizontal, sincronización vertical, vectores de salida para el color rojo, el color verde y el color azul. En este ejemplo solo se lleva una señal de conteo dependiendo en qué posición se encuentre horizontalmente mostrará un color o no.

Figura 78. **Declaración de señales de entrada y salida en el módulo VGA**



```
entity DispCtrl is
    Port (ck: in std_logic;  -- 125MHz

        HS: out std_logic;                  -- horizontal synchro signal
        VS: out std_logic;                  -- verical synchro signal

          outck25MHz : out std_logic;

        outRed  : out std_logic_vector(4 downto 0);  -- final color
        outGreen: out std_logic_vector(5 downto 0);  -- outputs
        outBlue : out std_logic_vector(4 downto 0)
          );
end DispCtrl;
```

Fuente: elaboración propia.

Las señales y constantes necesarias en el módulo se muestran en la siguiente figura.

Figura 79. **Señales y constantes en el módulo VGA**

```
-- PAL-1+HFP
-- Constantes para el módulo de sincronización
constant PAL:integer:=640;      --Linea Pixeles/Activos (pixels)
constant LAF:integer:=480;      --Frame Linea/Activa (lineas)
constant PLD: integer:=800;     --Divisor Pixel/Line
constant LFD: integer:=521;     --Divisor Line/Frame
constant HPW:integer:=96;       --Horizontal synchro Pulse Width (pixels)
constant HFP:integer:=16;       --Horizontal synchro Front Porch (pixels)
constant VPW:integer:=2;        --Verical synchro Pulse Width (lines)
constant VFP:integer:=10;       --Verical synchro Front Porch (lines)

type mem_t is array(0 to 70,0 to 70) of integer range 0 to 255;
signal mem : mem_t;

-- Señales para el módulo VGA
signal intHcnt: integer range 0 to 800-1; --PLD-1 - contador horizontal
signal intVcnt: integer range 0 to 521-1; -- LFD-1 - contador verical

signal ck25MHz: std_logic;      -- clk 25MHz

signal cont_p      : integer range 0 to 4;
signal cont_n      : integer range 0 to 4;
```

Fuente: elaboración propia.

Llevar el control de la sincronización vertical y horizontal se realiza en un *process* donde simplemente se ejecuta una serie de condicionales dependiendo del tamaño de la pantalla.

Tomar en cuenta que los contadores deben ser más grandes que las dimensiones de la pantalla para que dé tiempo justo a que el haz de electrones regrese a la posición inicial, esto sucede tanto en el contador horizontal como en el contador vertical. El segmento del diseño en donde se realiza se muestra a continuación.

Figura 80. **Proceso generación de funcionamiento de VGA**

```
syncro: process (ck25MHz)
    begin

    if ck25MHz'event and ck25MHz='1' then
        if intHcnt=PLD-1 then
            intHcnt<=0;
            if intVcnt=LFD-1 then
                intVcnt<=0;
            else
                intVcnt<=intVcnt+1;
            end if;
        else
            intHcnt<=intHcnt+1;
        end if;

        -- Generar la HS
        if intHcnt=PAL-1+HFP then
            HS<='0';
        elsif intHcnt=PAL-1+HFP+HPW then
            HS<='1';
        end if;

        -- Generar la VS
        if intVcnt=LAF-1+VFP then
            VS<='0';
        elsif intVcnt=LAF-1+VFP+VPW then
            VS<='1';
        end if;
    end if;
end process;
```

Fuente: elaboración propia.

La funcionalidad del módulo puede definirse en un último *process* que será el encargado de llevar el control de los colores que se mostrará en la salida de las distintas señales. Como se puede observar en la figura se asignan rangos dependientes del contador horizontal para generar un color específico en la salida.

Figura 81. **Proceso para validación de señales en VGA**

```
mixer: process(ck25MHz,intHcnt, intVcnt)
begin
    if intHcnt < PAL and intVcnt < LAF then

        if intHcnt<85 and intHcnt>15 then
            outRed <= (others => '1');
            outGreen <= (others => '1');
            outBlue <= (others => '1');
        elsif (intHcnt>=90 and intHcnt<155) then
            outRed <= (others => '0');
            outGreen <= (others => '1');
            outBlue <= (others => '0');
        elsif (intHcnt>=185 and intHcnt<240) then
            outRed <= (others => '0');
            outGreen <= (others => '0');
            outBlue <= (others => '1');
        else
         outRed <= (others => '0');
         outGreen <= (others => '0');
         outBlue <= (others => '0');
        end if;
    else
        outRed <= (others => '0');
        outGreen <= (others => '0');
        outBlue <= (others => '0');
    end if;

end process;
```

Fuente: elaboración propia.

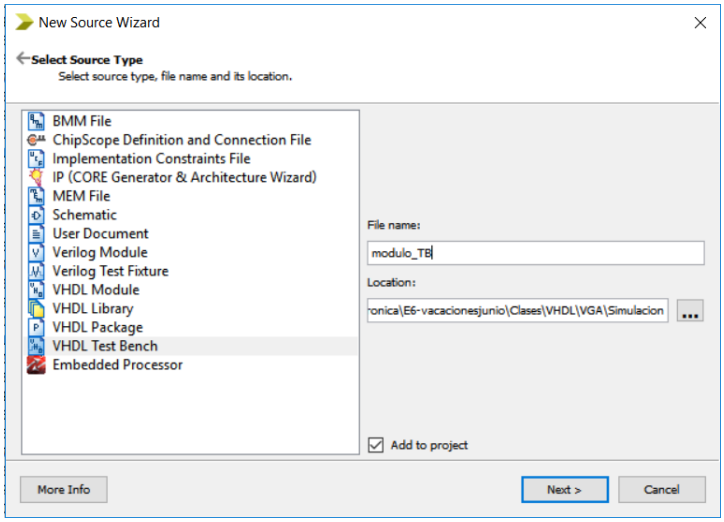
Es en este *process* donde el diseñador podría hacer cualquier funcionalidad que necesite como la lectura de una memoria donde se almacenen pixeles con su respectivo valor RGB y asignarlos a las salidas correspondientes.

1.14.1. Simulación de un diseño de hardware con la herramienta iSim

La herramienta iSim es un programa para simulación de diseños de hardware en el cual se puede mostrar la funcionalidad de los diseños generados en VHDL. Es un simulador de un analizador lógico, por lo tanto, tiene a su izquierda un panel con las señales de entrada y salida que irán siendo analizadas en líneas de tiempo, la escala de tiempo puede estar dado en nano segundos, micro segundos, mili segundos y segundos.

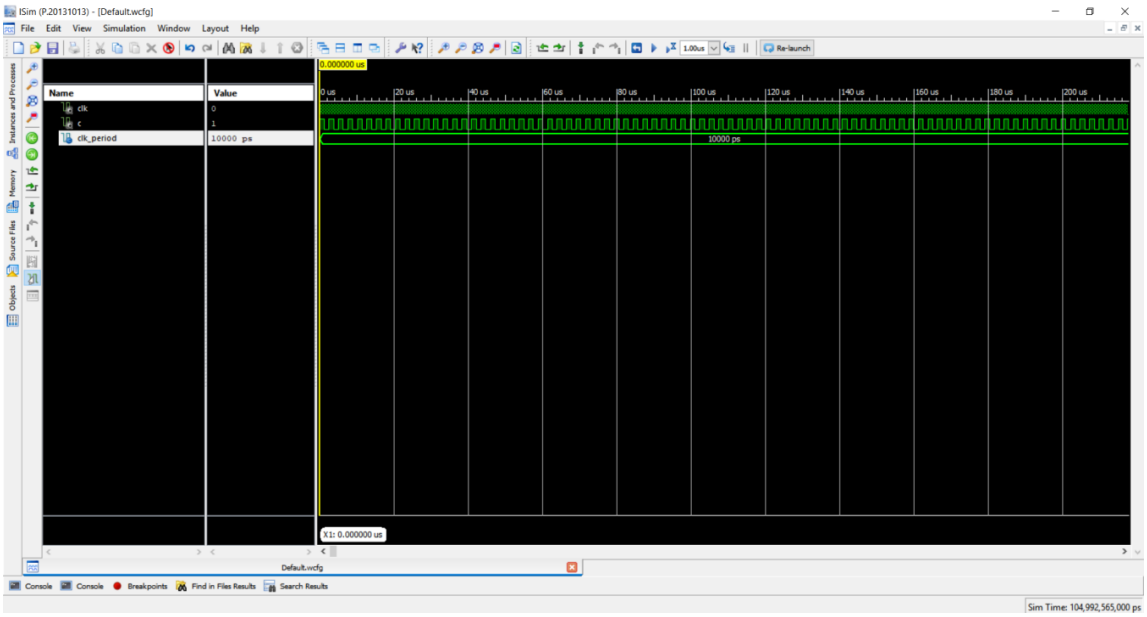
La pantalla principal del simulador se muestra a continuación. Para crear una simulación debe agregar un nuevo archivo, de la misma forma en la que se crea un módulo VHDL, solamente que eligiendo la opción de VHDL *Test Bench.* Se sugiere agregar al final del archivo "_TB" para saber que ese módulo es de simulación.

Figura 82. **Diálogo para generación de pruebas**



Fuente: elaboración propia.

Figura 83. **Simulación del módulo generado**



Fuente: elaboración propia.

Para realizar una simulación primero se tener un diseño de hardware en VHDL luego se podrá proceder a crear un Testbench. Este archivo es también de extensión vhd como cualquier otro módulo en VHDL. Un Testbench está conformado por las mismas partes que un módulo, un *entity* y un *architecture*, con la diferencia de que no se definen puertos de entrada y salida. El Testbench crea tres partes, las cuales se describirán a continuación.

La primera es la generación de un componente del módulo que se desea simular.

Figura 84. **Componente de la simulación**

```
COMPONENT modulo
PORT(
     clk : IN  std_logic;
     c : OUT  std_logic
    );
END COMPONENT;
```

Fuente: elaboración propia.

Figura 85. **Mapa de puertos del módulo simulador**

```
-- Instantiate the Unit Under Test (UUT)
uut: modulo PORT MAP (
       clk => clk,
       c => c
     );
```

Fuente: elaboración propia.

La segunda es un proceso para generar la señal de reloj de la simulación, esta se conecta a la entrada de reloj de la instanciación del módulo por simular.

Figura 86. **Reloj del módulo simulador**

```
-- Clock process definitions
clk_process :process
begin
   clk <= '0';
   wait for clk_period/2;
   clk <= '1';
   wait for clk_period/2;
end process;
```

Fuente: elaboración propia.

La tercera es un proceso para colocar las acciones requeridas en la simulación.

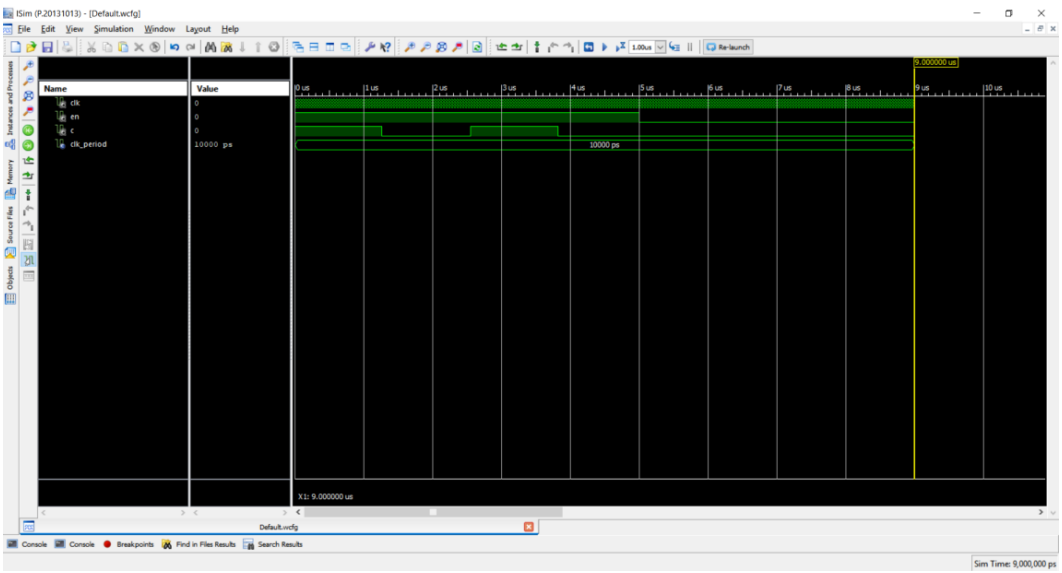
Figura 87. **Proceso principal para la simulación**

```
-- Stimulus process
stim_proc: process
begin


   wait;
end process;
```

Fuente: elaboración propia.

Un ejemplo para explicar cómo se utilizan las simulaciones podría ser la generación de un pulso de reloj dependiente de la habilitación de un pin de entrada llamado en. Es simplemente utilizar un process para contar ciclos de reloj y dependiendo de esta señal de conteo se habilitará una salida en uno o en cero.

Figura 88. **Simulación del módulo de pruebas**



Fuente: elaboración propia.

2. ARQUITECTURA INTERNA DE UNA FPGA

2.1. Características generales de una FPGA

Una FPGA, *Field Programmable Gate Array,* en español como arreglos de compuertas programable en el campo. Se define como un dispositivo semiconductor compuesto por arreglos de compuertas digitales las cuales son completamente programables.

La empresa de semiconductores Xilinx creó la idea de combinar el control del diseñador y el tiempo de desarrollo de los dispositivos lógicos programables o en sus siglas PLDs con la densidad y bajo costo de arreglos de compuertas existentes, con esto surgieron las FPGA con mejoras en respuestas de tiempo de sus predecesores programables.

Una de las mejoras que existieron al surgir estos dispositivos fue el tiempo de implementación de diseño de hardware, ya que anteriormente para crear un diseño se utilizaban los llamados *Application Specific Integrated Circuit* o ASICs que simplemente eran dispositivos creados para aplicaciones específicas.

Por ejemplo, alguna empresa necesitaba de algún circuito integrado digital para suplir alguna necesidad de procesamiento, contrataba a otra empresa que se dedicara a la creación de ASICs para su fabricación. Esta empresa tardaba un tiempo considerable en la producción de este circuito integrado cobrando mucho dinero ya que normalmente estos circuitos integrados se producen en masa para bajar el costo de producción.

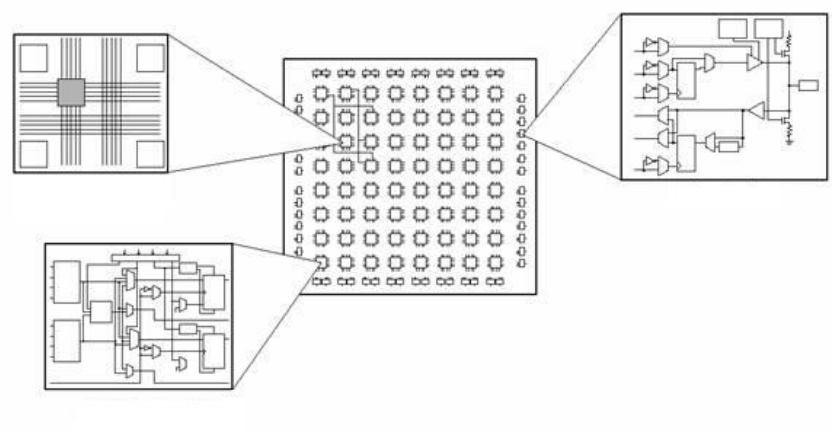
Como se puede notar el tiempo y el costo eran un factor negativo al solicitar un circuito integrado, por lo cual, al crear dispositivos lógicos programables se bajaban los costos y el tiempo de fabricación. A un inicio los PLD eran muy lentos al procesar las señales, por lo tanto, la FPGA también fue una mejora en esta característica.

Con la tecnología de FPGA se mejora sustancialmente el tiempo en respuesta a las señales, el costo de producción es mucho más bajo en comparación a los ASIC y se reduce en gran medida el tiempo de fabricación de circuitos integrados.

Hoy en día algunos de los fabricantes de estos dispositivos son Xilinx Corporation, Altera Corporation, Actel Corporation y Lattice Semiconductor. Todas las FPGA independientemente del fabricante comparten el arreglo tipo matricial de elementos lógicos como *flips-flops* y lógica combinacional, los cuales se configuran utilizando cierta lógica de programación.

Cada una de estas incluye también terminales de entrada y de salida, las cuales utilizan celdas especiales que son distintas a las celdas de elementos lógicos. La programación de las interconexiones y de los elementos lógicos puede o no ser permanente, lo cual dependerá de la tecnología utilizada.

Figura 89. **Estructura interna de una FPGA**



Fuente: *FPGA Fundamentos.* International instruments. http://www.ni.com/white-paper/6983/es/. Consulta: 5 de diciembre de 2018.

2.1.1. Bloques de CLB

Conocidos como bloque lógico programable, en una FPGA estos son la parte lógica y constituyen la mayor parte de la misma. De una manera que se pueda comprender con mayor facilidad, básicamente un bloque de CLB provee a una FPGA tanto de la lógica combinacional como la implementación de memorias, es decir, en esta parte es en donde se podrían generar una serie de lógica de compuertas o una memoria construida con *flip-flops*.

Un bloque CLB está formado por cuatro capas (*slices*), rutas de conexión para el acarreo aritmético (acarreo de entrada y acarreo de salida) y uniones a la matriz de conexiones (*switch matrix)* la cual provee el acceso a las rutas de conexiones generales y conexiones locales entre los *slices* del mismo CLB y los CLB´ vecinos.

Cada *slice* se individualiza en coordenadas x-y, si se refiere a un *slice* que se ubica en la parte inferior izquierda, esta tomará una coordenada (0,0) es decir recibirá por denominación *slice* X0 Y0 hasta llegar a la última ubicada en la parte superior derecha denominada *slice* X1 Y1.

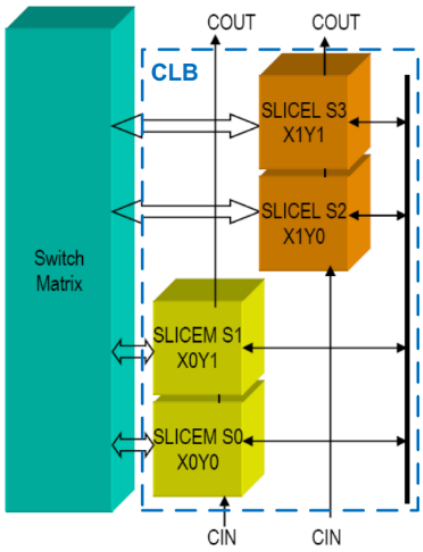
Como se puede observar la constitución de un CLB es matricial de igual manera que los elementos que conforman una FPGA, por lo tanto, es evidente que la localización de cada bloque de CLB dentro de esta también se ubica de forma matricial con coordenadas x-y.

Por ejemplo, el primer bloque CLB que se localiza en la parte inferior izquierda se identificará como CLB 00 y la última posición dependerá de la cantidad de elementos que tenga la FPGA.

Dependiendo del nivel de capa, cada *slice* recibe uno de los siguientes nombres:

- Slicem S0
- Slicem S1
- Slicel S2
- Slicel S3

Figura 90. **Niveles estructurales de un bloque CLB**



Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 12 noviembre de 2018.

2.2. *Slice* de un CLB

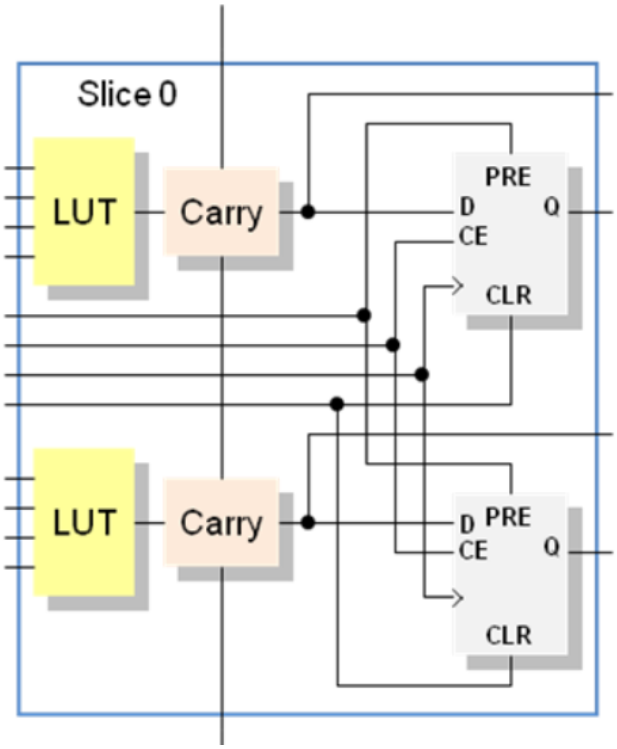
Existen dos tipos de capas (*slice)* dentro de un CLB:

- Slicel: este tipo de capa solo puede implementar funciones lógicas.
- Slicem: este tipo de capa puede implementar además de funciones lógicas de un *slice* una pequeña memoria.

En cada *slice* de una manera simplificada se pueden describir los siguientes elementos:

- Dos tablas de búsqueda (*look up tables*)
- Dos *flip flops*
- Cuatro salidas, dos combinacionales y dos con registros
- Entradas de control para los *flip flops*
- Entradas para las tablas de búsquedas
- Entrada y salida para la cadena de acarreo (*carry chain*)

Figura 91. **Diagrama de bloques de un slice**



Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 5 de diciembre de 2018.

2.2.1. Tablas de búsqueda o *Look up tables* (LUT)

La lógica combinacional es la parte de la electrónica digital donde se crean funciones lógicas en base al álgebra de Boole, creando un sistema donde, dependiendo de una serie de variables lógicas de entrada, se generará una serie de variables de lógica de salida.

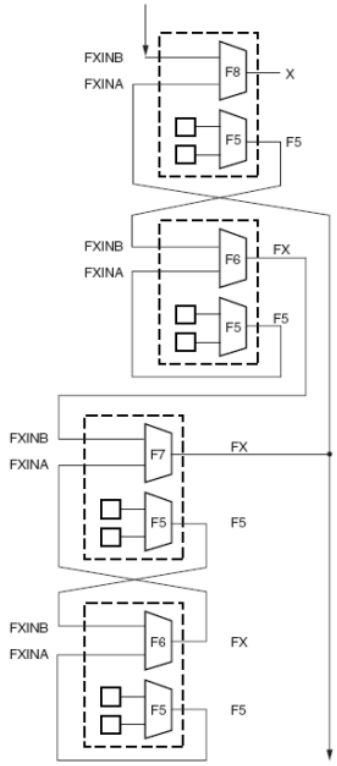
Estas funciones lógicas se implementan en una FPGA utilizando tablas de búsqueda o LUT. Estas tablas también son llamadas generadores de funciones ya que mantienen una similitud con una tabla de verdad.

Usualmente, para referirse a una LUT se utiliza la forma nxm, donde n es el total de posibles combinaciones de las entradas y m es el número de salidas, por ejemplo, 16x1 se refiere a que existen 4 entradas generando 16 posibles combinaciones y una salida para estas entradas.

Para implementar funciones lógicas con más de cuatro entradas en una LUT, se utilizan multiplexores dedicados que están distribuidos en *slice* (L o M) y en el CLB para implementar cualquier función con un mayor número de entradas. Los multiplexores más importantes son:

- F5MUX: multiplexa las salidas de las LUTs dentro del slice
- F6MUX: multiplexa las salidas de los F5MUX de un slice
- F7MUX: multiplexa las salidas de los F6MUX de un CLB
- F8MUX: multiplexa las salidas de los F7MUX de dos CLB

Figura 92. **Bloque interno de multiplexación en una FPGA**



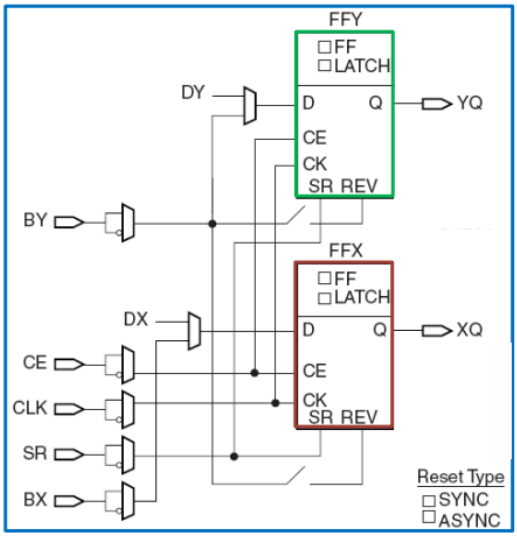
Fuente: PONG, Chu. *FPGA prototyping by VHDL examples*. p. 40.

En una LUT el retardo que se genera es constante independientemente de la función implementada, esto es muy útil en sistemas donde se utiliza alta frecuencia. Las conexiones entre los multiplexores se hacen a través de rutas de conexiones dedicadas para que estas tengan un retardo cero. Según investigaciones de la empresa Xilinx las entradas de una LUT de un bloque de CLB deben ser 6 para estas tengan un número óptimo de entradas.

2.3. *Flip-flops*

Estos son los elementos de almacenamiento. Cada SLICE posee dos de estos elementos programables y sus respectivas señales de control, estos pueden funcionar como *flip-flop* tipo D o como un *latch* transparente. Estos elementos en un SLICE se ubican en la parte superior (llamado FFY) o en la parte inferior (llamado FFX), ambos poseen un multiplexor de selección para la entrada D, permitiendo seleccionar entre la salida de la respectiva LUT, DY o DX, o una entrada externa al SLICE, llamada BY para el elemento FFY y BX para el FFX.

Figura 93. **Distribución de flip-flops en una FPGA**



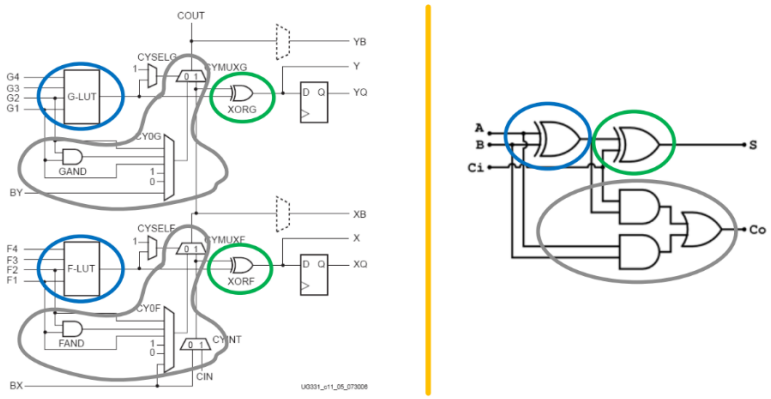
Fuente: PONG, Chu. *FPGA prototyping by VHDL examples*. p. 40.

2.3.1.1. Lógica de acarreo

Los bloques de CLB tiene lógica dedicada para el acarreo de la suma aritmética para mejorar el funcionamiento de sumadores, contadores, comparadores y otras funciones lógicas.

El bloque lógico básico de un sumador total mantiene una relación en diseño con la lógica de acarreo en un *slice*. Este diseño de lógica de acarreo es construido para que sea más eficiente que hacerla con simples LUT.

Figura 94. **Bloques de acarreo dentro de un slice**



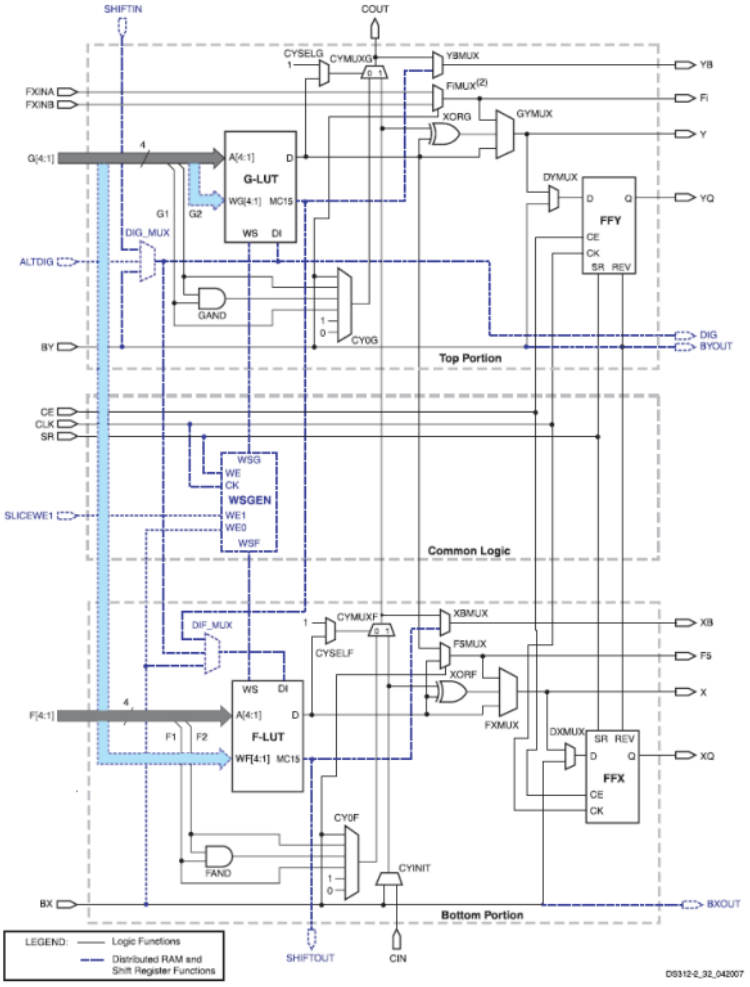
Fuente: PONG, Chu. *FPGA prototyping by VHDL examples*. p. 40.

2.4. Componentes del *SLICE*

Un *SLICE* se divide en dos los *SLICEL y los SLICEM.* Los *SLICEM* permiten implementar bloques de memoria, registros de corrimiento y lógica combinacional. En las FPGA actuales, un 50 % del total de los CLB contienen SLICEM y el otro 50 % SLICEL. Anteriormente en las FPGA el 100 % de los CLB eran SLICEM, pero esto fue cambiando a la relación actual mencionada.

Esto se debe a que el SLICEM es más genérico y físicamente ocupa más espacio que el SLICEL y esto hace que el dispositivo sea más costoso. También se realizaron amplios estudios del uso del CLB, rara vez se usaba más del 50 % de ellos como SLICEM, por lo tanto, se redujo su cantidad de relación a la actual.

Figura 95. **Distribución de slice en un bloque CLB**



Fuente: PONG, Chu. *FPGA prototyping by VHDL examples*. p. 40.

Principalmente se tienen cuatro líneas de señal de entradas llamadas F(4:1) estas entran directamente a la LUT. Como se mencionó anteriormente una LUT genera la lógica combinacional. La salida de la LUT, llamada D, tiene cuatro caminos posibles:

- Salir en forma directa o negada (XORF) por la salida X, pasando por el multiplexor FXMUX.
- Ingresar por la entrada de datos D al elemento de almacenamiento FFX, cuya salida es XQ.
- Controlar el multiplexor CYMUXF de la cadena de acarreo.
- Entrada de datos al multiplexor F5Mux para implementar funciones combinacionales de más de 4 entradas.

Los dos componentes de almacenamiento del SLICE son controlados por el mismo reloj y las mismas señales de habilitación de reloj y de reset. La salida de una mitad del SLICE puede entrar en la otra mitad del SLICE, utilizando rutas de conexión local. Esto es muy usado para sincronizar una señal asincrónica utilizando doble flip-flop, con un mínimo retardo entre ellos.

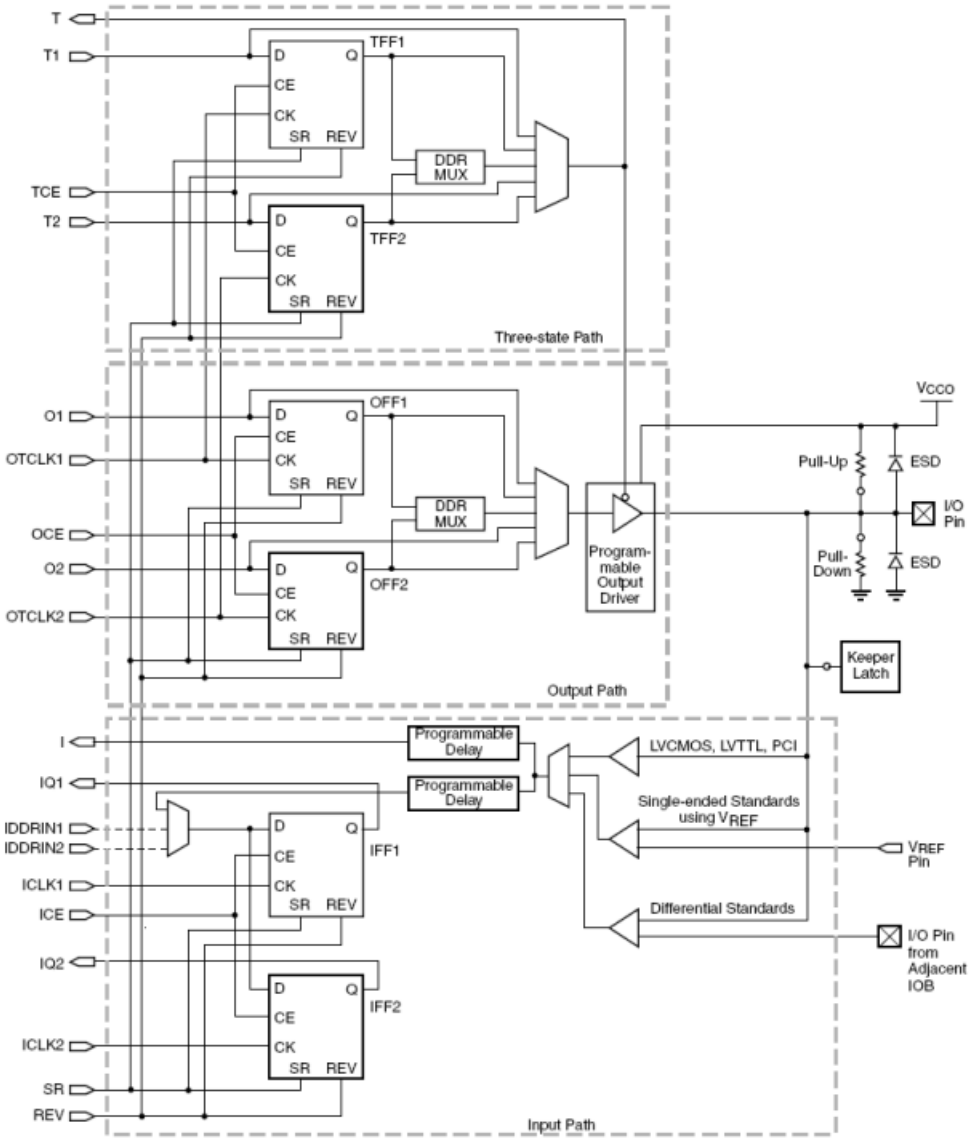
2.5. Bloques de entrada y salida

Las FPGA s contienen bloques que permiten la posibilidad de recibir y transmitir señales digitales, las cuales están preparados para usarlos con diversos rangos de voltaje, frecuencias de trabajo, estándares de señales digitales, y otros.

Existe un bloque de entrada y salida por cada terminal de entrada y salida del FPGA, por lo tanto, cada uno puede ser configurado como bloque de

entrada, salida o bidireccional. En cada uno de estos bloques existe un buffer que tienen diversas funciones configurables.

Figura 96. **Distribución de entradas y salidas en una FPGA**



Fuente: PONG, Chu. *FPGA prototyping by VHDL examples*. p. 40.

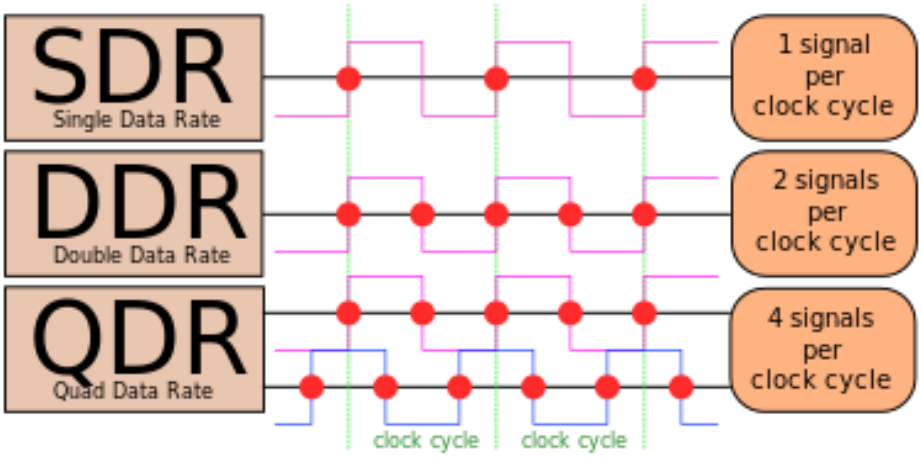
Existen tres caminos posibles en un bloque de IOB de una FPGA:

- Camino de entrada: este es el que conecta la entrada o salida de la terminal del circuito integrado a la lógica interna del FPGA. Esta conexión puede ser como se enuncia a continuación.

    - Directa, a través de un elemento de retardo
    - Con registro usando el *flip-flop* IFF1
    - Con registro usando el *flip-flop* IFF2

Las tres salidas del IOB, I, IQ1 e IQ2, se conectan al ruteo general interno del FPGA. Los *flip-flops* IFF1 e IFF2 se usan para interfaces tipo DDR (dual data rate). Esta interfaz se utiliza para que el dato de entrada sea obtenido en ambos flancos con distintos *flip-flops* y con esto el proceso será más eficiente.

Figura 97. **Tipos de interfaces para salidas en una FPGA**



Fuente: PONG, Chu. *FPGA prototyping by VHDL examples*. p. 40.

- Camino de salida: este comunica la lógica interna con la entrada y salida física de la FPGA. Este pasa por un par de *flip-flops* para transmitir datos en ambos flancos del reloj es decir se crea una interfaz DDR.

- Camino de alta impedancia: comunica la señal de control de alta impedancia con el buffer de salida. Esta ruta también incluye un par de *flip-flops* para interfaces tipo DDR. También se utiliza una realimentación de la señal la cual controla la alta impedancia del buffer de salida.

2.5.1. Buffer de entrada y salida

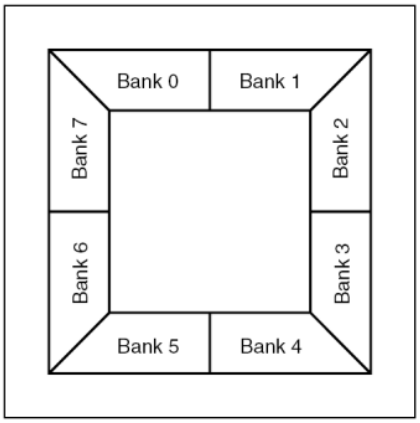
Este es un buffer de entrada y salida del IOB configurable de acuerdo con el estándar de la entrada o salida que se requiera. En el caso del buffer de salida se puede configurar para un extenso rango de estándares de entrada o salida.

También se puede configurar su corriente de salida. Cada IOB tiene un diodo de protección a Vcco y otro a GND. También dispone de resistencias tipo *pull-up* y *pull-down* configurables.

2.5.2. Bancos de entrada y salida

Se le llama banco a la agrupación de los bloques de entrada y salida. Dependiendo de la FPGA, cada banco puede tener entre 20 a 40 IOBs.

Figura 98. **Distribución de los bloques de memorias en una FPGA**

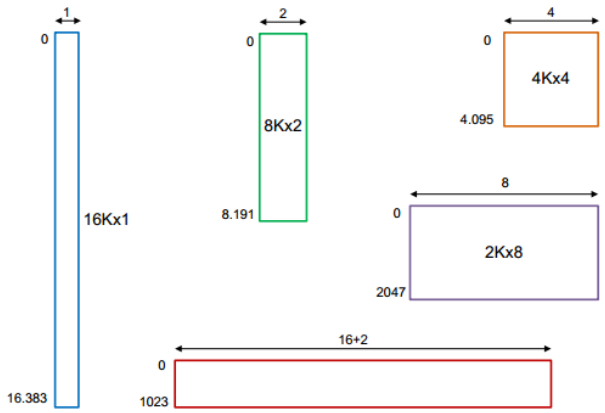


Fuente: PONG, Chu. *FPGA prototyping by VHDL examples*. p. 40.

2.5.3. Bloques de memoria RAM

Las FPGA internamente tienen bloques de memoria RAM llamados BRAM. La cantidad de estos bloques depende del tamaño del FPGA. Cada bloque de RAM contiene 18432 bits de RAM estática rápida, de los cuales 16k son dedicados para datos, y los otros 2K para bits de paridad o para algunos bits extras de los datos almacenados. Se permite conectar distintas BRAM en cascada. Los tamaños disponibles para implementar son:

Figura 99. **Distribución de memoria RAM en una FPGA**



Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 11 de noviembre de 2018.

Los BRAM son bloques configurables de acuerdo a la necesidad del diseño, es decir, el mismo bloque puede ser configurado para que funcione como RAM, ROM, FIFO, convertidor de ancho de palabra, buffers circulares y registros de corrimiento. También cada una de estas configuraciones soporta distintos anchos de la palabra de datos y distintos tamaños del bus de direcciones.

Físicamente el bloque RAM tiene dos puertos de acceso completamente independientes, llamados puerto A y puerto B. Cada puerto de memoria tiene su propia señal de reloj, habilitación de reloj y habilitación de escritura. La lectura de la memoria es sincrónica y requiere un reloj y una habilitación de reloj. Los BRAM se utilizan en las siguientes aplicaciones:

- Almacenamiento de programas para procesadores embebidos.
- Variables para cálculos matemáticos, por ejemplo, en coeficientes para filtros FIR.
- Buffers circulares.
- Registros de desplazamiento.
- Líneas de retardo.
- Contadores muy largos.
- Memorias direccionales de alto rendimiento.
- Almacenamiento de formas de onda o tablas de funciones trigonométricas para salidas de tipo *direct digital synthesis*.

Se debe recordar que en el tema de SLICE de un CLB se implementaron pequeñas memorias con los SLICEM los cuales tienen la opción de ser configurados como bloques de memoria, concatenáundolos se puede tener memorias de tamaño pequeño y de rápido acceso.

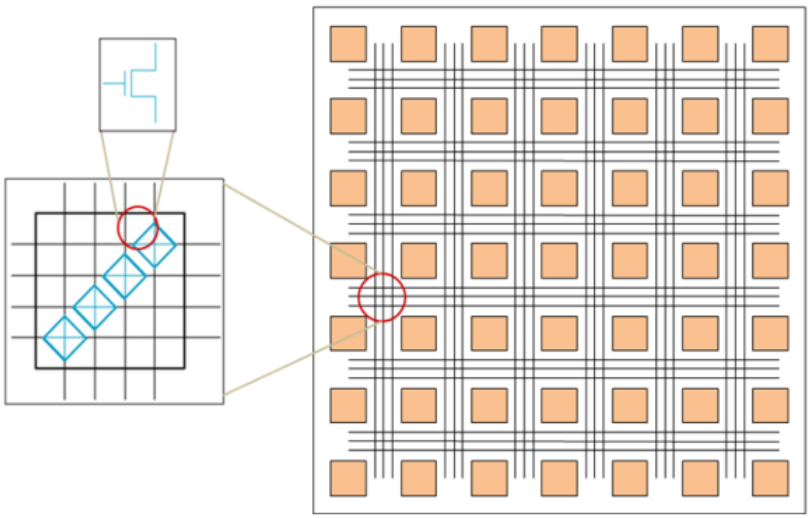
2.5.4. Bloques de multiplicación o bloque DSP

Como se sabe, las aplicaciones de procesamiento digital de señales basan sus cálculos básicamente en dos elementos, multiplicadores y sumadores. Si se necesitaran cálculos complejos, se requeriría un gran número de estos elementos, cuya implementación en LUTs resultaría muy compleja, consumiendo una gran cantidad de lógica disponible en la FPGA. Es por esto que los fabricantes de FPGA han incluido en su arquitectura elementos lógicos configurables dedicados a la multiplicación y otras FPGA más avanzadas, la multiplicación y la suma en paralelo.

2.5.5. Interconexiones de los FPGA

Adicionalmente de las celdas de lógica programable, las FPGA tienen celdas de interconexión programables. Estas celdas definen el camino o ruta a seguir por cada señal interna de la FPGA. Cada bloque de interconexión de rutas es programable a través de transistores.

Figura 100. **Matriz de interconexiones en una FPGA**

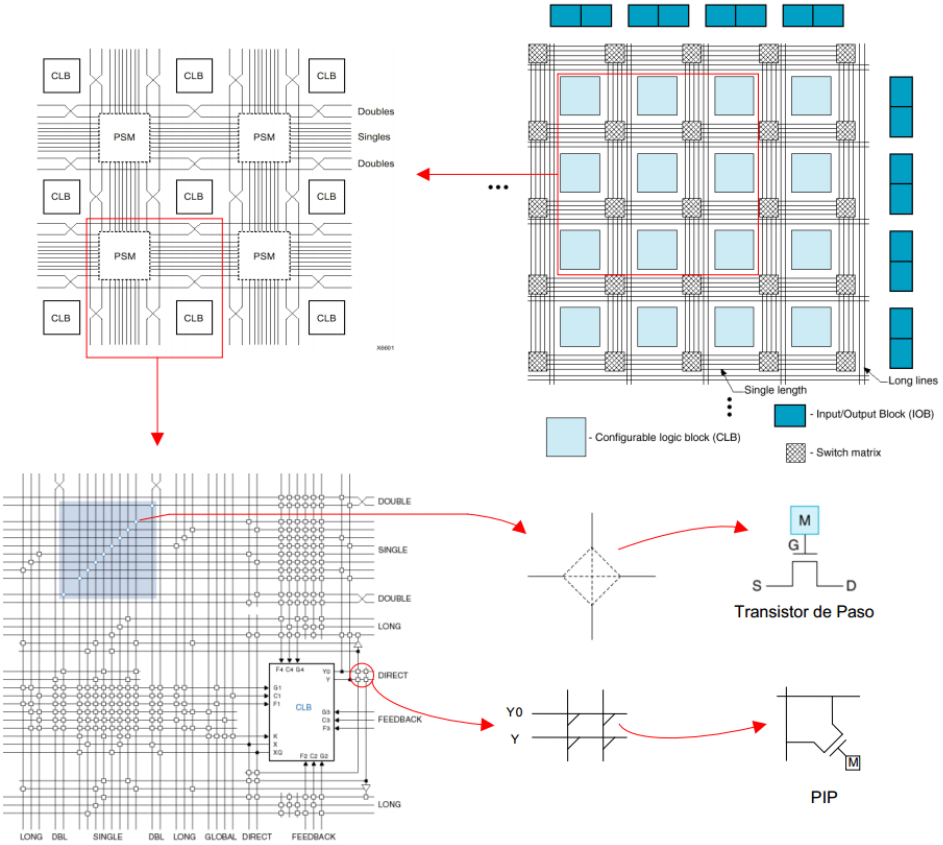


Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 11 de noviembre de 2018.

Esta programación de interconexión agrega retardos a la señal que pasa por el transistor. Si una señal debe pasar por varios elementos de interconexión la suma total de los retardos puede ser considerable y debe ser tenida en cuenta en diseños de alta frecuencia. Con objetivo de no sumar demasiados retardos en cada interconexión se crean rutas largas.

También se crean rutas cortas para crear conexiones directas que comunican un bloque lógico con sus bloques vecinos. Para señales que tienen mucha cargabilidad de salida en el sistema se crean rutas dedicadas. Hay una relación directa entre la cantidad de interconexiones y el tamaño físico de la FPGA. La herramienta encargada de generar las rutas se denomina *Place and Route.*

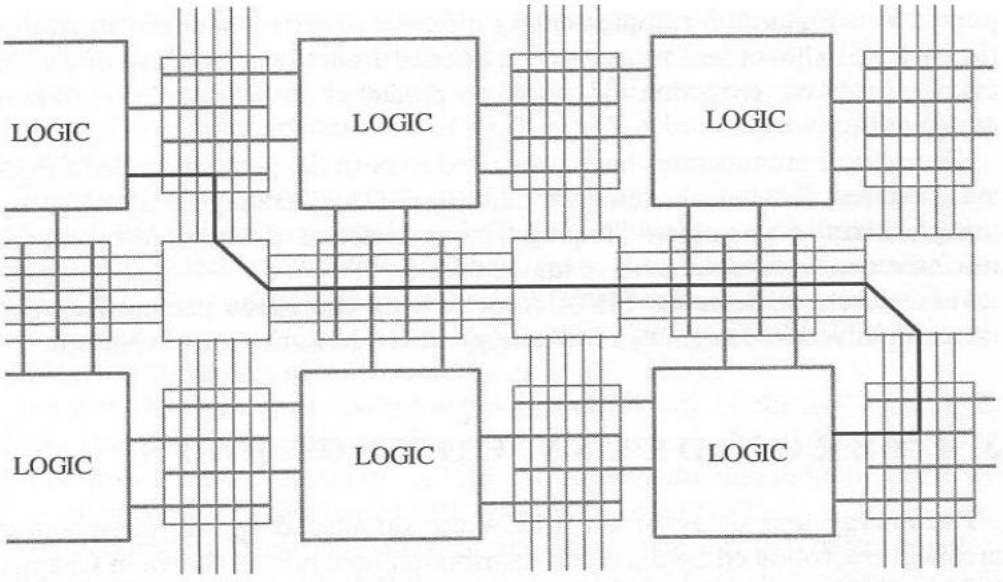
Figura 101. **Descripción detallada de las interconexiones de bloques en una FPGA**



Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 11 de noviembre de 2018.

La matriz de interconexiones programable (*Programmable Switching Matrix, PSM*) conecta las diferentes rutas dentro del FPGA a través de los transistores de paso. Las rutas largas cruzan todo el FPGA. Las rutas de conexión directa saltan sobre las matrices de interconexión para conectar directamente bloques lógicos adyacentes. Son importantes los transistores que conectan las entradas y salidas de los bloques lógicos a las rutas generales de interconexiones, llamados puntos de interconexión programables o PIP.

Figura 102. **Ejemplo de una interconexión de bloques en una FPGA**

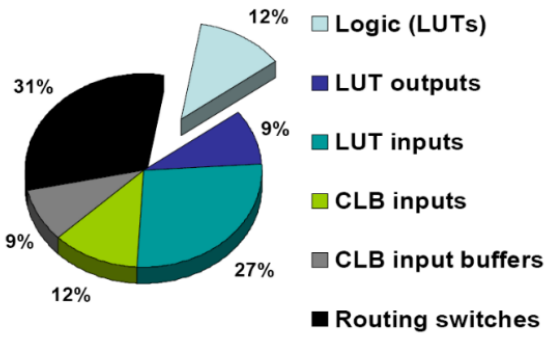


Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 11 de noviembre de 2018.

2.6. Total de recursos de una FPGA

Este gráfico muestra el porcentaje de retardo en los conectores del ruteo.

Figura 103. **Gráfica para la distribución de recursos en una FPGA**



Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 11 de noviembre de 2018.

2.6.1. Generación de reloj y su distribución

Las FPGA tienen bloques de lógica dedicada para funciones de control y generación de señales de reloj. En las FPGA de Xilinx a estos bloques se les llama genéricamente *Digital Clock Managers* DCM). La cantidad de estos bloques disponibles en una FPGA depende de su tamaño, puede haber desde 2 DCM en las FPGA más pequeños hasta 12 DCM en las FPGA grandes.

Los DCM incluyen capacidades avanzadas del reloj dentro de una red de distribución dedicada del reloj de la FPGA. Entre las funciones del DCM se pueden mencionar:

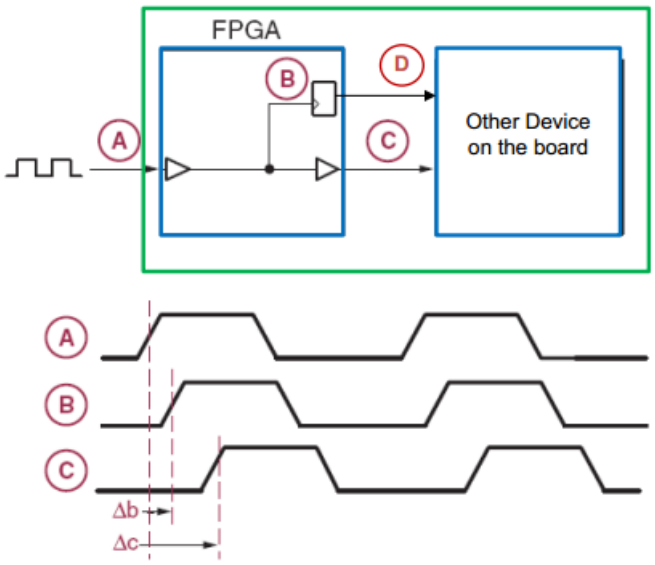
Eliminar el sesgo del reloj o *clock skew*, al conectar dos puntos dentro de una FPGA se crea una modificación en el tiempo de llegada de la señal de reloj debido a la distancia que debe recorrer, es por esto que se crea esta red

de distribución de reloj para que en ambos puntos se tenga un acceso más próximo de la señal de reloj.

- Producir corrimiento de fase de una señal de reloj
- Multiplicar o dividir la frecuencia de entrada del reloj
- Acondicionar la señal de entrada del reloj
- Amplificar de nuevo una señal de reloj

Para configurar un DCM se usa un software del fabricante del FPGA. En el caso Xilinx se llama CoreGen (Core Generator) el cual tiene una interfaz gráfica que facilita la configuración del DCM.

Figura 104. **Generación de clock en una FPGA**



Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 11 de noviembre de 2018.

3. COMUNICACIONES IMPLEMENTADAS EN UNA FPGA

3.1. Tipos de codificación (conversión digital a digital)

En un flujo de datos se utilizan códigos para la transmisión de datos de una señal digital a través de una línea de transmisión. Este proceso de codificación se elige para evitar la superposición y la distorsión de la señal, como la interferencia entre símbolos.

Las propiedades de la codificación son las siguientes:

- Se realiza para hacer que más bits se transmitan en una sola señal ya que el ancho de banda utilizado es muy reducido.
- Para un ancho de banda dado, la potencia se usa de manera más eficiente.
- La probabilidad de error es muy reducida.
- Se realiza la detección de errores.
- La densidad de potencia es muy favorable.
- Se evitan cadenas largas de unos y ceros para mantener la transparencia.

Existen tres tipos de codificación lineal:

- Unipolar
- Polar
- Bipolar

3.1.1. Codificación unipolar

Las codificaciones unipolares también se conocen como *On-Off Keying* o simplemente OOK, es la forma más simple de codificación. La presencia de pulso representa un 1 y la ausencia de pulso representa un 0.

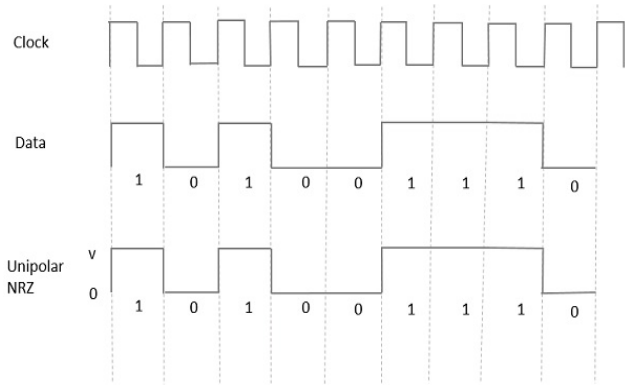
Existen dos variaciones en la codificación unipolar:

- *Unipolar Non-Return to Zero* (NRZ)
- *Return to Zero* (RZ)

3.1.1.1. Unipolar Non-Return to Zero (NRZ)

En este tipo de codificación en el cual un valor alto en los datos se representa mediante un impulso positivo, que tiene una duración igual a la duración del bit que se desea transmitir. Cuando existe un valor cero en la entrada de datos no se genera un pulso.

Figura 105. **Diagrama de codificación no retorno de cero**



Fuente: elaboración propia.

Ventajas:

- Simpleza
- Requiere un bajo ancho de banda

Desventajas:

- No se corrige ningún error.
- La presencia de componentes de baja frecuencia puede causar la caída de la señal.
- No hay un reloj presente.
- Es probable que ocurra una pérdida de sincronización (especialmente para cadenas largas de 1s y 0s).

3.1.1.2. Unipolar Return to Zero (RZ)

En este tipo de codificación que representa un pulso alto en datos pero su duración es menor que la duración del bit de los datos portransmitir. La mitad de la duración del bit permanece alta, pero inmediatamente vuelve a cero durante la mitad restante de la duración del bit.

Ventajas:

- Es muy simple
- Desventajas:
- No tiene detección de errores
- Ocupa el doble de ancho de banda que el NRZ.
- La caída de la señal se produce en los lugares donde la señal no es cero a 0 Hz.

3.1.2. Codificación polar

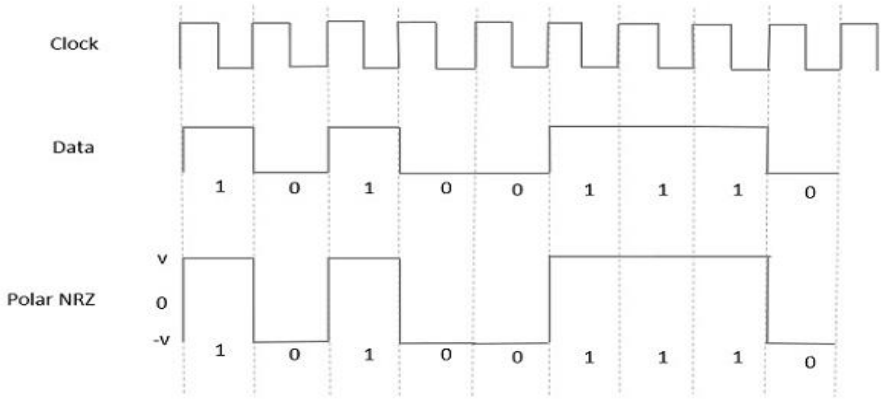
Existen dos métodos para codificaciones polares:

- Polar NRZ
- Polar RZ

3.1.2.1. Polar NRZ

Este tipo de codificación, un alto en datos se representa por un pulso positivo, mientras que un bajo en datos se representa por un pulso negativo.

Figura 106. **Diagrama de codificación polar NRZ**



Fuente: elaboración propia.

Ventajas:

- Es muy simple
- No hay presentes componentes de baja frecuencia

Desventajas:

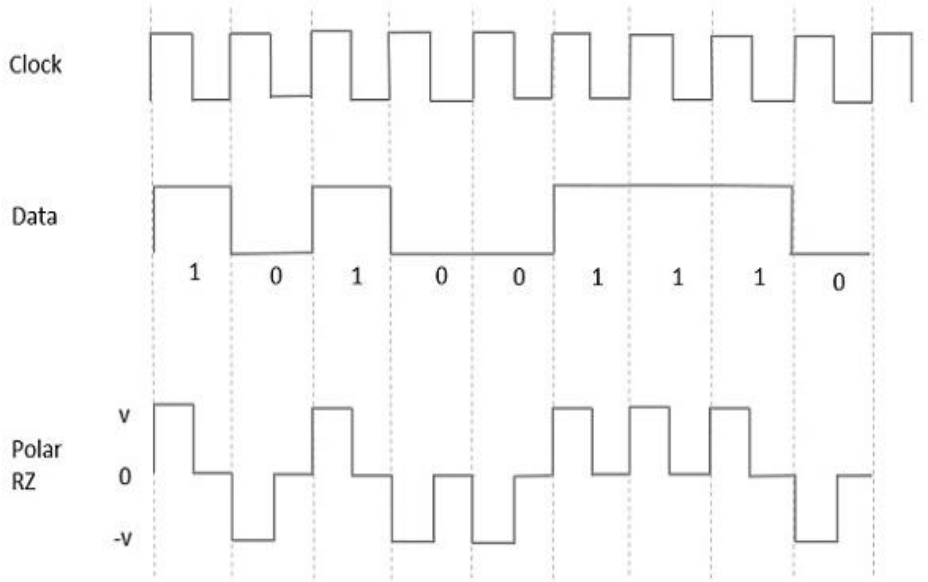
- No hay corrección de errores.

3.1.2.2. Polar RZ

En este tipo de codificación, un alto en datos su duración es menor que la duración del bit del símbolo. La mitad de la duración del bit permanece alta pero inmediatamente vuelve a cero y muestra la ausencia de pulso durante la mitad restante de la duración del bit.

Sin embargo, para una entrada baja, un pulso negativo representa los datos, y el nivel cero permanece igual para la otra mitad de la duración del bit.

Figura 107. **Diagrama de codificación polar RZ**



Fuente: elaboración propia.

Ventajas:

- Simple.
- No hay presentes componentes de baja frecuencia

Desventajas:

- No hay corrección de errores
- Ocupa el doble de ancho de banda que el polar NZ

3.1.3. Codificación bipolar

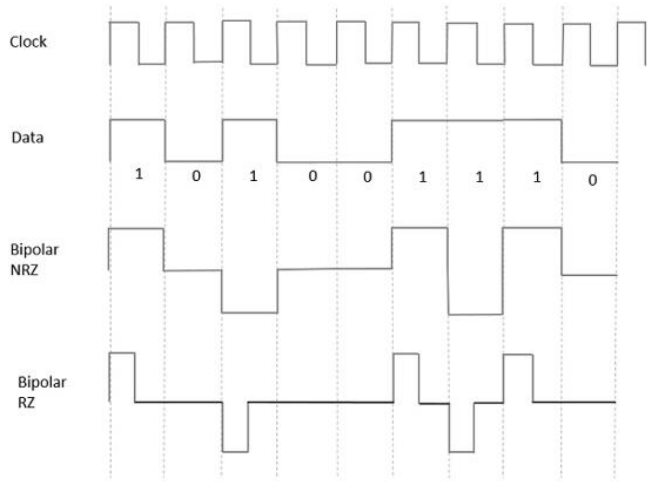
Es una técnica de codificación que tiene tres niveles de tensión, positivo, negativo y cero.

Un ejemplo de este tipo es la Inversión de Mark alternativa (AMI). Para un 1, el nivel de voltaje obtiene una transición de positivo a negativo o de negativo a positivo, que tienen una alternación de un segundo para ser de igual polaridad. A cero tendrá un nivel de voltaje cero.

Esta codificación tiene dos tipos:

- Bipolar NRZ
- Bipolar RZ

Figura 108. **Diagrama de codificación bipolar**



Fuente: elaboración propia.

En la figura se presentan las formas de onda Bipolar NRZ y RZ. La duración del pulso y la duración del bit por transmitir son iguales en el tipo NRZ, mientras que la duración del pulso es la mitad de la duración del bit por transmitir en el tipo RZ.

Ventajas:

- Es muy simple.
- No tiene componentes de baja frecuencia.
- Ocupa menos ancho de banda que una codificación unipolar y polar NRZ.
- Esta técnica es adecuada para la transmisión sobre líneas acopladas de AC, ya que la caída de señal no ocurre aquí.
- Existe una sola capacidad de detección de errores.

Desventajas:

- Largas cadenas de datos causan pérdida de sincronización.

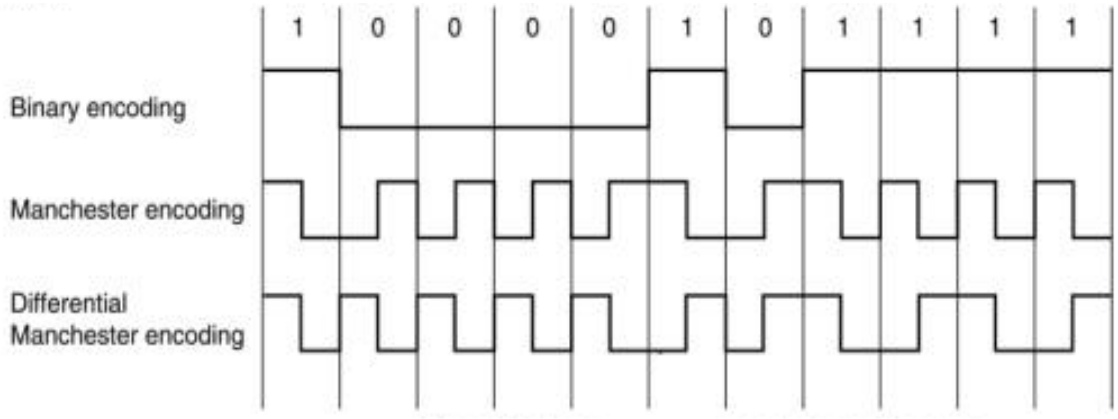
3.1.4. Codificación Manchester

La codificación de Manchester es una forma de codificación digital en la que los bits de datos se representan mediante transiciones de un estado lógico al otro. Esto es diferente del método de codificación común, en el que un bit se representa mediante un estado alto o un estado bajo.

Cuando se utiliza el código de Manchester, la longitud de cada bit de datos se establece de forma predeterminada. Esto hace que la señal se sincronice automáticamente. El estado de un bit se determina según la dirección de la transición.

La principal ventaja de la codificación de Manchester es que la señal se sincroniza a sí misma. Esto minimiza la tasa de errores y optimiza la confiabilidad. La principal desventaja es que una señal codificada en Manchester requiere que se transmitan más bits que los bits de la señal original.

Figura 109. **Diagrama de codificación Manchester**



Fuente: elaboración propia.

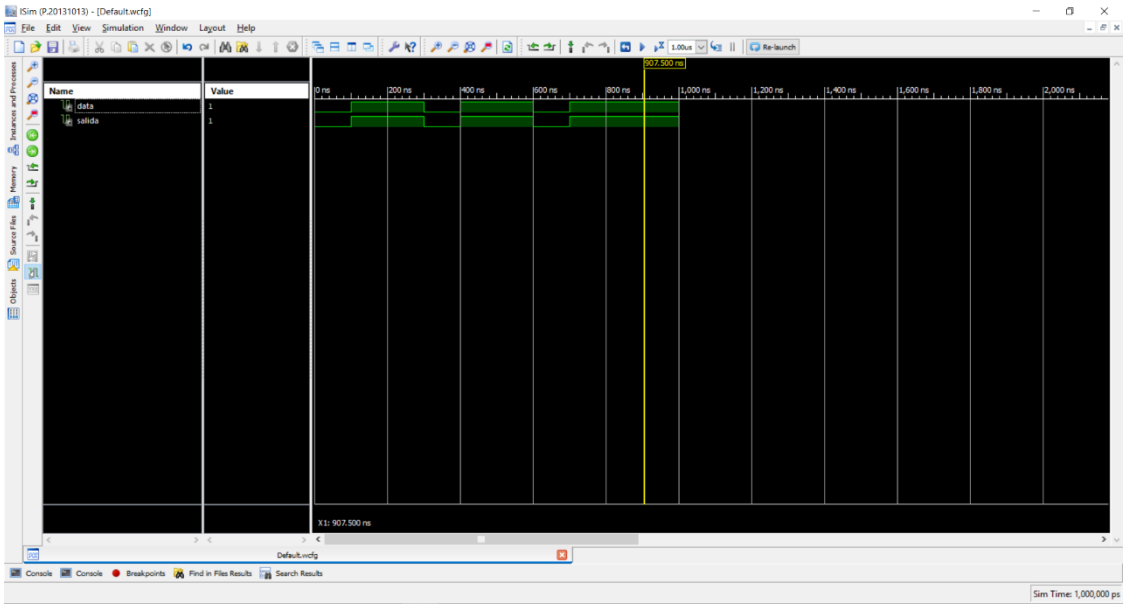
A continuación, se muestra la implementación de algunos tipos de codificación en lenguaje de descripción de hardware.

3.2. Implementación de codificación NRZ

Se implementa un diseño para una codificación NRZ y una codificación NRZ invertida. La codificación NRZ se refiere a no retorno a cero, la cual como se hizo referencia anteriormente puede dividirse en polar o no polar.

Polar hace referencia a una asignación a tensiones de +V y -V, y no polar se refiere a una asignación de tensión de + V y 0, por los valores binarios correspondientes de 0 y 1. En este caso debido a la tecnología utilizada dentro de una FPGA se toma como un tipo no polar.

Figura 110. **Simulación de codificación NRZ**



Fuente: elaboración propia.

En la simulación se muestra que simplemente se toma el valor del bit binario y se habilita una salida la cual estaría conectada a un circuito de activación para generar un valor de voltaje requerido. El código VHDL, es esencialmente una estructura de selección concurrente, el cual se muestra a detalle a continuación.

Figura 111. **Módulo de descripción codificación NRZ**

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

entity NRZ is
port(
    data : in std_logic;
    salida : out std_logic
);
end NRZ;

architecture Behavioral of NRZ is

begin

salida <= '1' when data = '1' else '0';

end Behavioral;
```

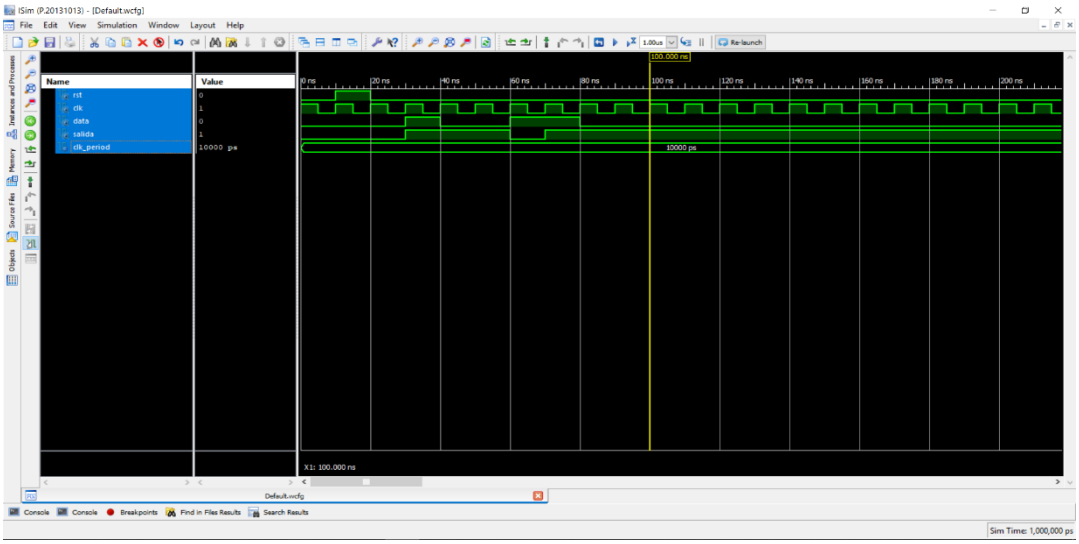
Fuente: elaboración propia.

Otra implementación que se puede realizar de codificación digital es el tipo NRZI, el cual se refiere a un NRZ invertida, para generar esta se debe seguir el siguiente algoritmo:

- Si el de entrada bit es cero: la señal se mantiene tal y como se encuentra.
- Si el bit es uno: la señal cambia de estado, es decir, si está a nivel bajo pasa a ser de nivel alto, y viceversa.

En la siguiente imagen se muestra la simulación que ilustra la implementación de este tipo de codificación.

Figura 112. **Simulación de módulo NRZ**



Fuente: elaboración propia.

Se puede observar que efectivamente al momento que se recibe un uno la señal de salida cambia su estado presente. El módulo por implementar en lenguaje VHDL se describe a continuación.

Figura 113. **Módulo de descripción NRZI**

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

entity NRZI is
port(
    rst : in std_logic;
    clk,data : in std_logic;
    salida : out std_logic
);
end NRZI;

architecture Behavioral of NRZI is
type FSM is (cero,uno);
signal state: FSM;
begin
    process(clk)
    begin
        if(rst='1') then
            state <= cero;
            salida <= '0';
        else
            if rising_edge(clk) then
                case state is
                    when cero =>
                        if data = '1' then
                            salida <= '1';
                            state <= uno;
                        else
                            salida <= '0';
                            state <= cero;
                        end if;
                    when uno =>
                        if data = '1' then
                            salida <= '0';
                            state <= cero;
                        else
                            salida <= '1';
                            state <= uno;
                        end if;
                    when others =>
                        state <= cero;
                        salida <= '0';
                end case;
            end if;
        end if;
    end process;
end Behavioral;
```

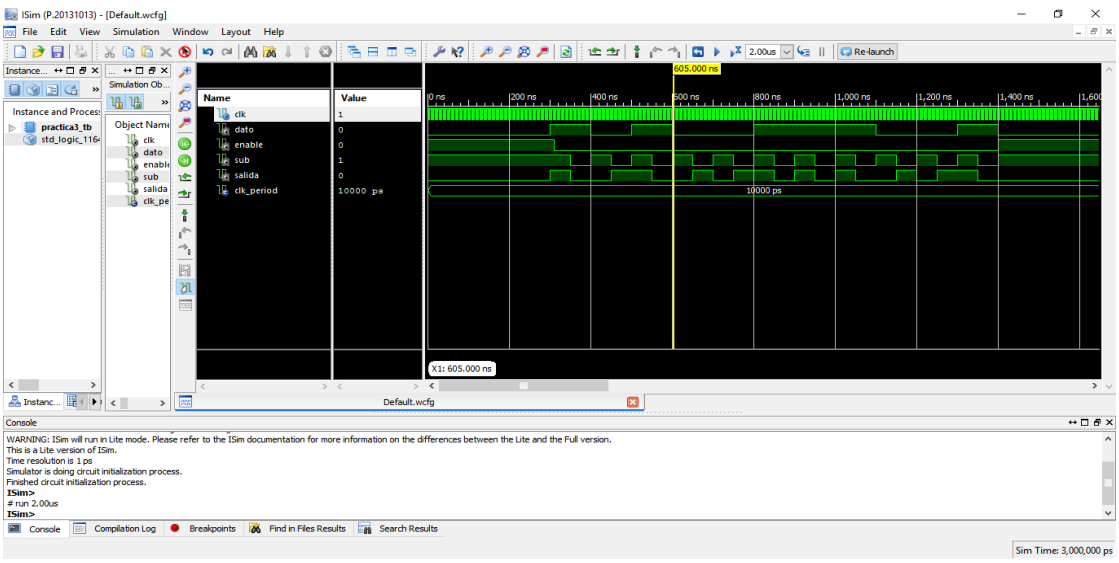
Fuente: elaboración propia.

La implementación describe una máquina de estados finita, la cual cuenta con dos estados, cuando detecta un estado de uno se tienen dos opciones, la primera es que el estado actual del bit es uno, esto ocasiona que el estado siguiente sea de cero para realizar la alteración de estado característico de este tipo de codificación. Si se detecta que se está en el estado uno y la entrada de datos es cero se mantiene en el mismo estado que se tenía. Si se está en el estado de cero se mantiene en el mismo estado que determina el dato de entrada.

3.2.1. Implementación de codificación Manchester

Por último, se presentará una codificación Manchester, anteriormente se describe que en una codificación Manchester cada bit codificado contiene una transición en la mitad del intervalo de duración del bit de entrada, es decir, una transición de negativo a positivo representa un uno y una transición de positivo a negativo representa un cero. Debido a que la alimentación de una FPGA es desde 3,3V hasta 0 V las transiciones serán en este rango de voltajes.

Figura 114. **Simulación de la codificación Manchester**



Fuente: elaboración propia.

En la implementación del módulo en lenguaje de descripción de hardware se utiliza un contador que registra cuántos ciclos de reloj interno de la FPGA deben realizarse para llegar a la mitad del bit de entrada. El valor del contador depende de la velocidad a la que se estén recibiendo los bits, en este ejemplo se eligió un conteo de cero a diez.

Se puede observar que se tienen dos procesos, el primerose utiliza simplemente para llevar el conteo de pulsos de la señal de reloj de la FPGA, es decir, cuenta todas las veces que existe una caída de flanco en el reloj interno, y así mantener de manera concurrente actualizada una señal llamada contador.

En el segundo proceso se valida en qué estado se encuentra una señal llamada sub, esta señal recibe un valor de uno cuando el contador es menor a la mitad del contador en el ejemplo ya que el conteo es diez, esta señal toma el valor de cinco en caso contrario la señal subtomará un valor de cero.

Esta señal es importante en el segundo proceso ya que determinará si la salida es cero o uno. En el caso en el que sub sea uno y el dato de entrada en ese momento sea uno la salida será cero. Si la señal sub contiene un valor de uno y el bit de entrada está en cero la salida será cero.

Esto se debe a que si la entrada es un uno la primera mitad deberá estar en alto y deberá seguirle un estado bajo cuando el conteo pase de la mitad del bit de entrada. En caso contrario y se recibe un cero en el bit de entrada, se debe empezar la primera mitad de la salida con un cero y cuando se pase a la mitad del bit de entrada deberá cambiar a un valor de cero.

Figura 115. **Módulo de descripción de una codificación Manchester**

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

entity manchester is
Port(
    clk: in std_logic;
    salida: out std_logic;
    dato: in std_logic;
    sub: inout std_logic;
    enable: in std_logic
);
end manchester;

architecture Behavioral of manchester is
signal contador: integer range 0 to 9;
begin
process(clk,enable)
begin
    if (enable = '1') then
        contador <= 0;
    else
        if (falling_edge(clk)) then
            if (contador = 9) then
                contador <= 0;
            else
                contador <= contador + 1;
            End if;
        End if;
    End if;
End Process;

process(sub, dato)
begin
    if (sub='1')then
        if dato = '1' then
            salida <= '1';
        else
            salida <= '0';
        End if;
    else
        if dato = '1' then
            salida <= '0';
        else
            salida <= '1';
        End if;
    End if;
End process;

sub <= '1' when contador < 5 else '0';

end Behavioral;
```

Fuente: elaboración propia.

3.2.2. Código Hamming

El código Hamming es algoritmo para detección y corrección de errores lineales. Este puede detectar errores hasta dos bits o corregir errores de un bit sin detectar errores. El código de paridad simple no puede corregir errores y solo puede detectar un número impar de bits por error. El código Hamming es llamado código perfecto, es decir, alcanza la tasa más alta posible para códigos con su longitud de bloque y distancia mínima de tres. Richard Hamming inventó

los códigos de Hamming en 1950 como una forma de corregir automáticamente los errores introducidos por los lectores de tarjetas perforadas.

Este es una clase de código binario lineal. Para cada número entero $r \geq 2$ hay un bloque de código con longitud $n = 2r - 1$ y longitud de mensaje $k = 2r - r - 1$. Por lo tanto, la tasa del código de Hamming es $R = k / n = 1 - r / (2r - 1)$. La matriz de verificación de paridad de un código de Hamming se construye enumerando todas las columnas de longitud r que no son cero.

Debido a la redundancia limitada que los códigos de Hamming agregan a los datos, solo pueden detectar y corregir errores cuando la tasa de errores es baja. Este es el caso en la memoria de la computadora, donde los errores de bit son extremadamente raros y los códigos de Hamming son ampliamente utilizados.

En algunos casos se usa un código de Hamming extendido que tiene un bit de paridad adicional. Los códigos Hamming extendidos alcanzan una distancia de Hamming de cuatro, que permite que el decodificador distinga entre cuando se produce un error de un bit y cuando se producen errores de dos bits. En este sentido, los códigos Hamming extendidos son de corrección de error simple y detección de doble error.

3.2.2.1. Implementación de código Hamming en una FPGA

El pequeño tamaño de los transistores o condensadores, combinado con factores externos como los efectos de los rayos cósmicos, ocasiona errores ocasionales en la información almacenada en memorias RAM grandes y densos, particularmente aquellas que son dinámicas. Estos errores se pueden

detectar y corregir empleando la detección de errores y la corrección de códigos. El esquema de detección de errores más común es el bit de paridad. Se genera un bit de paridad y se almacena junto con la palabra de datos en la memoria.

La paridad de la palabra se comprueba después de leer la palabra de la memoria. Se acepta la palabra si la paridad de los bits leídos es correcta. Si la paridad de los bits leídos es incorrecta, se detecta un error, pero no se puede corregir. Un código de corrección de errores utiliza varios bits de comprobación de paridad que se almacenan con la palabra de datos en la memoria.

Cada bit de verificación es un bit de paridad para un grupo de bits en la palabra de datos. Cuando se lee la palabra de la memoria, se evalúa la paridad de cada grupo, incluido el bit de verificación. Si la paridad es correcta para todos los grupos, significa que no se ha producido ningún error detectable. Si uno o más de los valores de paridad recién generados son incorrectos, se obtiene un patrón único llamado síndrome que puede identificar qué bit está en error.

Se produce un solo error cuando un bit cambia de valor de 1 a 0 o de 0 a 1 mientras está almacenado o si cambia erróneamente durante una operación de lectura o escritura. Si se identifica el bit específico en error, entonces el error se puede corregir complementando el bit erróneo.

Los tipos más comunes de códigos de corrección de errores utilizados en RAM se basan en los códigos diseñados por R. W. Hamming. En el código de Hamming, se agregan k bits de paridad a una palabra de datos de n bits, formando una nueva palabra de n+k bits. Las posiciones de los bits están numeradas en secuencia de 1 a n+k. Las posiciones numeradas con potencias

de dos están reservadas para los bits de paridad. Los bits restantes son los bits de datos. El código se puede utilizar con palabras de cualquier longitud. Se mostrará su operación con una palabra de datos de ocho bits. Considerando, la palabra de datos de 8 bits 11000100. Se incluyen cuatro bits de paridad en esta palabra y se organizan los 12 bits de la siguiente manera:

Figura 116. **Ejemplo de bits de paridad**

1	2	3	4	5	6	7	8	9	10	11	12
P_1	P_2	1	P_4	1	0	0	P_8	0	1	0	0

Fuente: elaboración propia.

Los 4 bits de paridad P1 a P8 están en las posiciones 1, 2, 4 y 8, respectivamente. Los 8 bits de la palabra de datos están en las posiciones restantes. Cada bit de paridad se calcula de la siguiente manera:

Figura 117. **Asignación de operaciones para calcular de bit de paridad**

$$P_1 = \text{XOR}(3, 5, 7, 9, 11) = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$$
$$P_2 = \text{XOR}(3, 6, 7, 10, 11) = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$
$$P_4 = \text{XOR}(5, 6, 7, 12) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$
$$P_8 = \text{XOR}(9, 10, 11, 12) = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

Fuente: elaboración propia.

Se debe tomar en cuenta que la operación OR exclusiva o XOR realiza una función impar. Es igual a 1 para un número impar de unos y a 0 para un

número par de unos. Por lo tanto, cada bit de paridad se establece de modo que el número total de 1 en las posiciones marcadas, incluido el bit de paridad, sea siempre equitativo. La palabra de datos de 8 bits se escribe en la memoria junto con los 4 bits de paridad como una palabra compuesta de 12 bits. Sustituyendo los 4 bits de paridad en sus posiciones correctas, se obtiene la palabra compuesta de 12 bits, desde la posición 1 hasta la posición 12:

Figura 118. **Palabra compuesta para código Hamming**

1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	1	0	0	1	0	1	0	0

Fuente: elaboración propia.

Cuando los 12 bits se leen de la memoria o transmisión de datos, se revisan nuevamente para detectar errores. La paridad de la palabra se verifica en los mismos grupos de bits, incluidos sus bits de paridad. Los cuatro bits de verificación se evalúan de la siguiente manera:

Figura 119. **Bits de verificación en código Hamming**

$$C_1 = \text{XOR}(1, 3, 5, 7, 9, 11)$$
$$C_2 = \text{XOR}(2, 3, 6, 7, 10,11)$$
$$C_4 = \text{XOR}(4, 5, 6, 7, 12)$$
$$C_8 = \text{XOR}(8, 9, 10, 11, 12)$$

Fuente: elaboración propia.

Un bit 0 de chequeo designa una paridad uniforme sobre los bits marcados, y un 1 designa una paridad impar. Dado que los bits se escribieron con paridad uniforme, el resultado, C = C8 C4 C2 C1 = 0000, esto indica que no se ha producido ningún error.

Sin embargo, si el número binario de 4 bits formado por los bits de verificación da un número distinto de cero ha ocurrido un error. Este número indicará la posición del bit erróneo si existe solo un bit de error. Por ejemplo, considere los siguientes tres casos:

Figura 120. **Ejemplo de codificación Hamming**

1	2	3	4	5	6	7	8	9	10	11	12	
0	0	1	1	1	0	0	1	0	1	0	0	No hay Error
1	0	1	1	1	0	0	1	0	1	0	0	Error en bit 1
0	0	1	1	0	0	0	1	0	1	0	0	Error en bit 5

Fuente: elaboración propia.

En el primer caso, no hay error en la palabra de 12 bits. En el segundo caso, hay un error en la posición del bit uno porque cambió de 0 a 1. El tercer caso muestra un error en la posición del bit cinco con un cambio de 1 a 0. Al evaluar el XOR de los bits correspondientes, se determina que los cuatro bits de verificación serán los siguientes:

Figura 121. **Verificación de errores en codificación Hamming**

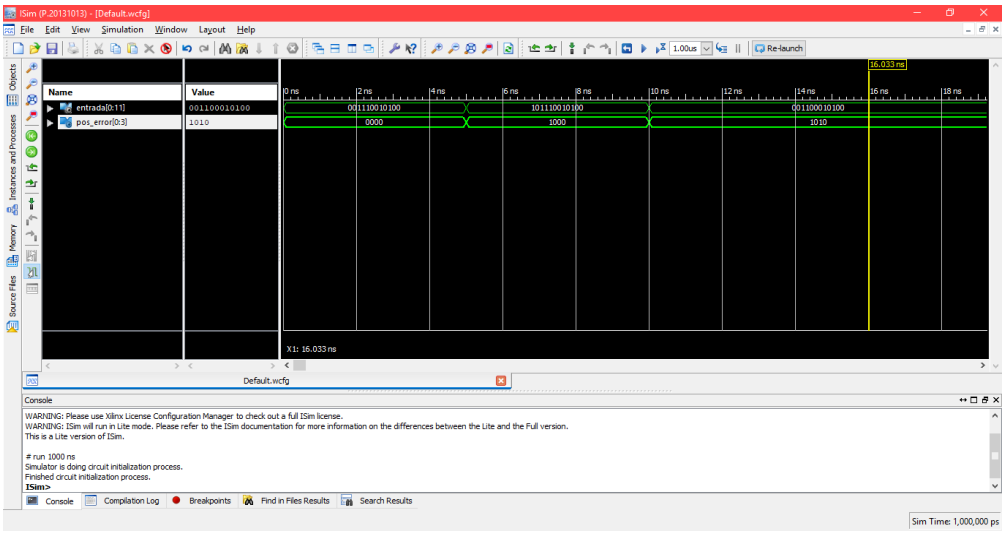
	C_8	C_4	C_2	C_1
No hay Error	0	0	0	0
Error en bit 1	0	0	0	1
Error en bit 5	0	1	0	1

Fuente: elaboración propia.

Por lo tanto, para cero errores se tiene C = 0000, con un error en el bit uno, se obtiene C = 0001 y con un error en el bit cinco, se obtiene C = 0101. Por lo tanto, cuando C no es igual a 0, el valor decimal de C da la posición del bit en error. El error se puede corregir complementando el de error bit correspondiente. Se debe tener en cuenta que puede producirse un error en los datos o en uno de los bits de paridad. El código de Hamming se puede utilizar para palabras de datos de cualquier longitud.

En una implementación de hardware en una FPGA, simplemente se aprovecha la concurrencia y la generación de las compuertas para generar un código Hamming o para realizar la verificación de un código transmitido. En este diseño propuesto se asume que existe una etapa de recepción de datos en la cual transformará de una transmisión serial a una paralela, guardando así los datos recibidos en una memoria, conectando la salida de esta memoria al módulo siguiente por medio de la señal llamada entrada. En la simulación se muestra cómo se genera el vector de salida que indica la posición del bit con error.

Figura 122. **Simulación de codificación Hamming**



Fuente: elaboración propia.

Se muestra a continuación el diseño en VHDL del hardware encargado de realizar la detección de errores utilizando código Hamming.

Figura 123. **Módulo descrito para la codificación Hamming**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity hamming is
generic(
    width: natural := 12;
    width_error: natural := 4
);
port(
    entrada : in std_logic_vector(0 to width-1);
    pos_error : out std_logic_vector(0 to width_error-1)

);
end hamming;

architecture Behavioral of hamming is

begin

    pos_error(0) <= entrada(0) xor entrada(2) xor entrada(4) xor entrada(6) xor entrada(8) xor entrada(10);
    pos_error(1) <= entrada(1) xor entrada(2) xor entrada(5) xor entrada(6) xor entrada(9) xor entrada(10);
    pos_error(2) <= entrada(3) xor entrada(4) xor entrada(5) xor entrada(6) xor entrada(11);
    pos_error(3) <= entrada(7) xor entrada(8) xor entrada(9) xor entrada(10) xor entrada(11);

end Behavioral;
```

Fuente: elaboración propia.

Como se puede observar, a manera de optimizar recursos y procesos este diseño es simplemente una operación de la compuerta XOR entre los bits previamente determinados.

3.3. Aplicación para módulos seriales

La comunicación en serie es una técnica de comunicación utilizada en telecomunicaciones en la que la transferencia de datos se produce transmitiendo datos bit por bit en orden secuencial a través de un bus informático o un canal de comunicación. Es la forma más simple de comunicación entre un emisor y un receptor.

Debido a las dificultades de sincronización implicadas en la comunicación paralela, junto con el costo del cable, la comunicación serial se considera mejor para la comunicación a larga distancia.

A diferencia de la comunicación en paralelo, que es semidúplex, la comunicación en serie es full dúplex, es decir, la transmisión y la recepción de señales pueden ocurrir simultáneamente. Es el modo más popular de protocolo de comunicación para la mayoría de los dispositivos de instrumentación. También es popular en dispositivos de computadora, dispositivos periféricos y circuitos integrados, que están provistos de uno o más puertos seriales, lo que no genera requisitos de hardware adicionales para la comunicación en serie.

Hay varias ventajas con la comunicación serial. Como hay menos conductores en contraste con la comunicación paralela, el problema de las interferencias es significativamente menor. Los cables de interconexión son menos, y no hay necesidad de un serializador o deserializador en ningún caso. Sin embargo, la velocidad de transferencia de datos puede ser baja en

comparación con la comunicación paralela. No obstante, el problema de inclinación del reloj, que a menudo ocurre entre los diferentes canales de comunicación, no es un problema con la comunicación serial.

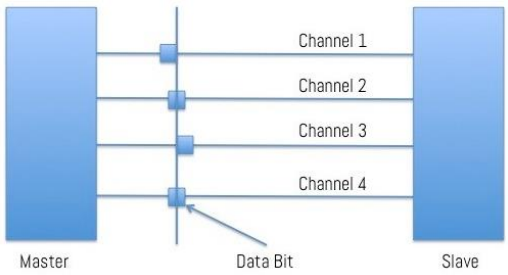
En comparación con la comunicación paralela, la comunicación serial tiene una mejor integridad de la señal. Además, la comunicación en serie es uno de los modos más económicos de comunicación que se puede implementar y, a través de la comunicación de larga distancia, puede proporcionar numerosos beneficios.

3.4. Terminología necesaria

- *Bit Rate:* es la cantidad de bits que se transmiten (enviados / recibidos) por unidad de tiempo.

- *Clock Skew:* en un circuito paralelo, el reloj skew es la diferencia de tiempo en la llegada de dos registros secuencialmente adyacentes, es decir, existe una diferencia de tiempo que se suele llamar también segado de reloj. Esto se muestra mejor en la siguiente imagen:

Figura 124. **Descripción del sesgo de reloj**



Fuente: elaboración propia.

Hay un retraso de tiempo en los bits de datos a través de diferentes canales del mismo bus. El Clock skew es inevitable debido a las diferencias en las condiciones físicas de los canales, como la temperatura, la resistencia, o la longitud de la ruta.

El *crosstalk* es el fenómeno por el cual una señal transmitida en un canal de un bus de transmisión crea un efecto no deseado en otro canal. El acoplamiento capacitivo, inductivo o conductivo no deseado suele ser lo que se denomina *crosstalk*, de un circuito, parte de un circuito o canal, a otro. El *clock skew* y el *crosstalk* son inevitables.

3.4.1. Factores principales que limitan la comunicación paralela

Antes del desarrollo de las tecnologías seriales de alta velocidad, la elección de enlaces paralelos sobre enlaces en serie se basaba en estos factores:

- Velocidad: idealmente la velocidad de un enlace paralelo es igual a la velocidad de bits por la cantidad de canales. En la práctica, la inclinación del reloj reduce la velocidad de cada enlace al más lento de todos los enlaces.

- Longitud del cable: la diafonía crea interferencia entre las líneas paralelas, y el efecto aumenta con la longitud del enlace de comunicación. Esto limita la longitud del cable de comunicación que se puede usar.

Estos dos son los principales factores que limitan el uso de la comunicación paralela.

3.5. Ventajas de serial sobre paralelo

Aunque un enlace en serie puede parecer inferior a uno paralelo, dado que puede transmitir menos datos por ciclo de reloj, a menudo los enlaces serie se pueden sincronizar considerablemente más rápido que los enlaces paralelos para lograr una mayor velocidad de datos. Varios factores permiten que la comunicación serie se sincronice a una velocidad mayor:

- El *clock skew* entre diferentes canales no es un problema (para enlaces de comunicación serie asíncronos no sincronizados).

- Una conexión en serie requiere menos cables de interconexión (por ejemplo, cables / fibras) y, por lo tanto, ocupa menos espacio. El espacio adicional permite un mejor aislamiento del canal de su entorno.

- El *crosstalk* no es un problema significativo, ya que hay menos conductores en las proximidades.

En muchos casos, el serial es una mejor opción porque es más económico de implementar. Muchos circuitos integrados tienen interfaces seriales, a diferencia de las paralelas, de modo que tienen menos pines y, por lo tanto, son menos costosas. Debido a estos factores, se prefiere la comunicación en serie sobre la comunicación paralela.

3.6. ¿Cómo se envían los datos en serie?

Cuando un conjunto de datos particular está en el microcontrolador, está en forma paralela, y se puede acceder a cualquier bit independientemente de su

número de bit. Cuando este conjunto de datos se transfiere al búfer de salida a transmitir, todavía está en forma paralela.

Este búfer de salida convierte estos datos en datos en serie (salida serial en paralelo), el bit más significativo primero o el bit menos significativo primero según el protocolo. Ahora esta información se transmite en modo serie.

Cuando estos datos son recibidos por otro microcontrolador en su búfer receptor, el búfer receptor lo convierte nuevamente en datos en paralelo para su posterior procesamiento.

3.6.1. Modos de transmisión en serie

Los datos en serie se pueden transferir en dos modos: asincrónico y sincrónico.

3.6.2. Transferencia de datos asíncrona

La transferencia de datos se denomina asíncrona cuando los bits de datos no están sincronizados con una línea de reloj, es decir, no hay reloj en absoluto.

La transferencia de datos asincrónica tiene un protocolo, que generalmente es el siguiente:

- El primer bit es siempre el bit START, esto significa el inicio de la comunicación serie, seguido de los bits DATA, generalmente 8 bits, seguido de un bit STOP, indica que es el final del paquete de datos. Puede haber un bit de paridad justo antes del bit STOP. El bit de paridad

se utilizó anteriormente para la comprobación de errores, pero rara vez se utiliza ahora.

- El bit START siempre es bajo (0) mientras que el bit STOP siempre es alto (1).

3.6.3. Transferencia de datos síncrona

La transferencia de datos síncrona es cuando los bits de datos están sincronizados con un pulso de reloj.

El concepto para la transferencia de datos sincrónicos es simple, y de la siguiente manera:

El principio básico es que el muestreo de bit de datos se realiza con respecto a las ventajas del reloj. Dado que los datos se muestrean dependiendo de los pulsos del reloj, y dado que las fuentes del reloj son muy confiables, hay mucho menos error en la sincronización que en la asincrónica.

3.7. Terminologías de comunicación en serie

- MSB / LSB: se refiere a bit más significativo o bit menos significativo. Como los datos se transfieren bit por bit en la comunicación en serie, es necesario saber qué bit se envía primero MSB o LSB.

- Comunicación Simplex: en este modo de comunicación serial, los datos solo se pueden transferir de un transmisor a otro y no viceversa.

- Comunicación semidúplex: significa que la transmisión de datos puede ocurrir solo en una dirección a la vez, es decir, de maestro a esclavo, o esclavo a maestro, pero no a ambos.

- Comunicación *Full Duplex*: significa que los datos pueden transmitirse desde el maestro al esclavo, y desde el esclavo al maestro al mismo tiempo.

- *Baud Rate:* es la velocidad de transmisión.

3.7.1. Importancia de la velocidad en baudios

Para que dos microcontroladores se comuniquen en serie, deben tener la misma velocidad en baudios, de lo contrario, la comunicación en serie no funcionará. Esto se debe a que cuando configura una velocidad en baudios, usted dirige el microcontrolador para que transmita o reciba los datos a esa velocidad particular. Entonces, si establece diferentes velocidades de transmisión, el receptor puede perder los bits que envía el transmisor.

Diferentes tasas de baudios están disponibles para su uso. Los más comunes son 2 400, 4 800, 9 600, 19 200, 38 400. No puede elegir ninguna velocidad de transmisión arbitraria, hay algunos valores fijos que debe usar como 2 400, 4 800. Tener en cuenta que la unidad de velocidad en baudios es bps (bits por segundo).

3.7.2. UART y USART

UART son las siglas de *Universal Asynchronous Receiver Transmitter*, mientras que USART son las siglas de *Universal Synchronous Asynchronous*

Receiver Transmitter. Básicamente, son componentes de hardware que convierte los datos en paralelo en datos en serie. La única diferencia entre ellos es que UART solo admite el modo asíncrono, mientras que USART admite modos asíncronos y síncronos. A diferencia de Ethernet o Firewire, no hay un puerto específico para UART o USART. Se usan comúnmente en conjugación con protocolos como RS-232, RS-434, etc.

En la transmisión síncrona, los datos del reloj se recuperan por separado del flujo de datos y no se utilizan bits de inicio o parada. Esto mejora la eficacia de la transmisión en canales adecuados, ya que la mayoría de los bits enviados son datos útiles.

El USART tiene los siguientes componentes:

- Un generador de reloj, generalmente un múltiplo de la velocidad de bits para permitir el muestreo en el medio de un período de bit.
- Registros de corrimiento de entrada y salida.
- Control para transmitir o recibir.
- Lógica de control de lectura o escritura.
- Bandejas de transmisión o recepción.
- Buffer de bus paralelo de datos.
- Memoria intermedia de primera entrada, primera salida (FIFO).

3.8. Protocolos de comunicación en serie

Se han desarrollado muchos tipos de protocolos de comunicación basados en la comunicación serial. Algunos de ellos son:

- Interfaz de periféricos seriales (SPI): es un sistema de comunicación basado en tres hilos. Un cable cada uno para maestro a esclavo y viceversa, y uno para pulsos de reloj. Hay una línea adicional SS (*Slave Select*), que se usa principalmente cuando queremos enviar o recibir datos entre múltiples circuitos integrados.

- Circuito inter integrado (I2C): esta es una forma avanzada de USART. Las velocidades de transmisión pueden llegar a los 400 KHz. El bus I2C tiene dos cables, uno para reloj, y el otro es la línea de datos, que es bidireccional; esta es la razón por la que a veces hay algunas condiciones llamadas Interfaz de dos cables (TWI). Es una tecnología bastante nueva y revolucionaria inventada por Philips.

- FireWire: desarrollado por Apple, son buses de alta velocidad capaces de transmisión de audio o video. El bus contiene una cantidad de cables dependiendo del puerto, que puede ser uno de 4 pines, uno de 6 pines o uno de 8 pines.

- Ethernet: se utiliza principalmente en conexiones LAN, el bus consta de 8 líneas, o 4 pares Tx y Rx.

- Bus serial universal (USB): este es el más popular de todos. Se usa para prácticamente todo tipo de conexiones. El bus tiene 4 líneas: VCC, Ground, Data+ y Data-.

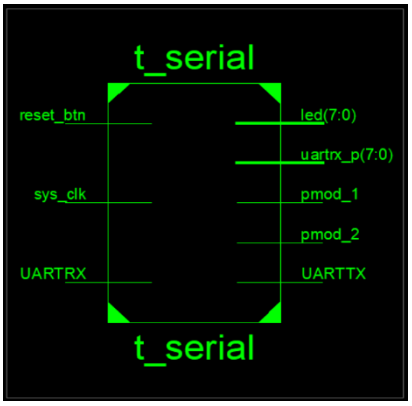
- RS-232: El RS-232 se conecta normalmente utilizando un conector DB9, que tiene 9 pines, de los cuales 5 son de entrada, 3 son de salida, y uno es tierra. Este puerto se puede encontrar en algunas computadoras antiguas.

A continuación, se describirá la implementación de un módulo serial UART en VHDL.

El módulo TOP de este diseño se muestra en la siguiente figura, como se puede observar se tienen tres entradas y cinco salidas. Las entradas incluyen un reset del sistema, un reloj de 125MHz y un pin para la entrada de Rx de datos.

Las salidas incluyen un puerto llamado led para visualizar la salida del último dato recibido y dos buses llamados pmod_1 y pmod_2, estos guardarán el último dato recibido, también se incluye una salida de Tx para realizar alguna transmisión serial.

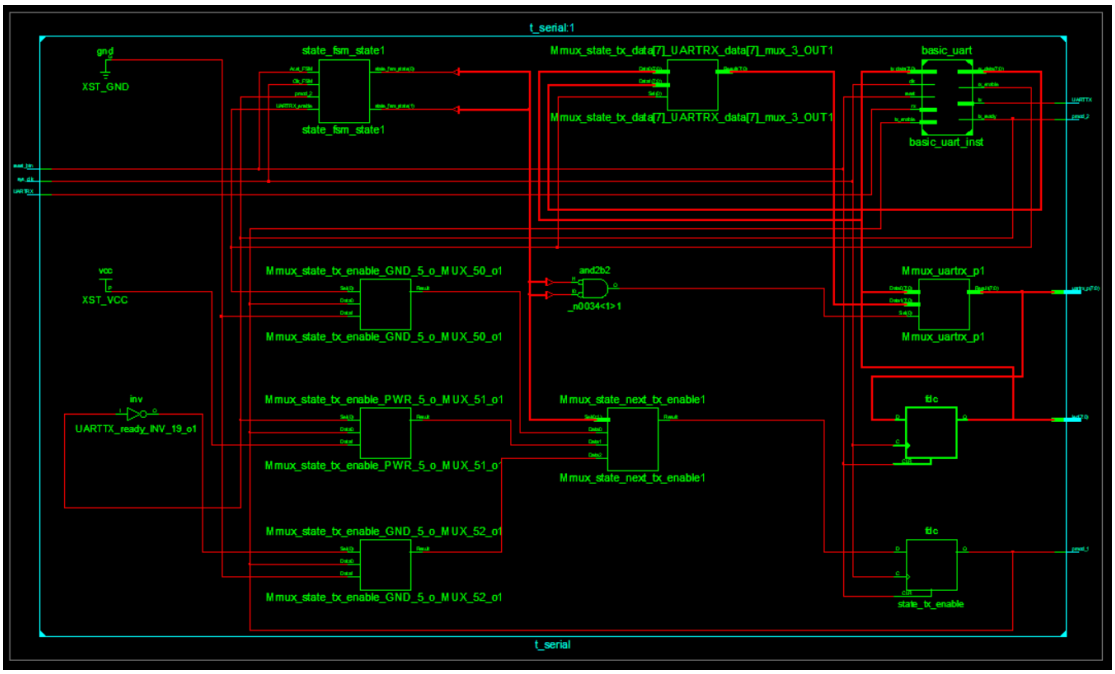
Figura 125. **Módulo para transmisión serial**



Fuente: elaboración propia.

La lógica por implementar de una forma de diagrama de componentes se muestra en la siguiente figura:

Figura 126. **Diagrama interno de módulo serial implementado en una FPGA**



Fuente: elaboración propia.

Este diseño incluye dos módulos VHDL, el primero realiza la lógica referente al algoritmo de una comunicación UART descrito anteriormente. La primera parte describe las librerías necesarias para la implementación y la declaración de puertos.

Figura 127. **Descripción de librerías y puertos en el módulo serial**

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

entity basic_uart is
  generic (
    DIVISOR: natural
    -- 2400 -> 2604
    -- 9600 -> 651
    -- 115200 -> 54
    -- 1562500 -> 4
    -- 2083333 -> 3
  );
  port (
    clk: in std_logic;
    reset: in std_logic;


    rx_data: out std_logic_vector(7 downto 0);
    rx_enable: out std_logic;
    tx_data: in std_logic_vector(7 downto 0);
    tx_enable: in std_logic;
    tx_ready: out std_logic;


    rx: in std_logic;
    tx: out std_logic
  );
end basic_uart;
```

Fuente: elaboración propia.

Dentro de la asignación de puertos se tiene una señal de reloj o clk, una señal de reset, un vector de salida rx_data, una habilitación de recepción llamada rx_enable, una señal para la recepción de bits llamada tx_data. También una para habilitar la transmisión serial llamada tx_enable, una de salida que indicará cuando la transmisión haya finalizado, una señal de entrada para los bits de recepción llamada rx y una de salida llamada tx para el envío de información serial.

Dependiendo la velocidad de transmisión establecida se crea una constante para la división de tiempo llamada divisor, se colocarán los posibles valores que puede llegar a tener este divisor para que se cree la velocidad de transmisión requerida.

Como siguiente parte de este módulo se tienen las constantes, máquinas de estados y records requeridos para la implementación. Principalmente se tiene un tipo de señal creada por el diseñador, la cual se utiliza para llevar control de la máquina de estados general. Luego se tienen dos records, uno para los estados de transmisión y otro para los estados de recepción. Se debe recordar que utilizar esta herramienta es solamente por facilidad asociativa en la lógica, ya que es simplemente una agrupación de señales. Por último, se tiene la asignación de las señales a los tipos implementados.

Figura 128. **Asignación de señales en el módulo serial descrito**

```
architecture Behavioral of basic_uart is
  constant COUNTER_BITS : natural := integer(ceil(log2(real(DIVISOR))));
  type fsm_state_t is (idle, active);
  type rx_state_t is
  record
    fsm_state: fsm_state_t;
    counter: std_logic_vector(3 downto 0);
    bits: std_logic_vector(7 downto 0);
    nbits: std_logic_vector(3 downto 0);
    enable: std_logic;
  end record;

  type tx_state_t is
  record
    fsm_state: fsm_state_t;
    counter: std_logic_vector(3 downto 0);
    bits: std_logic_vector(8 downto 0);
    nbits: std_logic_vector(3 downto 0);
    ready: std_logic;
  end record;

  signal rx_state,rx_state_next: rx_state_t;
  signal tx_state,tx_state_next: tx_state_t;
  signal sample: std_logic;
  signal sample_counter: std_logic_vector(COUNTER_BITS-1 downto 0);

begin
```

Fuente: elaboración propia.

A manera de que se genere un reloj que esté acorde con la velocidad de transmisión se genera un proceso para crear un sub reloj. Esto es simplemente un conteo de ciclos de reloj mediante una señal y la constante establecida previamente.

Figura 129. **Módulo para generación de reloj para transmisión**

```
Sub_clock: process (clk,reset) is
begin
  if reset = '1' then
    sample_counter <= (others => '0');
    sample <= '0';
  elsif rising_edge(clk) then
    if sample_counter = DIVISOR-1 then
      sample <= '1';
      sample_counter <= (others => '0');
    else
      sample <= '0';
      sample_counter <= sample_counter + 1;
    end if;
  end if;
end process;
```

Fuente: elaboración propia.

Como se ha establecido ya la máquina de estados más óptima es la tipo A, por lo tanto este diseño de UART utiliza este tipo de máquina de estado. Se debe recordar que este tipo de máquina separa la lógica en tres procesos, uno para la lógica actual, otro para la lógica siguiente y otro para la lógica de salida.

A continuación, se muestra el proceso que genera la lógica actual, este simplemente asigna un valor a la señal encargada de la máquina de estados para que se dispare el proceso de la lógica siguiente o se resetea las señales dependiendo de la entrada de reset. Este proceso incluye, tanto la señal de recepción como la de transmisión.

Figura 130. **Descripción del proceso de sincronización de señales en el módulo serial**

```
reg_process: process (clk,reset) is
begin
  if reset = '1' then
    rx_state.fsm_state <= idle;
    rx_state.bits <= (others => '0');
    rx_state.nbits <= (others => '0');
    rx_state.enable <= '0';
    tx_state.fsm_state <= idle;
    tx_state.bits <= (others => '1');
    tx_state.nbits <= (others => '0');
    tx_state.ready <= '1';
  elsif rising_edge(clk) then
    rx_state <= rx_state_next;
    tx_state <= tx_state_next;
  end if;
end process;
```

Fuente: elaboración propia.

Se puede notar que existen dos señales, rx_state y tx_state, las cuales se declararon como records, cada uno tiene como un conjunto de señales. Se tiene fsm_state para guardar el estado actual del proceso, la señal bits guarda la información de la transmisión, la señal nbits incluye los bits de parada e inicialización, la señal enable habilita la transmisión o recepción. Por último, se tiene una señal de ready, la cual se utiliza para indicar cuándo está lista la transmisión o recepción.

El proceso de la lógica siguiente está compuesto por dos estados uno de espera llamada IDLE y otro de procesamiento, en este estado se debe realizar un conteo de cero a ocho ya que se recibirá un un byte de información. Se mantendrá en este estado hasta que haya finalizado el conteo. Al finalizar se guardará en una memoria el dato recibido pasando nuevamente al estado de espera.

Figura 131. **Descripción de proceso para la recepción serial**

```
rx_process: process (rx_state,sample,rx) is
begin
  case rx_state.fsm_state is
  when idle =>
    rx_state_next.counter <= (others => '0');
    rx_state_next.bits <= (others => '0');
    rx_state_next.nbits <= (others => '0');
    rx_state_next.enable <= '0';
    if rx = '0' then

      rx_state_next.fsm_state <= active;
    else

      rx_state_next.fsm_state <= idle;
    end if;
  when active =>
    rx_state_next <= rx_state;
    if sample = '1' then
      if rx_state.counter = 8 then

        if rx_state.nbits = 9 then
          rx_state_next.fsm_state <= idle;
          rx_state_next.enable <= rx;
        else
          rx_state_next.bits <= rx & rx_state.bits(7 downto 1);
          rx_state_next.nbits <= rx_state.nbits + 1;
        end if;
      end if;
      rx_state_next.counter <= rx_state.counter + 1;
    end if;
  end case;
end process;
```

Fuente: elaboración propia.

Figura 132. **Descripción de proceso para la transmisión serial**

```
tx_process: process (tx_state,sample,tx_enable,tx_data) is
begin
  case tx_state.fsm_state is
      when idle =>
        if tx_enable = '1' then
          -- start a new bit
          tx_state_next.bits <= tx_data & '0';  -- data & start
          tx_state_next.nbits <= "0000" + 10; -- send 10 bits (includes '1' stop bit)
          tx_state_next.counter <= (others => '0');
          tx_state_next.fsm_state <= active;
          tx_state_next.ready <= '0';
        else
          -- keep idle
          tx_state_next.bits <= (others => '1');
          tx_state_next.nbits <= (others => '0');
          tx_state_next.counter <= (others => '0');
          tx_state_next.fsm_state <= idle;
          tx_state_next.ready <= '1';
        end if;
      when active =>
        tx_state_next <= tx_state;
        if sample = '1' then
          if tx_state.counter = 15 then
            -- send next bit
            if tx_state.nbits = 0 then
              -- turn idle
              tx_state_next.bits <= (others => '1');
              tx_state_next.nbits <= (others => '0');
              tx_state_next.counter <= (others => '0');
              tx_state_next.fsm_state <= idle;
              tx_state_next.ready <= '1';
            else
              tx_state_next.bits <= '1' & tx_state.bits(8 downto 1);
              tx_state_next.nbits <= tx_state.nbits - 1;
            end if;
          end if;
          tx_state_next.counter <= tx_state.counter + 1;
        end if;
      end case;
end process;
```

Fuente: elaboración propia.

Como se observa en los códigos anteriores se tienen dos procesos que dependen de qué tipo de acción se requiere. El primer *process* es para la recepción de códigos seriales y el segundo es para la transmisión. Ambos procesos tienen una funcionalidad parecida, solamente que se guarda en diferente record la información. Los dos incluyen un estado de active y dentro de este se tiene una estructura de control de comparación.

Cuando el conteo es menor a nueve se hace un corrimiento de bits y se va incluyendo el bit recibido al vector total de salida, el cual está almacenado en rx_state_next.bits o en tx_state_next.bits dependiendo de qué proceso se analice, luego de esto se aumenta el campo nbits para saber en qué bit va la transmisión en determinado momento.

La última parte de este módulo se refiere al proceso que realiza las salidas en el módulo, siguiendo el modelo de la máquina de estado tipo A.

Figura 133. **Procesos para salidas del módulo serial**

```
rx_output: process (rx_state) is
begin
  rx_enable <= rx_state.enable;
  rx_data <= rx_state.bits;
end process;

-- TX output
tx_output: process (tx_state) is
begin
  tx_ready <= tx_state.ready;
  tx <= tx_state.bits(0);
end process;
```

Fuente: elaboración propia.

El primer proceso se refiere a la salida de la recepción serial, el cual simplemente es la asignación del vector recibido a una salida llamada rx_data.

El segundo proceso es simplemente la asignación de la primera posición del vector asignado para la transmisión al bit de salida llamado tx que es una señal de salida en el módulo.

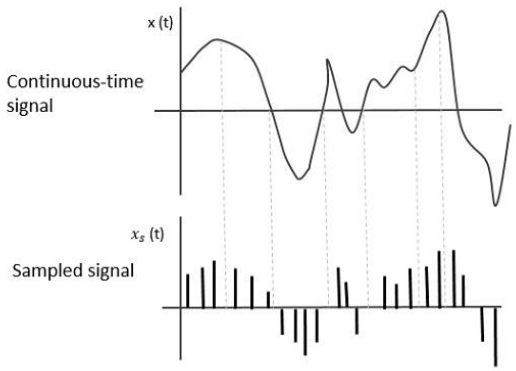
3.8.1. Muestreo de señales

El muestreo se define como, el proceso de medir los valores instantáneos de la señal de tiempo continúo en una forma discreta. En el muestreo se genera un *sample*, el cual es una pieza de datos tomada de la información continua en el dominio del tiempo.

Cuando una fuente genera una señal analógica y esta tiene que digitalizarse, teniendo unos y ceros como altos o bajos, como la señal discretizada en el tiempo. Esta discretización de la señal analógica se denomina muestreo.

La siguiente figura indica una señal de tiempo continuo x(t) y una señal muestreada xs(t). Cuando x(t) se multiplica por un tren de impulso periódico, se obtiene la señal muestreada xs(t).

Figura 134. **Ejemplo de muestreo de una señal**



Fuente: elaboración propia.

3.8.1.1. Tasa de muestreo

Para discretizar las señales, el espacio entre las muestras debe ser fijo. Esa brecha puede denominarse como un período de muestreo Ts.

$$SamplingFrequency = 1/T_S = f_S$$

Donde:

Ts: tiempo de muestreo
fs: frecuencia de muestreo

La frecuencia de muestreo es el recíproco del período de muestreo.

Esta frecuencia de muestreo, se puede llamar simplemente como índice de muestreo. La frecuencia de muestreo indica el número de muestras tomadas por segundo o un conjunto finito de valores.

Para que una señal analógica se reconstruya a partir de la señal digitalizada, la velocidad de muestreo debe ser considerada. La tasa de muestreo debe ser tal que los datos en la señal del mensaje no se pierdan ni se sobrepongan. Por lo tanto, se propuso una tasa para esto, llamada tasa de Nyquist.

La frecuencia de muestreo es el recíproco del período de muestreo. Esta frecuencia de muestreo, se puede llamar simplemente como índice de muestreo. La frecuencia de muestreo indica el número de muestras tomadas por segundo o un conjunto finito de valores.

3.8.1.1.1. Tasa Nyquist

Supongamos que una señal tiene una banda limitada sin componentes de frecuencia superiores a W hercio. Eso significa que W es la frecuencia más alta. Utilizando esta señal, para una reproducción efectiva de la señal original, el teorema dice que la frecuencia de muestreo debería ser el doble de la frecuencia más alta. Esta tasa es llamada tasa de muestreo de Nyquist el cual es el teorema de muestreo de señales.

3.8.1.1.2. Teorema de muestreo

El teorema de muestreo, que también se conoce como el teorema de Nyquist, muestra la frecuencia de muestreo en términos de ancho de banda para las funciones que están limitadas en banda.

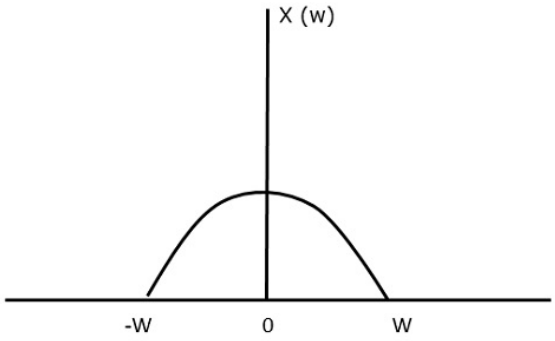
El teorema de muestreo establece que una señal puede reproducirse exactamente si se muestrea a una velocidad con frecuencia fs, la cual es mayor que el doble de la frecuencia máxima W.

Para comprender este teorema de muestreo, hay que considerar una señal de banda limitada, es decir, una señal cuyo valor no es cero y está entre algunos W y W hercio.

Tal señal se representa como $x(f) = 0$ para $| f | > W$

Para la señal de tiempo continuo x(t), la señal de banda limitada en el dominio de la frecuencia se puede representar como se muestra en la siguiente figura;

Figura 135. **Señal de banda limitada en el dominio de la frecuencia**



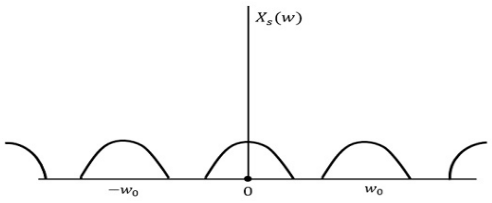
Fuente: elaboración propia.

Se necesita una frecuencia de muestreo, una frecuencia en la que no debe haber pérdida de información, incluso después de terminar el muestreo. Para esto, existe la tasa de Nyquist la cual indica que la frecuencia de muestreo debe ser dos veces la frecuencia máxima. Es la tasa crítica de muestreo.

Si la señal x(t) se muestrea por encima de la tasa de Nyquist, la señal original puede recuperarse, y si se muestrea debajo de la tasa de Nyquist, la señal no puede recuperarse.

La siguiente figura explica una señal, si se muestrea a una velocidad mayor que 2W en el dominio de la frecuencia.

Figura 136. **Representación del muestreo a una velocidad al doble de ancho de banda**



Fuente: elaboración propia.

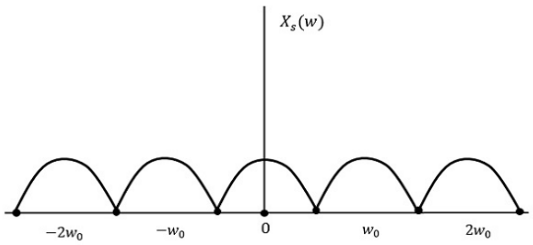
La figura anterior muestra la transformada de Fourier de una señal xs(t). Aquí, la información se reproduce sin ninguna pérdida. No hay confusión y, por lo tanto, la recuperación es posible.

Qué ocurre si la frecuencia de muestreo es igual al doble de la frecuencia más alta(2W)

Eso significa:

$$fs = 2W$$

Figura 137. **Representación del muestreo a una velocidad igual al ancho banda de la señal**



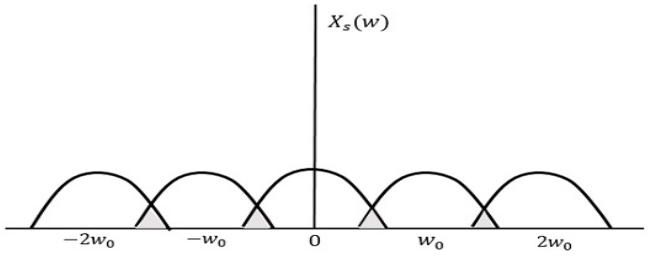
Fuente: elaboración propia.

El resultado será como se muestra en la figura anterior. La información se reemplaza sin ninguna pérdida. Por lo tanto, esta es también una buena tasa de muestreo.

Ahora, si la condición es fs <2W

El patrón resultante se verá como la siguiente figura.

Figura 138. **Representación del muestreo a una velocidad menor al doble de ancho de banda de una señal**



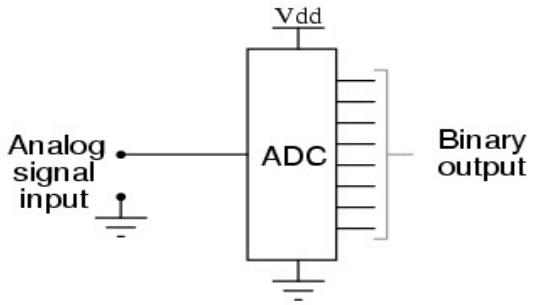
Fuente: elaboración propia.

Se puede observar a partir del patrón anterior que la superposición de información se realiza, lo que conduce a la confusión y la pérdida de información. Este fenómeno no deseado de traslape se llama Aliasing.

3.8.2. Conversor análogo digital

Un convertidor analógico al digital es una herramienta muy útil que convierte una tensión analógica de una terminal en un valor digital. Al convertir el mundo analógico al mundo digital, se puede empezar a utilizar la electrónica para interactuar con el mundo analógico.

Figura 139. **Módulo conversor análogo digital**



Fuente: elaboración propia.

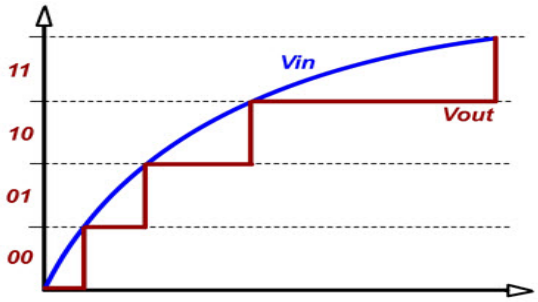
3.9. Cuantización

Una vez realizado el muestreo, el siguiente paso es la cuantización de la señal analógica. Para esta parte del proceso los valores continuos de la señal analógica se convierten en series de valores numéricos decimales discretos correspondientes a los diferentes niveles o variaciones de voltajes que contiene la señal analógica original.

Por tanto, la cuantización representa el componente de muestreo de las variaciones de valores de tensiones o voltajes tomados en diferentes puntos de la señal analógica, llamados pasos, que permite medirlos y asignarles sus correspondientes valores en el sistema numérico decimal, antes de convertir esos valores en sistema numérico binario.

Para proceder a terminar la conversión se realiza un proceso de digitalización, que es simplemente asignar valores binarios a las cantidades decimales obtenidas.

Figura 140. **Cuantización de señales**



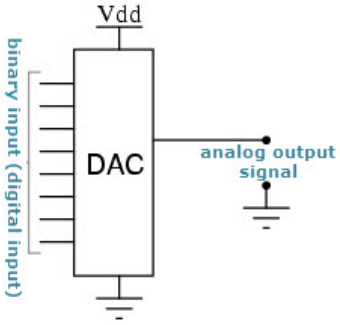
Fuente: elaboración propia.

3.9.1. Conversor digital análogo (DAC)El convertidor digital a analógico

Es un dispositivo que transforma datos digitales en una señal analógica. De acuerdo con el teorema de muestreo de Nyquist-Shannon, cualquier dato muestreado se puede reconstruir perfectamente con el ancho de banda y los criterios de Nyquist.

Un conversor digital análogo puede reconstruir los datos muestreados en una señal analógica con precisión. Los datos digitales pueden producirse desde un microprocesador, circuito integrado de aplicación específica o arreglo de puerta programable de campo. Pero, finalmente, los datos requieren la conversión a una señal analógica para interactuar con el mundo real.

Figura 141. **Módulo conversor digital análogo**

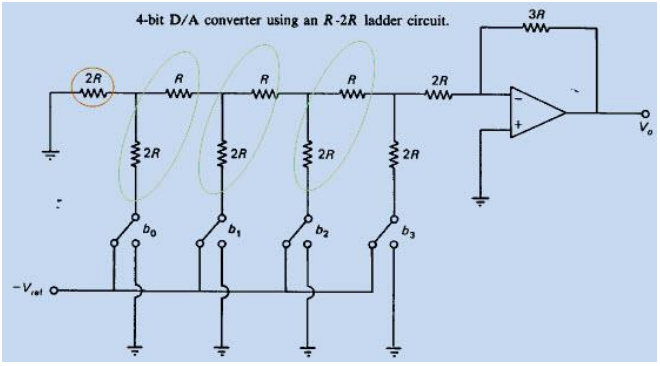


Fuente: elaboración propia.

3.9.1.1. Red escalera R-2R

Este conversor digital análogo construido como un DAC ponderado binario que usa una estructura repetida en cascada de valores de resistencia R y 2R. Esto mejora la precisión debido a la relativa facilidad de producir resistencias de igual valor.

Figura 142. **Conversor digital análogo escalera**



Fuente: elaboración propia.

3.9.1.2. Aplicaciones del convertidor digital a analógico

Los DAC se utilizan en muchas aplicaciones de procesamiento de señal digital y en muchas más aplicaciones. Algunas de las aplicaciones más importantes se muestran a continuación.

- Amplificador de audio: los DAC se usan para producir ganancia de voltaje de corriente continua con los comandos del microcontrolador. Usualmente, el DAC se incorporará a un códec de audio completo que incluye funciones de procesamiento de señales.

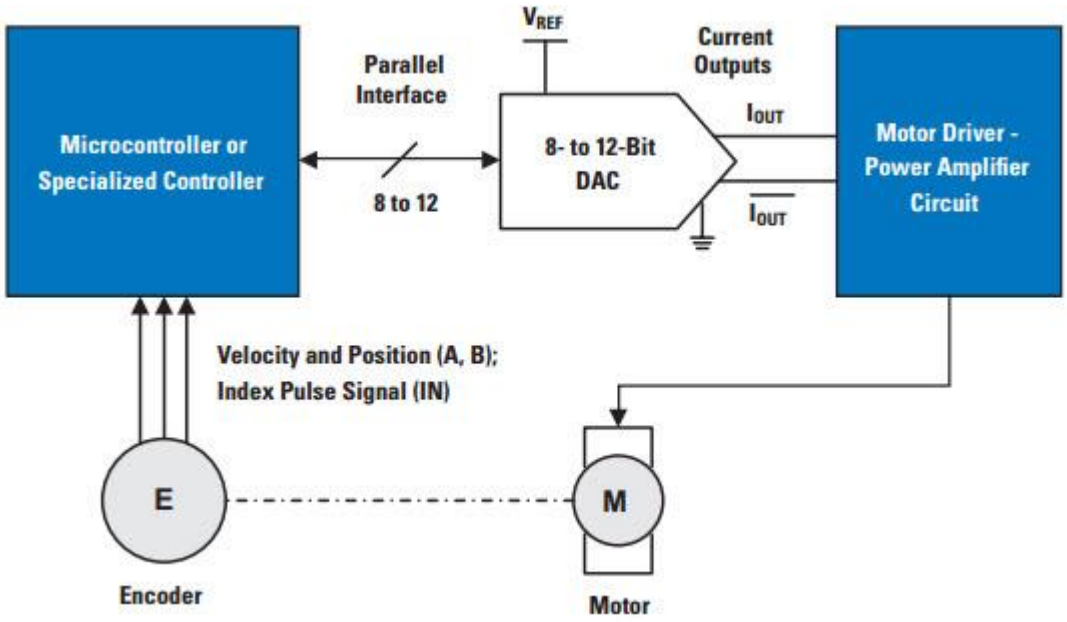
- Video Encoder: el sistema de codificador de video procesa una señal de video y envía señales digitales a una variedad de DAC para producir señales de vídeo analógico de varios formatos, junto con la optimización de los niveles de salida.

- Display electrónico: un controlador gráfico generalmente usará una tabla de búsqueda para generar señales de datos enviadas a un DAC de video para salidas analógicas como señales rojas, verdes, azules (RGB) para controlar una pantalla.

- Calibración: el DAC proporciona calibración dinámica para compensación de ganancia y voltaje para precisión en sistemas de prueba y medición.

- Control del motor: muchos controles de motor requieren señales de control de voltaje, y un DAC es ideal para esta aplicación el cual puede ser impulsado por un procesador o controlador.

Figura 143. **Control de motor utilizando un convertidor digital análogo**



Fuente: elaboración propia.

3.10. Descripción de hardware en VHDL de un módulo ADC

El siguiente módulo VHDL es utilizado para manejar el DAC TLV5616 y el ADC ADS5500. Este se implementa utilizando una máquina de estados finitos (FSM) de Moore Tipo B. En esta sección se hará la síntesis de un receptor de SPI para realizar un ADC utilizando el circuito integrado.

Se describe a continuación todos los puertos y señales que implementa este módulo, estos son implementados con interfaz Wishbone. Se describirán todas las interfaces de entrada y salida a continuación.

Constantes:

- bus_width := 8 : determina el número de bits por utilizar.
- clkDiv := 30 : ajusta la frecuencia del divisor de reloj.

Entradas:

- Rst_I [std_logic] : entrada para resetear el sistema.
- Clk_I [std_logic]: entrada de reloj general del sistema.
- DAT_I [std_logic_vector (bus_width-1 downto 0]: entrada de un dato paralelo para ser convertido en un dato serial en protocolo SPI.
- WE_I [std_logic]: Bit para activar el emisor SPI.

Salidas:

- DOUT [std_logic]: Salida serial SPI

Señales:

- count_dout: contador del número de bits de transmisión SPI.
- current_state: máquina de estados para variar el comportamiento del emisor SPI.
- Data: bus que contiene temporalemente el bus de entrada del emisor SPI.
- SPIClk: reloj con frecuencia derivada del reloj principal, es decir, esta es la salida de una división de reloj.
- CLKCounter: contador para realizar SCLK = CLK / 10.

3.10.1.1. Sintetizando el divisor de reloj

En la comunicación serial se genera una velocidad de transferencia de datos, es decir se crea un reloj auxiliar que es derivado del reloj general del funcionamiento del módulo SPI. Para realizar este divisor de reloj se deberá sintetizar un proceso paralelo a los otros procesos necesarios para ejecutar el protocolo de comunicación.

Primero se generará una señal llamada CLKCounter la cual es de tipo entero, esta señal llevará el conteo de ciclos de reloj, al iniciar el conteo la salida SPIClk se mantendrá en uno. Habiéndose ejecutado cierta cantidad de ciclos de reloj la salida SPIClk cambiará a cero, generando así un tren de pulsos a una frecuencia determinada por el contador de pulsos CLKConter. Se muestra a continuación la descripción de hardware utilizando VHDL a continuación.

Figura 144. **Módulo generador de reloj**

```
process (CLK_i, RST_i)
begin
If (RST_i = '1') Then
    CLKCounter <= 0;
-- Si hay un reset, se inicializa en 0 el contador del divisor
    SPIClk <= '0';
Else
    If (rising_edge(CLK_i)) Then
        If CLKCounter < clkDiv Then
-- Si hay overflow, regresa a su valor inicial (0)
            CLKCounter <= CLKCounter + 1;
        Else
CLKCounter <= 0;
SPIClk <= not SPIClk;
-- Si existio overflow en el contador de reloj secundario, genera una transicion en el reloj de SPI
                End If;
End If;
End If;
end process;
```

Fuente: elaboración propia.

El primer *if* verifica el cambio de la variable sensitiva del reseteo, cuando esto sucede el sistema inicializa la variable contador llamada SPICounter y la salida del reloj derivado llamada SPIClk la inicializa con cero.

Si cambia alguna otra variable sensitiva como lo es la señal CLK_i pasa a la sentencia Else, se procede a verificar si este cambio de la variable sensitiva es ascendente o descendente, en este caso sí es ascendente o el cambio es de flanco de subida se comparará si la señal contadora CLKConter es menor al número de pulsos requeridos.

Este número de pulsos requeridos es una constante Generic llamada clkDiv. Sí está condición es verdadera, se le sumará un uno al contador, hasta que se deje de cumplir esta condición se reiniciará el contador CLKCounter y se negará la salida del reloj derivado SPIClk.

Para la implementación del módulo principal de recepción del protocolo SPI se utilizará una señal llamada *current_state* de tipo FSM, que describe una máquina de estados con los siguientes estados:

- DATA_FETCH: estado de esperar, se genera antes de que se empiece a recibir los datos seriales.

- SPI_OUT: estado de recepción, cuando se pasa del estado de espera al de recepción se espera en el bit de entrada la trama de datos dependiendo el número de bits del dispositivo que trabajará con el protocolo SPI. Para los integrados DAC TLV5616 y el ADC ADS5500 se utilizarán 16 bits de trama.

- RESYNC: estado previo al de espera, se toma un ciclo de reloj para estabilizar la comunicación y pasar al estado DATA_FETCH.

Se muestran las entradas y salidas del módulo de SPI receptor el cual está dentro de un módulo general de prueba descrito más adelante.

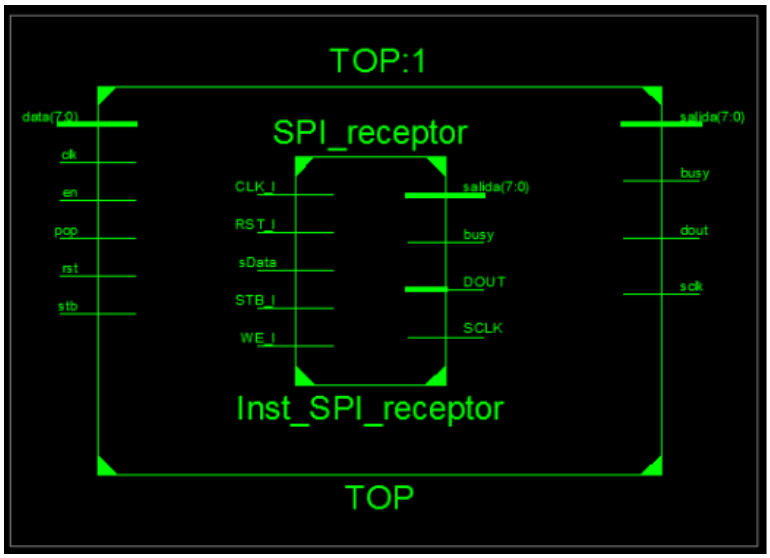
Por último, se requiere un proceso para llevar la lógica de la recepción serial para obtener el dato obtenido del integrado ADC. Se describe a continuación la lógica de descripción de hardware utilizando VHDL.

Figura 145. **Módulo para generación de SPI**

```
process (SPIClk)
begin
    If rising_edge (SPICLK) Then -- Revisa si el reloj derivado está en flaco de subida
        If (RST_I = '1') Then
            -- Si se receta la transmisión se pasa al estado de espera y se inicializan las señales a cero
            current_state <= DATA_FETCH; -- Valor inicial de la máquina de estado
            EnableDout <= '0';           -- Deshabilitar SCLK y SPI Data Output
            count_dout <= 0;             -- Iniciar contador de shift-register SPI
            mem_tmp <= (others => '0');
            busy_s <= '0';
        Else
            Case current_state is  -- Validación de la máquina de estados tipo B de moore
                When DATA_FETCH =>
                    If (WE_I='1') Then
                        -- Cuando se activan los bits WE_I (write enable) se pasa al estado de
                        -- recepción de la trama de datos
                        current_state <= SPI_OUT;
                        count_dout <= 0; -- Se inicializa el contador de
                        -- la trama de bits
                        busy_s <= '0';
                    End If;
                When SPI_OUT    => -- Estado donde se cuentan los bits de la trama,
                    -- para este dispositivo se hace un conteo de 16
                    -- bits.
                    If (count_dout <= bus_width and count_dout > 0) Then
                        -- Se inicia el conteo de bits con la señal count_dout
                        EnableDout <= '1';
                        -- Se da la opción de guardarlo en una memoria
                        -- una señal std_logic_vector llamada
                        -- mem_tmp.
                        mem_tmp(count_dout-1)<=sData;
                        -- sData es el bit de recepción serial
                        count_dout <= count_dout + 1;
                        busy_s <= '0';
                    elsif count_dout = 0 then
                        -- Se espera un ciclo de reloj para sincronización.
                        count_dout <= count_dout + 1;
                        busy_s <= '0';
                    Else -- Se espera un ciclo de reloj para estabilizar la recepción
                        current_state <= RESYNC;
                        EnableDout <= '0';
                        busy_s <= '1';
                    End If;
                When Others     =>
                    -- En cualquier otro estado se procede a llevarlo al estado de espera
                    busy_s <= '0';
                    If (WE_I='0' AND STB_I='0') Then
                        busy_s <= '0';
                        current_state <= DATA_FETCH;
                    End If;
            End case;
        End If;
    End If;
End Process;
```

Fuente: elaboración propia.

Figura 146. **Módulo general de SPI**



Fuente: elaboración propia.

El *process* encargado de cumplir la función de un receptor del estándar SPI se ejecutará en paralelo con el reloj derivado creado con anterioridad.

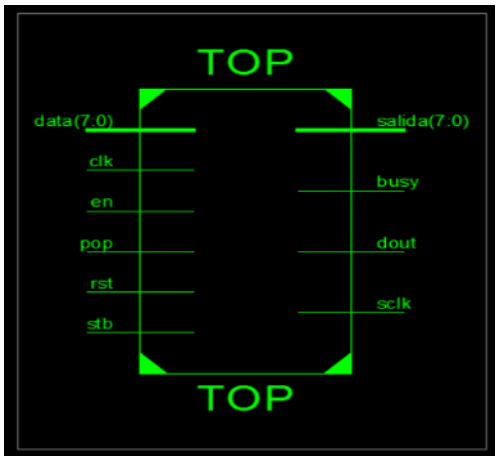
En síntesis, este *process* será ejecutado por este reloj, por lo tanto, cada ciclo de reloj revisará en qué estado se encuentra actualmente la máquina, al inicio se encuentra en el estado de espera hasta que la variable sensitiva pase a ser un uno lógico, cuando esto sucede se procederá a pasar al estado de recepción el cual contiene un contador para saber si ya se han recibido todos los bits seriales.

Al finalizar, es decir, que el contador llegue al número total de bits, en este ejemplo 16 bits, se pasará al siguiente estado el cual tomará un ciclo de reloj para estabilizar la comunicación. Cuando este ciclo de reloj termina se regresa nuevamente al estado de espera cerrando así el ciclo.

Para efecto de prueba se crea un emisor de comunicación SPI para revisar el funcionamiento y poder ilustrar la recepción del módulo SPI. A continuación, se muestra la caja negra general que contiene lo siguiente:

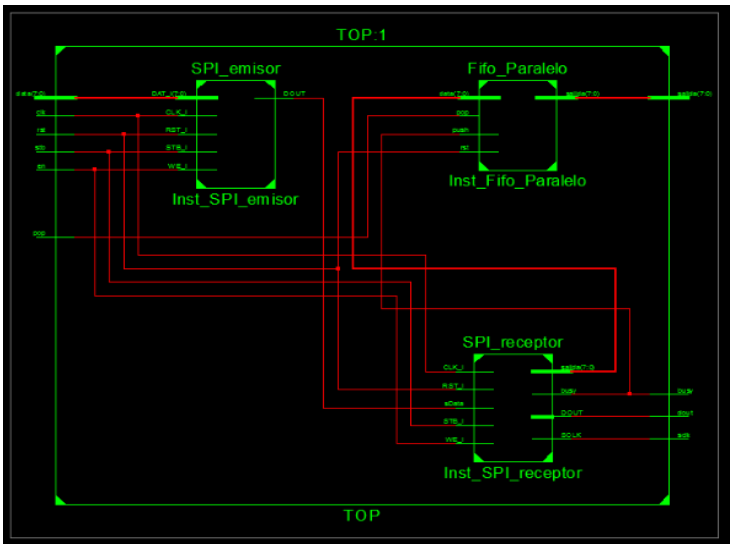
- Emisor SPI
- Receptor SPI
- Memoria FIFO (*First in, first out*)

Figura 147. **Puertos de módulo SPI**



Fuente: elaboración propia.

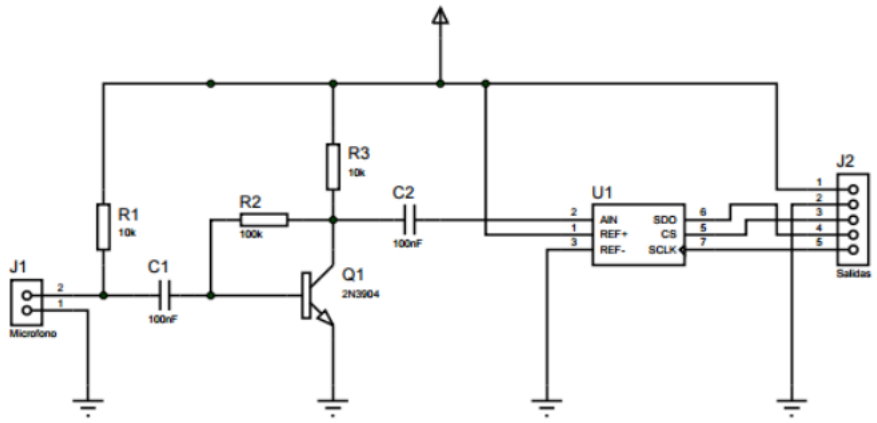
Figura 148. **Descripción de módulos generados en una comunicación SPI**



Fuente: elaboración propia.

Si se desea realizar conversiones de valores digitales a analógicos utilizando comunicación SPI, se propone utilizar un preamplificador conectado al integrado TLV5616 propuesto en esta sección. Este módulo se debe conectar al circuito sintetizado en la fpga, conectando las entradas respectivamente a las salidas del integrado.

Figura 149. **Diagrama de circuitos de un módulo SPI**



Fuente: MORALES, Iván.
http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf.
Consulta: 19 diciembre de 2018.

3.11. Modulación por ancho de pulso o PWM

PWM es un término elegante para describir un tipo de señal digital. La modulación por ancho de pulso se usa en una variedad de aplicaciones que incluyen circuitos de control sofisticados. La modulación de ancho de pulso permite variar la cantidad de tiempo que la señal es alta de forma analógica.

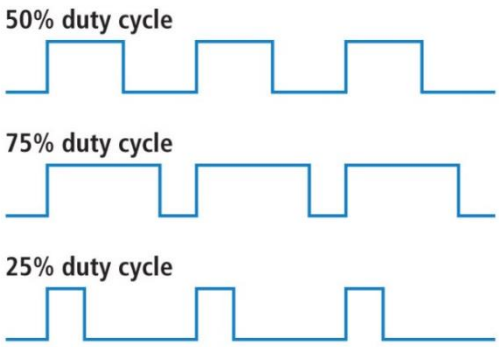
Si bien la señal solo puede ser alta o baja en cualquier momento, se puede cambiar la proporción de tiempo que la señal es alta en comparación con cuando es baja durante un intervalo de tiempo constante.

3.11.1. Ciclo de trabajo

Para describir la cantidad que un pulso está en alto, se utiliza el concepto de ciclo de trabajo o *duty cycle.* El ciclo de trabajo se mide en porcentaje. El ciclo de trabajo porcentual describe específicamente el porcentaje de tiempo que una señal digital está activa durante un intervalo o período de tiempo. Este período es el inverso de la frecuencia de la forma de onda.

Si una señal digital pasa la mitad del tiempo activada y la otra mitad apagada, se dice que la señal digital tiene un ciclo de trabajo del 50 % y se asemeja a una onda cuadrada ideal. Si el porcentaje es superior al 50 %, la señal digital pasa más tiempo en estado alto que en estado bajo, y viceversa si el ciclo de trabajo es inferior al 50 %.

Figura 150. **Tipos de ciclo de trabajo PWM**



Fuente: elaboración propia.

Este módulo contiene una constante generic llamada *bus_width* el cual determinará la resolución del PWM. Se agrega una entrada de reloj llamada clk, una entrada de un bit para reset llamada rst y otra señal de un bit llamada *load*, que se utilizará para cambiar el ciclo de trabajo en un determinado momento.

Se incluye una salida de un bit para mostrar el pulso de salida de PWM. En el módulo se crea una señal de conteo de pulsos de reloj llamada cnt, esto se utiliza para comparar este contador contra la entrada de ciclo de reloj, el cual es asignado cuando la señal de *load* es igual a uno, este se asigna a una señal temporal llamada comp_val.

Por último, se utiliza una validación concurrente donde la señal de salida pwm_*out* va a ser cero cuando el contador cnt sea menor al valor cargado en la señal temporal comp_val, de caso contrario la salida pwm_out será uno. A continuación, se muestra el módulo de descripción de hardware en VHDL.

Figura 151. **Módulo de descripción de PWM**

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

entity PWM_mod is
   generic (
      bus_width : natural := 8
   );
   port (
      clk, rst, load : in std_logic;
      duty_cycle : in std_logic_vector(bus_width-1 downto 0);
      pwm_out : out std_logic
   );
end PWM_mod;

architecture Behavioral of PWM_mod is
signal cnt, comp_val : unsigned (bus_width-1 downto 0);
begin
   count_proc : process (clk, rst)
   begin
      if (rst = '1') then
         cnt <= to_unsigned(0, bus_width);
      elsif (rising_edge(clk)) then
         if (load = '1') then
            comp_val <= unsigned(duty_cycle);
         else
            cnt <= cnt + 1;
         end if;
      end if;
   end process count_proc;
   pwm_out <= '0' when (cnt < comp_val) else '1';
end Behavioral;
```

Fuente: elaboración propia.

3.12. Filtros digitales con respuesta finital al impulso (FIR)

En el procesamiento de señal digital, un FIR es un filtro cuya respuesta de impulso es de período finito, como resultado de que se establece en cero en tiempo finito. A diferencia de los filtros con respuesta infinita al impulso o IIR estos no pueden tener retroalimentación interna y responder indefinidamente.

La respuesta de impulso de un filtro FIR de tiempo discreto de orden N toma exactamente N+1 muestras antes de establecerse en cero. Los filtros FIR son los tipos más populares de filtros ejecutados en software y estos filtros pueden ser de tiempo continuo, analógico o digital y discreto. Los tipos especiales de filtros FIR son *Boxcar, Hilbert Transformer, Differentiator, Lth-Band y Raised-Cosine.*

Los filtros son acondicionadores de señal y la función de cada filtro es que permite un componente de corriente alterna y bloquea los componentes de corriente continua. Un ejemplo de filtro es una línea telefónica, ya que limita las frecuencias a una serie significativamente menor equivalente a la que los seres humanos que pueden escuchar.

Existen diversos tipos de filtros como el filtro pasa bajos o LPF, el cual solo permite señales de baja frecuencia, por lo que este filtro se usa para eliminar las frecuencias altas. Un LPF es conveniente para controlar el rango más alto de frecuencias en una señal de audio. Otro ejemplo de filtro es el pasa altos o HPF, este es opuesto a LPF, ya que rechaza solo componentes de frecuencia por debajo de un umbral.

Los métodos de diseño del filtro FIR se basan en la aproximación del filtro ideal. El filtro resultante se acerca a la característica perfecta porque el orden

del filtro aumentará, por lo que la creación del filtro y su implementación son complicadas.

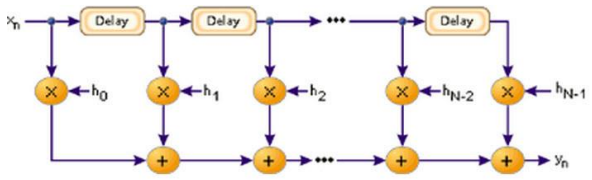
El proceso de diseño comienza con las necesidades y especificaciones del filtro FIR. El método utilizado en el proceso de diseño del filtro depende de la implementación y las especificaciones. Hay muchas ventajas y desventajas de los métodos de diseño.

Por lo tanto, es muy importante elegir el método correcto para el diseño del filtro FIR. Debido a la eficiencia y simplicidad del filtro FIR, comúnmente se usa el método de ventana.

3.13. Estructura lógica del filtro FIR

Un filtro FIR se utiliza para implementar casi cualquier tipo de respuesta de frecuencia digital. Por lo general, estos filtros están diseñados con multiplicadores, sumadores y una serie de retrasos para crear la salida del filtro. El resultado de los retrasos opera en muestras de entrada. Se deben incluir valores característicos del filtro, los cuales son los coeficientes que se usan para la multiplicación. Por lo tanto, la salida resultante es la suma de todas las muestras retrasadas multiplicadas por los coeficientes apropiados.

Figura 152. **Diagrama de bloques de un filtro digital**



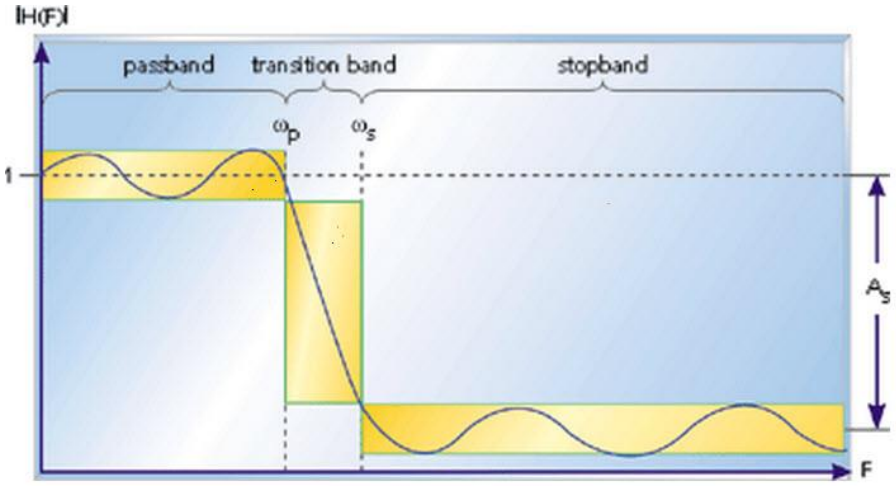
Fuente: elaboración propia.

Diseñar un filtro es el proceso de elegir la longitud y los coeficientes del filtro. La intención es establecer los parámetros requeridos, como lo es la banda de detención y la banda de paso. Una buena opción es utilizar el software MATLAB para diseñar el filtro.

Normalmente, los filtros se definen por sus respuestas a los componentes de frecuencia. Las respuestas de los filtros se clasifican en tres tipos en función de las frecuencias, la banda de paro, la banda de paso y la banda de transición. La respuesta de la banda de paso es permitir que los componentes de frecuencia pasen sin ser muy afectados.

Las frecuencias en la banda de detención de un filtro son muy reducidas. La banda de transición afecta a las frecuencias en el medio, que pueden recibir alguna reducción, pero no están separadas por completo de la señal.

Figura 153. **Respuesta en frecuencia de un filtro digital**



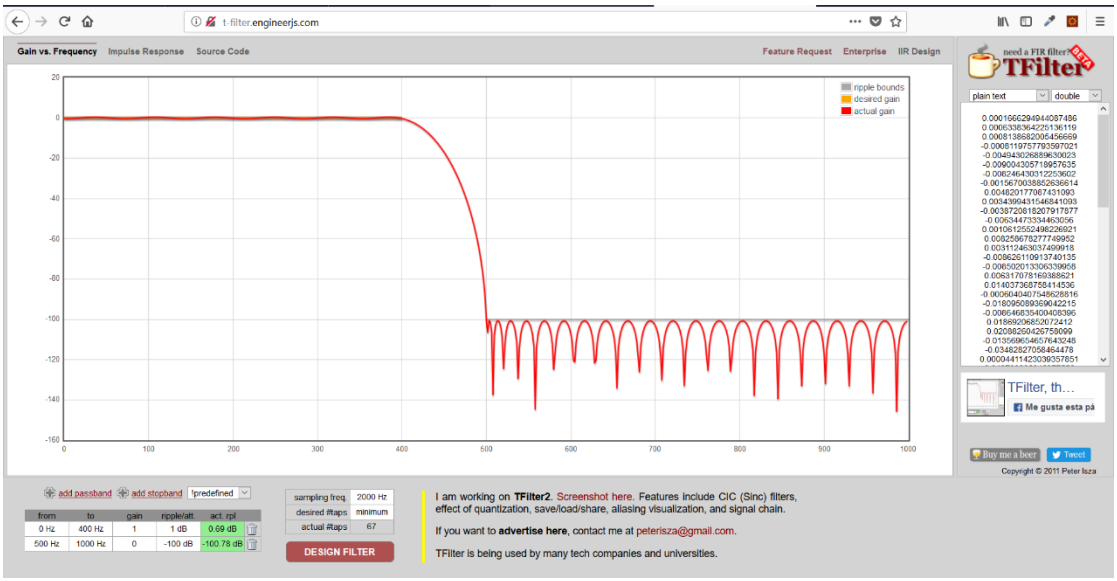
Fuente: Mathworks. *Introducción práctica al diseño de filtro digital*. https://la.mathworks.com/help/signal/examples/practical-introduction-to-digital-filter-design.html. Consulta: 23 de diciembre de 2018.

El diagrama de respuesta de frecuencia del filtro incluye una frecuencia final de banda de paso, como ωs la cual es la frecuencia de inicio de banda de parada, ωp es la cantidad de atenuación en la banda de detención. Las frecuencias ωp y ωs caen en la banda de transición y se reducen. Esto confirma que el filtro cumple con las especificaciones preferidas, incluido el ancho de banda de transición, la fluctuación, la longitud del filtro y los coeficientes. Cuanto más largo sea el filtro, más finamente se puede ajustar la respuesta. Con la longitud N y los coeficientes decididos la implementación del filtro FIR es muy simple.

Una buena opción para diseñar filtros por medio de una página web que se puede encontrar en internet es TFilter. En esta aplicación web se coloca en la parte inferior la frecuencia de muestreo y se agregan las necesarias, ya sea una frecuencia de paso, transición o de detención. Por último, se presiona el botón de diseñar filtro y muestra cuál sería el diagrama de respuesta, la gráfica de respuesta al impulso y el código necesario para generarlo en el lenguaje C.

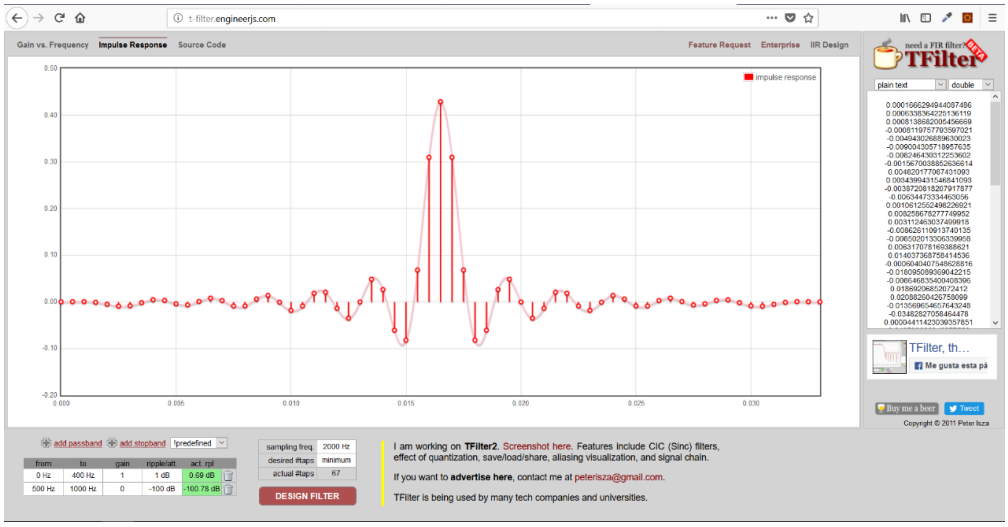
En la parte derecha se muestra una lista de números decimales, este es el arreglo de coeficientes necesarios para implementar el filtro diseñado.

Figura 154. **Gráfica de coeficientes generados para el filtro digital**



Fuente: elaboración propia.

Figura 155. **Gráfica de la respuesta al impulso del filtro digital**



Fuente: elaboración propia.

3.13.1. Implementación de un filtro FIR en una FPGA

Un filtro FIR responde a la ecuación:

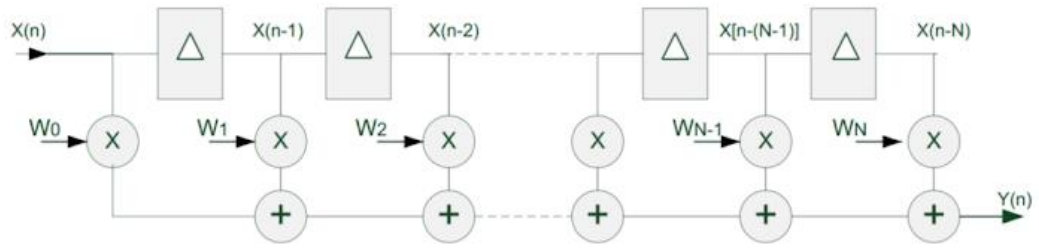
Figura 156. **Ecuación general del filtro digital**

$$y_n = w_0.x_n + w_1.x_{n-1} + w_2x_{n-2} + \ldots + w_{k-1}.x_{n-k-1} + w_{N-1}.x_{n-(N-1)}$$

Fuente: elaboración propia.

Donde Xn es el valor de la última muestra de la señal de entrada y Xn-1 hasta Xn(N-1) son las muestras anteriores. Wo hasta WN-1 son los N coeficientes del filtro e Yn es la señal de salida. El diagrama de bloques de un FIR puede representarse de la manera siguiente.

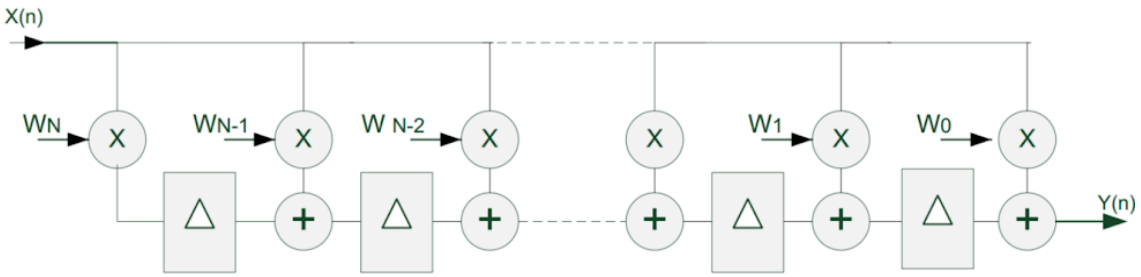
Figura 157. **Diagrama de bloques de un filtro digital**



Fuente: elaboración propia.

Ya que en una FPGA en cada ciclo de reloj solo hay que hacer una opción de multiplicación y otra se suma existe una estructura más adecuada para implementar y así mejorar la frecuencia máxima de trabajo.

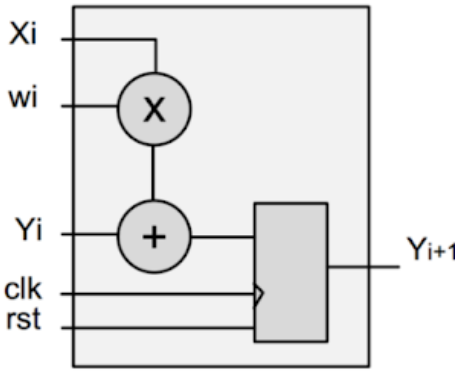
Figura 158. **Diagrama de bloques reducido de un filtro digital**



Fuente: elaboración propia.

Por lo tanto, el bloque básico para diseñar un filtro, que incluye un multiplicador y un sumador, se muestra a continuación.

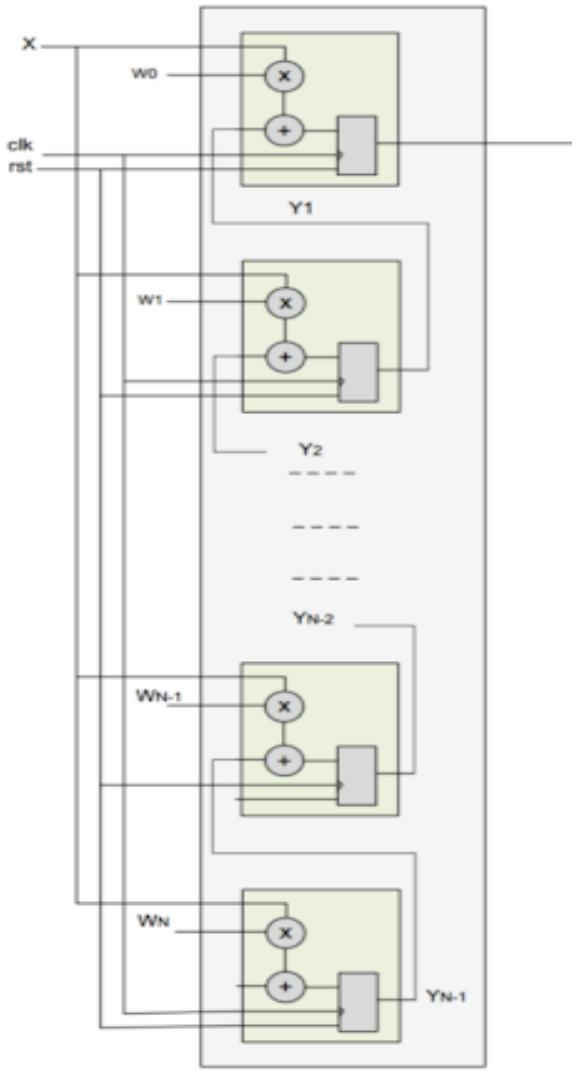
Figura 159. **Componente básico de un filtro digital**



Fuente: elaboración propia.

El diseño del filtro por implementar dentro de una FPGA incluye cada uno de estos bloques dependiendo de los coeficientes necesarios.

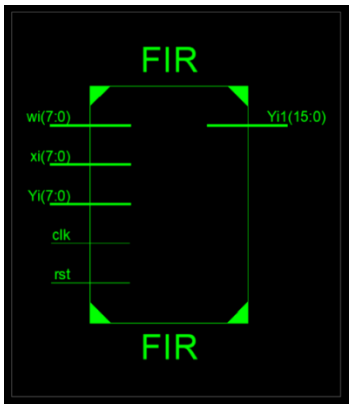
Figura 160. **Diagrama de filtro digital con sus componentes básicos**



Fuente: elaboración propia.

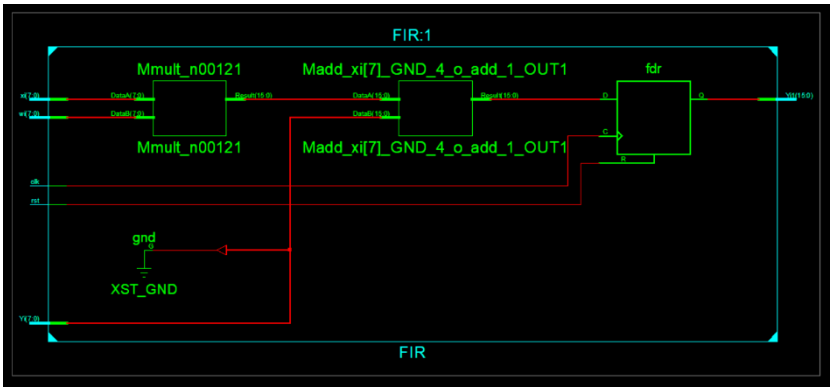
El módulo del bloque básico sintetizado en una FPGA se muestra a continuación.

Figura 161. **Módulo general de un filtro FIR**



Fuente: elaboración propia.

Figura 162. **Diagrama de bloques de un filtro FIR**



Fuente: elaboración propia.

Este módulo es muy simple de describir ya que solamente son operaciones muy básicas. Se debe convertir las señales xi, wi e Yi a señales tipo entero para procesarlas. Se procede a realizar la multiplicación y suma dentro de un proceso ya que se requiere un *flip-flop*. El módulo VHDL se muestra a continuación.

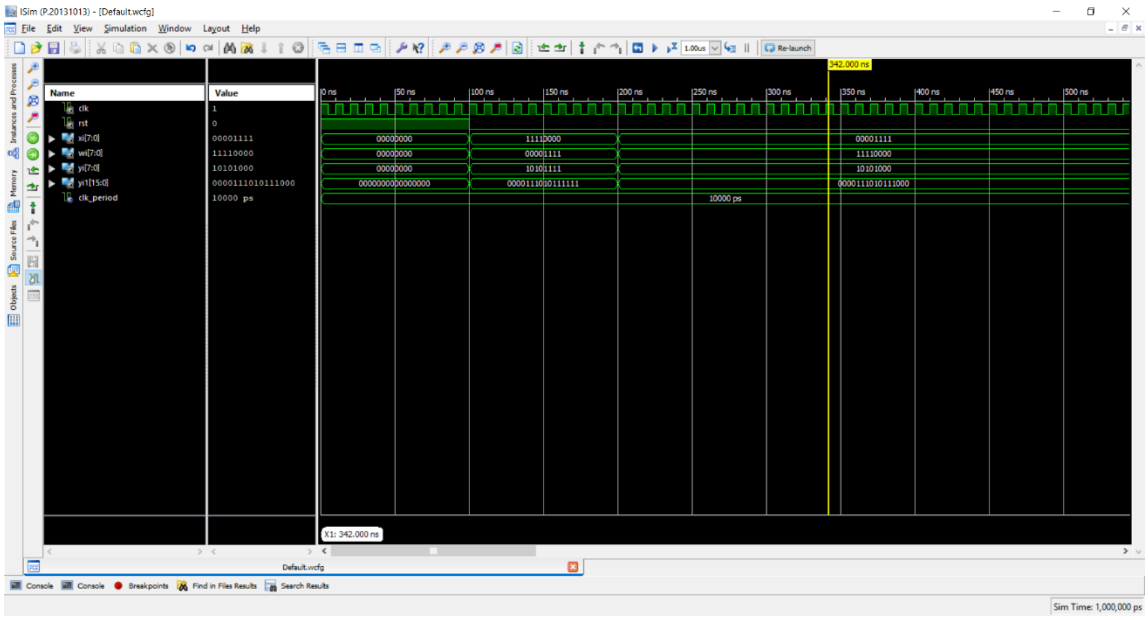
Figura 163. **Descripción de un módulo básico para un filtro FIR**

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

entity FIR is
generic(
    width : natural := 8
);
port(
    clk,rst : in std_logic;
    xi,wi,Yi : in std_logic_vector(width-1 downto 0);
    Yi1      : out std_logic_vector(width*2-1 downto 0)
);

end FIR;

architecture Behavioral of FIR is

signal tmp : integer range 0 to 2**(width*2)-1;

begin
    process(clk)
    begin
        if(rising_edge(clk))then
            if(rst = '1')then
                Yi1 <= (others=>'0');
            else
                Yi1 <= std_logic_vector(to_unsigned(to_integer(unsigned(xi))*to_integer(unsigned(wi))+to_integer(unsigned(Yi)),width*2));
            end if;
        end if;
    end process;

end Behavioral;
```

Fuente: elaboración propia.

Cuando se realiza la simulación de este módulo se puede observar que las operaciones se realizan correctamente.

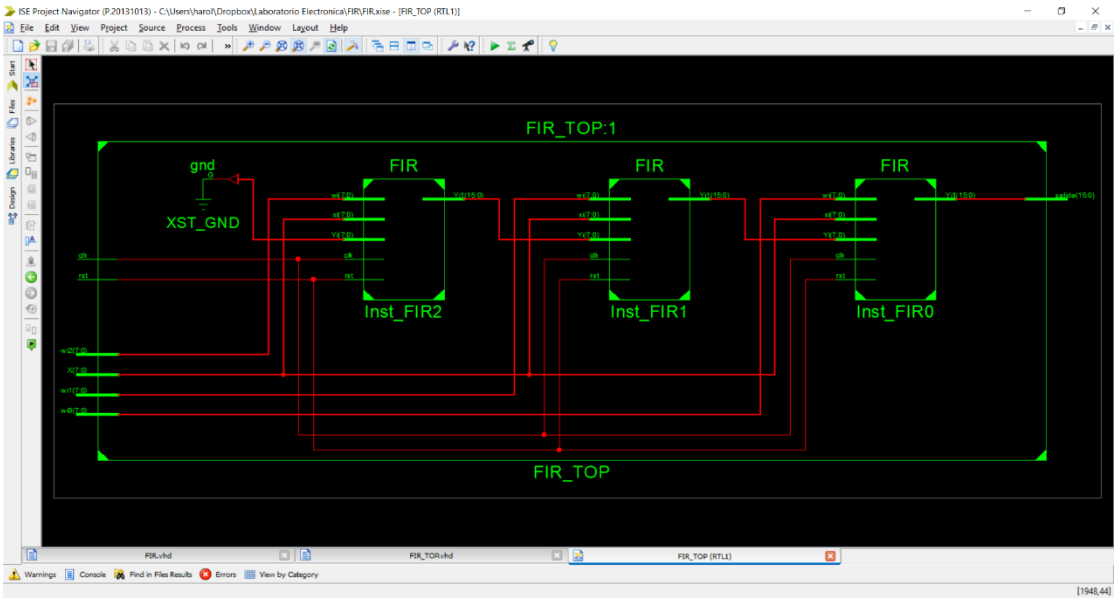
Figura 164. **Simulación del módulo básico de un filtro FIR**



Fuente: elaboración propia.

Por último, se debe realizar el acoplamiento de todos los módulos requeridos dependiendo de la cantidad de coeficientes que se obtengan del filtro. En este diseño solo se realizaron tres coeficientes para ejemplificar su conexión. El módulo final de la implementación del filtro FIR se muestra a continuación.

Figura 165. **Interconexiones de módulos básicos para la generación de un filtro FIR**



Fuente: elaboración propia.

Como se puede observar la conexión es la misma a la que se ilustró en la teoría. Se utilizan señales para almacenar el valor temporal requerido en las entradas de Yi-1, se debe tener muy en cuenta el valor del bus ya que las operaciones de multiplicación y suma causa que los números puedan tener valores muy altos.

En el caso de la señal de entrada de Yi al conectar la señal temporal se realiza un truncamiento de los valores más significativos para aproximar la operación. A continuación, se muestra la descripción de hardware con el lenguaje VHDL.

Figura 166. **Descripción de la interconexión de módulos básicos de un filtro FIR**

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

entity FIR_TOP is
generic(
    width: natural := 8
);
port(
    clk,rst : in std_logic;
    X       : in std_logic_vector(width-1 downto 0);
    wi0     : in std_logic_vector(width-1 downto 0);
    wi1     : in std_logic_vector(width-1 downto 0);
    wi2     : in std_logic_vector(width-1 downto 0);
    salida  : out std_logic_vector(width*2-1 downto 0)
);
end FIR_TOP;
architecture Behavioral of FIR_TOP is
COMPONENT FIR
    PORT(
        clk : IN std_logic;
        rst : IN std_logic;
        xi : IN std_logic_vector(width-1 downto 0);
        wi : IN std_logic_vector(width-1 downto 0);
        Yi : IN std_logic_vector(width-1 downto 0);
        Yil : OUT std_logic_vector(width*2-1 downto 0)
        );
    END COMPONENT;
    signal Ytmp1,Ytmp2 : std_logic_vector(width*2-1 downto 0);
begin
    Inst_FIR0: FIR PORT MAP(
        clk => clk,
        rst => rst,
        xi => X,
        wi => wi0,
        Yi => Ytmp1(15 downto 8),
        Yil => salida
    );
    Inst_FIR1: FIR PORT MAP(
        clk => clk,
        rst => rst,
        xi => X,
        wi => wi1,
        Yi => Ytmp2(15 downto 8),
        Yil => Ytmp1
    );
    Inst_FIR2: FIR PORT MAP(
        clk => clk,
        rst => rst,
        xi => X,
        wi => wi2,
        Yi => (others=>'0'),
        Yil => Ytmp2
    );
end Behavioral;
```

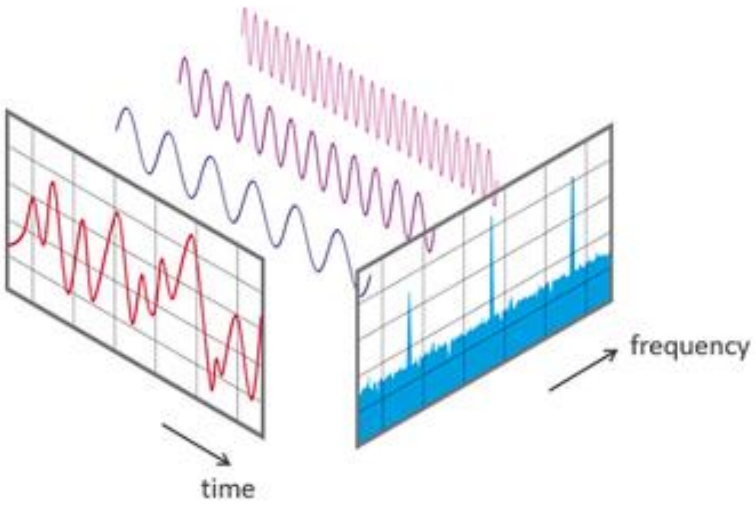
Fuente: elaboración propia.

3.13.2. Transformada rápida de *Fourier* (FFT)

Es un algoritmo que muestrea una señal en un período de tiempo y la divide en sus componentes de frecuencia. Estos componentes son oscilaciones senoidales únicas a frecuencias distintas, cada una con su propia amplitud y fase.

Este algoritmo calcula la transformada discreta de *Fourier* (DFT) de una secuencia. El análisis de *Fourier* convierte una señal de su dominio original en una representación en el dominio de frecuencia y viceversa. Una FFT computa rápidamente tales transformaciones al factorizar la matriz DFT en un producto de factores dispersos principalmente cero. Esto logra reducir la complejidad de calcular el DFT.

Figura 167. **Gráfica de la representación de *Fourier* de una señal en el tiempo**

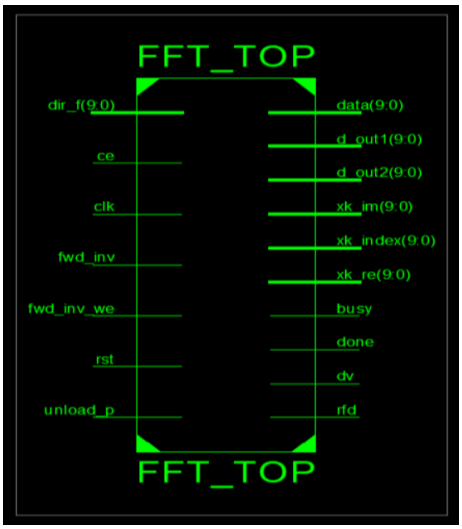


Fuente: elaboración propia.

3.14. Implementación de un módulo de FFT en una FPGA

En esta implementación se requiere que se tengan guardadas las muestras dentro de una memoria, la cual irá entregando cada dato una vez terminada la operación de transformada. A continuación, se muestran todas las señales de entrada y salida que se incluyen en el módulo.

Figura 168. **Módulo general de la transformada rápida de *Fourier***



Fuente: elaboración propia.

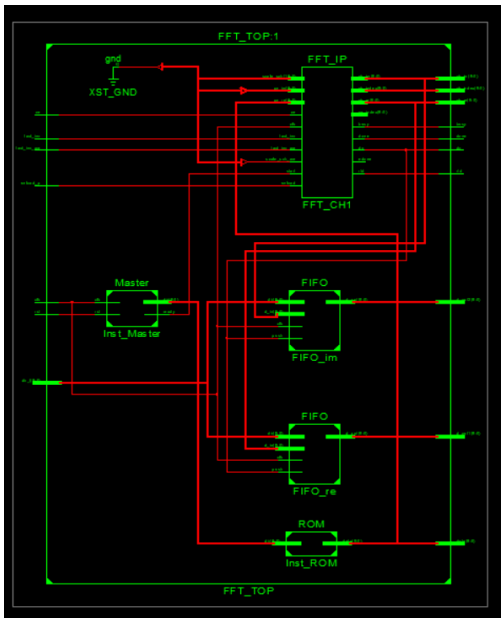
Las entradas de dir_f describe la entrada de dirección de las memorias de la señal transformada, hay una dirección para la parte real y otra para la parte imaginaria estas dos son almacenada en dos memorias distintas. La señal ce se utiliza para habilitar el módulo de transformada. Se tiene una entrada de reloj de sistema está puede funcionar bien con 100MHz.

La señal fwd_inv y la señal fwd_inv_we se utilizan cuando se desea habilitar la transformada inversa, si estos son cero se genera la operación de

transformada. Se incluye una señal de reset llamada rst para inicializar la operación de transformada. La entrada unload_p se utiliza para empezar a realizar la operación de transformada y se coloca en uno la salida llamada busy, luego de haber colocado un uno en esta señal, se pasa a un tiempo de espera a que finalice la operación.

Cuando esto sucede se pone a uno la salida llamada done y los datos se colocan en las salidas xk_im y xk_re, las cuales resultan en la parte real e imaginaria respectivamente de la transformada rápida de *Fourier*. Se describe a continuación el diseño interno de cómo funciona este diseño.

Figura 169. **Interconexiones del módulo de transformada rápida de *Fourier***

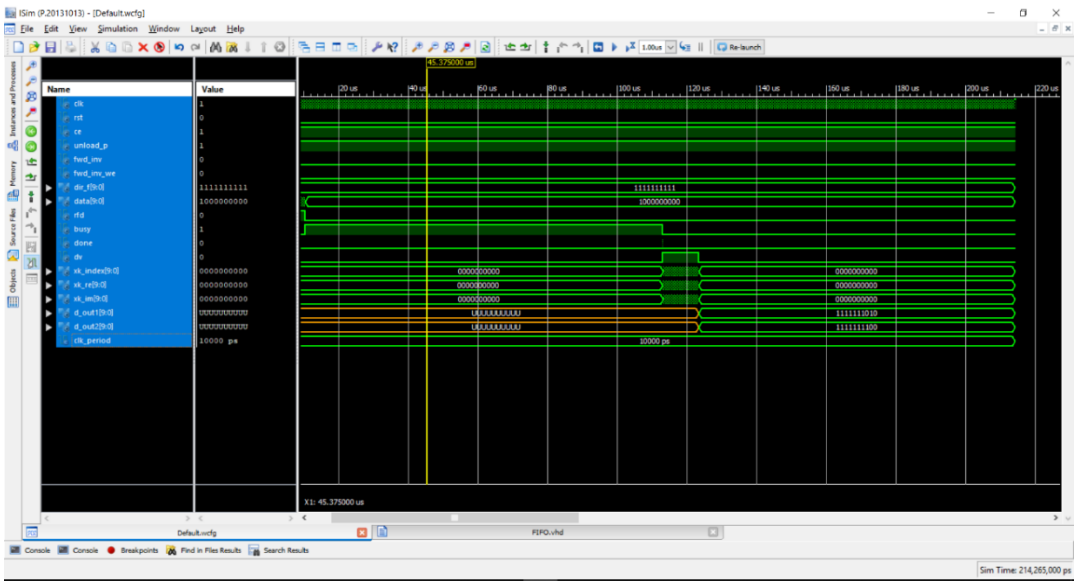


Fuente: elaboración propia.

Se muestra en la figura cinco módulos que componen este diseño. El primer módulo es el llamado ROM, este es el encargado de entregar los datos al puerto de datos del módulo de FFT_IP, esta entrada se llama xn_re en el módulo. Se utiliza solamente la entrada de la parte real ya que son valores muestreados de voltaje.

Esta memoria recibe la dirección desde el módulo llamado máster el cual lanza mil veinticuatro datos a la velocidad de reloj del sistema, cuando se completa esta cantidad de datos se empieza a realizar la transformada de todos los datos. Por último, se tienen dos memorias las cuales guardan cada valor que se genera en la transformada, estos valores se obtienen a cada ciclo de reloj del sistema, por lo tanto, se debe ir guardando uno a uno. Se muestra a continuación una simulación donde se resume todo el comportamiento del módulo implementado.

Figura 170. **Simulación de la transformada rápida de *Fourier***



Fuente: elaboración propia.

Se puede observar que desde el tiempo cero se pone a uno la señal de rfd, aproximadamente a los 10 milisegundos se pone a uno la señal de busy. El proceso toma aproximadamente 100 milisegundos y cuando finaliza se levanta la señal done la señal dv, esto permanecerá así hasta que se hayan entregado todos los datos. Se puede notar en la imagen que hay una serie de datos que duran un tiempo aproximado diez milisegundos, estos son los datos de la parte real e imaginaria que se van entregando a la frecuencia de reloj.

La descripción de hardware en lenguaje VHDL por cada módulo se realiza a continuación. La primera imagen define los puertos ya descritos anteriormente.

Figura 171. **Descripción de los puertos utilizados en el módulo de transformada rápida de *Fourier***

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

entity FFT_TOP is
generic(
    width : natural := 10
);
port(
    data : out std_logic_vector(width-1 downto 0);
    clk  : in std_logic;
    rst  : in std_logic;

    -- FFT ports
    ce : in std_logic;
    unload_p: in std_logic;
    fwd_inv: in std_logic;
    fwd_inv_we: in std_logic;
    rfd : out std_logic;
    busy : OUT STD_LOGIC;
    done : OUT STD_LOGIC;
    dv : OUT STD_LOGIC;
    xk_index : OUT STD_LOGIC_VECTOR(width-1 DOWNTO 0);
    xk_re : OUT STD_LOGIC_VECTOR(width-1 DOWNTO 0);
    xk_im : OUT STD_LOGIC_VECTOR(width-1 DOWNTO 0);

    -- FIFO;
    dir_f: in   std_logic_vector(width-1 downto 0);
    d_out1: out std_logic_vector(width-1 downto 0);
    d_out2: out std_logic_vector(width-1 downto 0)

);
end FFT_TOP;
```

Fuente: elaboración propia.

Se realiza la definición de componentes para luego hacer la instancia de cada uno.

Figura 172. **Descripción de componentes en el módulo de transformada rápida de *Fourier***

```
COMPONENT Master
    PORT(
        clk : IN std_logic;
        rst : IN std_logic;
        dir : OUT std_logic_vector(9 downto 0);
        ready : OUT std_logic
        );
    END COMPONENT;


 COMPONENT ROM
     PORT(
         dir : IN std_logic_vector(9 downto 0);
         data : OUT std_logic_vector(9 downto 0)
         );
     END COMPONENT;


COMPONENT FFT_IP
  PORT (
    clk : IN STD_LOGIC;
    ce : IN STD_LOGIC;
    start : IN STD_LOGIC;
    unload : IN STD_LOGIC;
    xn_re : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    xn_im : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    fwd_inv : IN STD_LOGIC;
    fwd_inv_we : IN STD_LOGIC;
    scale_sch : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    scale_sch_we : IN STD_LOGIC;
    rfd : OUT STD_LOGIC;
    xn_index : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
    busy : OUT STD_LOGIC;
    edone : OUT STD_LOGIC;
    done : OUT STD_LOGIC;
    dv : OUT STD_LOGIC;
    xk_index : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
    xk_re : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
    xk_im : OUT STD_LOGIC_VECTOR(9 DOWNTO 0)
  );
END COMPONENT;


COMPONENT FIFO
    PORT(
        push : IN std_logic;
        clk : IN std_logic;
        dir : IN std_logic_vector(9 downto 0);
        d_in : IN std_logic_vector(9 downto 0);
        d_out : OUT std_logic_vector(9 downto 0)
        );
    END COMPONENT;
```

Fuente: elaboración propia.

Las señales declaradas para la intercomunicación de módulos se describen a continuación.

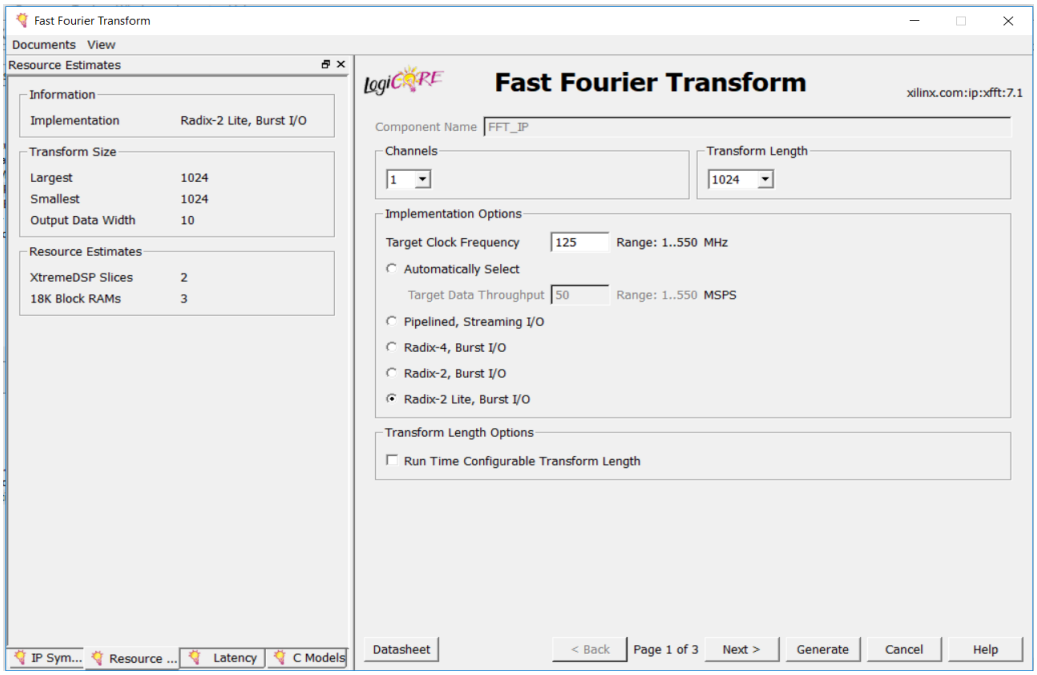
Figura 173. **Descripción del módulo de transformada rápida de *Fourier***

```
signal dir_s : std_logic_vector(width-1 downto 0);
signal start_s: std_logic;
signal data_s : std_logic_vector(width-1 downto 0);
signal dv_s : std_logic;
signal xk_re_s : STD_LOGIC_VECTOR(width-1 DOWNTO 0);
signal xk_im_s : STD_LOGIC_VECTOR(width-1 DOWNTO 0);

signal d_out1_s: std_logic_vector(width-1 downto 0);
signal d_out2_s: std_logic_vector(width-1 downto 0);


          Inst_Master: Master PORT MAP(
                  clk => clk,
                  rst => rst,
                  dir => dir_s,
                  ready => start_s
              );

          Inst_ROM: ROM PORT MAP(
                  dir => dir_s,
                  data => data_s
              );

          FFT_CH1 : FFT_IP
            PORT MAP (
              clk => clk,
              ce => ce,
              start => start_s,
              unload => unload_p,
              xn_re => data_s,
              xn_im => (others=>'0'),
              fwd_inv => fwd_inv,
              fwd_inv_we => fwd_inv_we,
              scale_sch => (others=>'0'),
              scale_sch_we => '0',
              rfd => rfd,
              xn_index => open,
              busy => busy,
              edone => open,
              done => done,
              dv => dv_s,
              xk_index => xk_index,
              xk_re => xk_re_s,
              xk_im => xk_im_s
            );


          FIFO_re: FIFO PORT MAP(
                  push => dv_s,
                  clk => clk,
                  dir => dir_f,
                  d_in => xk_re_s,
                  d_out => d_out1_s
              );

          FIFO_im: FIFO PORT MAP(
                  push => dv_s,
                  clk => clk,
                  dir => dir_f,
                  d_in => xk_im_s,
                  d_out => d_out2_s
              );

              data <= data_s;
              dv <= dv_s;
              xk_re <= xk_re_s;
              xk_im <= xk_im_s;
              d_out1 <= d_out1_s;
              d_out2 <= d_out2_s;
```

Fuente: elaboración propia.

El componente de FFT se implementa mediante la utilización de un ipcore, llamado *Fast Fourier Transform.*

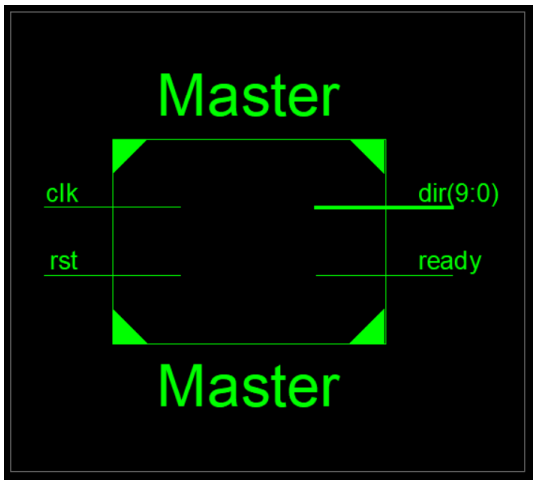
Figura 174. **Interfaz para la creación de transformada rápida de *Fourier***



Fuente: elaboración propia.

En el módulo de máster simplemente es una máquina de estados con tres estados, esperar, tomar y RYS. El primer estado se utiliza para el momento en que no se ha iniciado el proceso de toma de muestras. El estado tomar se utiliza para hacer el conteo de direcciones de la memoria y así realizar la lectura de la misma. Concurrentemente se hace la conversión de entero a vector y se asigna a la señal de dirección dir. Por último, se habilita la señal *ready* para poner el dato en módulo ipCore de FFT.

Figura 175. **Módulo general para el controlador de la transformada rápida de *Fourier***



Fuente: elaboración propia.

Figura 176. **Descripción del módulo de transformada rápida de *Fourier***

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Master is
generic(
    width : natural := 10
);
port(
    clk : in std_logic;
    rst : in std_logic;
    dir : out std_logic_vector(width-1 downto 0);
    ready:out std_logic
);
end Master;

architecture Behavioral of Master is

signal contador : integer range 0 to 2**width;
signal ready_s  : std_logic;

type FSM is (ESPERAR,TOMAR,RYS);
signal current_state : FSM;
```

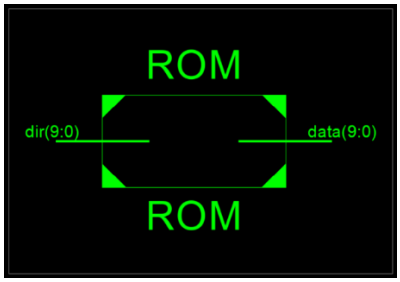
Continuación de la figura 176.

```
process(clk,rst)
begin
    if(rst='1')then
        contador <= 0;
        current_state <= RYS;
        ready_s <= '0';
    else
        if(rising_edge(clk))then
            case current_state is
                when RYS =>
                    current_state <= TOMAR;
                    ready_s <= '1';
                when TOMAR =>
                    if contador < 2**width then
                        contador <= contador + 1;
                        current_state <= TOMAR;
                    else
                        contador <= 0;
                        current_state <= ESPERAR;
                        ready_s <= '0';
                    end if;
                when others =>
                    current_state <= ESPERAR;
            end case;
        end if;
    end if;
end process;


dir <= std_logic_vector(to_unsigned(contador,width));
ready <= ready_s;

end Behavioral;
```

Fuente: elaboración propia.

El módulo de la memoria ROM, como su definición indica se guardan datos estáticos sintetizados previamente muestreados de la señal. Este puede ser una memoria de escritura, pero se debe implementar otro diseño distinto.

Figura 177. **Módulo general de memoria**



Fuente: elaboración propia.

La memoria en el módulo es una señal declarada como una memoria de n datos.

Figura 178. **Declaración de una memoria en la transformada rápida de *Fourier***

```
type srom is array (0 to 2**width-1) of integer range 2**width - 1 downto 0;
signal srom_1 : srom;
```

Fuente: elaboración propia.

Luego se asigna concurrentemente los valores para que sean estáticos y no cambien. Como también se asigna a un vector de salida llamado data en la dirección específica. El número total de datos es mil veinticuatro, los cuales fueron previamente elegidos de una señal muestreada. En la siguiente imagen se muestra la asignación de la entrada dir.

Figura 179. **Asignación de valores de la señal muestreada**

```
data <= std_logic_vector(to_signed(srom_1(to_integer(unsigned(dir))),width));

srom_1(0) <= 512;
srom_1(1) <= 584;
srom_1(2) <= 656;
srom_1(3) <= 724;
srom_1(4) <= 788;
srom_1(5) <= 846;
srom_1(6) <= 898;
srom_1(7) <= 942;
```

Fuente: elaboración propia.

Por último, se tienen los módulos de FIFO, este tipo de memoria ingresa el dato a cada posición hasta llenar todas las posiciones necesarias para luego ser leída. En estas se almacenan los resultados de la transformada de *Fourier*.

Como se describió anteriormente se implementaron dos memorias unas para la parte real y otra para la parte imaginaria.

En el diseño de VHDL se tiene simplemente un *flip flop* que coloca el dato en la posición de la señal cont y luego suma uno al conteo. Concurrentemente se muestra el dato contenido en la memoria en la posición que se requiera dependiendo de la señal dir. La descripción de hardware se muestra a continuación.

Figura 180. **Módulo de descripción para la memoria contenedora de la señal**

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

entity FIFO is
generic(
    width : natural := 10
);
port(
    push : in std_logic;
    clk  : in std_logic;
    dir  : in std_logic_vector(width-1 downto 0);
    d_in : in std_logic_vector(width-1 downto 0);
    d_out: out std_logic_vector(width-1 downto 0)
);
end FIFO;

architecture Behavioral of FIFO is
type mem is array(2**width-1 downto 0) of std_logic_vector(width-1 downto 0);
signal mem1 : mem;
signal cont : integer range 0 to 2**width-1;

begin

process(clk)
begin
    if(rising_edge(clk))then
        if(push = '1')then
            mem1(cont) <= d_in;
            cont <= cont + 1;
        else
            cont <= 0;
            end if;
    end if;
end process;

d_out <= mem1(to_integer(unsigned(dir)));

end Behavioral;
```

Fuente: elaboración propia.

3.15. Sintetizador digital directo o DDS.

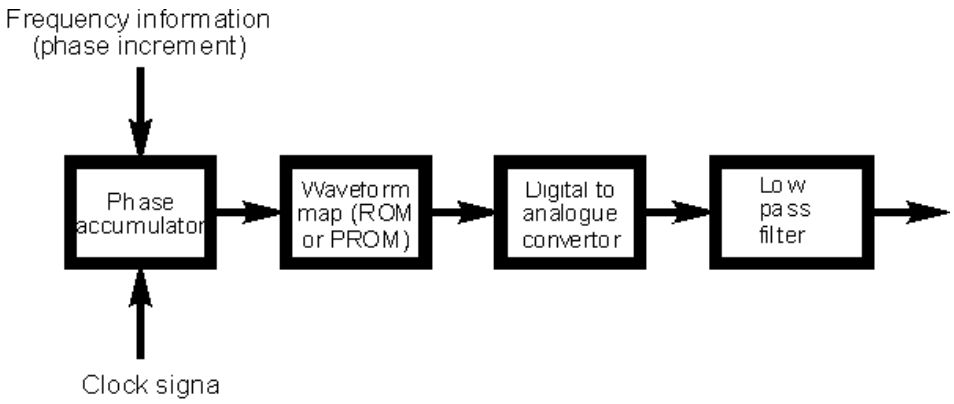
La síntesis digital directa (DDS) es una técnica utilizada en la generación de señales de radiofrecuencia para su uso en una variedad de aplicaciones, desde receptores de radio hasta generadores de señales. La técnica se ha extendido mucho más en los últimos años con los avances logrados en la tecnología de circuitos integrados que permiten manejar velocidades mucho más rápidas lo que, a su vez permite, que se realicen chips DDS de mayor frecuencia.

Aunque a menudo se usa solo, la síntesis directa digital se usa a menudo en conjunción con lazo de seguimiento de fase o PLL. Combinando ambas tecnologías es posible aprovechar los mejores aspectos de cada una.

Esta forma de síntesis genera la forma de onda directamente utilizando técnicas digitales. Esto es diferente a la forma en la que otros sintetizadores indirectos usuales funcionan ya que solo utilizan el PLL como base de su operación.

Un sintetizador digital directo funciona almacenando las muestras de una forma de onda en formato digital y luego se recuperan para generar la forma de onda. La velocidad a la que el sintetizador completa una forma de onda determina la frecuencia. El diagrama de bloques general se muestra a continuación.

Figura 181. **Diagrama de bloques de un sintetizador digital**



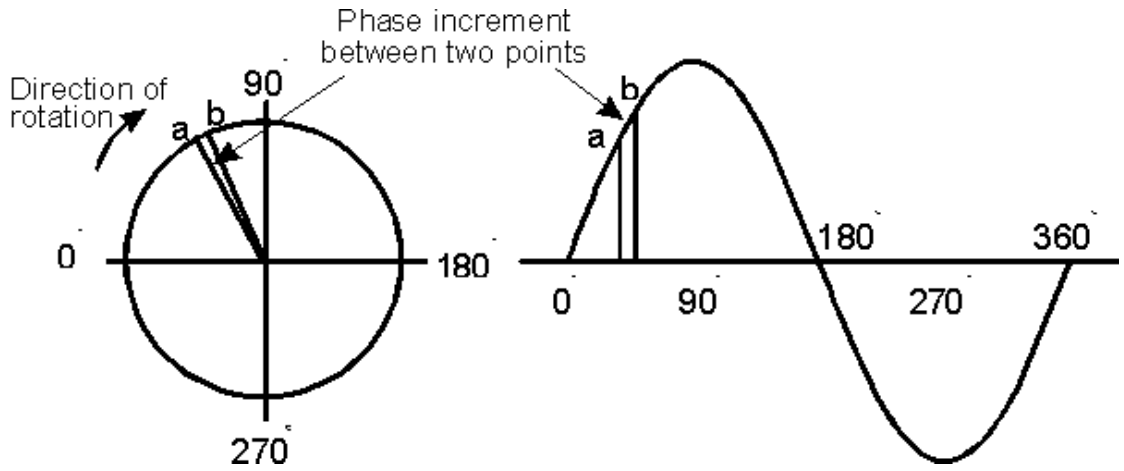
Fuente: elaboración propia.

Su funcionamiento se puede explicar con más detalle considerando que la fase avanza alrededor de un círculo. A medida que la fase avanza alrededor del círculo, esto corresponde a los avances en la forma de onda, es decir, cuanto mayor sea el número correspondiente a la fase, mayor será el punto que está a lo largo de la forma de onda. Al avanzar sucesivamente el número correspondiente a la fase, es posible avanzar más a lo largo del ciclo de la forma de onda.

El número digital que representa la fase se mantiene en el acumulador de fase. El número que se mantiene aquí corresponde a la fase y se incrementa a intervalos regulares. De esta manera, puede enviarse al acumulador de fase básicamente como una forma de contador.

Cuando está sincronizado, agrega un número preestablecido al que ya está guardado. Cuando se llena, se reinicia y comienza a contar desde cero nuevamente. En otras palabras, esto corresponde a alcanzar un círculo completo en el diagrama de fase y reiniciar nuevamente.

Figura 182. **Descripción de fase para un sintetizador de señales**



Fuente: elaboración propia.

Una vez que se ha determinado la fase, es necesario convertir esto en una representación digital de la forma de onda. Esto se logra utilizando un mapa de forma de onda. Esta es una memoria que almacena un número correspondiente al voltaje requerido para cada valor de fase en la forma de onda.

En la mayoría de los casos, la memoria es una memoria de solo lectura (ROM) o una memoria de solo lectura programable (PROM). Esto contiene una gran cantidad de puntos en la forma de onda, muchos más de los que se accede a cada ciclo. Se requiere de un número muy grande de puntos para que el acumulador de fase pueda incrementarse en un cierto número de puntos para establecer la frecuencia requerida.

La siguiente etapa en el proceso es convertir los números digitales provenientes de la tabla de búsqueda de seno en un voltaje analógico. Esto se logra utilizando un convertidor digital a analógico (DAC). Esta señal se filtra

para eliminar cualquier señal no deseada y se amplifica para proporcionar el nivel requerido según sea necesario.

La sintonización se logra aumentando o disminuyendo el tamaño del paso o incremento de fase entre diferentes puntos de muestra. Un incremento mayor en cada actualización del acumulador de fase significa que la fase alcanza el valor del ciclo completo más rápido y la frecuencia es más alta. Incrementos menores en el valor del acumulador de fase significa que lleva más tiempo aumentar el valor de ciclo completo y un valor de frecuencia correspondientemente bajo.

De esta manera es posible controlar la frecuencia. También se puede ver que los cambios de frecuencia se pueden hacer al instante simplemente cambiando el valor de incremento.

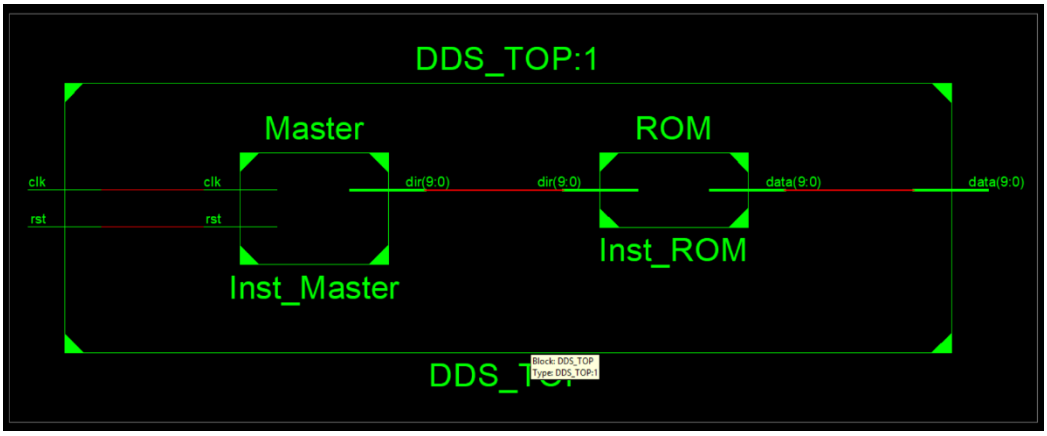
Con esto se llega a la conclusión que hay una diferencia finita entre una frecuencia y la siguiente, y que la diferencia de frecuencia mínima o resolución de frecuencia está determinada por el número total de puntos disponibles en el acumulador de fase. Un acumulador de fase de 24 bits proporciona algo más de 16 millones de puntos y proporciona una resolución de frecuencia de aproximadamente 0,25 Hz cuando se utiliza con un reloj de 5 MHz.

Estos sintetizadores tienen algunas desventajas. Hay una serie de señales espurias que son generadas por un sintetizador digital directo. El más importante de estos es el llamado señal de alias. Las imágenes de la señal se generan a cada lado de la frecuencia de reloj y sus múltiplos. Por ejemplo, si la señal requerida tenía una frecuencia de 3 MHz y el reloj estaba a 10 MHz, las señales de alias aparecerán a 7 MHz y 13 MHz, así como a 17 MHz y 23 MHz, y otros. Se pueden eliminar mediante el uso de un filtro de paso bajo.

3.15.1. Implementación de un DDS en una FPGA

Para este diseño de hardware se crearon dos módulos, el primero es llamado master, este es el encargado de crear la frecuencia a la que se desea mostrar la señal. Este módulo, además, accede a una memoria ROM que contiene todas las muestras de la señal deseada. A continuación, se muestran los módulos implementados en una FPGA.

Figura 183. **Diagrama general de un sintetizador digital**



Fuente: elaboración propia.

El módulo de master se compone de dos procesos el primero es un generador de señal el cual realiza una división de reloj. Esta división depende de una señal de conteo, se cuentan los flancos de subida, es decir, se hace un conteo de periodos. Quiere decir que el periodo deseado de muestreo debe ser un múltiplo del periodo del reloj del sistema.

Este subreloj que se crea será el encargado de activar el proceso que llevará el conteo y la asignación de la dirección de la memoria para acceder al dato deseado. Se muestra el módulo de la descripción de hardware de VHDL.

Figura 184. **Descripción del generador de reloj para el sintetizar de señal**

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

entity Master is
generic(
    width: natural := 10
);
port(
    clk,rst : in std_logic;
    dir     : out std_logic_vector(width-1 downto 0)
);
end Master;

architecture Behavioral of Master is

signal conta  : integer range 0 to 2**width-1;
signal cont_n : integer range 0 to 12;
signal sclk   : std_logic;

begin
    process(clk,rst)
    begin
        if(rst = '1')then
            cont_n <= 0;
        elsif(rising_edge(clk))then
            if(cont_n = 12)then
                cont_n <= 0;
            else
                cont_n <= cont_n + 1;
            end if;
        end if;
    end process;

    process(sclk)
    begin
        if(rising_edge(sclk)) then
            if(rst = '1') then
                conta <= 0;
            else
                dir <= std_logic_vector(to_signed(conta,width));
                conta <= conta + 1;
            end if;
        end if;
    end process;
    sclk <= '1' when (cont_n<6) else '0';
end Behavioral;
```

Fuente: elaboración propia.

La memoria en el módulo es una señal declarada como una memoria de n datos.

Figura 185. **Declaración de la memoria de la señal**

```
type srom is array (0 to 2**width-1) of integer range 2**width - 1 downto 0;
signal srom_1 : srom;
```

Fuente: elaboración propia.

Después de la declaración se asignan concurrentemente los valores para que sean estáticos y no cambien. Como también se asigna a un vector de salida llamado data en la dirección específica. El número total de datos es mil veinticuatro, los cuales fueron previamente elegidos de una señal muestreada. En la siguiente imagen se muestra la asignación de la entrada dir a la posición de la memoria para obtener el dato en esa posición y también la asignación de los valores a la misma.

Figura 186. **Asignación de la señal muestreada**

```
data <= std_logic_vector(to_signed(srom_1(to_integer(unsigned(dir))),width));

srom_1(0) <= 512;
srom_1(1) <= 584;
srom_1(2) <= 656;
srom_1(3) <= 724;
srom_1(4) <= 788;
srom_1(5) <= 846;
srom_1(6) <= 898;
srom_1(7) <= 942;
```

Fuente: elaboración propia.

Como se puede notar este módulo es simplemente una memoria con valores estáticos con acceso por dirección a sus valores.

La instancia de los componentes es conectarlos mediante una señal llamada dir_tmp esta une el componente ROM con el componente máster por

medio de la salida y entrada dir correspondiente a cada módulo. Se muestra a continuación el diseño propuesto en VHDL.

Figura 187. **Descripción del módulo sintetizador de señal**

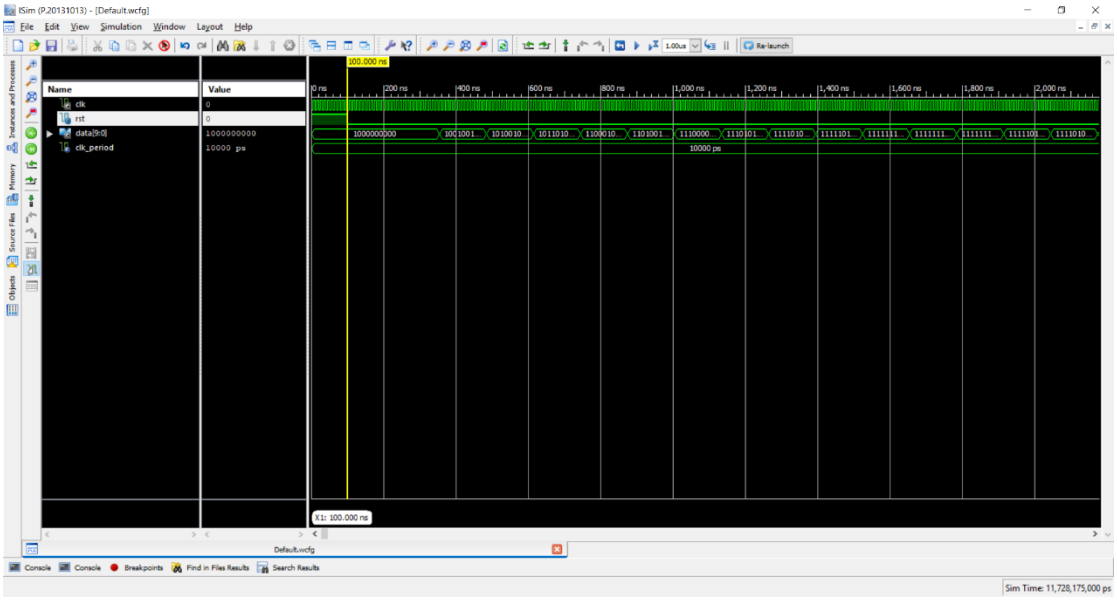
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DDS_TOP is
generic(
    width : natural := 10
);
port(
    clk,rst : in std_logic;
    data    : out std_logic_vector(width-1 downto 0)
);
end DDS_TOP;

architecture Behavioral of DDS_TOP is

COMPONENT Master
    PORT(
        clk : IN std_logic;
        rst : IN std_logic;
        dir : OUT std_logic_vector(width-1 downto 0)
        );
    END COMPONENT;

    COMPONENT ROM
    PORT(
        dir : IN std_logic_vector(width-1 downto 0);
        data : OUT std_logic_vector(width-1 downto 0)
        );
    END COMPONENT;

    signal dir_tmp : std_logic_vector(width-1 downto 0);

begin

    Inst_Master: Master PORT MAP(
        clk => clk,
        rst => rst,
        dir => dir_tmp
    );

    Inst_ROM: ROM PORT MAP(
        dir => dir_tmp,
        data => data
    );

end Behavioral;
```

Fuente: elaboración propia.

En la simulación de este módulo se puede notar que se va mostrando cambios en la salida data del módulo. Como se explica anteriormente el módulo muestra valores que están guardados en la memoria a la frecuencia que se

muestrearon, en esta implementación se desea mostrar una señal senoidal a tres mil hercios. Debido a esto el contador debe ser igual a trece y así obtener la división de reloj deseada.

Figura 188. **Simulación del sintetizador de señales**



Fuente: elaboración propia.

4. APLICACIONES DE FPGA A MICROCONTROLADORES

4.1. Microprocesador básico sintetizado programable

Una aplicación muy interesante en la síntesis de hardware es la de poder generar microcontroladores embebidos que funcionen conjuntamente con lógica concurrente. Para esta aplicación se describirá un diseño con conceptos básicos para que el diseñador comprenda cómo implementar la lógica necesaria de un microcontrolador que está compuesto por registros, memorias y unidades de funcionamiento lógico.

Un usuario que utilice el microcontrolador necesitará una interfaz y un lenguaje para poder programar el microcontrolador sintetizado, la opción que se utilizará es crear un lenguaje parecido a *assembler* para que se entienda cada una de las instrucciones.

El microcontrolador ejecuta cada una de las instrucciones, las cuales están almacenadas en una memoria interna. Este al estar procesando cada instrucción, dependiendo de cuál sea, modificará los registros, puertos, espacios de memoria, y otros. El manejo del puntero es muy importante ya que este es el encargado de ir ejecutando cada una de las instrucciones, el diseñador debe tomar en cuenta que existen los saltos y las llamadas a subrutinas.

Todas las instrucciones son programadas por un usuario, por lo tanto, es necesario un software que sea capaz de interpretar cada una de las instrucciones creadas en lenguaje *assembl*er y convertirlas a binario. Las

instrucciones binarias son enviadas por el protocolo de comunicación UART, desde el puerto serial de una computadora hasta el puerto serial de una FPGA. Cada instrucción binaria tiene un formato específico para luego ser decodificada e interpretada por el microcontrolador sintetizado.

El *software* que se utilizará como interprete simplemente debe leer un archivo, línea por línea guardarlo en arreglos de *Strings* y separarlo en partes las cuales se convertirán en binario, dependiendo de cada instrucción. Si el neumónico está seguido por otra palabra, estos deberán estar separados por un espacio en blanco.

Las direcciones y los datos tendrán que estar separados por comas. Si se desea acceder a una posición de memoria este debe estar limitado por paréntesis. Al enviar los datos solamente se trabajará con valores en sistema decimal enteros positivos o negativos, utilizar complemento a dos.

A continuación, se describen las instrucciones propuestas para la generación del microcontrolador.

4.2. Formato de las instrucciones binarias

Para enviar las instrucciones por comunicación UART se utilizarán formatos especiales. Este formato dependerá de la instrucción que se quiera interpretar. El tamaño de la trama de bits no se debe modificar. La longitud total de la trama por enviar por comunicación UART es de 22 bits.

Tabla III. **Asignación de bits para la comunicación de instrucciones**

Neumónico	**Dirección**	**Dirección**
[5 bits]	[8 bits]	[9 bits]

Neumónico	Dirección	Dato
[5 bits]	[8 bits]	[9 bits]

Neumónico	Indicador	Dirección
[5 bits]	[8 bits]	[9 bits]

Neumónico	Dirección
[5 bits]	[9 bits]

Neumónico
[5 bits]

Fuente: elaboración propia.

4.3. Registros por implementar en el microcontrolador

Se implementarán los siguientes registros debido a su simpleza y a su capacidad de dar un gran concepto al diseño de microntroladores.

- A: este es un registro de 8 bits que funciona como el acumulador. Este registro se utiliza generalmente como destino de muchas operaciones aritméticas, de comparaciones y testeos.

- B, C, D, E, H, L: estos son registros de 8 bits para propósito general, utilizados para operaciones, almacenamiento de valores y otros.

- SP: registro de 16 bits llamado puntero de pila, este apunta a la posición actual del lector de la pila.

- PC: el *Program Counter* o contador de programa es un registro de 16 bits. El cual contiene la dirección de la instrucción actual por ejecutar. El PC no se modifica directamente moviendo valores a este registro, solamente se modificará mediante instrucciones de salto (JP, JR, CALL, y otros)

- Registro de flags: será llamado F, este no es un registro de propósito general en donde se pueda introducir valores a voluntad. Cada uno de los diferentes bits del registro F tiene un significado propio que cambia automáticamente según el resultado de las operaciones anteriores.

Los bits por implementar en los registros serán los siguientes:

- Flag S (sign o signo): este flag se pone a uno si el resultado de la operación realizada en complemento a dos es negativo (es una copia del bit más significativo del resultado).

- Flag Z (zero o cero): este flag se pone a uno si el resultado de la última operación que afecte a los flags es cero.

- Flag P/V (*Parity/Overflow* o paridad/desbordamiento): en las operaciones que modifican el bit de paridad, este bit vale 1 si el número de unos del resultado de la operación es par, y 0 si es impar. Si, por contra, el resultado de la operación realizada necesita más bits para ser representado de los que provee el registro, se tendrá un desbordamiento, con este flag a 1. Este mismo bit sirve pues para 2 tareas, y indicará una u otra (paridad o desbordamiento) según sea el tipo de operación que se

haya realizado. El flag de desbordamiento se activará cuando en determinadas operaciones se pase de valores 11111111b a 00000000b, por “falta de bits” para representar el resultado o viceversa.

- Flag N (*substract* o resta): se pone a 1 si la última operación realizada fue una resta. Se utiliza en operaciones aritméticas.

- Flag C (*carry* o acarreo): este flag se pone a uno si el resultado de la operación anterior no cupo en el registro y necesita un bit extra para ser representado. Este bit es ese bit extra. Se verá su uso cuando se trate de las operaciones aritméticas, en esta misma entrega.

Figura 189. **Banderas implementadas en el microcontrolador**

Fuente: elaboración propia.

4.3.1. El set de instrucciones por implementar en el microcontrolador sintetizado

Como se estableció anteriormente la serie de instrucciones es necesaria para la interacción de los usuarios programadores y el hardware generado. En este diseño se implementarán 15 instrucciones básicas. A continuación, se describirá cada una de las instrucciones. La representación binaria de la instrucción se indica entre corchetes.

- LD [00000]: instrucción para la carga de datos en los diferentes registros, este no afectará ningún valor del registro de banderas.

  La sintaxis de esta instrucción es:

  - LD destino, origen

  Las funciones de esta instrucción serán las siguientes:

  - Colocar un valor en un registro

    - LD A, 10 ; A = 10
    - LD B, 200 ; B = 200

  - Copiar el valor de un registro a otro registro

    - LD A, B *; A = B*

- Escribir en memoria (en una dirección determinada) un valor.
  - LD (12345), 10 *; Memoria[12345] = valor en A*
- Escribir en memoria (en una dirección determinada) el contenido de un registro.
  - LD (12345), **A** *; Memoria[12345] = valor en A*
- Asignarle a un registro el contenido de una dirección de memoria.
  - LD **A**, (12345) *; A = valor en Memoria[12345]*

- INC [00001] y DEC [00010]: estás instrucciones son utilizadas para incrementar o decrementar respectivamente el valor de un registro o posición de memoria.

  - LD A, 0 *; A = 0*
  - INC A *; A = A+1 = 1*
  - LD B, A *; B = A = 1*
  - INC B *; B = B+1 = 2*
  - INC B *; B = B+1 = 3*
  - INC (12345) *;* (12345) = (12345)+*1*

Estas instrucciones activan las banderas de P/V (paridad y *overflow*) y la bandera C (*substract* o resta).

Tabla IV. **Instrucción para incremento y decremento**

Instrucción	S	Z	P/V	N	C
INC r	*	*	V	0	-
DEC r	*	*	V	1	-

Fuente: elaboración propia.

Donde r es un registro de 8 bits, * el *flag* se ve afectado por la operación acorde al resultado, V cambia cuando existe un *overflow* acorde al resultado y - significa que la instrucción no afecta el resultado.

- ADD [00011]: instrucción utilizada para realizar sumas

  Sintaxis de la instrucción:

  - ADD destino, origen

  Las operaciones implementadas serán las siguientes:

    - ADD A, B *; A = A + B*
    - ADD A, 100 *; A = A + 100*
      - ADD A, (1023) ; A = A + (1023)

  Las banderas que se ven afectadas a esta instrucción son:

Tabla V. **Banderas en la operación de agregar**

Instrucción	S	Z	P/V	N	C
ADD	*	*	V	0	*

Fuente: elaboración propia.

Donde r es un registro de 8 bits, * el flag se ve afectado por la operación acorde al resultado, V cambia cuando existe un *overflow* acorde con el resultado y - significa que la instrucción no afecta el resultado.

- SUB [00100]: instrucción utilizada para realizar restas.

  Sintaxis de la instrucción:

  - Sub destino, origen

  Las operaciones implementas por esta instrucción son las siguientes:

    - SUB A, B *; A = A - B*
    - SUB A, 100 *; A = A - 100*
    - SUB A, (1023) ; A = A - (1023)

  Las banderas que se ven afectadas a esta instrucción son:

Tabla VI. **Banderas afectadas por la instrucción de resta**

Instrucción	S	Z	P/V	N	C
SUB	*	*	V	1	*

Fuente: elaboración propia.

Donde r es un registro de 8 bits, * el flag se ve afectado por la operación acorde al resultado, V cambia cuando existe un *overflow* acorde al resultado y - significa que la instrucción no afecta el resultado.

- NOP [00101]: esta instrucción dejará pasar la línea de instrucción sin hacer nada.

- JP [00110]: instrucción de salto, realiza un traslado a una sección específica de la pila de instrucciones.

  Se realizarán dos tipos de saltos:

  - Salto para un bucle infinito:

    - bucle:
    - LD A, 20
    - NOP
    - (...)
    - JP bucle

- Salto para generar un bucle condicional:

  - bucle:
  - DEC D
  - JP N , bucle

La instrucción de salto condicional funcionará con los siguientes estados del registro de *flags* F:

- ✓ JP NZ, dirección: salta si el indicador de cero (Z) está a cero (resultado no cero).

- ✓ JP Z, dirección: salta si el indicador de cero (Z) está a uno (resultado cero).

- ✓ JP NC, dirección: salta si el indicador de carry (C) está a cero.

- ✓ JP C, dirección: salta si el indicador de carry (C) está a uno.

- ✓ JP PO, dirección: salta si el indicador de paridad/desbordamiento (P/V) está a cero.

- ✓ JP PE, dirección: salta si el indicador de paridad/desbordamiento (P/V) está a uno.

- ✓ JP P, dirección: salta si el indicador de signo S está a cero (resultado positivo).

- ✓ JP M, dirección**:** salta si el indicador de signo S está a uno (resultado negativo).

- CALL [00111]: esta instrucción genera una subrutina y retornará la instrucción siguiente en donde se hizo la llamada cuando se encuentre la instrucción RET.

  - LLAMADA:
  - LD **A**, 20
  - (...)
  - RET
  - (...)
  - CALL LLAMADA

Hay que recordar que la pila de llamadas es limitada ya que cada posición de salto se guarda en una memoria tipo LIFO (*last in first out*). Esta memoria está contenida en donde se encuentran los registros. El número de llamadas máximo será de 4.

- AND [01000]: instrucción que realiza la operación lógica and.

Las banderas que se ven afectadas a esta instrucción son:

Tabla VII. **Banderas afectadas por la instrucción de AND**

Instrucción	S	Z	P/V	N	C
AND	*	*	V	0	0

Fuente: elaboración propia.

Donde * significa que el flag se ve afectado por la operación acorde al resultado, V cambia cuando existe un *overflow* acorde con el resultado y - significa que la instrucción no afecta el resultado.

- OR [01001]: instrucción que realiza la operación lógica or.

Las banderas que se ven afectadas a esta instrucción son:

Tabla VIII. **Banderas afectadas por la instrucción de OR**

Instrucción	S	Z	P/V	N	C
OR	*	*	V	0	0

Fuente: elaboración propia.

Donde * significa que el flag se ve afectado por la operación acorde al resultado, V cambia cuando existe un *overflow* acorde al resultado y -significa que la instrucción no afecta el resultado.

- XOR [01010]: instrucción que realiza la operación lógica xor.

Las banderas que se ven afectadas a esta instrucción son:

Tabla IX. **Banderas afectadas por la instrucción XOR**

Instrucción	S	Z	P/V	N	C
XOR	*	*	V	0	0

Fuente: elaboración propia.

Donde * significa que el flag se ve afectado por la operación acorde al resultado, V cambia cuando existe un *overflow* acorde al resultado y -significa que la instrucción no afecta el resultado.

- NEG [01011]: instrucción que realiza la operación lógica not. Las banderas que se ven afectadas a esta instrucción son:

Tabla X. **Banderas afectadas por la instrucción NEG**

Instrucción	S	Z	P/V	N	C
NEG	-	-	-	0	0

Fuente: elaboración propia.

Donde r es un registro de 8 bits, * el flag se ve afectado por la operación acorde al resultado, V cambia cuando existe un *overflow* acorde con el resultado y -significa que la instrucción no afecta el resultado.

- HALT [01100]: la instrucción HALT suspende el funcionamiento de la CPU hasta que una interrupción la habilita o se recibe un reset. Mientras existe el estado HALT, el procesador ejecuta NOP para mantener la lógica de la memoria.

- IN [01110]: esta instrucción se utilizará para acceder a una dirección específica de puerto para ser guardado en un registro.

    - El valor que se encuentra en la dirección de memoria n se guarda en el registro A.
    - IN A, (n).

- OUT [01111]: Esta instrucción se utilizará para mostrar en una dirección específica de puerto el registro especificado.

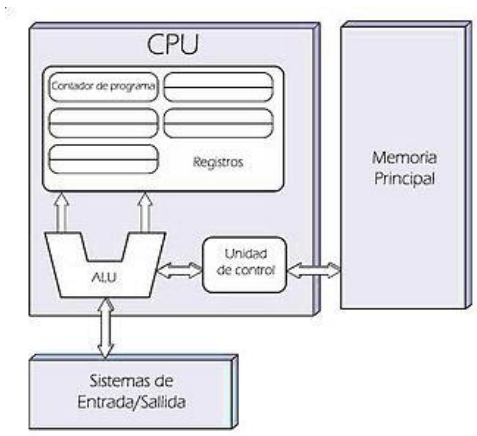
    - El valor que se encuentra en la dirección de memoria n se guarda en el registro A.
    - OUT (n), A .

4.3.2. Descripción del hardware sintetizado

El hardware sintetizado consta de una unidad central de procesos, constituida por memorias para el manejo de los registros y pilas descritos anteriormente, una unidad de control y una unidad lógica aritmética, la cual podrá manejar entradas y salidas de espacios de memoria, como banderas, espacios en memoria RAM y puertos. La unidad de control puede acceder a los registros, a la memoria y a los puertos conectados.

Las memorias por implementar son dos, una memoria ROM en la cual se guardará cada una de las instrucciones por ejecutar y una memoria RAM en la cual se guardarán los registros, las *flags*, la LIFO para manejar las llamadas y los puertos de entrada y salida. Se requiere de dos puertos, estos pueden ser de entrada o de salida. El siguiente diagrama muestra un esquema del hardware por describir.

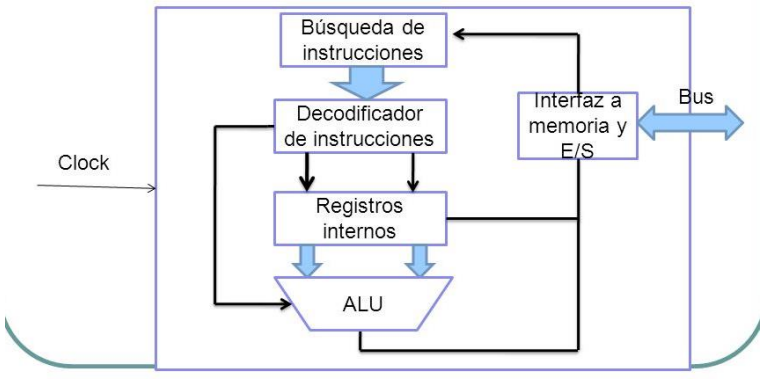
Figura 190. **Diagrama de bloques para la creación del microcontrolador**



Fuente: elaboración propia.

La lógica del comportamiento del hardware se muestra en el siguiente diagrama de bloques básico de un procesador.

Figura 191. **Bloques para el tratamiento de instrucciones**



Fuente: elaboración propia.

Como se describió anteriormente, las instrucciones deberán ser recibidas por la FPGA por medio de la comunicación UART, estas estarán guardadas en una memoria ROM interna para ser leídas al ejecutar el programa recibido. El periodo de reloj podría ser modificable a uno de más duración al que utiliza la FPGA y con esto verificar si cada instrucción se está ejecutando correctamente.

El módulo de UART sintetizado se diseñó con una máquina de estados de codificación A. La lógica de descripción consiste en una serie de puertos de verificación, el puerto de rx, el puerto de tx y el puerto de *data_led,* el cual se utiliza para enviar los datos de 8 bits al módulo de control y guardado de la memoria ROM.

Figura 192. **Descripción del módulo serial para la carga de instrucciones**

```
entity InstanciarTOP is

port(
        sys_clk :       in      std_logic;
        led : out               std_logic_vector(7 downto 0);
        UARTRX :    in              std_logic;
        UARTTX :    out         std_logic;
        pmod_1 :    out             std_logic;
        pmod_2 :    out         std_logic;
        reset_btn :  in     std_logic;
        data_led:   out     std_logic_vector(7 downto 0)
    );

end InstanciarTOP;

 COMPONENT t_serial
  PORT(
      sys_clk          : IN std_logic;
      UARTRX       : IN std_logic;
      reset_btn    : IN std_logic;
      led              : OUT std_logic_vector(7 downto 0);
      UARTTX       : OUT std_logic;
      pmod_1       : OUT std_logic;
      pmod_2       : OUT std_logic;
      uartrx_p     : out std_logic_vector(7 downto 0)
      );
  END COMPONENT;
```

Continuación de la figura 192.

```
Inst_t_serial: t_serial PORT MAP(
        sys_clk => sys_clk  ,
        led => led_s,
        UARTRX => UARTRX,
        UARTTX => UARTTX,
        pmod_1 => pmod_1,
        pmod_2 => pmod_2,
        reset_btn => reset_btn,
        uartrx_p => data_led
    );
```

Fuente: elaboración propia.

El siguiente módulo importante es el ingreso de las instrucciones a la memoria ROM para ser ejecutadas posteriormente. Se realizó una máquina de estados de codificación B el cual lleva el control del estado de entrada del bus de entrada. La declaración de los puertos se describe a continuación.

Figura 193. **Declaración de puertos para el módulo microcontrolador**

```
entity Estados is
port(
    clk        : in std_logic;
    uart_pmod2 : in std_logic;
    data       : in std_logic_vector(7 downto 0);
    con_data   : out std_logic_vector(21 downto 0);
    rdy_master : out std_logic;
    cu         : out std_logic_vector(1 downto 0)
);
end Estados;
```

Fuente: elaboración propia.

Las señales utilizadas incluyen tanto la máquina de estados como señales de conteo y la declaración de la memoria para definir la ROM.

Figura 194. **Máquina de estados para tratamiento de las instrucciones**

```
type FSM is (esperar,transmitiendo,procesando);
signal maquina : FSM;
signal cont : integer range 0 to 2;
signal tmp_mem : std_logic_vector(23 downto 0);

type ROM_t is array(0 to 9) of std_logic_vector(23 downto 0);
signal ROM : ROM_t;
```

Fuente: elaboración propia.

El *process* principal es en el cual se describe la funcionalidad del guardado de la memoria ROM y el paso de cada uno de los estados definidos.

Figura 195. **Proceso para asignación de instrucciones**

```
process(clk,uart_pmod2)
begin
    if(uart_pmod2 = '0') then
        rdy_master <= '1';
        tmp_mem <= (others => '0');
        maquina <= procesando;
    elsif rising_edge(clk) then
        case maquina is
            when transmitiendo =>
                if uart_pmod2 = '1' then
                    maquina <= procesando;
                else
                    maquina <= transmitiendo;
                end if;
            when procesando =>
                if cont = 0 then
                    tmp_mem(23 downto 16) <= data;
                    ROM(cont)(23 downto 16) <= data;
                    cont <= cont + 1;
                elsif cont = 1 then
                    tmp_mem(15 downto 8) <= data;
                    ROM(cont)(15 downto 8) <= data;
                    cont <= cont + 1;
                else
                    tmp_mem(7 downto 0) <= data;
                    ROM(cont)(7 downto 0) <= data;
                    cont <= 0;
                    rdy_master <= '0';
                end if;

                maquina <=esperar;

            when others =>
                maquina <= esperar;
        end case;
    end if;
end process;

con_data <= tmp_mem(23 downto 2);
cu       <= tmp_mem(1 downto 0);
```

Fuente: elaboración propia.

El módulo que sigue es el que lleva el control de la separación de cada segmento de instrucción y habilita el procesamiento en el módulo de ALU. La entidad del módulo se describe a continuación.

Figura 196. **Señales del módulo principal para el microcontrolador**

```
entity Master is
port(
    clk        : in std_logic;
    entrada    : in std_logic_vector(21 downto 0);
    en_master  : in std_logic;
    rdy_alu    : in std_logic;

    neum       : out std_logic_vector(4 downto 0);
    A          : out std_logic_vector(7 downto 0);
    ins        : out std_logic;
    B          : out std_logic_vector(7 downto 0);
    salida_alu : out std_logic

);
end Master;
```

Fuente: elaboración propia.

La arquitectura muestra la utilización de una máquina de estados juntamente con lógica concurrente para hacer la asignación de cada instrucción a puertos de salida y así simplificar el proceso.

Figura 197. **Descripción para la decodificación de instrucciones**

```
architecture Behavioral of Master is

type FSM is (reception,processing,synq);
signal current_state : FSM;

begin

neum    <= entrada(21 downto 17);
A       <= entrada(16 downto 9);
B       <= entrada(7 downto 0);
ins     <= entrada(8);


process(entrada,en_master,clk)
begin
    if en_master = '1' then
        current_state <= processing;
        salida_alu <= '1';
    elsif(rising_edge(clk))then
        case current_state is
            when processing =>
                if rdy_alu = '1' then
                    current_state <= synq;
                    salida_alu <= '0';
                else
                    current_state <= processing;
                end if;
            when others =>
                current_state <= reception;
                salida_alu <= '0';
        end case;
    end if;

end process;


end Behavioral;
```

Fuente: elaboración propia.

El siguiente módulo utilizado es el de ALU uno de los más importantes, ya que en este se lleva el control de las operaciones aritméticas y lógicas descritas anteriormente. Los puertos de entradas y salidas incluyen las dos entradas de datos, el puerto de instrucción, el puerto de dirección, el puerto de dato de entrada y dato de salida, un puerto para habilitación de escritura, un puerto para habilitar la entrada ALU y un puerto de salida para indicar que el módulo está ocupado. Estos puertos se muestran a continuación.

Figura 198. **Puertos de entrada y salida para el módulo aritmético lógico**

```
entity Alu is
port(
    clk         : in  std_logic;
    neum        : in  std_logic_vector(4 downto 0);
    A           : in  std_logic_vector(7 downto 0);
    ins         : in  std_logic;
    B           : in  std_logic_vector(7 downto 0);
    dir         : out std_logic_vector(7 downto 0);
    data_in     : in  std_logic_vector(7 downto 0);
    data_out    : out std_logic_vector(7 downto 0);
    wr          : out std_logic;
    entrada_alu : in  std_logic;
    busy_alu    : out std_logic
);
end Alu;
```

Fuente: elaboración propia.

Entre las señales utilizadas en este módulo está una máquina de estados que lleva el control de las acciones del módulo. El estado *load* es utilizado para cargar la información de los puertos a la operación requerida y siguiente se coloca en un estado de esperar.

Figura 199. **Declaración de señales para el módulo aritmético lógico**

```
signal data1 : std_logic_vector(7 downto 0);
signal flag  : std_logic;

type FSM is (esperar,load);
signal current_state: FSM;
```

Fuente: elaboración propia.

El *process* principal de este módulo es la utilización de la máquina de estados para la verificación de la operación. En este proceso se hace la verificación del nemonico y dependiendo de este será diferente la operación o asignación de los registros. A manera de que no se haga muy extenso se describirá el primer nemonico el de carga. El nemonico LD básicamente es cargar a un registro de una dirección específica.

La parte principal de este proceso está en la lógica del estado presente ya que se tiene una serie de *if* para validar qué caso de *load* se necesita. Este operador también activa o desactiva la *flag* y dependiendo esto y un bit de instrucción se seleccionará el caso. Como se puede observar en la imagen, el primer caso es la asignación de un valor en un registro con una dirección en otra dirección especificada solamente utilizando puertos para que sean almacenados en una memoria llamada RAM en este diseño.

Figura 200. **Descripción de módulo para la interpretación de instrucciones**

```
process(clk,entrada_alu)
begin
    if entrada_alu='1' then
        current_state <= load;
        dir           <= (others => '0');
        data_out      <= (others => '0');
        flag          <= '0';
        datal         <= (others=>'0');
        wr            <= '0';
        busy_alu      <= '1';
    elsif rising_edge(clk) then
        case current_state is
            when load =>
                if (neum = "00000") then -- cargar direccion dirección
                    if(ins = '1' and flag = '0') then
                        dir           <= B;
                        datal         <= data_in;
                        data_out      <= (others=>'0');
                        wr            <= '0';
                        flag          <= '1';
                    elsif(ins = '1' and flag = '1')then
                        wr            <= '1';
                        dir           <= A;
                        data_out      <= datal;
                        current_state <= esperar;
                        flag          <= '0';
                    elsif(ins = '0')then -- direcion/dato
                        wr            <= '1';
                        dir           <= A;
                        data_out      <= B;
                        current_state <= esperar;
                        flag          <= '0';
                    end if;
                elsif (neum = "01000") then -- incrementar
                    if(flag = '0') then
                        dir           <= A;
                        datal         <= data_in;
                        data_out      <= (others=>'0');
                        wr            <= '0';
                        flag          <= '1';
                    elsif(flag = '1')then
                        wr            <= '1';
                        dir           <= A;
                        data_out      <= std_logic_vector(to_unsigned(to_integer(unsigned(datal))+2,8));
                        current_state <= esperar;
                        flag          <= '0';
                    end if;
                elsif (neum = "10000") then -- decrementar
                    .
                    .
                    .
                end if;
            when others =>
                current_state       <= esperar;
                wr                  <= '0';
                busy_alu            <= '0';
        end case;
    end if;
end process;
```

Fuente: elaboración propia.

Las demás instrucciones son parecidas a esta solamente depende el operador, la dirección y el dato de entrada que se quiera manipular. Algo muy importante por tomar en cuenta es que al realizar una operación aritmética se debe convertir las señales por operar a entero y tomar en cuenta la longitud de la operación.

El último módulo importante en este diseño es al que se le llama memoria RAM, este módulo simplemente es una declaración de memoria con puertos de comunicación con los demás módulos. A continuación, se presenta la descripción del módulo.

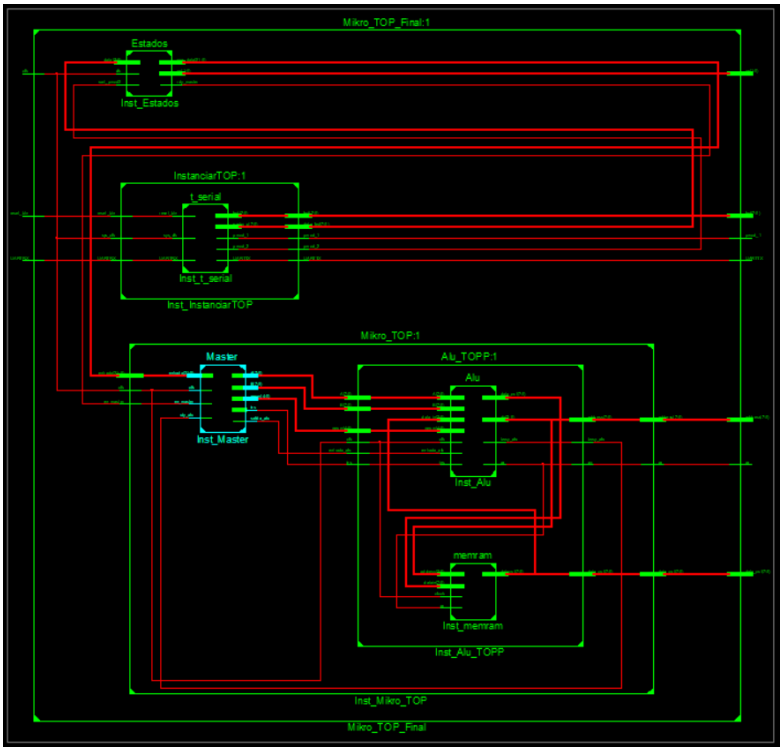
Figura 201. **Descripción de la memoria de instrucción**

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

entity memram is
port (
    clock   : in  std_logic;
    wr      : in  std_logic;
    address : in  std_logic_vector(7 downto 0 );
    datain  : in  std_logic_vector(7 downto 0 );
    dataout : out std_logic_vector(7 downto 0 )
);
end entity memram;

architecture Behaivoral of memram is

type ramtipo is array (0 to (2**8)-1) of std_logic_vector(7 downto 0);
signal memoria : ramtipo;

begin

process(clock) is
begin
    if (rising_edge(clock)) then -- En el flanco de subida
        if (wr = '1') then
            memoria(to_integer(unsigned(address))) <= datain; -- el array se posiciona en una de las direcciones
        end if;
        dataout <= memoria(to_integer(unsigned(address)));
    end if;

end process ;

end architecture Behaivoral;
```

Fuente: elaboración propia.

Un posible diseño de hardware para sintetizar un microprocesador programable incluye crear componentes con los módulos anteriores. La generación del hardware empieza desde la comunicación UART luego se pasa al módulo de estados de memoria donde se almacenan las instrucciones para luego pasar al módulo *TOP* y específicamente pasa al módulo maáster para separar las instrucciones. Cada instrucción separada se tomará dependiendo el nemónico para que sea operado por el módulo ALU, el módulo ALU interactúa directamente con la memoria RAM ya que cada operación dejará cambios en los registros almacenados.

Figura 202. **Diagrama general del microcontrolador**



Fuente: elaboración propia.

4.4. Utilización del ARM Cortex-A9 de la tarjeta de desarrollo Zybo

La distribución de Xillinux está pensada como una plataforma de desarrollo, y no solo como una demostración. El entorno está listo para su desarrollo e integración como lógica personalizada antes de generarse en hardware.

Para iniciar la distribución de Xillinux se debe utilizar una tarjeta *Micro SD*, la cual debe tener dos componentes:

- Un sistema de archivos FAT32 en una partición de arranque, que consiste en cargadores de arranque, una configuración bitstream para la parte FPGA y los binarios para arrancar Linux.

- Un sistema de archivos raíz ext4 montado por Linux.

La imagen descargada de Xillinux ya tiene casi todo configurado, solo faltan tres archivos en la partición de arranque, uno de los cuales se debe generar con las herramientas de Xilinx, y dos que se copian del kit de partición de arranque.

Las diversas operaciones para preparar la *Micro SD* se detallan paso a paso en esta sección.

Este proceso consta de los siguientes pasos, estos deben realizarse en el siguiente orden:

- Descomprimir el kit de partición de arranque
- Generación de la lista de conexiones del procesador

- Generating Xilinx' IP cores
- Implementar el diseño VHDL
- Escribir la imagen de Xillinux en la *Micro SD*
- Copiar el árbol de archivos dentro de la *Micro SD* en la partición *boot*

A continuación, se describe cada uno de los pasos antes mencionados.

4.4.1. Descomprimir el kit de partición de arranque

Este kit de partición se descargará de la página oficial de Xillybus en la ruta de Xillinux. Descomprima el archivo xillinux-eval-board-XXX.zip previamente descargado en un directorio de trabajo, la ruta del directorio de trabajo no debe incluir espacios en blanco. Si se está trabajando en un entorno Windows, el escritorio no es adecuado, ya que su ruta incluye "Documents and Settings". Lo incluido consta de los siguientes directorios.

- *Verilog*: contiene el archivo de proyecto para la lógica principal y algunas fuentes en *Verilog.*

- VHDL: contiene el archivo de proyecto para la lógica principal y algunas fuentes, con el archivo fuente editable por el usuario en VHDL.

- *Blockdesign*: este directorio contiene los archivos relacionados con el flujo de diseño del bloque (consulte la guía del flujo de diseño del bloque Xillybus para usuarios que no usan HDL).

- *Cores*: binarios precompilados de los núcleos Xillybus IP.

- *System*: directorio para generar lógica relacionada con el procesador.

- *Runonce*: directorio para generar lógica de propósito general (núcleos IP FIFO CoreGen).

- *Bootfiles*: contiene dos archivos específicos de la placa, para copiar en la partición de arranque.

- *Vivado-essentials*: archivos de definición y directorios de compilación para lógica relacionada con el procesador y de uso general para uso de *Vivado*.

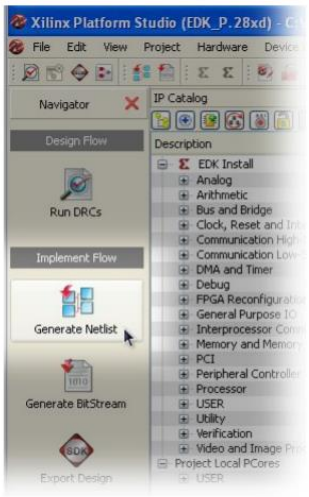
En la página web donde se encuentran estos archivos están disponibles las versiones para las placas Zedboard, MicroZed y Zybo. Para proyectos en ISE se utilizan el archivo UCF en el directorio src en la carpeta vhdl. También se debe tener en cuenta que el directorio vhdl contiene archivos *Verilog*, pero ninguno de ellos debería necesitar edición por usuario.

La interfaz con el núcleo *Xillybus IP* se lleva a cabo en xillydemo.vhd archivos en los respectivos subdirectorios. Este es el archivo para editar y probar Xillybus con sus propias fuentes de datos e interfaces.

4.4.2. Generación de la lista de conexiones del procesador

Dentro del kit de partición de arranque, debe hacer doble clic en el archivo system.xmp en el directorio *system*. Esto abre *Xilinx Platform Studio* (XPS). Luego debe hacer clic en *Generate Netlist* a la izquierda como se muestra en la imagen a continuación.

Figura 203. **Generar conexiones para el procesador**



Fuente: elaboración propia.

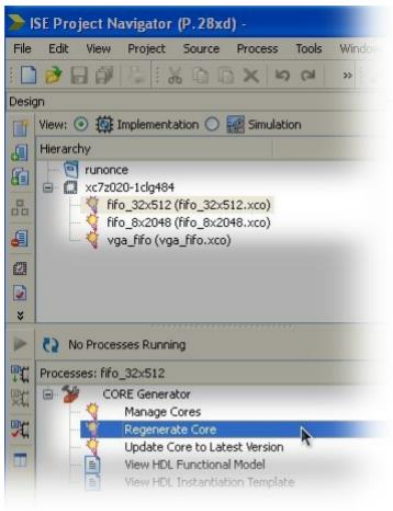
El proceso toma aproximadamente diez minutos, dependiendo de la computadora que lo ejecute. Se pueden generan advertencias, pero no se deben tolerar errores. Cuando la salida de la consola dice "XST completed" y "Done!" significa una finalización exitosa del proceso. Cuando se llegue a este punto puede cerrar el XPS por completo.

No es necesario repetir este proceso en el futuro. Es posible que aparezca un cuadro de diálogo de error de licencia Xilinx seguido de una configuración de licencia Xilinx pero estos deben ser ignorados. Si continúan apareciendo errores de licencia, evitando la generación de una lista de conexiones, asegúrese de tener una licencia WebPack u otra instalada más avanzada en la máquina.

4.4.3. Generating Xilinx' IP cores

Dentro del kit de partición de arranque, haga doble clic en el archivo runonce.xise en el directorio *runonce*. Esto abre el Xilinx ISE Project Navigator. En la parte superior izquierda de la ventana abierta, haga clic en fifo 32x512, luego en el panel inferior expanda la opción Generador CORE y finalmente haga doble clic *Regenerate Core*, como se muestra en la imagen a continuación.

Figura 204. **diagrama de memoria para generación del Core de Xilinux**



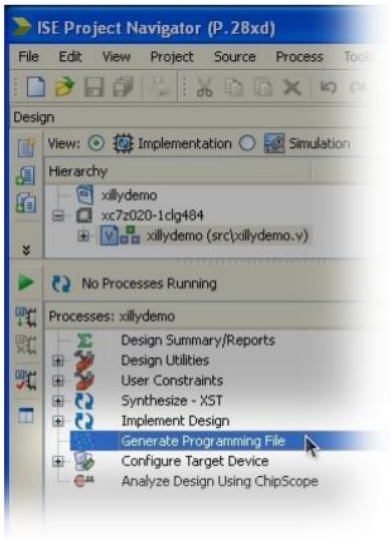
Fuente: elaboración propia.

El proceso produce muchas advertencias, pero no errores y finaliza con un mensaje diciendo que el proceso se ha completado con éxito. Repita esto para los otros dos *Cores IP, fifo 8x2048* y *vga fifo*. Terminado esto puede proceder a cerrar completamente el *ISE Project Navigator*. No hay necesidad de repetir este proceso de nuevo una vez realizado.

4.4.4. Implementar el diseño VHDL

El siguiente paso es generar el diseño de hardware donde se sintetiza el diseño que se utilizará para comunicar el micropocesador con la FPGA. Primero debe buscar en el subdirectorio vhdl del kit el archivo *xillydemo.xise*. *El Project ISE Navigator* se iniciará y abrirá el proyecto con la configuración correcta. Para que se genere el diseño sólo se debe hacer clic en la opción *Generate Programming File*.

Figura 205. **Generación de archivo para sintetizar el hardware de xilinux**



Fuente: elaboración propia.

El procedimiento producirá varias advertencias, pero no debe presentar ningún error. El proceso debe finalizar con mensaje de completado con éxito. El archivo resultante se puede encontrar como xillydemo.bit en el directorio vhdl junto con varios otros archivos. Si ocurre un error, verifique que los pasos anteriores hayan sido realizados correctamente, en particular, el paso donde se

regeneran los tres *Cores IP*. Cierre completamente ISE Project Navigator cuando esta tarea se complete con éxito.

4.5. Escribir la imagen de Xillinux en la Micro SD

A continuación, para generar la imagen del sistema operativo Xillinux se describirán los pasos a realizar en el sistema operativo *Windows.*

En *Windows* se necesita una aplicación especial para copiar la imagen como la llamada *USB Image Tool*. Esta herramienta es adecuada cuando se usa un adaptador USB para acceder a la *Micro SD* tarjeta. Algunas computadoras tienen una ranura *Micro SD* integrada y pueden necesitar usar otra herramienta como por ejemplo *Win32 Disk Imager*. Este también puede ser el caso cuando se ejecuta Windows 7. Ambas herramientas están disponibles de forma gratuita para descargar desde varios sitios de internet. Para crear la imagen ejecute el programa *USBImageTool.exe , c*uando la ventana principal se muestre conecte el adaptador USB, seleccione el ícono del dispositivo que aparece arriba a la izquierda. En el menú haga clic en restaurar y establezca el tipo de archivo como "Archivos de imagen comprimidos (gzip)".

Seleccione el archivo de imagen descargado xillinux.img.gz. Todo el proceso debería tomar alrededor de 4 a 5 minutos aproximadamente. Cuando termine el proceso desmonte el dispositivo de forma segura. Puede utilizar este programa o utilizar cualquier otro que utilice para crear imágenes en memorias.

4.5.1. Copiar el árbol de archivos dentro de la Micro SD en la partición boot

Esta es la etapa final, debe coloca los archivos necesarios para el arranque:

- Copie los archivos boot.bin y devicetree.dtb del bootfiles subdirectorio del kit de partición de arranque, en la partición de arranque de la tarjeta *Micro SD* es decir en la raíz del directorio de la *Micro SD*.
- Copie el archivo xillydemo.bit que se generó en el subdirectorio vhdl.

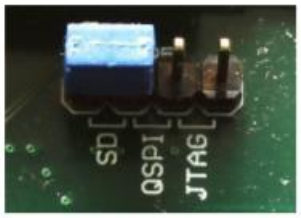
Antes de copiar estos archivos si la imagen en la Micro SD se acaba de escribir en esta, desmonte el dispositivo USB y luego vuelva a conectarlo a la computadora. Esto es necesario para asegurarse que la computadora esté actualizada con la tabla de partición de la *Micro SD.*

Antes de intentar iniciar el sistema operativo verifique que la partición de arranque contenga los archivos correctos. Para arrancar, deben existir cuatro archivos en la primera partición de la *Micro SD*:

- uImage: el binario del kernel de Linux. Este es el único archivo en la partición de arranque después de escribir la imagen Xillinux en la *Micro SD*.
- boot.bin: es el gestor de arranque inicial. Este archivo contiene las inicializaciones del procesador y la utilidad *U-boot*.
- devicetree.dtb: a este se le llama Device Tree Blob file, el cual contiene información de hardware para el kernel de Linux.
- xillydemo.bit: es el archivo de programación de la FPGA, que se generó anteriormente.

Para que funcione es importante que el modo de arranque por SD se seleccione mediante un puente cerca del conector VGA, que debe ser establecido en los dos pines marcados con SD, como se muestra en esta imagen.

Figura 206. **Pines para lectura desde SD**



Fuente: elaboración propia.

Terminando todos estos pasos el sistema operativo Xillinux y la conexión por medio de Xillybus ya puede realizarse.

4.5.2. El sistema operativo Xillinux

La distribución Xillinux es un kit de software que incluye código para comunicarse con FPGA, ejecutando un escritorio gráfico completo con las tarjetas Zedboard, ZyBo o SocKit, conectando un monitor, teclado y mouse a la placa para su utilización.

Este *software* no es solo una demostración, sino un kit de desarrollo haciendo que la integración entre el host de Linux y la parte de FPGA sea simple, intuitiva y relativamente sencilla. Una configuración de demostración del núcleo Xillybus IP se incluye en la lógica de la distribución. Se puede configurar y descargar un núcleo de Xillybus IP personalizado en la fábrica de IP Core.

El paquete completo, junto con instrucciones de instalación fáciles de seguir, se puede descargar de forma gratuita desde es sitio oficial. La configuración lleva entre 30 a 60 minutos aproximadamente, la mayoría de los cuales consiste en esperar a que las herramientas de Xilinx o Altera implementen algunos componentes lógicos. No se necesitan conocimientos previos en FPGA ni Linux para llevar a cabo el proceso de configuración.

Figura 207. **Muestra del sistema operativo Xilinux funcionando**

Fuente: elaboración propia.

4.5.2.1. Conexión entre Xillinux y la FPGA por medio del Xillybus

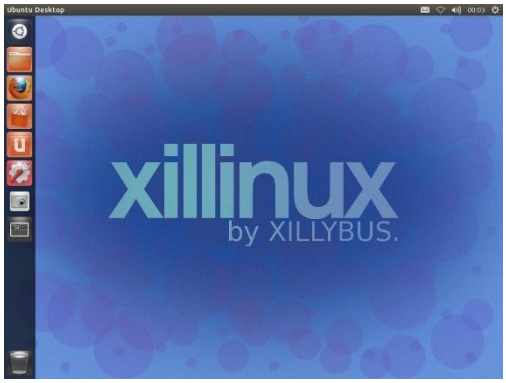
A continuación, se presentará cómo trabajar en el sistema operativo Xillinux para utilizar la conexión con el puerto de alta velocidad Xillybus.

El primer paso es iniciar la interfaz gráfica de Xillinux. Cada vez que se inicia el dispositivo, se ha de notar que solo hay una interfaz de shell iniciada como usuario root. Para iniciar la interfaz gráfica de usuario escriba en la terminal:

$ root@localhost:~# **startx**

Después de algunos segundos debería ser capaz de ver una interfaz similar a Ubuntu 12.04.

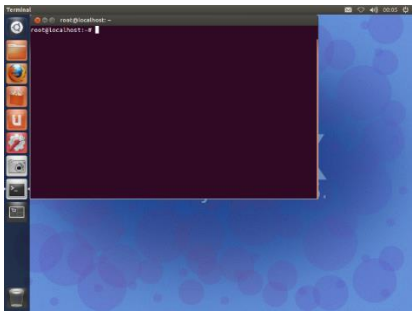
Figura 208. **Entorno de escritorio Xillinux**



Fuente: elaboración propia.

Xillinux es una distribución de Linux basada en Ubuntu, pero su núcleo ha sido compilado para ejecutarse dentro de la arquitectura ARM A9 de Zynq y para proporcionar acceso directo a Xillybus. Ahora se navegará a través de los dispositivos disponibles para descubrir las interfaces de Xillybus a las que se puede acceder directamente desde el sistema operativo. Lo primero que se debe hacer es abrir una ventana de terminal de linux presionando Ctrl+Alt+T.

Figura 209. **Terminal de Xillinux**



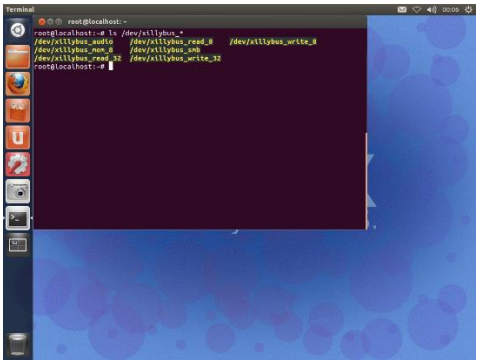
Fuente: elaboración propia.

Ahora, se mostrarán los archivos que están conectados a las señales de Xillybus. Para hacerlo se debe escribir:

root@localhost:~# ls /dev/xillybus_*

Una lista de las interfaces de Xillybus disponibles pueden aparecer como se muestra en la siguiente figura.

Figura 210. **Interfaces de Xillybus**



Fuente: elaboración propia.

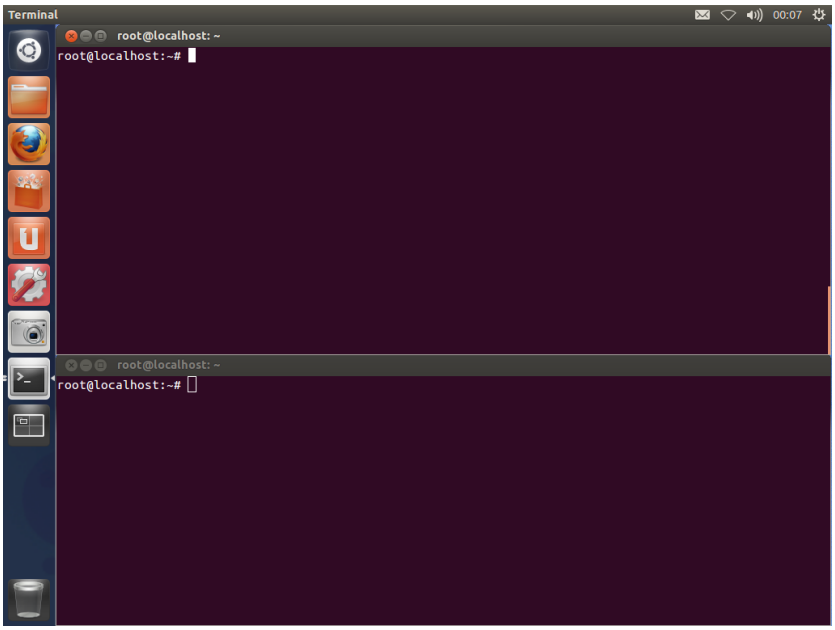
Este documento se centrará solamente en las interfaces de lectura y escritura de 8 y 32 bits. Algo importante es que se puede acceder directamente a estas interfaces a través de cualquier lenguaje de programación, o incluso desde el *prompt* del shell de Linux como archivos.

En este momento se empezará a trabajar con Xillybus. Las siguientes pruebas están destinadas a realizarse exclusivamente si se utilizaron los archivos de descripción de hardware xillybus_demo.vhd predeterminados en el lado de la lógica programable. En este ejemplo se utiliza una FIFO como *loopback* con el propósito de responder todo lo que se envía desde el ARM.

Se escribirán datos directamente en los buses *xillybus_write_* (8/32) y se esperará una respuesta en el xillybus_read_ (8/32), ya sea desde Linux bash, o formando un usuario hecho a la medida. Algo muy útil es que Xillybus proporciona un conjunto de API escritas en C, que no solo facilitan el acceso a sus interfaces de bus, sino que también proporcionan una manera eficiente de leer y escribir en ellas.

Inicialmente se deben abrir dos ventanas de terminal organizandolas como lo desee. Estas se utilizarán simultáneamente para visualizar lo generado.

Figura 211. **Muestra de terminales simultáneas**

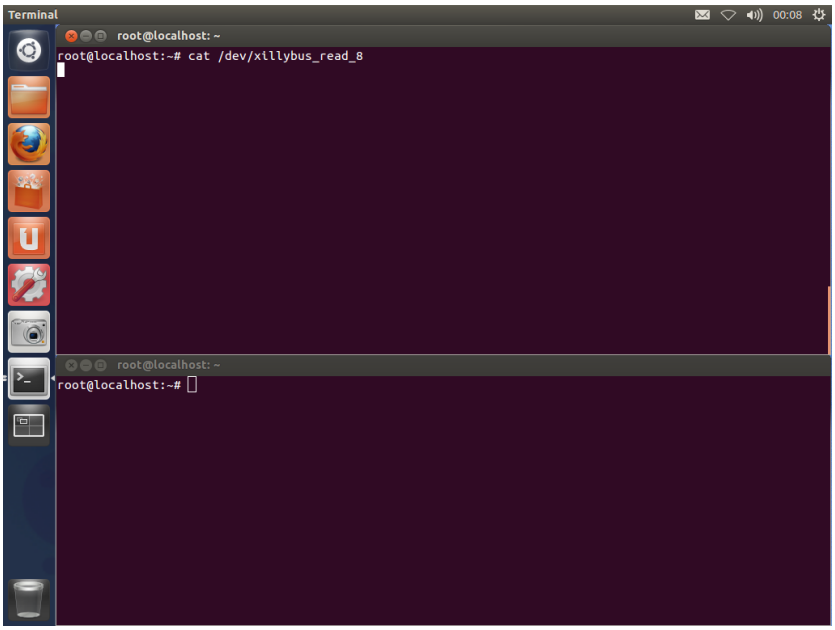


Fuente: elaboración propia.

Para esta prueba, la herramienta cat de Linux se usará para leer y escribir en las interfaces de Xillybus. El primer paso es comenzar a escuchar en el bus de lectura. En la primera ventana de terminal se debe escribir:

```
root@localhost:~# cat /dev/xillybus_read_8
```

Figura 212. **Utilización de la comunicación de Xillybus con terminales simultáneas**



Fuente: elaboración propia.

Como se puede notar se utilizará la interfaz de bus de 8 bits para este ejemplo. Cuando se presione *enter* después de escribir el comando anterior, no debería pasar nada, ya que todavía no se ha escrito nada en esa interfaz.

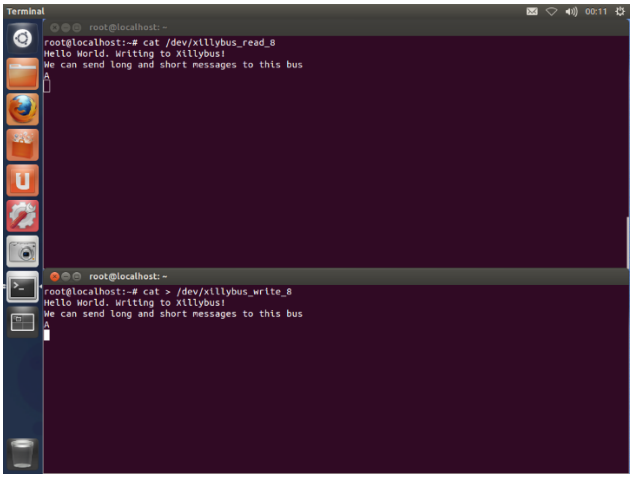
Se procederá a escribir algo en el mismo *bus*. Evidentemente, si se intenta leer algo de la interfaz de bus de 8 bits se necesita escribir en una interfaz de 8 bits no de 32 bits. Para hacerlo, deberá escribir en la segunda terminal de Linux, mientras que la primera sigue abierta y ejecutando el comando cat anterior, de la siguiente manera:

```
root@localhost:~# cat > /dev/xillybus_write_8
```

Esto enviará todo lo que está escribiendo al archivo de interfaz de bus.

Es en este momento cuando ya se permite enviar datos al bus. Para ver resultados se debe escribir algo en la segunda ventana, se debe notar que no pasa nada hasta que se presiona la tecla *enter*. Esto se debe a la forma en que la herramienta *cat* maneja la entrada del teclado, dejando fluir los datos hasta que se envíe el comando con la tecla mencionada.

Figura 213. **Funcionalidad de lectura y escritura en la terminal de Xillinux**



Fuente: elaboración propia.

Cuando se termine de transferir información a través del Xillybus, se puede cerrar el canal de transmisión presionando Ctrl+C.

Ahora para utilizar la interfaz del bus de 32 bits se deben abrir otro par de terminales o simplemente usar el comando clear para limpiar las instrucciones del comando anterior. Se procederá a realizar la misma operación, pero con la interfaz de bus de 32 bits. Para realizar esto se debe escribir la siguiente primera terminal:

```
root@localhost:~# cat /dev/xillybus_read_32
```

En este orden en la segunda terminal se deberá escribir la siguiente instrucción:

```
root@localhost:~# cat > /dev/xillybus_write_32
```

Al intentar escribir un texto se debe notar que los mensajes están incompletos en el lado de lectura. Una buena forma para probar esto es tratar de enviar un solo carácter a la vez, se debe notar que no será transferido inmediatamente ya que los datos se transfieren en trozos de 4 bytes (8 * 4 = 32), y como cada carácter es una representación ASCII de 8 bits, los datos se eliminan de la escritura FIFO cada cuatro caracteres.

Es importante destacar que esto no es un mal funcionamiento del bus, es simplemente la manera en que se comparten los datos con el bus. Esta es la razón por la que las interfaces de bus asíncronas de 32 bits de ancho están destinadas a ser utilizadas en la transmisión de operaciones de alto rendimiento, donde la latencia del *pipeline* no es una preocupación.

Ahora que se ha logrado una familiarización básica con el bus, es momento de entrar en las funcionalidades API incluidas por Xillybus. Pero antes de comenzar, no hay que olvidar que se deben cerrar ambas comunicaciones con Ctrl+C y abrir un conjunto de nuevas indicaciones de la terminal o simplemente limpiarlas con el comando borrar.

Antes de usar los ejemplos API proporcionados, las aplicaciones deben compilarse con gcc. Xillybus ha tenido la amabilidad de proporcionar también un

archivo *MAKE* para facilitar aún más las cosas. Bajo el directorio root del símbolo del sistema, debe ir al directorio ~ /xillybus/demoapps.

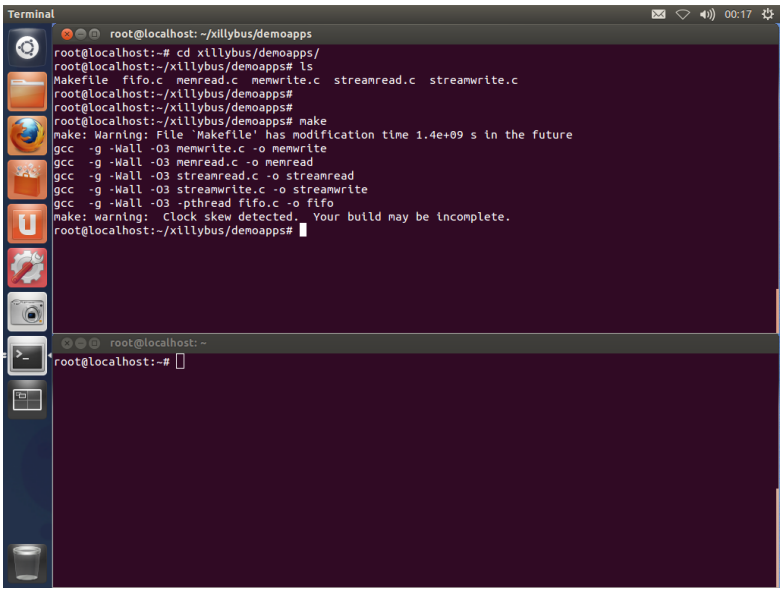
Luego para listar los archivos disponibles, escribir:

root@localhost:~# **ls**

Para compilar las aplicaciones de demostración (suponiendo que el directorio actual todavía es ~/xillybus/demoapps/) solo se debe escribir:

root@localhost:~# **make**

Figura 214. **Compilación de la aplicación para lectura y escritura**



Fuente: elaboración propia.

Se debe proceder a limpiar la primera ventana de la terminal para comenzar a usar las aplicaciones. Se utilizará la aplicación de transmisión incluida en los ejemplos. Para utilizarlo se debe escribir en la primera terminal el siguiente comando:

root@localhost:~/xillybus/demoapps# **./streamread /dev/xillybus_read_8**

En la segunda terminal, si se está en el directorio de inicio se debe cambiar al directorio demoapps y luego ejecutar la aplicación de escritura de secuencia.

root@localhost:~# cd xillybus/demoapps
root@localhost:~/xillybus/demoapps# ./streamwrite /dev/xillybus_write_8

Se puede notar que los comandos a utilizar para escribir o leer de forma continua o hacer un *streaming* se muestra a continuación.

- Sintaxis para *streamread*:

./streamread <xillybus_interface_to_read_from>

- Sintaxis para *streamwrite*:

./streamwrite <xillybus_interface_to_write_to>

El uso de estas aplicaciones implica el uso de entradas y salidas estándar del sistema, es decir, teclado y pantalla a través del símbolo del sistema, pero las funciones de la API pueden modificarse para usar fuentes alternativas y otros receptores, en su mayoría.

Se puede comenzar a utilizar el Xillybus, usando funciones API, que conducen a una forma más eficiente de transferir datos. Hay que tener en cuenta que cada golpe de teclado genera automáticamente una operación de transferencia de datos al bus.

Para utilizar el bus de 32 bits simplemente se debe ejecutar el archivo que contiene el número 32 al final, en este caso sería ejecutar los comandos siguientes de lectura y escritura respectivamente:

root@localhost:~/xillybus/demoapps# **./streamread /dev/xillybus_read_32**
root@localhost:~/xillybus/demoapps# **./streamwrite /dev/xillybus_write_32**

Para utilizar un lenguaje de programación diferente a C como lo es python, se puede realizar un script que ejecute comandos bash utilizando la librería *os.* Utilizando esta librería de python puede manejar las interfaces descritas anteriormente, como se realizó utilizando la herramienta CAT.

La propuesta de *software* es simplemente manejar el bus con una adaptación de Python a las interfaces de Xillybus. Se realiza la lógica de programación en Python y el manejo de escritura y lectura del bus se realizaría con la librería *os* de Python pero realmente es una ejecución Bash. Con esto podría hacer cualquier tipo de solución utilizando las ventajas del sistema operativo Xilinux y el bus de alta velocidad Xillybus con el diseño de hardware generado en la FPGA.

CONCLUSIONES

1. En el análisis de señal los proceso con microcontroladores o microprocesadores llevan más tiempo en ejecutarse es por esto que la tecnología que utiliza una FPGA es más eficiente ya que ejecuta los procesos en paralelo, es decir, optimiza los procesos de análisis de señal.

2. Es necesario comprender la estructura interna de una FPGA para comprender los conceptos del lenguaje de descripción de hardware VHDL.

3. Realizar un diseño de hardware con FPGA reduce costos y reduce el tiempo de implementación de los circuitos digitales.

4. La modularidad que presenta el lenguaje VHDL simplifica el diseño de circuitos como es el caso de un filtro digital.

5. Dependiendo de las capacidades de una FPGA se puede sintetizar cualquier diseño de hardware digital como es el caso de un microprocesador funcional.

RECOMENDACIONES

1. Comprender toda la teoría de electrónica digital antes de adentrarse a la síntesis de circuitos con FPGA.

2. Conocer las características de la FPGA con la que se desea sintetizar hardware para tener cuantiada la cantidad de hardware que se puede generar.

3. Comprender primero la lógica de descripción de hardware mediante a otro lenguaje convencional para entender de mejor manera la síntesis de circuitos digitales.

4. Para un mejor aprendizaje utilizar como dispositivo FPGA Zybo ya que esta contiene un microprocesador embebido ARM Cortex-A9.

BIBLIOGRAFÍA

1. CHU, Pong. *FPGA Prototyping by VHDL Examples*. 3a ed. New Jersey, Estados Unidos: John Wiley & Sons, Inc, 2008. 471 p.

2. Comunicacionyredesinfo. *Tipos de codificación*. [en línea]. <http://comunicacionyredesinfo.blogspot.com/2012/08/tipos-de-codificacion.html>. [Consulta: 26 de octubre 2018].

3. Laboratorio de electrónica. *Implementación de unIP Corecompatible con Wishbone.* [en línea]. <http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/memorias.pptx/>. [Consulta: 2 de abril 2018].

4. _____. *Introducción al diseño de hardware con FPGA utilizando VHDL*. [en línea]. <http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/presentaci%C3%B3n_fpga.pdf/>. [Consulta: 20 de enero 2018].

5. _____. *Memorias.* [en línea]. <labelectronica.weebly.com/uploads/8/1/9/2/8192835/memorias.pptx/>. [Consulta: 21 de marzo 2018].

6. MANO, M. Morris. *Diseño digital*. 3a ed. Estados Unidos: Pearson Educacion, 2005. 538 p.

7. READLER, Blaine. *VHDL by Example*. 2a ed. Estados Unidos: Full Arc Press, 2014. 120 p.

8. SIMPSON, Philip Andrew. FPGA Design: *Best Practices for Team-based Reuse.* 2a ed. Estados Unidos: Springer, 2015. 257 p.

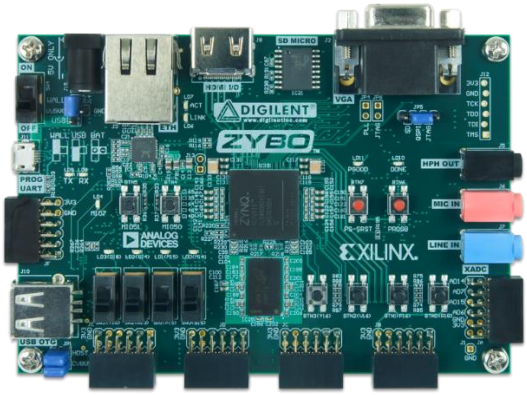
9. Xilinx. *Diseñando con VHDL*. [en línea]. <https://www.xilinx.com/content/dam/xilinx/training/languages/lang-vhdl.pdf/>. [Consulta: 11 de febrero 2018].

10. Xillybus. Xillinux: *A Linux distribution for Z-Turn Lite, Zedboard, ZyBo and MicroZed*. [en línea]. <http://xillybus.com/xillinux>. [Consulta: 5 de agosto 2018].

11. ZHANG, Weijun. *VHDL Tutorial: Learn by Example*. [en línea]. <http://esd.cs.ucr.edu/labs/tutorial/>. [Consulta: 14 de enero 2018].

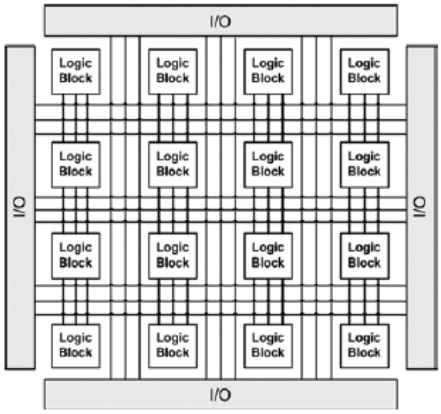
APÉNDICES

Apéndice 1. **Imagen de muestra de una FPGA Zybo, utilizada en las pruebas**



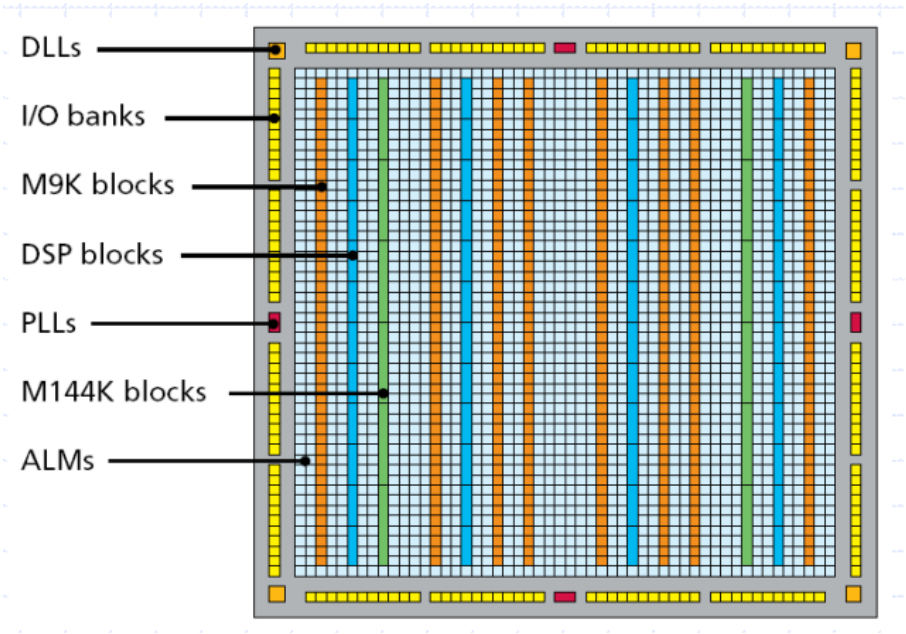
Fuente: elaboración propia.

Apéndice 2. **Bloques internos de una FPGA**



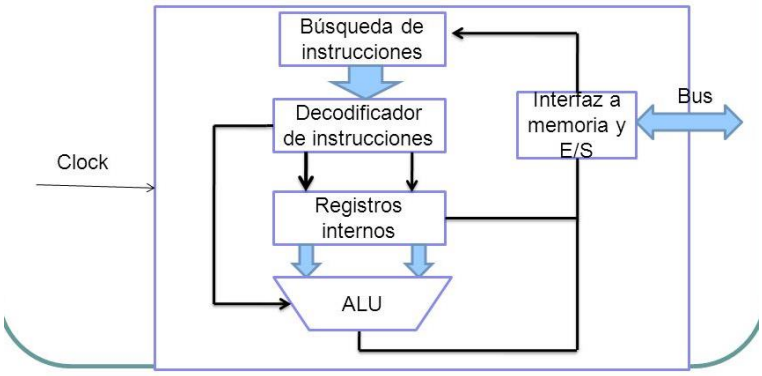
Fuente: elaboración propia.

Apéndice 3. **Diagrama de descripción de los componentes de una FPGA**



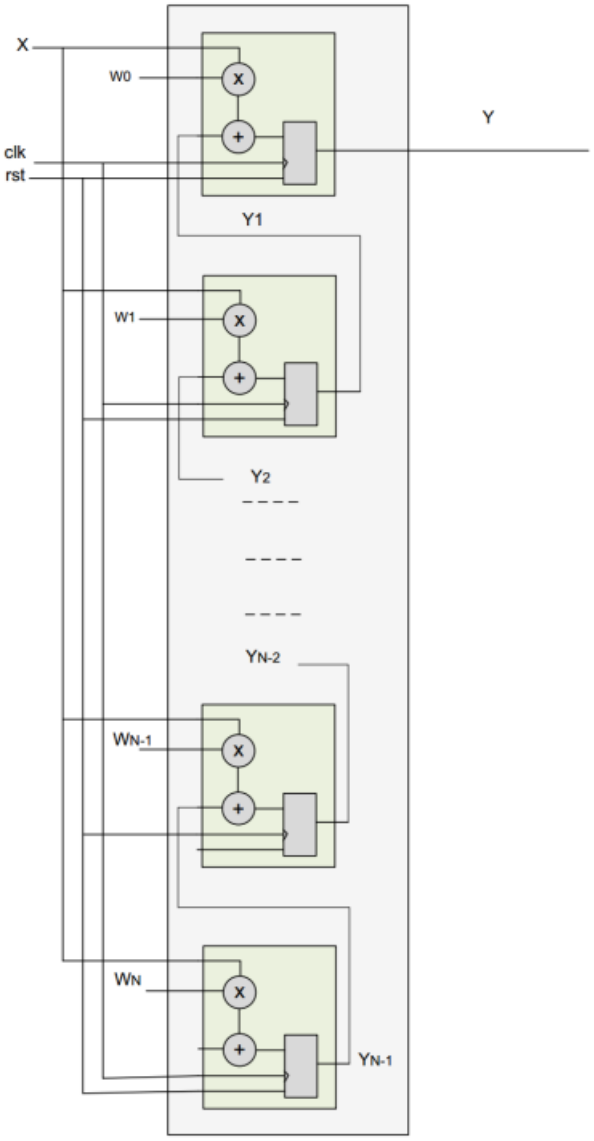
Fuente: elaboración propia.

Apéndice 4. **Diagrama de los bloques propuestos de un microprocesador sintetizado**



Fuente: elaboración propia.

Apéndice 5. **Diagrama general de un filtro FIR para ser generado en una FPGA**



Fuente: elaboración propia.