UNIVERSIDAD POLITÉCNICA DE VALENCIA

ESCUELA POLITÉCNICA SUPERIROR DE GANDIA

I.T TELECOMUNICACIÓN (SIST. ELECTRONICOS)

UNIVERSIDAD
POLITECNICA
DE VALENCIA

# "Diseño e implementación de un sintetizador de pulsos de alto rendimiento con dispositivos FPGAs"

TRABAJO FINAL DE CARRERA

Autor: Herrera Santi José Alan

Director: Javier Valls Coquillat

Gandía, 2011

*Agradecimientos.*

*Antes de terminar me gustaría agradecer a mi familia, por todo el apoyo y motivación recibido a lo largo de los años de mi formación académica.*
*Muchas gracias a mi tutor Javier Valls Coquillat por la ayuda y consejos, por estar siempre disponible para resolver mis dudas.*
*Quiero agradecer también al grupo de investigación del laboratorio de Gandía por la ayuda técnica aportada.*

Índice General

# Índice de figuras.

## Resumen

Debido a la complejidad alcanzada actualmente en el campo de la tecnología, necesitamos ciertos instrumentos para medir, verificar, corregir los problemas que surgen, para seguir progresando y poder realizar cada vez diseños más complejos.

En el campo tecnológico una solución son los sistemas digitales los cuales como se ha podido comprobar son una fuerte herramienta para descubrir, innovar y progresar.

Haremos uso de uno de los dispositivos más avanzados la "*FPGA*" ya que presenta características muy importantes como flexibilidad, facilidad de uso, adaptación a diversas aplicaciones y sobre todo muy económicos por ser programables.

El diseño consta de un dispositivo *FPGA* y el lenguaje utilizado es el *VHDL* con el cual podremos simular, modelar y sintetizar, es decir, traducir a la implementación real.

El proyecto tiene como objeto diseñar e implementar un instrumento capaz de generar pulsos programables con un alto rendimiento mediante dispositivos FPGA.



## Abstract

Due to the complexity currently achieved in the field of technology, we need certain tools to measure, verify, correcting the problems that arise, to move forward and to realize more and more complex designs.

Technologically, the fundamental solutions are the digitals systems which has been shown to be a powerful tool to discover, innovate and grow.

We will use one of the most advanced devices the FPGA as it has very important characteristics such as flexibility, ease of use, adaptability to various applications and above all very cheap to be programmable.

Design consists of an FPGA and VHDL language is used with which we can simulate, model, and synthesize that is translated into actual implementation.

The project aims to design and implement a tool capable of generating pulses and implement with high performance using FPGA devices.

# 1. INTRODUCCIÓN.

## 1.1 OBJETIVOS.

- El objetivo general es el de realizar un dispositivo capaz de generar pulsos programables con un alto rendimiento para testear un equipo en cuestión.

- El instrumento será capaz de generar pulsos de duración arbitraria a elevadas frecuencias obteniendo así la posibilidad de tener pulsos estrechos de periodos del orden de ns(nanosegundos).

- Por otra parte los objetivos específicos son de realizar un programa basado en varios módulos utilizando el lenguaje *VHDL* para generar y controlar los pulsos. Seguidamente se realizará una simulación del diseño y sucesivamente se pasará a la implementación con el dispositivo *FPGA*.

## 1.2 INTRODUCCIÓN FPGA.

*FPGA* ( *Field Programmable Gate Array*) que surge de la evolución de los dispositivos PAL y CPLD, es un dispositivo semiconductor que contiene bloques de lógica. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip.

Las *FPGAs* se utilizan en aplicaciones similares a los *ASICs* sin embargo son más lentas, tienen un mayor consumo de potencia y no pueden abarcar sistemas tan complejos como ellos. A pesar de esto tienen las ventajas de ser reprogramables (lo que añade una enorme flexibilidad al flujo de diseño), sus costes de desarrollo y adquisición son mucho menores para pequeñas cantidades de dispositivos y el tiempo de desarrollo es también menor.

## 1.3. ARQUITECTURA BÁSICA Y FUNCIONAMIENTO DEL *FPGA*.

La fabricación de dispositivos lógicos programables se puede dividir en dos partes:

1. Funcionalidad completa donde cualquier función lógica se puede realizar mediante una suma de productos.

2. Celdas de funciones universales las cuales son bloques lógicos preparados para procesar cualquier función lógica, siendo similares en su funcionamiento a una memoria.

La arquitectura de un *FPGA* consiste en varias celdas lógicas que se comunican mediante canales de interconexión verticales y horizontales. Sus entradas funcionan como un bus de direcciones y por medio de las diferentes combinaciones las celdas seleccionan el resultado correcto. Hay una gran cantidad de celdas lógicas lo que hace posible la implementación de grandes funciones con la utilización de varias celdas lógicas interconectadas en forma de cascada.

Las tecnologías utilizadas en la creación de las conexiones pueden ser:

- *ANTIFUSE* utilizada por Ciprés, Actel, Xilinx y QuickLogic;
- *SRAM* utilizada por Altera, Lucent Technologies, Atmel, Xilinx y otros.

Esta arquitectura proporciona grandes ventajas en el ámbito industrial, además de promover un desarrollo tecnológico. En la figura 1.1 podemos ver la representación de la estructura del dispositivo *FPGA*.

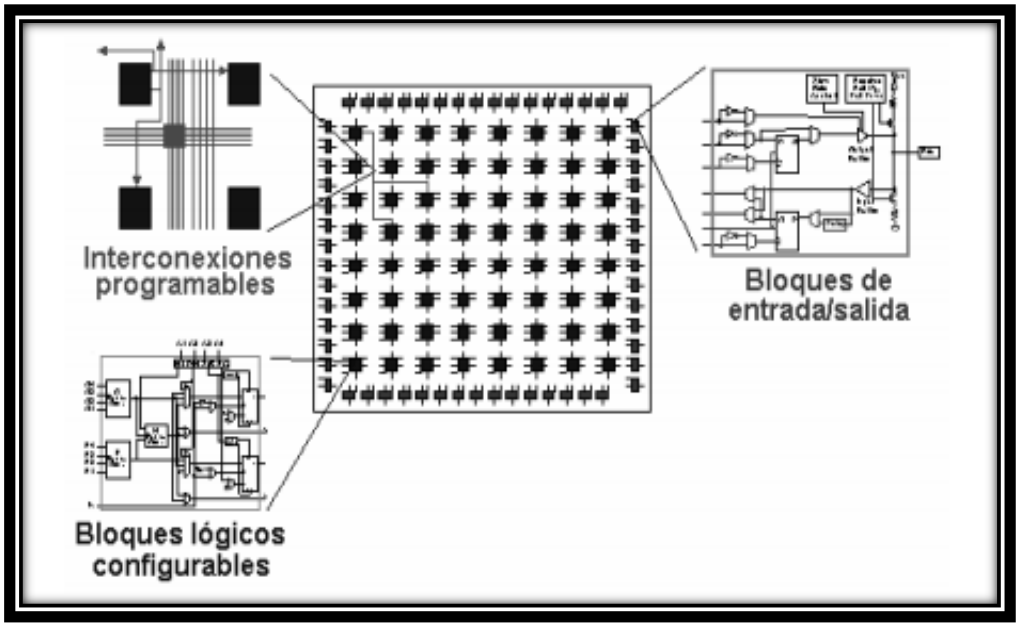



Figura 1.1: Estructura del dispositivo FPGA.

Para implementar el proyecto se hará uso del dispositivo *FPGA* de la familia Virtex-5 por sus excelentes prestaciones.

## 1.4. CARACTERÍSTICAS DE LA FAMILIA VIRTEX-5.

La familia Virtex-5 proporciona las funciones más recientes y más poderosas en el mercado de *FPGA*. Usa la segunda generación *ASMBL* (*Advanced Silicon Modular Block*), utilizando una arquitectura basada en columnas que contienen cinco plataformas distintas (sub-familias). Cada plataforma contiene una proporción diferente de características para satisfacer las necesidades de una amplia variedad de diseños de lógica avanzada.

Contiene una poderosa memoria *RAM* de 36 Kbit / *FIFOs*, una fuente de bloques sincrónica *ChipSync*, la funcionalidad del sistema del monitor, el aumento en la gestión de relojes integrados con *DCM* (*Digital Clock Managers*) , los generadores de reloj (*PLL*) y opciones avanzadas de configuración.

Las características adicionales incluyen la plataforma de bloques de la optimización de la energía de las series de alta velocidad para la conexión del transmisor-receptor serial.

El *PCI Express* en esta familia es compatible con los bloques de extremo integrado, con *tri-modo Ethernet MAC* (*Media Access Controllers*) y con los bloques de alto rendimiento *PowerPC* incrustados en el microprocesador. Estas características permiten a los diseñadores la lógica avanzada para construir los más altos niveles de rendimiento y funcionalidad en sus sistemas basados en *FPGAs.*

## 1.5. LENGUAJE VHDL.

### 1.5.1. INTRODUCCIÓN *VHDL*.

Actualmente nos encontramos rodeados de sistemas electrónicos muy sofisticados, que demuestran un gran crecimiento en el desarrollo tecnológico, los cuales han cambiado nuestras vidas haciéndolas más cómodas. Con el tiempo hemos podido comprobar que las dimensiones y la potencia que caracteriza a estos sistemas han ido cambiando por diversos motivos económicos como de comodidad. Esto fue causa de la evolución de la tecnología y el diseño de la microelectrónica que ha permitido la realización de sistemas electrónicos digitales.

*VHDL* surge a principio de los ’80 de un proyecto *DARPA* (Departamento de Defensa de los EE.UU) y aparece como una manera de describir los circuitos integrados. Cada día los circuitos integrados eran más complicados y el coste de reponerlos cada vez era mayor. Debido a estos problemas nació *VHDL* como una manera *standar* de documentar los circuitos. Al mismo tiempo se vio que la expresividad de *VHDL* permitiría reducir el tiempo de diseño de los circuitos porque se podrían crear directamente de su descripción.

En 1987 el trabajo fue cedido al *IEEE* (*Institute of Electrical and Electronics Engineers*), y a partir de ese momento es un estándar abierto. Aunque puede ser usado de forma general para describir cualquier circuito se usa principalmente para programar *PLD* (*Programable Logic Device* ), *FPGA* (*Field Programmable Gate Array*), *ASIC* y similares.

La siguiente figura1.2 se muestra la evolución de *VHDL* a través de los años.

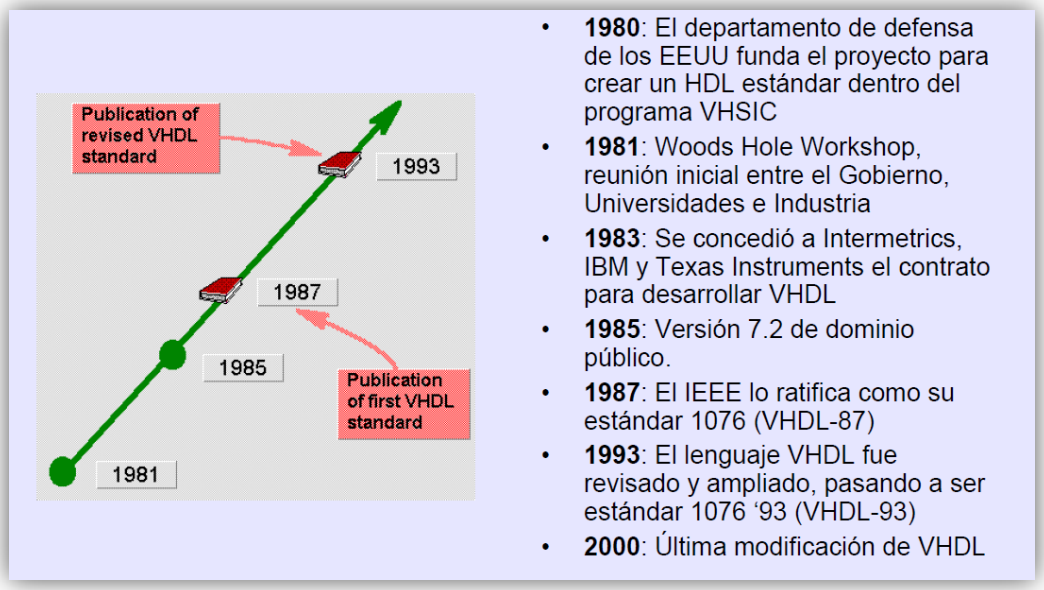



Figura 1.2: Evolución del lenguaje VHDL.

Historial de los circuitos de integración a lo largo de los años:

- de baja escala (*SSI o Small Scale Integration*),
- de mediana escala (*MSI o Médium Scale Integration*),
- de muy alta escala (*VLSI o Very Large Scale Integration*, ),
- de propósito específico ASIC (*Application Specific Integrated Circuit).*

Actualmente la tendencia en el diseño de circuitos para aplicaciones específicas se basa en la utilización de celdas programables preestablecidas dentro del circuito integrado. Con base en lo anterior, surgen los dispositivos lógicos programables (*PLD*) cuyo nivel de densidad de integración se ha venido incrementando al paso del tiempo.

Desde los dispositivos (*PAL*) hasta los dispositivos lógicos programables complejos (*CPLD*) y las compuertas programables en campo (*FPGA*), que han dado como resultado mayor facilidad en el desarrollo de estos circuitos y una disminución en su costo de fabricación.

Para este tipo de diseño de aplicaciones existe un lenguaje de programación que es considerado una de las mejores herramientas para el diseño de sistemas dentro de la industria. En nuestro caso el lenguaje de descripción de hardware *VHDL* nos da la posibilidad de integrar aplicaciones digitales de forma fácil y cómoda.

Por lo tanto se utilizó la herramienta *VHDL* por las siguientes razones:

- Es un lenguaje que permite integrar sistemas digitales en forma fácil y práctica.
- Permite la implementación de sistemas reales mediante la sintetización.
- Costos menores con respecto al tiempo de diseño y la implementación del mismo.
- Posee esquemáticos para facilitar el diseño y permite la división del diseño de un programa complejo en varios módulos con menor complejidad de programación.
- Permite modelar, simular y sintetizar circuitos digitales.
- Nos proporciona un sistema de la localización de fallos durante la compilación y simulación.

## 1.6. MODO DE TRABAJO.

El proyecto consta de dos fases de trabajo que a su vez cada fase está dividida en secciones. La fase 1 referente al software está compuesta de cinco etapas modulares mientras que la fase 2 está dividida en la sintetización e implementación.

Fases de diseño:

- Fase1: Diseño de software mediante la herramienta de lenguaje de programación *VHDL*.

- Fase2: Implementación y sintetización del sistema mediante el dispositivo lógico programable *FPGA*.

## 1.7. DESCRIPCIÓN GENERAL.

Como está especificado en los objetivos diseñaremos un generador de pulsos de alto rendimiento. Para ello utilizaremos el software *VHDL* del cual se hablará con detenimiento en el anexo 1 y el dispositivo *FPGA*.

Con el dispositivo *FPGA* junto con el lenguaje de programación *VHDL*, podemos crear software e implementarlo en hardware. Siendo un dispositivo lógico programable nos da la posibilidad de implementar y sintetizar el diseño creado. De esta forma desarrollamos programas sencillamente con la posibilidad de crear, dividir en módulos y trabajar con diferentes componentes que nos proporciona la *FPGA* por defecto.

El diseño estará compuesto por varias etapas de programación las cuales se describirán más adelante. Las etapas modulares están referidas a los componentes que utilizaremos así como su configuración.

Las etapas modulares son las siguientes:

- Módulo *PWM*.
- Módulo *PLL*.
- Módulo resolución.
- *Buffers* y registros.
- Bloque de memoria.

Cada módulo posee dentro de su arquitectura la declaración de procesos que equivale a una programación secuencial. En nuestro caso se hizo una descripción funcional utilizando el estilo de programación *behavioral*.

Para el desarrollo de los módulos *PWM* y *PLL* se han hecho uso de los componentes embebidos que posee la *FPGA*: *SERDES* y *PLL*. El componente *SERDES* nos brindará la posibilidad de la serialización y descerialización de datos. Será el encargado de recoger los datos enviados por el módulo *PWM* y los serializaremos obteniendo una salida de datos a alta velocidad. Este componente por lo tanto trabajará con altas frecuencias y será una pieza clave para cumplir con nuestro objetivo.

La red de distribución de reloj será controlada por el componente *PLL* el cual recibirá una frecuencia de 100Mhz y podrá obtener salidas de un máximo de 600Mhz.

Para la implementación de los componentes embebidos tendremos que hacer uso la librería *UNISIM*. Esta librería es la que hace referencia a un *core* interno, es decir, a un componente embebido que posee el dispositivo *FPGA*. Facilita y agiliza el diseño ya que permiten tener acceso a estructuras lógicas predeterminadas por el fabricante.

Hay un apartado dedicado a los buffers y registros utilizados ya que cumplen una función muy importante para el diseño. Con ellos conectaremos las entradas a los módulos, obtendremos salidas con alta velocidad y realizaremos un análisis en el tiempo.

Incorporaremos una memoria con una entrada de cuatro bits y una salida de 32 bits que se encargará de almacenar valores binarios referidos al periodo y ciclo activo de la señal.

Por último se diseño la etapa de resolución con la cual podremos insertar una resolución a nuestra señal y para ello necesitaremos también de los resultados de otros módulos. Esta etapa será una señal resultante lista para enviarse al componente *SERDES.*

El módulo resolución estará programado de forma que podamos tener varias posibilidades de trabajo, es decir, si queremos generar solo una señal de resolución o una señal sin resolución o las dos señales juntas.

Se ha dedicado un apartado sobre la implementación en donde describiremos el proceso realizado y los materiales de trabajo. A continuación se explicara con más detenimiento cada una de las etapas modulares. También hablaremos de las fases de programación, se detallará los pasos a seguir, su descripción y los ficheros de trabajo que se crean en cada momento.

## 1.8. COMPONENTES *PLL* Y *SERDES*.

### SERDES

El serializador/deserializer (*SerDes*) conforma un par de bloques funcionales de uso común en las comunicaciones de alta velocidad para compensar la entrada/salida. Estos bloques pueden convertir datos en serie y en paralelo en cada dirección.

Las interconexiones en serie actualmente ocupan un lugar muy importante en los sistemas de comunicación modernos debido a su impacto en el mundo de la tecnología por el rendimiento y sobre todo por el costo. Actualmente estos componentes se han adaptado rápidamente a las necesidades surgidas de aplicaciones específicas.

Como hemos descripto anteriormente, las piezas claves del diseño se denominan *PLL* y *SerDes.* Tales componentes fueron seleccionados para el desarrollo de proyecto porque nos proporcionan muchas ventajas a la hora de diseñar las etapas modulares.

El componente *SERDES* en particular se eligió por las diferentes prestaciones que nos proporciona en la tecnología moderna con un diseño que facilita la implementación de fuente síncrona de alta velocidad.

# PLL.

Los *PLLs*," lazos de seguimiento de fase, bucles de enganche de fase", (*Phase-Locked Loops*) son dispositivos muy populares en electrónica. Se trata de un sistema realimentado, en el que las magnitudes realimentadas son la frecuencia y la fase.
Su uso se vio restringido por décadas, debido a su complejidad y costo económico, a los ámbitos militares y de investigación científica hasta que en la década de 1960 pudo integrarse en un solo chip toda la circuitería de un sistema *PLL* completo, con lo cual su empleo comenzó a hacerse cada vez más masivo.

Unos de los objetivos del proyecto consiste en poder elegir la frecuencia de las señales que queremos generar y por lo tanto nuestro diseño tiene que cumplir con las características de un instrumento.
Por otra parte, el instrumento fue diseñado con la característica de poseer un alto rendimiento y para ello se hizo uso de los componentes necesarios para poder configurar las frecuencias y obtener los resultados correctos.

Por lo tanto con el propósito de manipular la frecuencia de salida y obtener un alto rendimiento hemos utilizado el componente embebido que posee el dispositivo *FPGA* denominado "*PLL*".

# 2. DESCRIPCIÓN DE LAS ETAPAS QUE CONSTITUYEN EL PROYECTO.

## 2.1 DISEÑO JERÁRQUICO DE NUESTRO DISEÑO.

El trabajo se estructuró de forma jerárquica y como se puede apreciar en la figura 1.3 podemos ver que los componentes pequeños son utilizados como elementos de otros más grandes. De este modo podemos reutilizar código y realizar diseños más legibles y portables.

El árbol de jerarquía nos muestra en la parte superior al componente TOP, seguido de un componente inferior a1 que a su vez está compuesto de los componentes IBUFG, PLL, BUFG1, BUFG2, MEMORIA, REGISTRO1, REGISTRO2, PWM, RESOLUCIÓN, REGISTRO DE SALIDA, OSERDE Y OBUFDS. Cada componente de la jerarquía está compuesto por una entidad y una arquitectura.

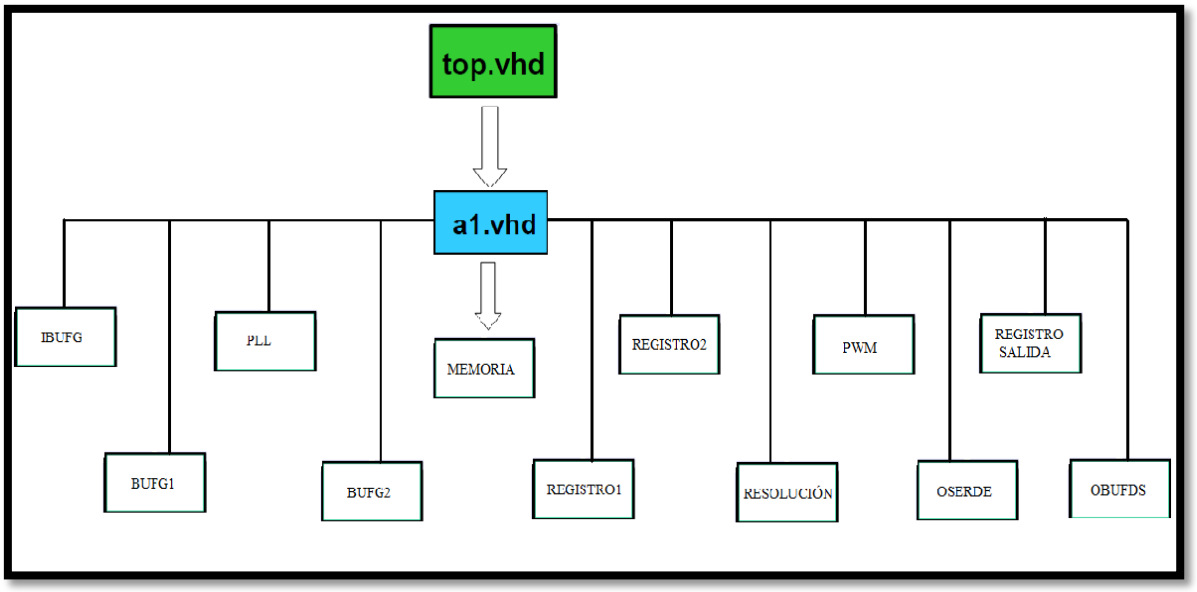



Figura 1.3: Árbol de jerarquía del diseño.

### 2.1.1. ETAPA 1: CONTROLADOR Y GENERADOR DE PWM.

La primera etapa consta de una unidad de control capaz de procesar ciertos valores binarios y generar una señal PWM. El objetivo de esta sección es poder manipular la frecuencia y el ciclo activo de la señal de salida PWM.

La técnica que se ha utilizado para generar la señal *PWM* es la de modulación por ancho de pulso que consiste en definir una señal a una determinada frecuencia con la posibilidad de poder variar el ciclo de trabajo (el trozo de señal que esta a nivel alto). La figura 1.4 nos detalla el ciclo de trabajo y el periodo de señal.

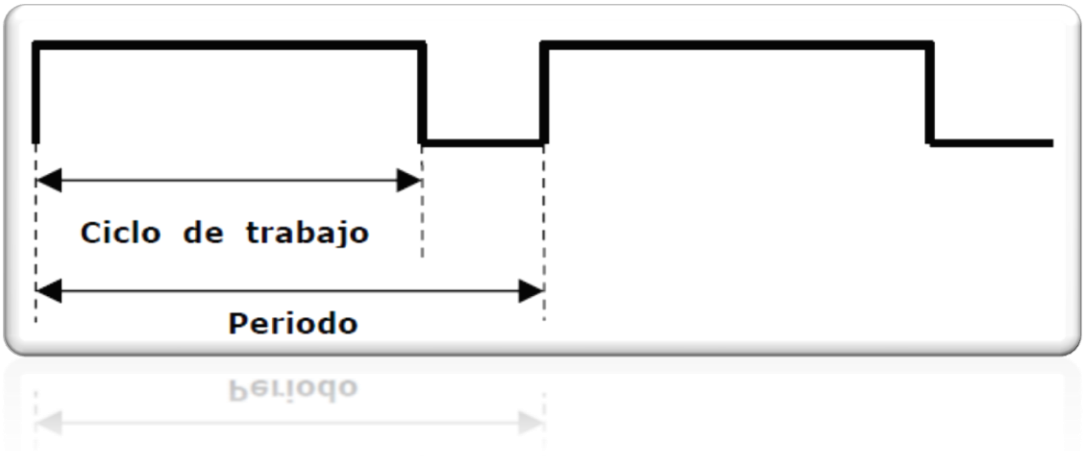



Figura 1.4: Modulación por ancho de pulso.

### 2.1.1.1. DIAGRAMA DE BLOQUES GENERAL.

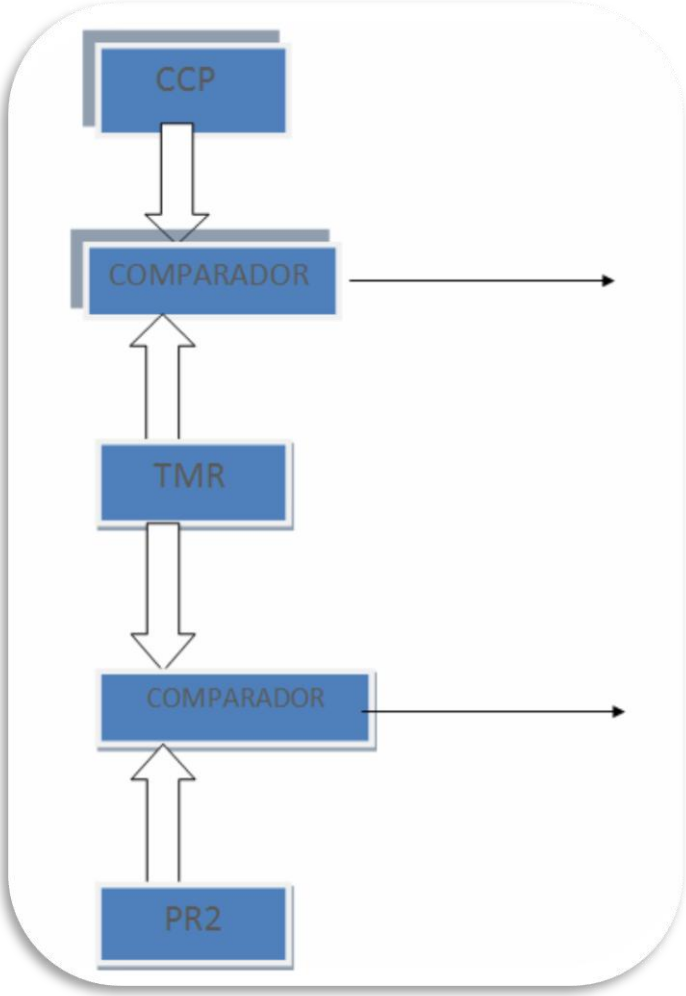



Figura 1.5: Diagramas de bloques general del módulo *PWM*.

Como podemos observar en el diagrama de bloques (figura 1.5), el sistema consta de un módulo *CCP*, referido al ciclo activo, el cual está configurado con 30 bits y conectado junto con el *TIMER* a un comparador.

Por otra parte también tenemos el módulo *PR2* de 30 bits con el cual introduciremos los valores del periodo de la señal. Este módulo también está conectado junto con el *TIMER* a un comparador. Por lo tanto la etapa de generación de *PWM* requiere de dos señales que se comparan.

Descripción de la etapa *PWM*:

Los valores desplegados como el ancho del periodo y el ciclo de trabajo serán introducidos por una memoria *RAM* y serán enviados al módulo *PWM*. El bloque carga todos los parámetros y los despliega hacia los módulos en los que también se utilizarán. La interacción de los módulos se ejecuta de forma síncrona y todo el sistema está conectado a una entrada de reloj de 100Mhz.

El contador (*TMR*), cuando se iguala con el registro *PR2*, provoca que la salida *PWM* tome el valor activo '1' y seguidamente el *TMR* se borra. Otra vez, se empieza a contar desde cero, y cuando el *TMR* nuevamente se iguala con el valor del registro *CCP* la salida toma el valor '0' y el *TMR* sigue contando. En el momento en que el *TMR* se iguala de nuevo con el registro *PR2* la salida *PWM* vuelve a tomar el valor '1', y así sucesivamente. El contador y los registros de comparación, en definitiva son los encargados de generar el ciclo de trabajo y el periodo.

## 2.2. ETAPA 2: MÓDULO GENERADOR DE RESOLUCIÓN.

### 2.2.1. ESTRUCTURA MODULAR DEL GENERADOR DE RESOLUCIÓN.

En esta etapa se diseño un programa capaz de generar una resolución mediante un registro de entrada con una amplitud de 3 bits.
Como podemos ver en el siguiente diagrama de bloques de la figura 1.6, para realizar el módulo generador de resolución dependemos de la etapa de modulación *PWM*.


Figura 1.6: Conexión entre el módulo *PWM* y Resolución.

Las señales internas AVCICLO, AVRESOL y PATRONciclo están conectadas entre la etapa *PWM* y la etapa resolución con el objetivo de poder generar una señal con cierta resolución.

El funcionamiento de las dos etapas juntas es el siguiente: la señal AVCICLO está programada de tal forma que avisa al módulo resolución cuando exista algún cambio en el ciclo de trabajo, es decir cuando tienen un valor '1' o '0'.

Por otra parte tenemos la señal AVRESOL conectada entre el módulo *PWM* y el módulo resolución con el fin de avisar a la etapa de resolución cuando ha terminado el ciclo activo de trabajo para poder añadir la resolución correspondiente elegida personalmente.

La señal interna PATRONciclo la utilizamos como medio de seguridad. Ésta señal de seis bits envía al módulo de resolución un dato de "111111" cuando en la etapa *PWM* se comprueba que el siguiente dato que se ha introducido por el registro *PR2* no es cero, es decir que hemos elegido un valor para el periodo. En el caso de no haber elegido ningún valor de periodo y si haber elegido un valor de resolución, el módulo de resolución pasará a generar solo el pulso correspondiente a la resolución. De este modo tenemos varias opciones para generar pulsos desde una frecuencia de 100Mhz de entrada hasta una de 600Mhz (explicado en el apartado de configuración de frecuencia).

Por lo tanto con las dos etapas juntas podremos obtener una señal con ciclos de trabajo y periodo hasta un ancho de 30 bits, una señal que solo corresponda a la parte de resolución o los dos pulsos juntos para formar la generación de una señal con resolución.

### 2.2.2. DIAGRAMA DE BLOQUES DEL MÓDULO RESOLUCIÓN.

La figura 1.7 muestra el diagrama de bloques de la etapa de resolución. (En el diagrama no se han añadido las señales de reloj para no complicar el diseño, pero todo está preparado para que funcione síncronamente.

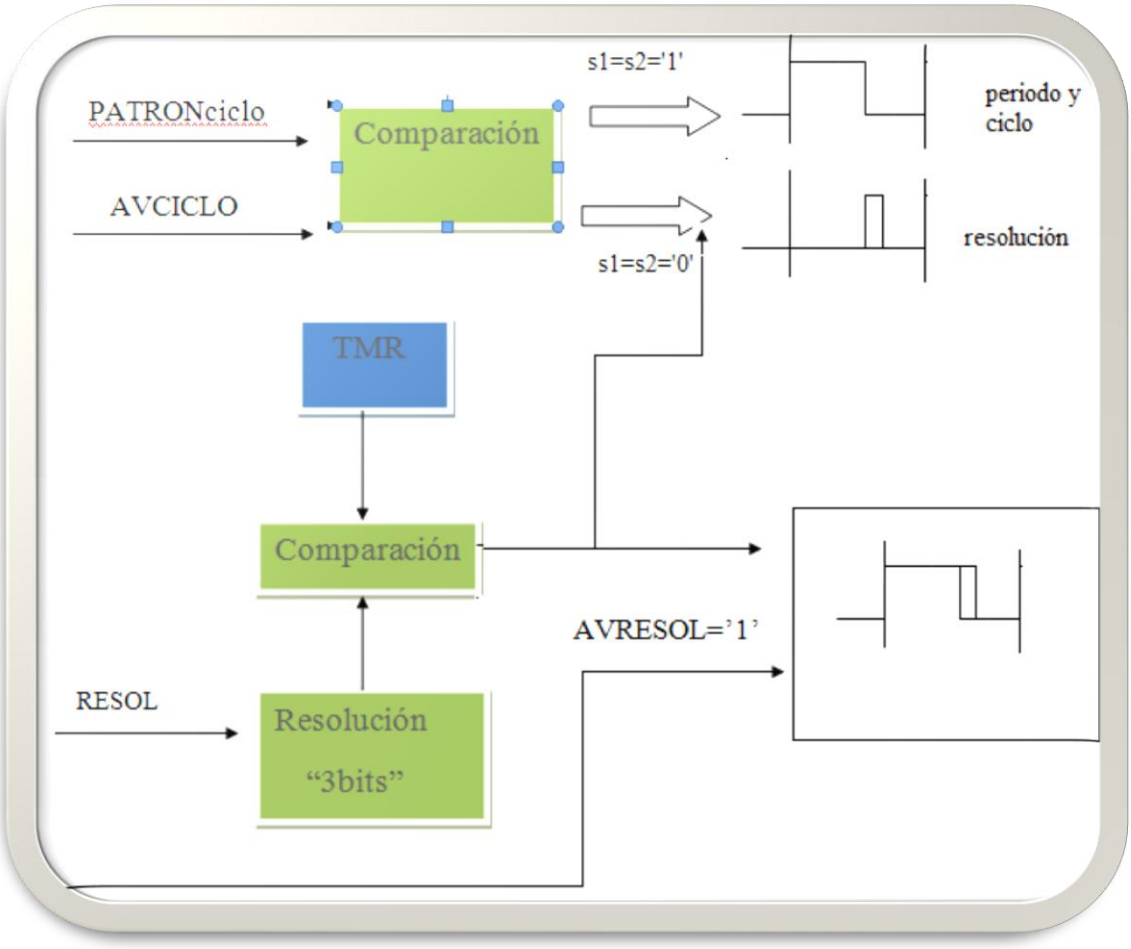



Figura 1.7: Diagramas de bloques del módulo Resolución.

En el diagrama se ha especificado s1=s2='1' como referencia a las señales PATROciclo y AVCICLO.

Como está descripto en el diagrama anterior la etapa de resolución posee dos registros de comparación un *TIMER* y el registro de resolución. A la entrada de unos de los registros de comparación aparecen las señales internas desde la etapa *PWM* las cuales serán comparadas y por lo tanto a la salida obtendremos una señal sin resolución o si lo deseamos solo una generación de pulsos muy pequeños correspondientes únicamente a la resolución.

Si llega la señal AVRESOL, el *TIMER* comienza a contar y a compararse con el registro de resolución al que anteriormente se le ha dado un cierto valor de resolución. Por lo tanto si hemos introducido valores tanto en el registro de periodo y en el de resolución se activarán las señales PATRONciclo, AVCICLO Y AVRESOL concluyendo con el funcionamiento deseado para obtener una señal completa.

## 2.3. ETAPA 3: CONFIGURACIÓN DE LA FRECUENCIA.

### 2.3.1. INTRODUCCIÓN AL *PLL*.

Hay que señalar que dentro de la gestión del reloj (*CMT*) en *FPGAs* Virtex-5 se incluye dos *MCD* y un *PLL*. Cada bloque dentro de este esquema embebido de la *FPGA* existe una ruta dedicada entre los bloques de creación. Al ser rutas locales se establece una ruta de reloj mejor reduciendo las posibilidades de acoplamiento de ruido.

### 2.3.2. DIAGRAMA DE BLOQUES DEL PLL.

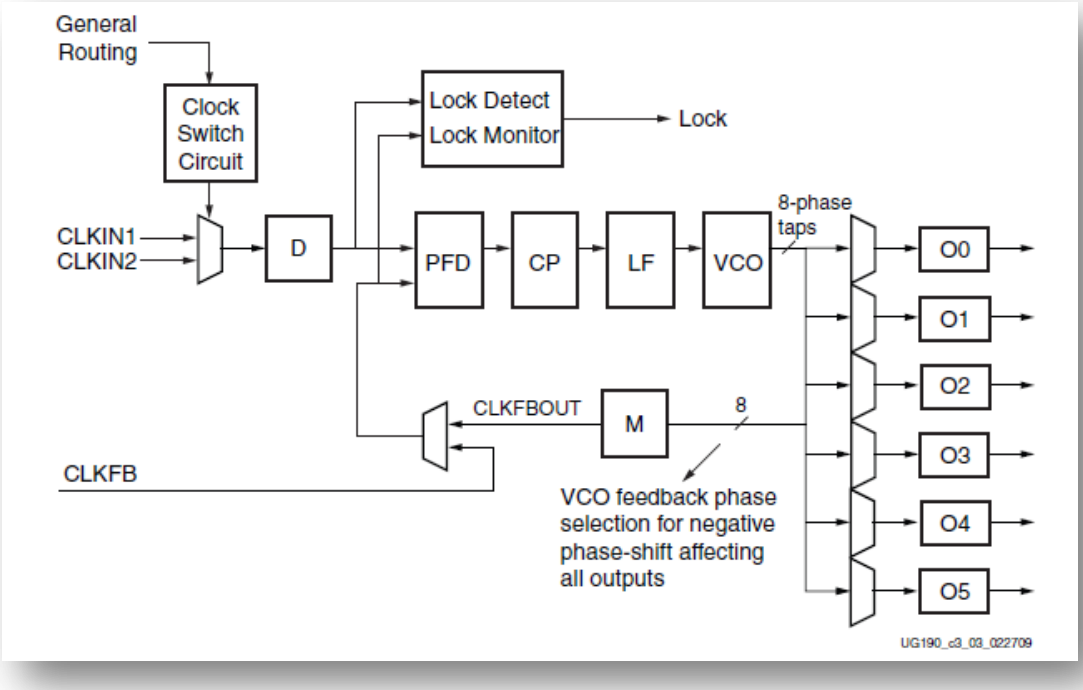



Figura 1.8: Diagramas de bloques del *PLL*.

Como se puede ver con detalle en el diagrama representado por la figura 1.8, cada entrada tiene un reloj programable D. El contador D puede tener diferentes valores dados por el programador con el fin de configurar la frecuencia de salida.

Siguiendo el lazo podemos ver que en primer lugar tenemos un bloque denominado *PFD (Phase-Frequency*) el cual compara la fase y la frecuencia de la entrada de referencia y el lazo de realimentación. En ésta parte del lazo determinamos una señal proporcional en fase y frecuencia entre dos relojes. La señal que obtenemos dirige el bloque *CP* (*Charge Pump*) y el boque *LF* (*Loop Filter*) con el fin de generar un voltaje de referencia para el *VCO*.

El *VCO* cuando opera a muy alta frecuencia provoca que el *PDF* controle el voltaje de manera que reduzca la frecuencia de operación del *VCO*. Y de lo contrario cuando el *VCO* está a muy baja frecuencia el *PDF* se comporta de manera para que se aumente la frecuencia de trabajo del *VCO*.

Como podemos ver en el diagrama de bloques, el *VCO* proporciona ocho salidas de fase las cuales pueden ser elegidas como relojes de referencia.

También el *PLL* consta con un contador denominado M cuya función se basa en controlar el reloj de realimentación permitiendo una amplia gama de síntesis de frecuencia.

La estructura presentada del *PLL BASE* posee una tecnología capaz de proporcionar muchas ventajas como la alineación de la red de reloj, las síntesis de frecuencia y la reducción del *jitter*.

### 2.3.3. AJUSTE DE LA FRECUENCIA DE SALIDA.

La frecuencia de operación del *VCO* puede ser determinada usando las ecuaciones siguientes:

- *FVCO= FCLKIN x M/D*
- *FOUT =FCLKIN x M/DO*

El *PLL* nos da la posibilidad de poder programar independientemente las seis salidas O siempre y cuando respetemos la limitación de la frecuencia de operación del *VCO*.

Con respecto a la red de relojes, se ha recurrido al uso de ciertos *bufers* para la E/S de reloj y para la realimentación del *PLL* (explicaremos en el apartado de *Bufers* y E/S digitales).

En el caso del *Jitter* el *PLL* siempre lo reduce de manera general pero también podemos configurarlo para poder trabajar en modo de filtro de *jitter*. Para usar un mayor filtro de *jitter* tenemos que utilizar el atributo *BANDWITH*.

En nuestra aplicación solamente necesitamos generar frecuencias de reloj de salida para otros bloques. En este modo, para la trayectoria de la generación del *PLL* podemos elegir la configuración de “interior”, ya que mantiene todas las rutas locales y minimizar el caso del *jitter*.
De acuerdo con las limitaciones del *PLL*, mostradas e la figura 1.9, la configuración elegida es la siguiente:

La frecuencia de entrada de nuestro sistema tiene un valor de f=100Mhz con lo que *FVCO= FCLKIN x M/D*=100Mhz x 6/1= 600Mhz será el valor de salida dentro del margen de valores posibles.

La *COMPENSATION* la señalamos con "*SYSTEM_SYNCHRONOUS*" para poder compensar todos los retardos que puedan llegar a tener los relojes para “0” *hold time*.

Como se ha especificado anteriormente el *PLL* posee seis salidas de las cuales solo utilizaremos dos. Para ello la configuración independiente de cada una de las salidas es la siguiente:

*CLKOUT0_DIVIDE => 1,*
*CLKOUT0_DUTY_CYCLE => 0.5*
*CLKOUT0_PHASE => 0.0*
*CLKOUT1_DIVIDE => 6*
*CLKOUT1_DUTY_CYCLE => 0.5*
*CLKOUT1_PHASE => 0.0*
*CLKOUT2_DIVIDE => 1*

Siguiendo con la especificación esperada del *jitter* sobre el reloj de referencia para realizar la optimización global se eligió una *BANDWITH=OPTIMIZED*.

| Symbol | Description | Speed Grade | | | Units |
|---|---|---|---|---|---|
| | | -3 | -2 | -1 | |
| $F_{INMAX}$ | Maximum Input Clock Frequency | 710 | 710 | 645 | MHz |
| $F_{INMIN}$ | Minimum Input Clock Frequency | 19 | 19 | 19 | MHz |
| $F_{INJITTER}$ | Maximum Input Clock Period Jitter | <20% of clock input period or 1 ns Max | | | |
| $F_{INDUTY}$ | Allowable Input Duty Cycle: 19—49 MHz | 25/75 | | | % |
| | Allowable Input Duty Cycle: 50—199 MHz | 30/70 | | | % |
| | Allowable Input Duty Cycle: 200—399 MHz | 35/65 | | | % |
| | Allowable Input Duty Cycle: 400—499 MHz | 40/60 | | | % |
| | Allowable Input Duty Cycle: >500 MHz | 45/55 | | | % |
| $F_{VCOMIN}$ | Minimum PLL VCO Frequency | 400 | 400 | 400 | MHz |
| $F_{VCOMAX}$ | Maximum PLL VCO Frequency | 1440 | 1200 | 1000 | MHz |
| $F_{BANDWIDTH}$ | Low PLL Bandwidth at Typical(1) | 1 | 1 | 1 | MHz |
| | High PLL Bandwidth at Typical(1) | 4 | 4 | 4 | MHz |
| $T_{STAPHAOFFSET}$ | Static Phase Offset of the PLL Outputs | 120 | 120 | 120 | ps |
| $T_{OUTJITTER}$ | PLL Output Jitter(2) | Note 1 | | | |
| $T_{OUTDUTY}$ | PLL Output Clock Duty Cycle Precision(3) | ±150 | ±200 | ±200 | ps |
| $T_{LOCKMAX}$ | PLL Maximum Lock Time(4) | 100 | 100 | 100 | µs |
| $F_{OUTMAX}$ | PLL Maximum Output Frequency for LX20T devices | N/A | 667 | 600 | MHz |
| | PLL Maximum Output Frequency for LX30, LX30T, LX50, LX50T, LX85, LX85T, LX110, LX110T, SX35T, SX50T, FX30T, and FX70Tdevices | 710 | 667 | 600 | MHz |
| | PLL Maximum Output Frequency for LX155, LX155T, and FX100T devices | 650 | 600 | 550 | MHz |
| | PLL Maximum Output Frequency for FX130T devices | 550 | 500 | 450 | MHz |
| | PLL Maximum Output Frequency for LX220, LX220T, LX330, LX330T, SX95T, SX240T, TX150T, TX240T, and FX200T devices | N/A | 500 | 450 | MHz |

Figura 1.9: Limitaciones de frecuencia de la salida del *PLL*.

## 2.4. ETAPA 4: *BUFFERS* Y E/S DIGITALES.

Cuando transmitimos y recibimos datos es muy importante considerar la terminación de la señal ya que dependiendo de las E/S estándar elegida el valor de la salida puede variar.

Debido a que las velocidades del reloj del sistema se vuelven más rápidas, el diseño de placa de circuito impreso y fabricación se hace más difícil y el mantenimiento de la señal, la integridad se convierte en un tema crítico.

Para abordar estos problemas y para lograr una mejor integridad de señal, Xilinx ha desarrollado el control digital Impedancia (*DCI*) de tecnología.

El sistema (*DCI*) es basa en que las señales provenientes de los conectores de E/S se encaminan directamente a la *FPGA* a través de resistencias en serie o paralelo para la terminación de la señal.

Con el dispositivo *FPGA* Virtex-5 podemos configurar las E/S para alto rendimiento ya que nos brinda una amplia variedad de interfaces estándar para lograrlo. El conjunto de funciones incluye un control programable para establecer las condiciones de velocidad y respuesta.

Cada *IOB* contiene entradas, salidas y tres estados de conductores E/S. Estos conductores pueden ser configurados por varios estándares de E/S.

Por otra parte también podemos utilizar un estándar diferencial *LVDS* en el que las señales son pares en los conectores, de tal manera que los pares de señales son asociados como positivos y negativos y se pueden utilizar dentro de la *FPGA* de Xilinx.

### 2.4.1. CLASIFICACIÓN DE LAS E/S.

Las E/S diferencial usa dos grupos juntos *IOBs* los cuales se han clasificado en los siguientes grupos:

• *Single-ended I/O standards* (LVCMOS, LVTTL, HSTL, SSTL, GTL, PCI)

• *Differential I/O standards* (LVDS, HT, LVPECL, BLVDS, Differential HSTL and SSTL)

• *Differential and VREF* son alimentados por *VCCAUX*.
Cada E/S de Virtex-5 *FPGA* contiene dos *IOBs*, y también dos *ILOGIC* y dos *OLOGIC* bloques.

### 2.4.2. PINES DE REFERENCIA.

#### PINES DE TENSIÓN DE REFERENCIA ($VREF$).

Con las normas de un *buffer* de entrada, un terminal de baja tensión requiere un voltaje de entrada de referencia (*VREF*). *VREF* es una entrada externa en los dispositivos Virtex-5. Dentro de cada banco de pines de E/S, uno de cada 20 pines E/S se configura automáticamente como una entrada *VREF*.

#### PINES DE SALIDA DE LA FUENTE DE VOLTAJE ($VCCO$).

Muchos de los pines de baja tensión de acuerdo con las normas O de dispositivos Virtex-5 requieren un voltaje de salida (*VCCO*).
Como resultado, cada dispositivo a menudo soporta múltiples voltajes de salida.

### 2.4.3. CARACTERÍSTICAS.

A efectos de registrar, el dispositivo Virtex-5 se divide en regiones. El número de regiones varía con el tamaño del dispositivo: 8 regiones en el dispositivo más pequeño y 24 regiones en el más grande.

Las E/S globales y regionales son los recursos que se necesitan para tal gestión. Cada dispositivo Virtex-5 cuenta con 32 líneas de relojes globales que pueden registrar todos los recursos secuenciales en el dispositivo completo (*CLB*, la memoria *RAM* del bloque, *CMT*, y E/S), y también las señales de unidad lógica. Cualquiera de estas 32 líneas globales de reloj puede ser usada en cualquier región, se mueven por un buffer de reloj global.

Cada región también tiene dos *buffers* de reloj regionales y cuatro árboles de relojes regionales. Cada una de estas entradas puede ser también de tipo diferencial. Una característica importante que nos ofrece el *buffer* de reloj regional es que puede ser programado para dividir la frecuencia de reloj de entrada por cualquier número del 1 al 8.

Podemos utilizar relojes globales para el uso normal de E/S. Hay 20 entradas globales y con la posibilidad de que todas las entradas puedan ser diferenciales pero en nuestro diseño solo la salida será diferencial.

### 2.4.4. TIPOS DE *BUFFERS* UTILIZADOS.

En el proyecto se utilizaron *buffers* genéricos de entrada de relojes como IBUFG, *buffers* para registrar las entradas de tipo BUFG y *buffers* diferenciales para la salida de alta velocidad. El *búfer* BUFG es un simple *búfer* genérico con un reloj de entrada y de salida. Las figuras 1.10, 2.1 y 2.2 nos muestran los tipos de *buffers* utilizados.

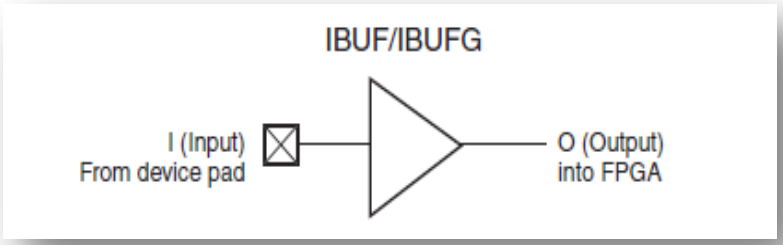



Figura 1.10: Búfer IBUFG.

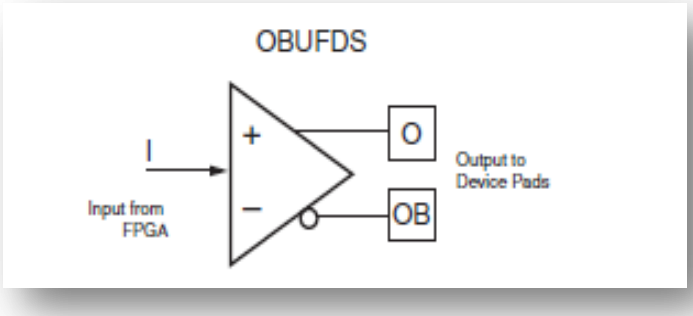



Figura 2.1: Búfer OBUFDS.

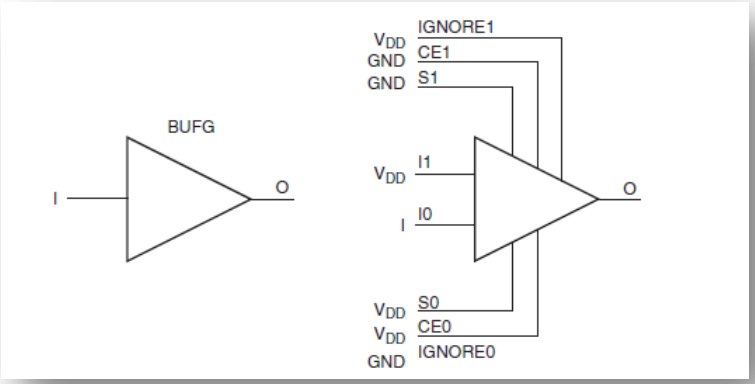



Figura 2.2: Búfer BUFG.

## 2.5. ETAPA 5: SERIALIZACIÓN / DESCERIALIZACIÓN (SERDES).

### 2.5.1. INTRODUCCIÓN.

El serializador / deserializer (SerDes) constituye un par de bloques funcionales de uso común en las comunicaciones de alta velocidad para compensar la E/S. Los bloques del SerDes pueden comunicar datos en serie y paralelo en cada dirección.

La función básica del SerDes se compone de dos bloques funcionales:

- El paralelo en serie (*PISO*) (convertidor de paralelo a serie).
- La serie En paralelo (*SIPO*) (convertidor de serie a paralelo).

Tipos de arquitecturas del SerDes:

(1) _Reloj paralelo SerDes.

(2) _Reloj incorporado SerDes.

(3) _SerDes 8b/10b.

(4) _Bits intercalados SerDes.

### 2.5.2. BLOQUES: PISO Y SIPO.

En nuestro proyecto utilizaremos el bloque *PISO*, donde las entradas son en paralelo y las salidas de serie. El sistema posee una entrada de reloj en paralelo, un conjunto de seis líneas de entrada de datos y los cierres de datos de entrada.

Con respecto al reloj incorporado en el SerDes, es el que se encarga de serializar los datos. En primer lugar actúa un ciclo de la señal de reloj y seguidamente se transmite el flujo de bits de datos. Cuando el reloj está explícitamente incorporado y se puede recuperar de la corriente de bits enviados, el serializador / descerializador consigue relajar la tolerancia del *jitter*.

El SerDes utiliza una fase interna o externa de bucle cerrado para multiplicar el reloj de entrada paralelo hasta la frecuencia de serie.

El bloque *PISO* tiene un registro de cambio único que recibe los datos en paralelo cada vez por ciclo de reloj en paralelo y lo desplaza en serie a alta velocidad.

Por otra parte cabe destacar el otro bloque no utilizado en este proyecto denominado *SIPO* (entrada serial, salida paralelo) el cual trabaja recibiendo un reloj de salida y obteniendo un conjunto de líneas de salida de datos. El reloj se puede recibir si se han recuperado los datos mediante la técnica de recuperación de reloj de serie.

Las implementaciones suelen tener dos registros conectados como un búfer doble. Un registro se utiliza para registrar en la corriente de serie y el otro se utiliza para mantener los datos.

Algunos tipos de SerDes incluye codificación / decodificación de bloques. El objetivo de esta codificación / decodificación suele colocar por lo menos los límites estadísticos sobre la tasa de transiciones de señal para permitir una fácil recuperación de reloj en el receptor, para proporcionar la *framing* y para proporcionar equilibrio de *DC*.

### 2.5.3. MÓDULO OSERDES.

Cada módulo OSERDES incluye un serializador para datos y un control de tres estados. Ambos pueden ser configurados en los modos *SDR* y *DDR* que pasaremos a explicar con detenimiento más adelante.

Los datos que introduciremos en el OSERDES son serializados en un orden de menor a mayor, el dato de entrada D1 será el primer bit en ser transmitido.

Como se puede ver en la figura 2.3 el OSERDES utiliza dos relojes CLK y CLKDIV que están obviamente en fase para la conversión de datos. El reloj CLK es el encargado de proporcionar alta velocidad mientras que el reloj CLKDIV es que se ocupa del mantenimiento de los datos en paralelo.

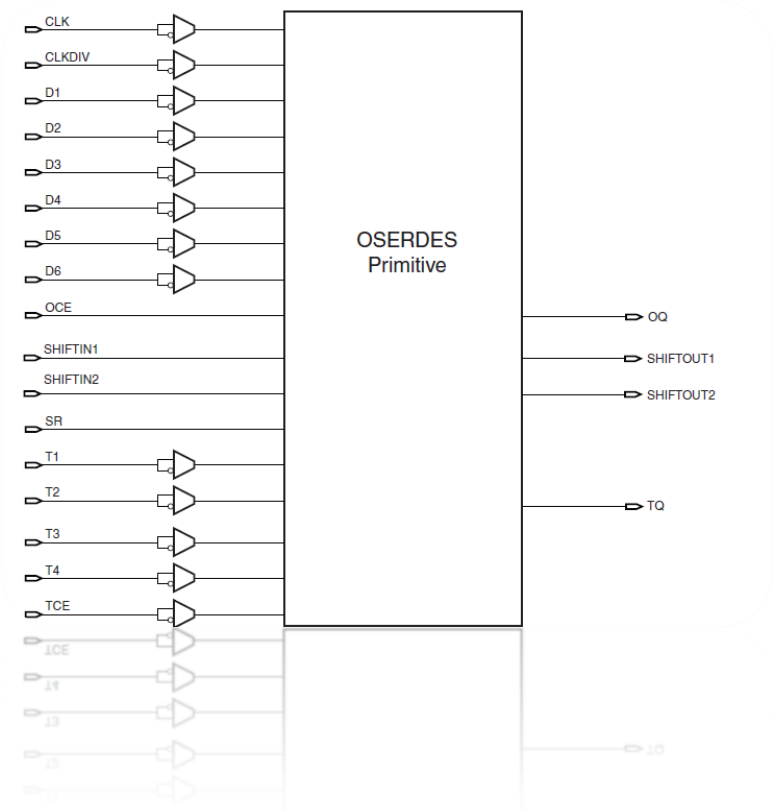



Figura 2.3: Módulo OSERDES.

### 2.5.4. CONFIGURACIÓN DE PINES.

El puerto OQ es el puerto de salida de datos del módulo OSERDES. Los datos que entran por el puerto de entrada D1aparecen por primera vez en OQ. Este puerto se conecta a la salida del convertidor de datos de paralelo a serie en la entrada del *IOB*.

El pin OCE es el encargado de habilitar el reloj de alta velocidad por lo tanto tiene que estar programado a nivel alto.

Cuando se utiliza el control *3-state* de la salida TQ, el puerto debe conectar la salida del convertidor de *3-state* con la entrada del control *3-state* del *IOB*.

Las entradas en paralelo (*3-state Inputs - T1 to T4)* también son puertos conectados a la *FPGA* y pueden ser programados como 1, 2 o 4 bits. Para controlar *3-state control* debemos antes activar el TCE. El ancho el módulo *3-state* (*TRISTATE_WIDTH)* depende del atributo *DATA_RATE_TQ*, el cual si es configurado como *DDR* coloca un valor de ancho de valor cuatro.

De modo contrario si el ancho de los datos es mayor que cuatro automáticamente el *TRISTATE_WIDTH* se configura con un valor de 1. En la siguiente figura 2.4 podemos ver los diferentes bloques, sus entradas y salidas

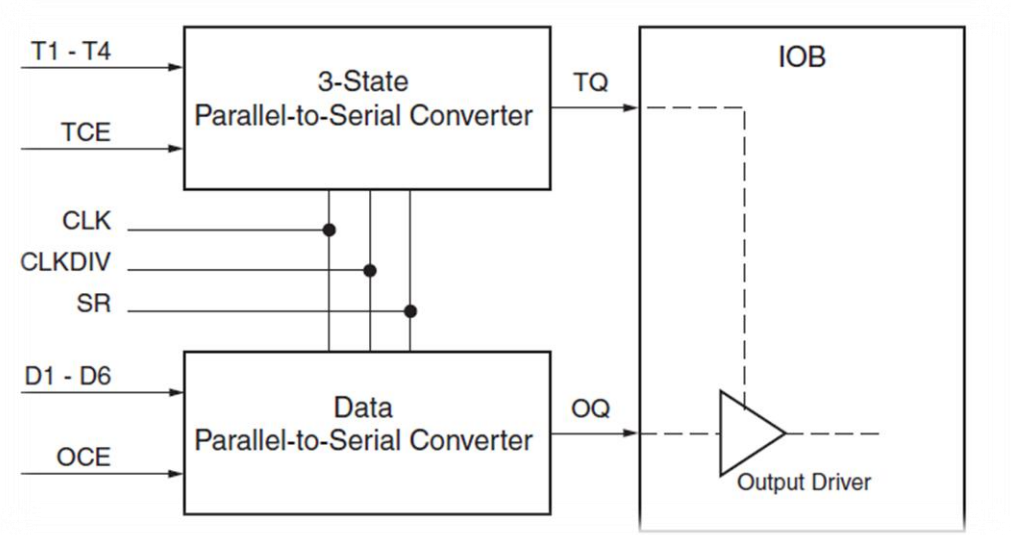



Figura 2.4: Control *3-state*.

Todos los datos de entrada que pasarán a través de los puertos D1-D6 pueden ser configurados hasta seis entradas. Utilizando otro OSERDE podríamos ampliar las entradas hasta 10 bits.

El ancho de los datos depende también del atributo *DATA_RATE_OQ*. Cuando el *DATA_RATE*_OQ está en modo SDR los posibles valores de la anchura de los datos pueden ser de 2, 3, 4, 5, 6, 7, y 8. Por esta razón la configuración de nuestro diseño es *SDR* ya que trabajamos con un ancho de seis bits.

A continuación podemos ver en la figura 2.5 la representación de la serialización de los datos trabajando en el modo *SDR*. Como se puede ver en la gráfica cada ciclo de CLKDIV que equivalen a dos ciclos reloj CLK realiza una muestra de los datos y obtiene por la salida OQ en el siguiente ciclo de reloj.

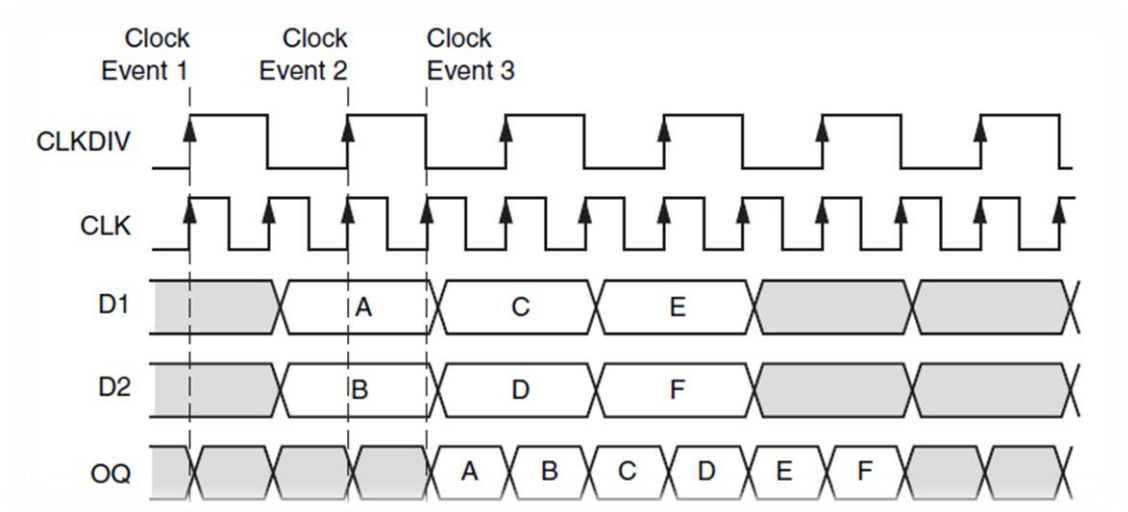



Figura 2.5: Serialización en modo SDR.

En cambio la figura 2.6 nos enseña un ejemplo del modo de trabajo *DDR* con un ancho de datos de cuatro. Dese T1-T4 y D1-D4 forman los caminos de la serialización de tal forma que los bits EFGH están siempre alineados con los valores “0010” presentados en T1-T4 durante el evento de reloj. Los datos se muestrean de acuerdo a los valores de T1-T4 y aparecen un ciclo de reloj después por la salida OQ.

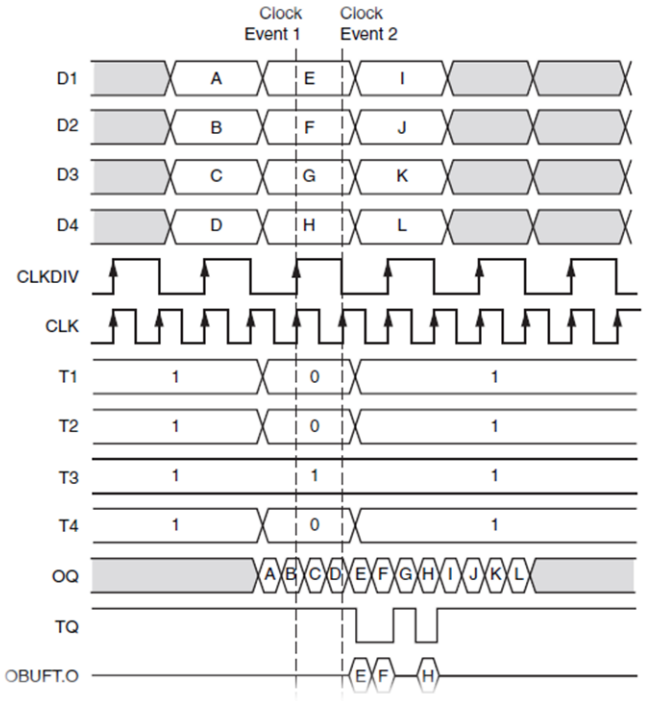



Figura 2.6: Serialización 8:1 en modo DDR.

### 2.5.5. EXPANSIÓN DE MÓDULO OSERDES.

En el caso de desear ampliar el ancho de los bits de entrada, el componente SERDES nos proporciona la posibilidad trabajar con dos OSERDES de los cuales uno trabajará en modo MAESTRO y el otro en modo ESCLAVO. Los bloques necesarios para la expansión se muestran a continuación en la figura 2.7

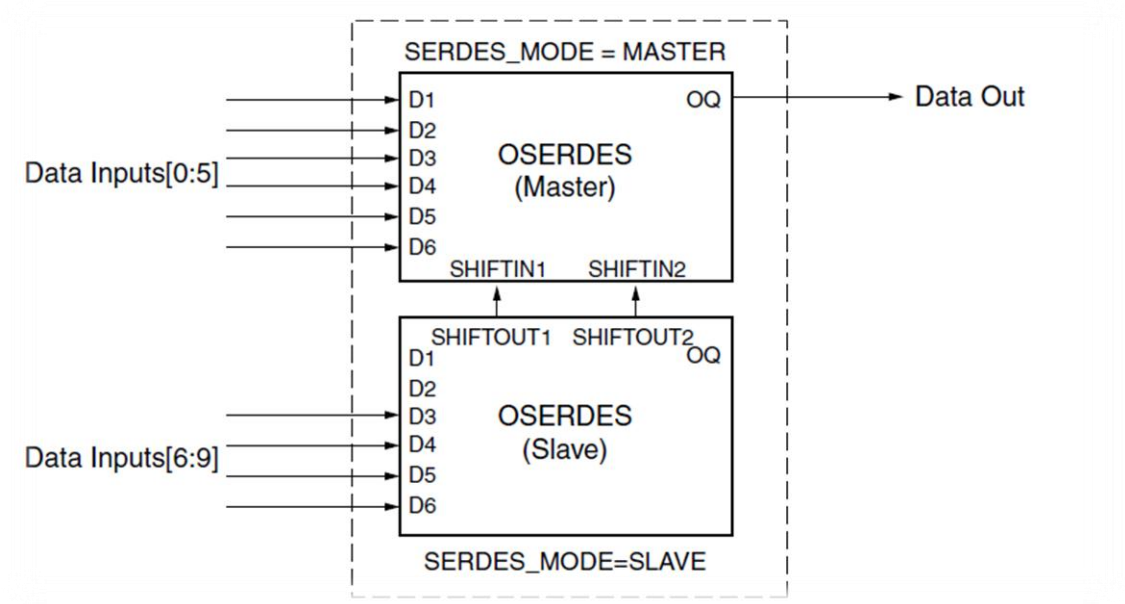



Figura 2.7: Expansión del módulo OSERDE.

### 2.5.6. ESTRUCTURA DE CONEXIÓN *PLL*-OSERDES.

La entrada de reloj CLKIN tiene un valor de 100Mhz la cual pasa por un *clock-buffer* denominado IBUFG para ser suministrado al módulo *PLL* el cual se encargará de proporcionar al OSERDE el CLK (reloj de alta velocidad) y el CLKDIV (reloj de para mantener los seis datos que se quieren enviar).

La salida del PLL, CLKOUT0, es la salida de alta velocidad con un valor de frecuencia de 600Mhz (cuya configuración fue especificada en el apartado del *PLL*) y la salida CLKOUT1 corresponde con el reloj CLKDIV, reloj de valor de frecuencia de 100Mhz que se encarga de mantener los seis datos que se enviaran.

Por otra parte el *PLL* tiene un lazo de realimentación que ayuda a realizar el cometido global de proporcionar tales frecuencias de reloj. Está conectado a un buffer global BUFG y realimentado hacia la entrada CLKFBIN. La salida OQ del OSERDE está conectada a un *buffer* diferencial OBUFDS para proporcionar los datos con alta velocidad.

A continuación la figura 2.8 nos mostrará únicamente las conexiones realizadas de los relojes entre el *PLL* y el OSERDE para trabajar con alto rendimiento.

El diagrama de bloques representa la conexión de todos los bloques a excepción de la memoria. Anteriormente se explicó todo acerca de los *buffers* que se utilizaron.

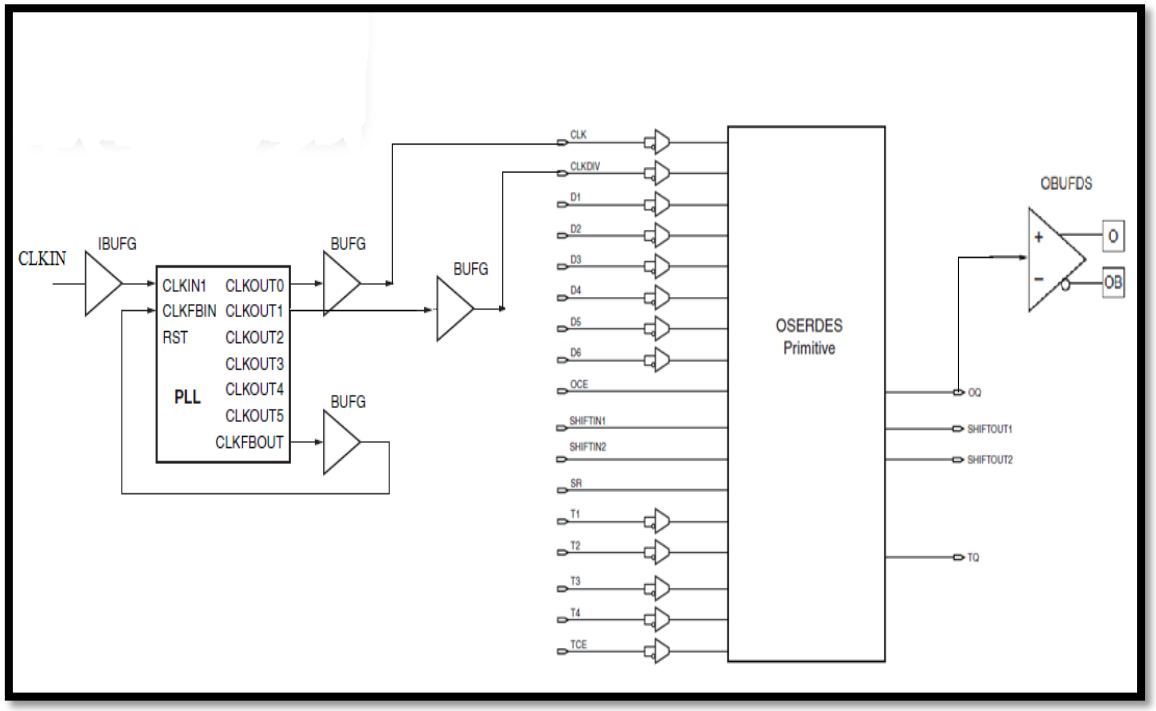



Figura 2.8: Distribución de reloj entre PLL y OSERDE.

El reloj de referencia CLKIN suministrado al *PLL* tiene un valor de 100Mhz y con la ayuda del *PLL* obtenemos dos salidas con un valor de frecuencias de 600Mhz y 100Mhz. Como se explicó en el apartado de generación de resolución tenemos la posibilidad por una parte de utilizar el modulo *PWM* para generar pulsos obteniendo como mínimo pulsos de un valor de 1/100Mhz=10ns, pero por otra parte también tenemos la posibilidad de utilizar los 3 bits del módulo de resolución obteniendo así pulsos más pequeños de un valor mínimo de 1/600Mhz=1.7ns.

En el caso de configurarlo para que trabajen ambas partes tanto módulo *PWM* como resolución tendríamos un pulso con un valor mínimo de 11,7ns. En el caso de la elección solamente de la resolución, el modulo está formado con tres bits lo que nos da la posibilidad de trabajar con ocho posibles valores partiendo del pulso más pequeño de 1,7ns.

Los posibles valores de resolución que puede elegir el programador para generar se pueden ver en la figura 2.9.

VALORES DE RESOLUCIÓN.

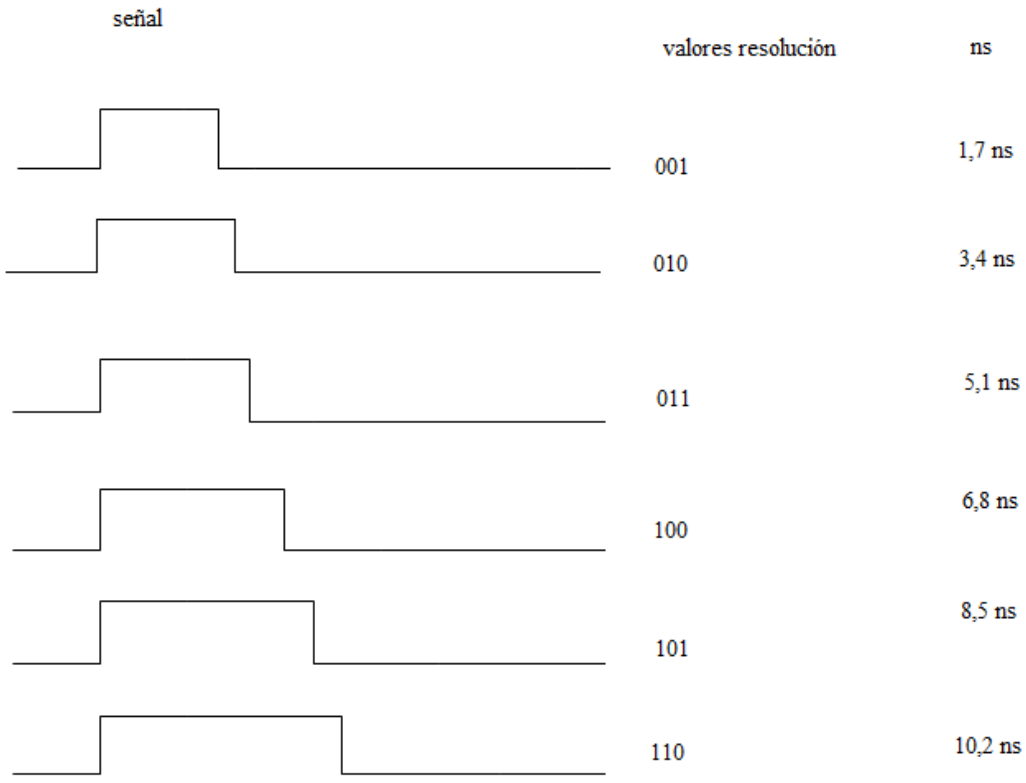



Figura 2.9: Valores de resolución.

Los valores referidos a la resolución como “000” y “111” no están presentes debido a que el valor de resolución “000” significa la elección de no introducir resolución y con respecto al valor “111” no lo necesitamos por que el valor “010” es el máximo que podemos entregar.

### 2.5.7. DISTRIBUCIÓN DE RELOJ.

Todos los módulos trabajan con un valor de frecuencia de 100Mhz (salida *PLL*, CLKOUT1) a excepción del módulo OSERDES el cual necesita tanto CLK (alta velocidad) como CLKDIV (frecuencia para mantener cada dato).

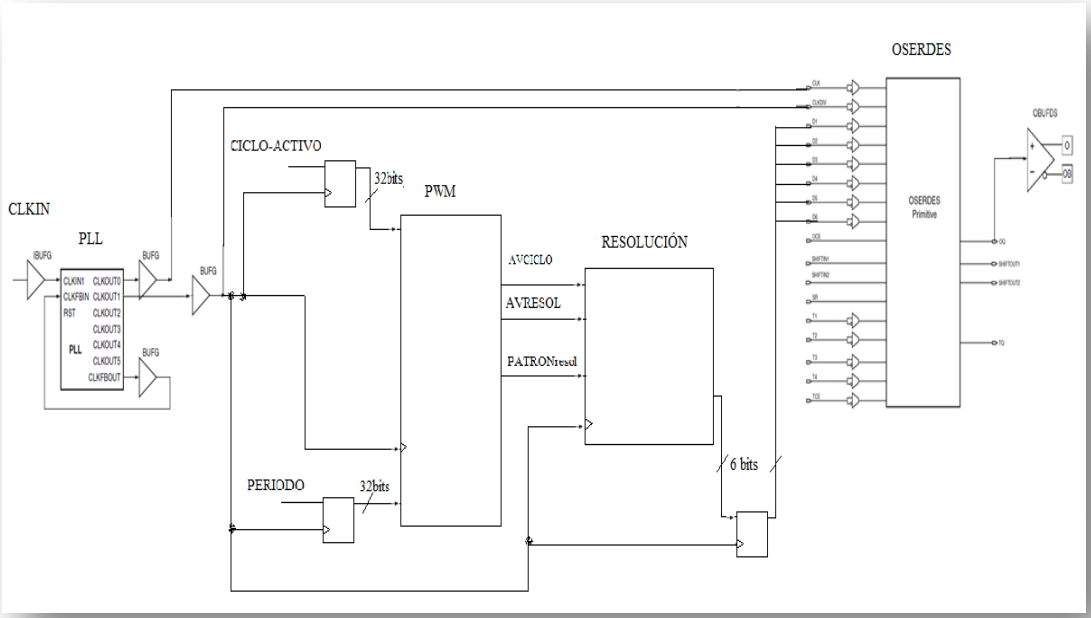



Figura 2.10: Conexión general entre bloques (distribución de reloj).

Como vemos en la figura 2.10 se ha hecho uso de tres registros, dos para poder registrar las señales de entrada del módulo *PWM* correspondientes al periodo y al ciclo de trabajo y uno situado en entre los bloques RESOLUCIÓN y OSERDE con el fin de registrar la señal que se enviará al OSERDES. Con estos registros establecemos los tiempos de *setup* y los tiempos de *hold* estableciendo el principio y el fin de nuestra lógica combinacional. Esto también ha sido útil para generar un análisis en el tiempo.

La fase final se basa en enviar los seis datos con el módulo Resolución al módulo OSERDES con el cual generaremos pulsos con alto rendimiento a una frecuencia máxima de 600Mhz obteniendo así pulsos con estrechos anchos del orden de ns(nanosegundos). Para ello hemos utilizado el *buffer* de salida OBUFDS trabajando así en modo diferencial.

## 2.6. MEMORIA.

Aparte del objetivo de almacenar los datos de entrada se ha decidido añadir una memoria *RAM* por las prestaciones que nos brinda tales como la interfaz de alta velocidad. En los dispositivos Virtex-5 tales dispositivos como las memorias ofrecen un gran número de bloques de 36Kb. Cada bloque contiene de 36Kb dos partes controladas de 18Kb. Los 36Kb están estructurados en cascada para permitir una aplicación de memoria más amplia y con menos penalización de tiempo mínimo. Los bloques *RAM* de doble o único puerto, módulos *ROM* pueden ser implementados con facilidad con el generador de *Xilinx CORE*.

### 2.6.1. CARACTERÍSTICAS DE LA MEMORIA.

Como hemos dicho antes la memoria posee doble puerto 36Kb y consistirá en un un área de almacenamiento de 36Kb y dos puertos de acceso totalmente independientes, A y B. Del mismo modo cada bloque de 18Kb de *RAM* de memoria de doble puerto consiste en un área de almacenamiento de 18Kb y dos puertos de acceso completamente independientes, A y B. La estructura es totalmente simétrica, y los puertos son intercambiables.

Cada operación de escritura es sincrónica, cada puerto tiene su propia dirección. Los datos pueden ser escritos por los puertos de ambos y se puede leer en uno o ambos puertos. La lectura y escritura son síncronas y requieren un flanco de reloj.

No hay monitor dedicado para arbitrar el efecto de direcciones idénticas en ambos puertos. Corresponde al usuario dar el tiempo a los dos relojes de manera adecuada.

La siguiente imagen nos muestra la forma física de la memoria de doble puerto.

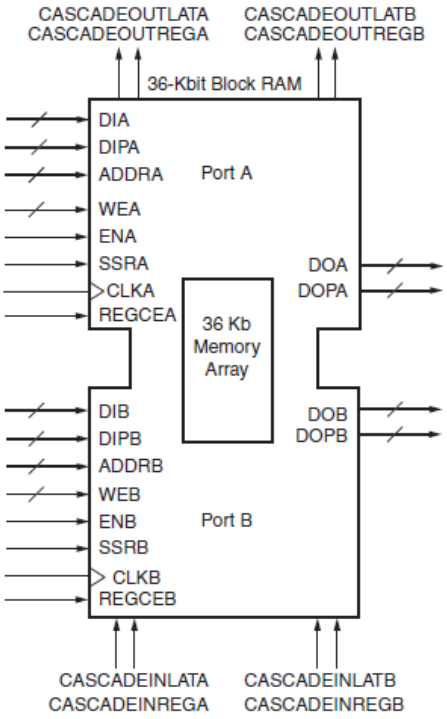



Figura 3.1: Memoria de doble puerto.

## 2.6.2. MÉTODOS DE TRABAJO.

Hay que distinguir dos métodos de trabajo:

### OPERACIÓN DE LECTURA

En el modo *latch*, la operación de lectura utiliza un flanco de reloj. La dirección de lectura se registra por el puerto de lectura y los datos almacenados se cargan al cierre de la salida después del tiempo de acceso de memoria *RAM*. Cabe destacar que cuando se utiliza el registro de la salida, la operación de lectura tendrá un ciclo de latencia adicional.

### OPERACIÓN DE ESCRITURA

Una operación de escritura es una operación simple en la que la dirección está registrada en el puerto de escritura y los datos almacenados en la memoria.

## 2.6.3. OPERACIONES DE MEMORIA.

### MODOS DE ESCRITURA.

El modo *WRITE_FIRST* trabaja de forma que mientras está escribiendo los nuevos datos que entran al mismo tiempo se está obteniendo por la salida los datos previamente almacenados.

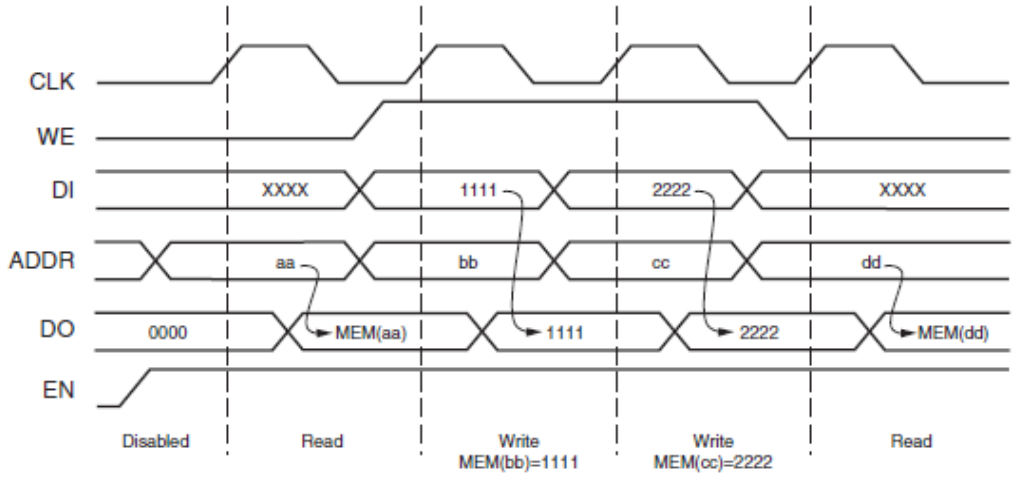



Figura 3.2: Operación de memoria *WRITE-FIRST*.

El modo *READ_FIRST* los datos previamente almacenados en la dirección son los que obtenemos a la salida mientras que los nuevos datos que entran están siendo almacenados en memoria.

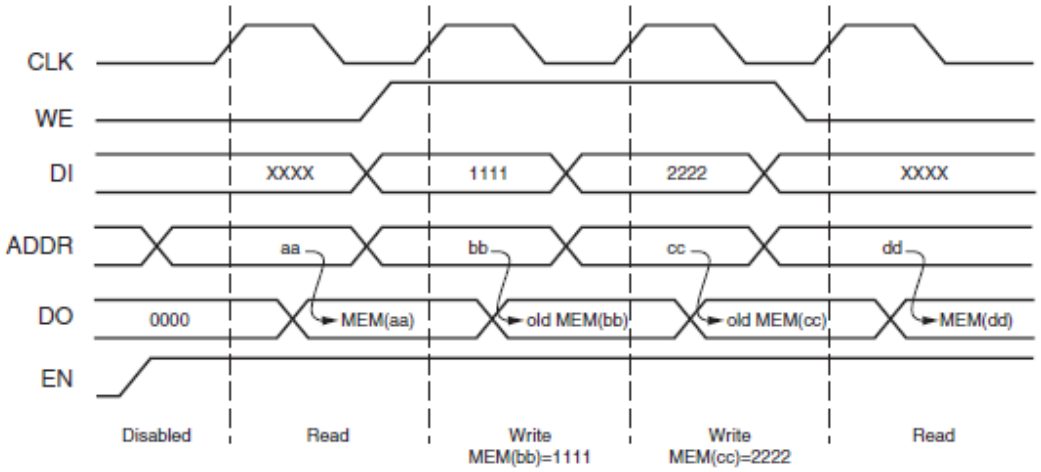



Figura 3.3: Operación de memoria *READ_FIRST*.

El modo *NO_CHANGE* mantiene la salida previamente generada durante una operación de escritura.

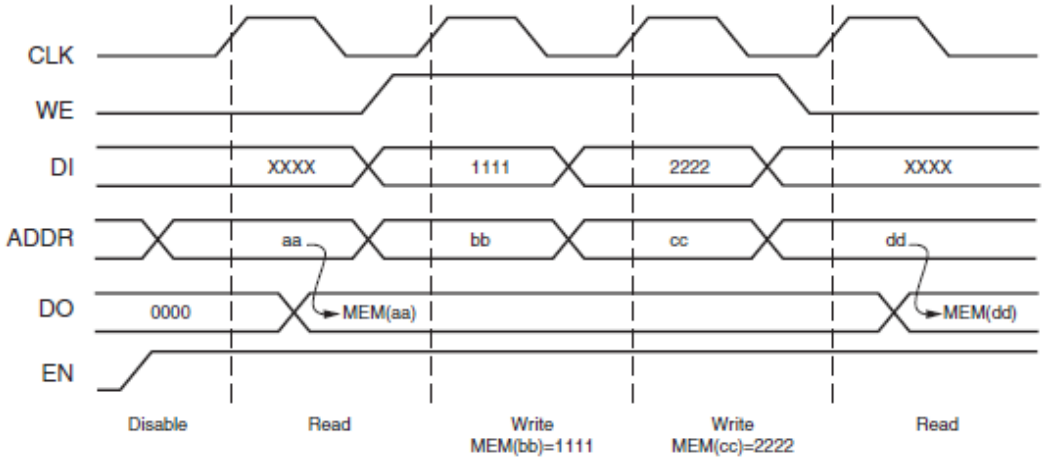



Figura 3.4: Operación de memoria *NO_CHANGE*.

Para lograr el sincronismo de los relojes debemos tener en cuenta lo siguiente:

- No hay limitaciones de tiempo cuando los dos puertos realizan una operación de escritura.
- Cuando un puerto realiza una operación de escritura el otro puerto de debe escribir en el mismo lugar a menos que en ambos puertos se escriban datos idénticos.
- Cuando un puerto realiza una operación de escritura con éxito el otro puerto puede leer los datos desde el mismo lugar si el puerto está en modo *READ_FIRST*.

### 2.6.4. SIMPLE BLOQUE *RAM DUAL-PORT.*

Cada bloque de 18Kb y 36Kb también se pueden configurar en un bloque simple de *RAM* de doble puerto, de este modo la memoria *RAM* de doble puerto tiene un ancho de 36 bits para el bloque de 18Kb y 72 bits para el bloque de 36Kb de *RAM*.

En este modo "simple de doble puerto" ocasionalmente pueden ocurrir simultáneamente operaciones de lectura y escritura y el puerto A puede ser designado como el puerto de lectura y el puerto B como puerto de escritura. La figura 3.5 nos enseña las entradas y salidas del bloque simple *dual-port*.

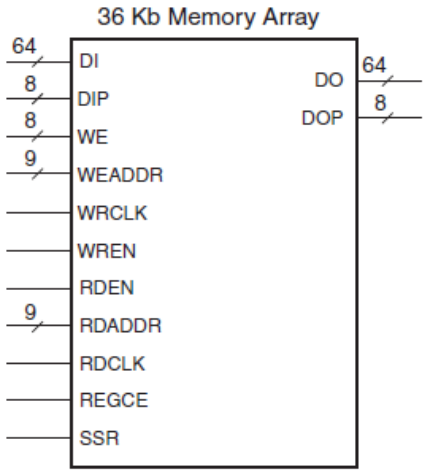



Figura 3.5: Simple bloque *dual- port.*

### 2.6.5. PROGRAMACIÓN DE LA MEMORIA *RAM*.

El programa posee una entrada de reloj (100Mhz) conectada de forma síncrona con el resto de bloques del diseño. La única entrada es ROMDIR, una entrada de direcciones por la cual elegiremos una dirección que corresponda con el periodo que deseamos generar.

Los valores de los periodos estarán almacenados en una *arrai-type* por una constante interna denominada *ROM*.

Entradas de la memoria:

```
CLKIN : in std_logic;
ROMDIR : in std_logic_vector(3 downto 0);
DOP : out std_logic_vector(29 downto 0);
DOA : out std_logic_vector(29 downto 0)
```

#### 2.6.5.1. HERRAMIENTAS DE SOFTWARE.

Por otra parte haremos uso de conversiones de tipo para poder dividir el valor de periodo elegido y así introducir un ciclo activo del 50%.

Conversiones de tipo:

```
M <=conv_integer(ROM(CONV_INTEGER(ROMDIR)));
S<= M/2;
AN <= CONV_STD_LOGIC_VECTOR(S,30);
```

Para ello también necesitamos definir las siguientes señales y realizar un *process* para obtener por las salidas DOP y DOA el periodo y el ciclo activo de la señal.

Descripción de las señales y proceso de memoria:

```
subtype sINT is integer range -2**15 to 2**30-1;
signal S,M: sINT:=0;
signal AN:std_logic_vector(29 downto 0);

begin
process (CLKIN)
        begin
                if CLKIN'event and CLKIN = '1' then

                        DOP<=ROM(CONV_INTEGER(ROMDIR));
                        DOA<=AN;
    end if;
end process;
```

# 3. SIMULACIONES Y RESULTADOS.

Como está especificado en el anexo1 referido al lenguaje *VHDL*, las simulaciones *test bench* que se detallan a continuación forman parte de la primera fase de comprobación. En ellas podremos tener un primer acercamiento, obteniendo así las primeras conclusiones y correcciones de nuestro diseño.

## 3.1. SIMULACIÓN DEL MÓDULO *PWM*.

### CÓDIGO DE SIMULACIÓN

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY gpwm_tb IS
END gpwm_tb;

ARCHITECTURE behavior OF gpwm_tb IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT gpwm
PORT(
CLKDIV : IN  std_logic;
ANCHOP : IN  std_logic_vector(31 downto 0);
PERIODO : IN  std_logic_vector(31 downto 0);
RST : IN  std_logic;
EN : IN  std_logic;
AVciclo : out  STD_LOGIC;
PATRONciclo: out  STD_LOGIC_vector(5 downto 0);
PWM_OUT : OUT  std_logic
 );
END COMPONENT;

--Inputs
signal CLKDIV : std_logic := '0';
signal ANCHOP : std_logic_vector(31 downto 0) := (others => '0');
signal PERIODO : std_logic_vector(31 downto 0) := (others => '0');
signal RST : std_logic := '0';
```

```
signal EN : std_logic := '0';

--Outputs
signal AVciclo : sTD_LOGIC;
signal PATRONciclo:  STD_LOGIC_vector(5 downto 0);
signal PWM_OUT : std_logic;

BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: gpwm PORT MAP (

CLKDIV => CLKDIV,
ANCHOP => ANCHOP,
PERIODO => PERIODO,
RST => RST,
EN => EN,
AVciclo => AVciclo,
PATRONciclo => PATRONciclo,
PWM_OUT => PWM_OUT
 );

CLKDIV <= not CLKDIV after 5 ns;
ANCHOP <=  "00000000000000001000011010100000" after 10 ns;
 --1ms/2
PERIODO <= "00000000000000011000011010100000" after 10 ns;
--1ms
RST <= '0' after 5 ns;
EN <= '1' after 5 ns;
END;
```

## RESULTADO DE SIMULACIÓN.

Como podemos ver en la asignación de estímulos se eligió una señal de de 1ms de periodo con un ancho activo de 1ms/2 para comprobar el funcionamiento del módulo *PWM*. Los cambios se efectúan en la señal de salida pwm_out cada vez que se cambia de estado. En el estado s1 la señal esta activa y en el estado s2 la señal pasa a tener un valor '0'. La señal avciclo cambia de estado igual que la señal de salida pwm_out como era previsto. Avciclo al ser al replica de la señal pwm_out (señal de prueba) es la que se comunicará con las demás etapas avisando cuando le señal esta activa o inactiva. Por otra parte vemos el funcionamiento de la señal de seguridad patronciclo que envía el dato "111111" diciéndonos que si existe periodo de señal. La siguiente imagen de simulación nos muestra los diferentes cambios de señales.

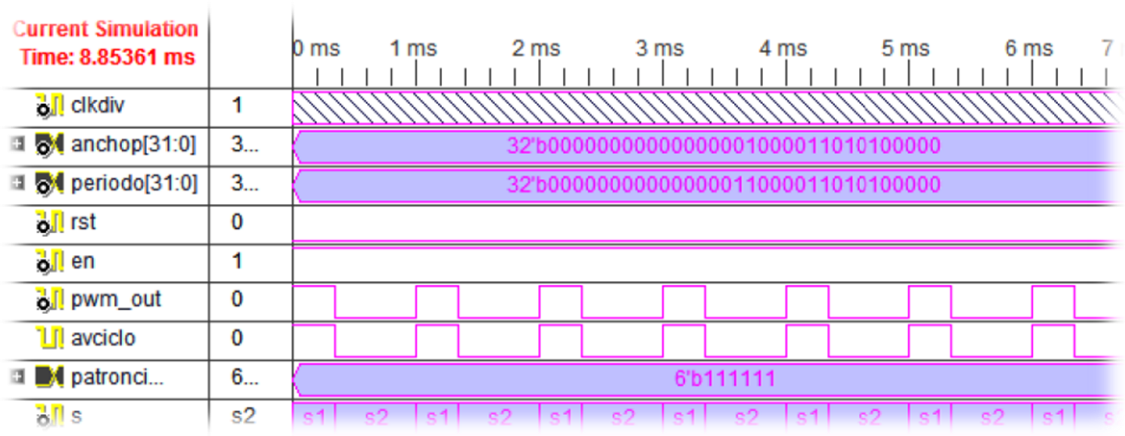



Figura 3.6: Imagen de simulación del modulo *PWM*.

## 3.2. SIMULACIÓN DEL MÓDULO RESOLUCIÓN (*FAST*).

### CODIGO DE SIMULACIÓN

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
 ENTITY fast_tb IS
END fast_tb;
 ARCHITECTURE behavior OF fast_tb IS
 -- Component Declaration for the Unit Under Test (UUT)
COMPONENT fast
PORT(
PATRONciclo : IN  std_logic_vector(5 downto 0);
AVCICLO : IN  std_logic;
AVRESOL : IN  std_logic;
RESOL : IN  std_logic_vector(2 downto 0);
CLKfast : IN  std_logic;
RST : IN  std_logic;
PATRONFINAL : OUT  std_logic_vector(5 downto 0)
);
END COMPONENT;

--Inputs
signal PATRONciclo : std_logic_vector(5 downto 0) := (others => '0');
signal AVCICLO : std_logic := '0';
signal AVRESOL : std_logic := '0';
signal RESOL : std_logic_vector(2 downto 0) := (others => '0');
signal CLKfast : std_logic := '0';
signal RST : std_logic := '0';

--Outputs
signal PATRONFINAL : std_logic_vector(5 downto 0);

BEGIN
 -- Instantiate the Unit Under Test (UUT)
uut: fast PORT MAP (
PATRONciclo => PATRONciclo,
AVCICLO => AVCICLO,
AVRESOL => AVRESOL,
RESOL => RESOL,
CLKfast => CLKfast,
RST => RST,
PATRONFINAL => PATRONFINAL
);


PATRONciclo <= "111111" after 5 ns, "000000" after 175 ns;
```

```
AVCICLO <= '0' after 20 ns, '1' after 30 ns, '0' after 50 ns, '1' after 80 ns, '0' after 120
ns, '1' after 140 ns, '0' after 160 ns;
AVRESOL <= '0' after 20 ns, '0' after 30 ns, '1' after 50 ns, '0' after 80 ns, '1' after 120ns;
RESOL <= "100" after 5 ns;
CLKfast <= not CLKfast after 5 ns;
RST <= '0' after 5 ns;
END;
```

## RESULTADO DE SIMULACIÓN.

El resultado de simulación del módulo resolución nos muestra cómo trabajan las señales avciclo y avresol para poder definir la señal final compuesta de la señal PWM y la señal de resolución. La señal avciclo se activa a nivel alto provoca el cambio de estado de s0 a s1 y avisa de esta forma a la señal de salida patronfinal que puede generar el pulso activo de periodo referido con la señal PWM. Seguidamente cuando la señal avciclo toma un valor inactivo, la señal avresol se activa provoca el cambio de estado de s1 a s6 y también da la orden para que se genere la señal de resolución. El estado s9 ejecuta un tiempo de espera introduciendo ceros hasta que llega el estado s0 el cual esperará a que llegue una nueva señal.

Al final la señal patronciclo envía la orden enviando ceros de desactivar y solo enviar los valores de resolución debido a que avciclo no está activa y avresol sigue estando activa, es decir que no existe señal proveniente del módulo *PWM* pero si existe señal de resolución para generar. A continuación se explica con más detenimiento el proceso de ejecución del módulo resolución. En la figura 3.7 podemos apreciar los cambios de las señales de la etapa Resolución.

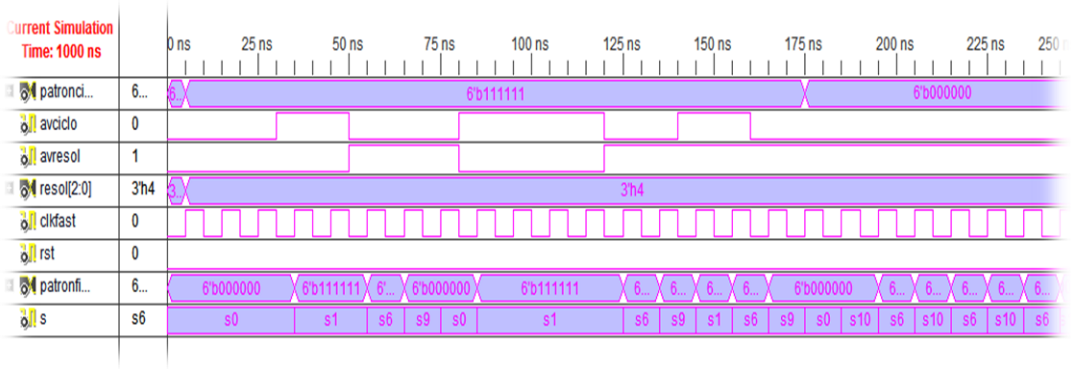



Figura 3.7: Imagen de simulación del modulo resolución.

## 3.3. PROCESO DE EJECUCIÓN DEL MODULO RESOLUCIÓN.

En la gráfica anterior (figura 3.7) de simulación podemos ver como las señales avciclo y avresol funciona perfectamente, cambiando de estados y obteniendo así los valores esperados por la señal de salida patronfinal. Cuando se activa la señales avciclo pasamos al estado s1 lo cual quiere decir que la señal *PWM* ha pasado el estado activo "1". Después de un cierto momento el módulo *PWM* envía la señal avresol para avisar que se ha seleccionado un valor de resolución (en el caso de haber introducido algún valor en el modulo *Fast*) y por lo tanto se genera automáticamente el valor de resolución y se adjunta al valor del ciclo activo de trabajo de la señal comenzada con el aviso de avciclo.

El estado s0 es un estado de reconocimiento de la señal con la cual nos proporciona el paso a los estados s1 el cual es activado con la señal avciclo para poder obtener por la señal de salida patronfinal un valor de seis bits "111111" y enviarla al OSERDES. Sucesivamente al activarse la señal avresol pasamos al estado correspondiente al valor de resolución elegida y generamos la resolución la cual se enviará también al OSERDES.

El estado s1ejecuta el salto a ciertos estados dependiendo del valor de resolución que el programador le ha dado previamente. En nuestra simulación el valor escogido fue "100" el cual está ligado al estado s6. Como se puede ver en la gráfica también tenemos un estado s9 que se utiliza como vía de separación de los patrones que enviamos, así cuando enviemos otro valor se deberá esperar a que suceda este estado el cual envía al OSERDES un valor de de seis bits "000000". También el estado s9 mientras está enviando sus respectivos valores al OSERDES ejecuta la función de control de la señal. El control llevado a cabo se basa en que cuando llega una señal durante el estado s9, se controla si la señal del ciclo de trabajo que llega está activa, es decir si la señal proveniente de avciclo es '1'. En el caso de que la señal avciclo está inactiva el estado s9 lo comprueba y se pasa al estado s0 que también posee la mima función de reconocimiento. Pero en el caso que la señal avciclo está activa durante el estado s3 se pasa automáticamente al estado s1 para generar el valor. De este modo no perdemos tiempos sino que generamos automáticamente los datos que provienen del modo *PWM*.

Al final podemos ver que la señal avciclo está inactiva lo que significa que hemos detenido la generación de pulsos y por lo tanto los estados s9 y s0 verifican esta situación y solo generan los pulsos correspondientes al valor de resolución representados por los estados s6 y su retorno a cero s10.

Cuando se cumplen los siguientes requisitos: avciclo='0' and patronciclo="000000" and avresol='1' el estado s0 nos lleva al estado s10 el cual ejerce la misma función que s1 pero con la diferencia que envía al oserde el valor “000000”. Por lo tanto en este caso trabajaríamos con el estado s10 y el estado de resolución s6 generando así los pequeños pulsos relacionados con la resolución.

La señal de seis bits patronciclo envía un valor de “111111” cuando existe una señal activa, es decir, cuando hemos introducido algún valor correspondiente al periodo de la señal. Por esta razón podemos ver que cuando patronfinal pasa a tener un valor de “000000” dejamos de generar una señal. En esta simulación como se ha especificado hemos hecho uso de la configuración que nos permite trabajar con pulsos de resolución cuando dejamos de generar señales con el módulo *PWM*.

## 3.4. SIMULACIÓN BLOQUE DE MEMORIA.

### CÓDIGO DE SIMULACIÓN.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY memoria_tb IS
END memoria_tb;
ARCHITECTURE behavior OF memoria_tb IS
 -- Component Declaration for the Unit Under Test (UUT)
COMPONENT memoria
PORT(
CLKIN : IN  std_logic;
ROMDIR : IN  std_logic_vector(3 downto 0);
DOP : OUT  std_logic_vector(29 downto 0);
DOA : OUT  std_logic_vector(29 downto 0)
);
END COMPONENT;
--Inputs
signal CLKIN : std_logic := '0';
signal ROMDIR : std_logic_vector(3 downto 0) := (others => '0');
--signal DI : std_logic_vector(2 downto 0) := (others => '0');
--Outputs
signal DOP : std_logic_vector(29 downto 0);
signal DOA : std_logic_vector(29 downto 0);
BEGIN

-- Instantiate the Unit Under Test (UUT)
 uut: memoria PORT MAP (
 CLKIN => CLKIN,
 ROMDIR => ROMDIR,
 DOP => DOP,
DOA => DOA
);

CLKIN<= not CLKIN after 10 ns;
process (CLKIN)
begin
if CLKIN'event and CLKIN = '1' then
ROMDIR<= "0001";
end if;
end process;
END;
```

## RESULTADO DE SIMULACIÓN.

Introducimos un valor “0001” por la entrada romdir eligiendo así el valor de periodo de 30 bits que está almacenado en esa dirección “000000000111101000010010000000” y su respectivo pulso activo de trabajo. La figura 3.8 enseña el resultado de la simulación del bloque de memoria.

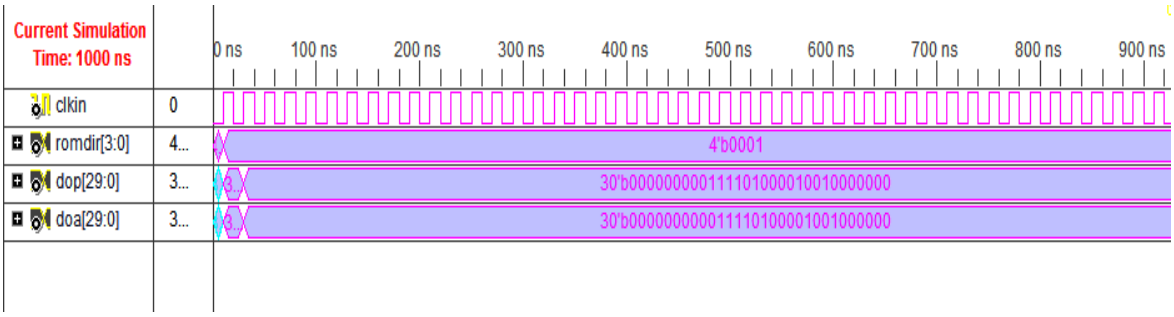



Figura 3.8: Imagen de simulación del bloque de memoria.

# 4. FASES DE PROGRAMACIÓN.

## 4.1. LA SÍNTESIS.

El software *ISE de Xilinx* incluye la Tecnología de síntesis (*XST*), que sintetiza *VHDL*, Verilog o diseños con lenguajes mezclados para crear archivos específicos *netlist* conocidos como archivos *NGD*. Los archivos *NCD* contienen los datos de diseño lógico y limitaciones.

Durante la síntesis de *HDL*, *XST* analiza el código *HDL* y los intentos de deducir bloques específicos de diseño o macros (como *Muxes*, *RAMs*) para crear una implementación eficiente. Para reducir la cantidad de macros infiere, *XST* realiza una comprobación de recursos compartidos. Esto por lo general conduce a una reducción de la zona, así como un aumento en la frecuencia de reloj.

El reconocimiento de "maquinas de estado" (*FSM*), es también parte de la etapa de síntesis de *HDL*. *XST* reconoce *FSM* independiente del estilo de modelización. Para crear la aplicación más eficiente, *XST* utiliza el objetivo de optimización de destino.

Podemos controlar el paso de las limitaciones de la síntesis de *HDL*. Las restricciones pueden ser introducidas utilizando cualquiera de los siguientes métodos:

- Archivo *HDL* de origen. Podemos introducir los atributos *VHDL*.

- *XCF*. Podemos introducir los parámetros globales y las restricciones a nivel de módulo en las limitaciones del archivo *XCF*.

En el panel de procesos que se muestra en la figura 3.9 podemos ver la herramienta *synthesize XST*.

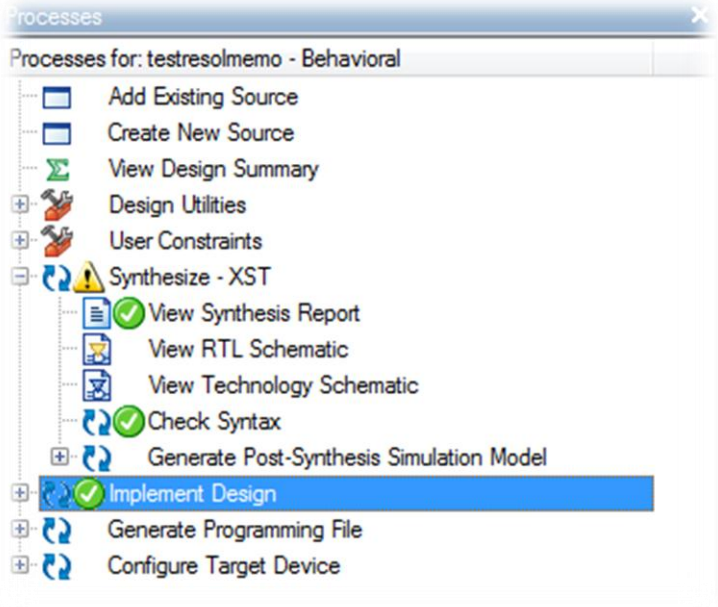



Figura 3.9: Panel de procesos de software *ISE de Xilinx*.

Después de la síntesis, se ejecuta la aplicación de diseño, que comprende los siguientes pasos:

1. *Translate*: combina los *netlist* de entrada y las restricciones en un archivo de diseño de Xilinx.
2. *Map*: ajusta el diseño en los recursos disponibles del dispositivo de destino.
3. *Place and route*: ejecuta un rutado y posicionamiento del diseño de acuerdo con las restricciones temporales.
4. *Programming file generation*: crea un archivo de flujo de bits que pueden ser descargados en el dispositivo.

## 4.2. PROCESOS Y ARCHIVOS.

### 4.2.1. *TRANSLATE*.

El proceso *translate* combina todas los *netlist* de entrada y las restricciones de diseño y entrega una "base de datos nativa genérica" (*NGD*), el cual describe el diseño lógico reducido a primitivas de Xilinx. La figura 3.10 detalla los archivos.

| Traducir Proceso | |
|---|---|
| **Herramienta de línea de comandos** | NGDBuild |
| **Archivos de entrada** | EDIF, SEDIF, EDN, EDF, la UCF NGC, NCF, URF, BMM NMC, |
| **Los archivos de salida** | BLD (informe), NGD |
| **Proceso de propiedades** | Traducir Propiedades |

Figura 3.10: Archivos del proceso *translate*.

### 4.2.2. *MAP*.

Los procesos *Map* mapean la lógica definida por un archivo *NGD* en las *FPGAs* como *CLBs* e *IOBs*. El diseño de salida es una descripción del circuito original de archivo (*NCD*) que representa físicamente el diseño asignado a los componentes de la *FPGA* de Xilinx. La figura 4.1 detalla los archivos.

| Mapa de Procesos | |
|---|---|
| **Herramientas de línea de comandos** | MAPA |
| **Archivos de entrada** | NGD, NMC, de las ENT, NGM<br><br>**Nota** Los archivos de las ENT y NGM son de guía. |
| **Los archivos de salida** | ENT, PCF, NGM, MRP (informe), GRF |
| **Proceso de Propiedades** | Mapa de Propiedades |
| **Herramientas disponibles después de ejecutar el proceso** | Floorplanner, Editor de FPGA, Analizador de Tiempo |

Figura 4.1: Archivos del proceso *map*.

### 4.2.3. *PLACE AND ROUTE.*

El proceso *Place and Route* toma un archivo mapeado *NCD*, ejecuta un posicionamiento- rutado y produce un archivo *NCD*, el cual es usado como entrada para la generación de flujo de bits. La figura 4.2 detalla los archivos.

| Lugar y proceso de Ruta | |
|---|---|
| **Herramientas de línea de comandos** | PAR |
| **Archivos de entrada** | ENT, PCF<br><br>**Nota** Además de los archivos de las ENT del MAP, RAP también se acepta un archivo de enfermedades no transmisibles para orientar. |
| **Los archivos de salida** | ENT, PAR (informe), PAD, CSV, TXT, GRF, DLY |
| **Proceso de Propiedades** | Lugar y Propiedades de la ruta |
| **Herramientas disponibles después de ejecutar el proceso** | Floorplanner, Editor de FPGA, analizador de sincronización, XPower More Info |

Figura 4.2: Archivos del proceso *place and route*.

### 4.2.4. *GENERATE PROGRAMMING FILE.*

El proceso *Generate Programming File* produce un flujo de bits para la configuración del dispositivo. Después que el diseño está completamente rutado configuramos el dispositivo apara que pueda ejecutar la función deseada. La figura 4.3 detalla los archivos.

| Generar procesos de programación Archivo | |
|---|---|
| **Herramientas de línea de comandos** | BitGen |
| **Archivos de entrada** | ENT, PCF, NKY |
| **Los archivos de salida** | BGN, BIN, BIT, República Democrática del Congo, LL ISC, MSD, MSK, NKY, ISC, RBA, RBB, RBD, RBT |
| **Proceso de Propiedades** | Opciones generales , las opciones de configuración , opciones de inicio , Opciones de repaso Opciones de cifrado |
| **Herramientas disponibles después de ejecutar el proceso** | Impact More Info |

Figura 4.3: Archivos del proceso *generating programming file*.

# 5. ANÁLISIS TEMPORAL (*POST PLACE AND ROUTE STATIC TIME*) *Y SIMULACIÓN POST PLACE AND ROUTE.*

## 5.1. ANÁLISIS TEMPORAL.

Después del *placing and routing* generamos un *Post Place and Route Static Time*, el cual es un proceso que incorpora información de los tiempos de retardo con el objetivo de proporcionar un resumen de tiempo completo del diseño.

Con esta herramienta podemos personalizar el contenido del informe para determinar si el diseño ha cumplido los requisitos de tiempo. Después que el proceso *Place and Route* ha cumplido con todas las limitaciones de tiempo creamos los datos de configuración.

En el caso de que se identifiquen problemas en el informe de tiempo, el software nos brinda la posibilidad de solucionarlos aumentando el nivel de esfuerzo, con re-enrutamiento o usando un *multi-pass place and route*.

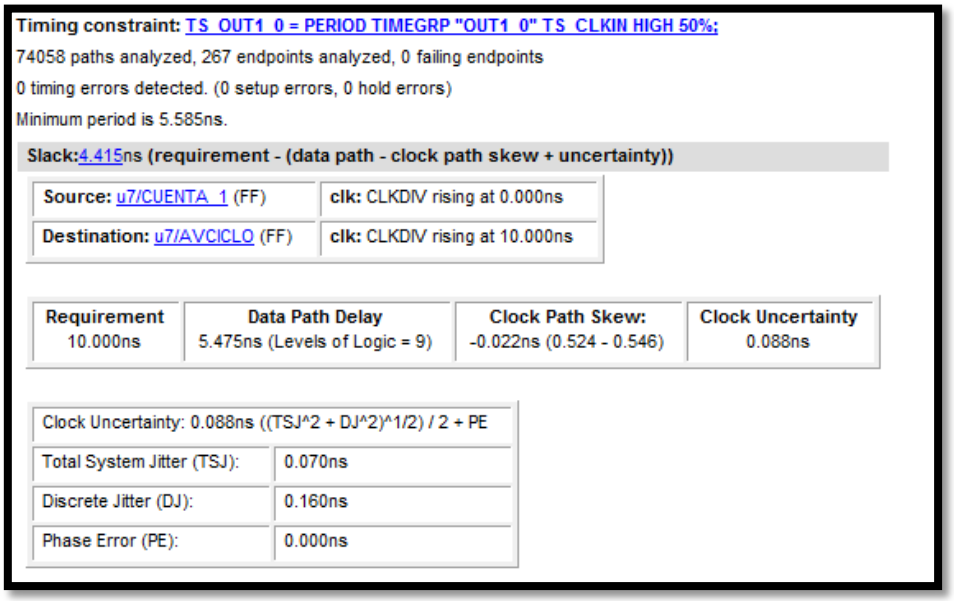
Para realizar el análisis de tiempo asignamos las restricciones en el archivo (.*UCF*). El archivo *UCF* es un archivo *ASCII* que tiene limitaciones de tiempo y lugar. De forma predeterminada cada proyecto *ISE* tiene un archivo *UCF* con el mismo nombre que el nivel de *netlist* superior.

Configuración del archivo *UCF*:

```
NET "OUT0" TNM_NET = OUT0;
TIMESPEC TS_OUT0 = PERIOD "OUT0" 600 MHz HIGH 50%;
NET "OUT1" TNM_NET = OUT1;
TIMESPEC TS_OUT1 = PERIOD "OUT1" 100 MHz HIGH 50%;
NET "CLKIN" TNM_NET = CLKIN;
TIMESPEC TS_CLKIN = PERIOD "CLKIN" 100 MHz HIGH 50%;
```

A continuación realizamos el análisis de tiempo con la herramienta (*Post Place and Route Static Time*) que nos brinda el programa *ISE de Xilinx*.

Como podemos ver en las figuras 4.4 y 4.5, se detallan los relojes, los tiempos mínimos requeridos y los tiempos de los *paths* concluyendo con un análisis del *jitter*.



**Timing constraint:** TS_OUT1_0 = PERIOD TIMEGRP "OUT1_0" TS_CLKIN HIGH 50%;

74058 paths analyzed, 267 endpoints analyzed, 0 failing endpoints

0 timing errors detected. (0 setup errors, 0 hold errors)

Minimum period is 5.585ns.

**Slack:**4.415ns **(requirement - (data path - clock path skew + uncertainty))**

| | |
|---|---|
| **Source:** u7/CUENTA_1 (FF) | **clk:** CLKDIV rising at 0.000ns |
| **Destination:** u7/AVCICLO (FF) | **clk:** CLKDIV rising at 10.000ns |

| Requirement | Data Path Delay | Clock Path Skew: | Clock Uncertainty |
|---|---|---|---|
| 10.000ns | 5.475ns (Levels of Logic = 9) | -0.022ns (0.524 - 0.546) | 0.088ns |

| Clock Uncertainty: 0.088ns ((TSJ^2 + DJ^2)^1/2) / 2 + PE | |
|---|---|
| Total System Jitter (TSJ): | 0.070ns |
| Discrete Jitter (DJ): | 0.160ns |
| Phase Error (PE): | 0.000ns |

Figura 4.4: Análisis temporal.

Maximum Data Path: u7/CUENTA_1 to u7/AVCICLO

| Delay type | Delay(ns) | Logical Resource |
|---|---|---|
| Tcko | 0.375 | u7/CUENTA_1 |
| net (fanout=1) | 0.753 | u7/CUENTA<1> |
| Topcyb | 0.415 | u7/CUENTA<1>_rt<br>u7/Madd_S_add0001_cy<3> |
| net (fanout=1) | 0.000 | u7/Madd_S_add0001_cy<3> |
| Tbyp | 0.091 | u7/Madd_S_add0001_cy<7> |
| net (fanout=1) | 0.000 | u7/Madd_S_add0001_cy<7> |
| Tbyp | 0.091 | u7/Madd_S_add0001_cy<11> |
| net (fanout=1) | 0.000 | u7/Madd_S_add0001_cy<11> |
| Tbyp | 0.091 | u7/Madd_S_add0001_cy<15> |
| net (fanout=1) | 0.000 | u7/Madd_S_add0001_cy<15> |
| Tbyp | 0.091 | u7/Madd_S_add0001_cy<19> |
| net (fanout=1) | 0.000 | u7/Madd_S_add0001_cy<19> |
| Tcinb | 0.295 | u7/Madd_S_add0001_cy<23> |
| net (fanout=5) | 1.095 | u7/S_add0001<21> |
| Topcyd | 0.336 | u7/Mcompar_S_cmp_eq0001_lut<7>1<br>u7/Mcompar_S_cmp_eq0001_cy<7> |
| net (fanout=1) | 0.000 | u7/Mcompar_S_cmp_eq0001_cy<7> |
| Tcinc | 0.306 | u7/Mcompar_S_cmp_eq0001_cy<10> |
| net (fanout=6) | 1.380 | u7/S_cmp_eq0001 |

Figura 4.5: Análisis temporal (señales de retardo).

En la siguiente imagen podemos ver el resultado total de todos los retardos de los recursos de lógica utilizados determinando así un valor final de t=5.475ns.

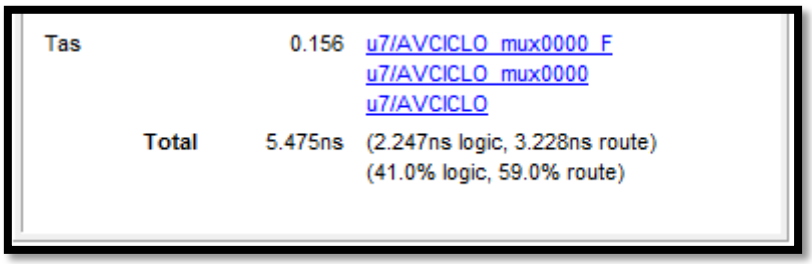



Figura 4.6: Resultado total de tiempo.

## 5.2. SIMULACIÓN *POST-PLACE AND ROUTE*.

Después de realizar el proceso *Place and Route* y la verificación temporal generamos una simulación *Post-Place and Route*. En este proceso se convierte los resultados del proceso *Place and Route* en un modelo de simulación y genera un archivo *SDF (Estándar Delay Format*). El archivo *SDF* contiene la información verdadera de los retardos de tiempos del diseño. Con el archivo del modelo de simulación y el archivo *SDF* podemos utilizarlos para verificar la funcionalidad y los tiempos del diseño.

### 5.2.1. CONFIGURACIÓN DEL MODULO TOP.

#### 5.2.1.1. SIMULACIÓN DE LOS PULSOS *PWM*.

Para realizar la simulación asignamos los siguientes estímulos:

```
ROMDIR<= "1010" after 10 ns;
RESOL<= "000" after 10 ns;
CLKIN <= not CLKIN after 10 ns;
OCE <= '1' after 5 ns;
SR <= '0' after 5 ns;
RSTfast <= '0' after 5 ns;
PWM_RST <= '0' after 5 ns;
PWM_LOAD <= '1' after 5 ns;
RSTPLL <= '0' after 5 ns;
```

Elegimos la dirección de memoria “1010” que corresponde con un valor de señal de periodo 20ns y un valor de resolución “000”. Activamos OCE para abilitar los relojes y PWM_LOAD para activar la generación de pulsos. Por otra parte colocamos a nivel bajo todos los RSTs de nuestro diseño: RSTfast, PWM_RST,  RSTPLL y SR.

Como podemos apreciar en la figura 4.7, se ha generado una señal de periodo de 20ns pero con la ausencia de valores de resolución. En la simulación se ha representado la salida *PWM* y las salidas diferenciales O y OB por que corresponden con las salidas del diseño.

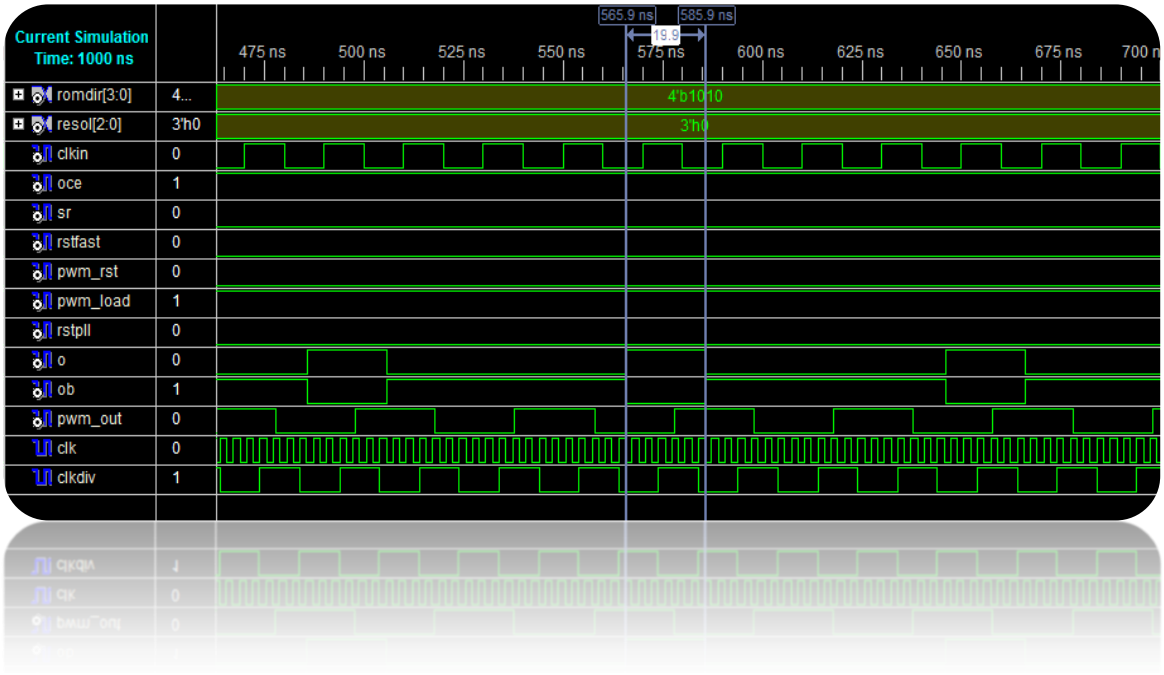



Figura 4.7: Simulación de los pulsos *PWM*.

### 5.2.1.2. SIMULACIÓN DE LOS PULSOS DE RESOLUCIÓN.

En la siguiente simulación agregamos un valor de resolución y realizamos una medida con los cursores de medida de *ISE*.

Los estímulos introducidos son los siguientes:

*ROMDIR<= "1010" after 10 ns;*
*RESOL<= "010" after 10 ns;*
*CLKIN <= not CLKIN after 10 ns;*
*OCE <= '1' after 5 ns;*
*SR <= '0' after 5 ns;*
*RSTfast <= '0' after 5 ns;*
*PWM_RST <= '0' after 5 ns;*
*PWM_LOAD <= '1' after 5 ns;*
*RSTPLL <= '0' after 5 ns;*

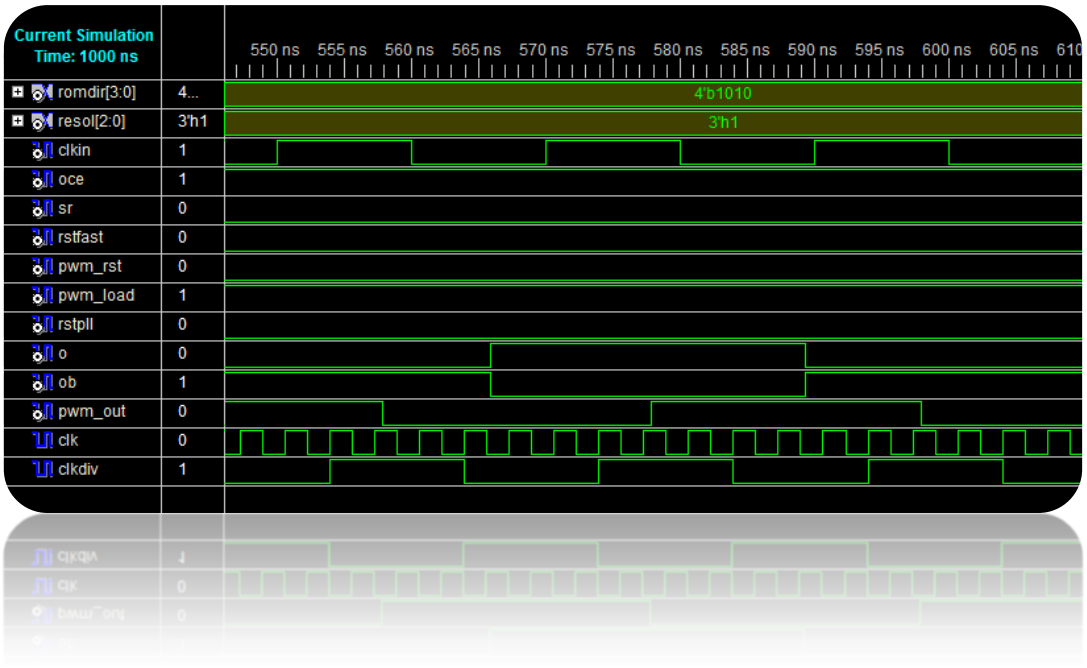



Figura 4.8: Simulación de los pulsos de resolución.

En la figura4.9 podemos ver que se seleccionó un valor de señal de dirección de memoria “1010” que equivale a una señal con periodo de 20ns. El valor de resolución elegido para esta simulación “010”correspondiente a 3,4ns, el cual lo medimos con el reloj de alta velocidad CLK. Al juntar las dos señales generamos una señal de 23.4ns de periodo.

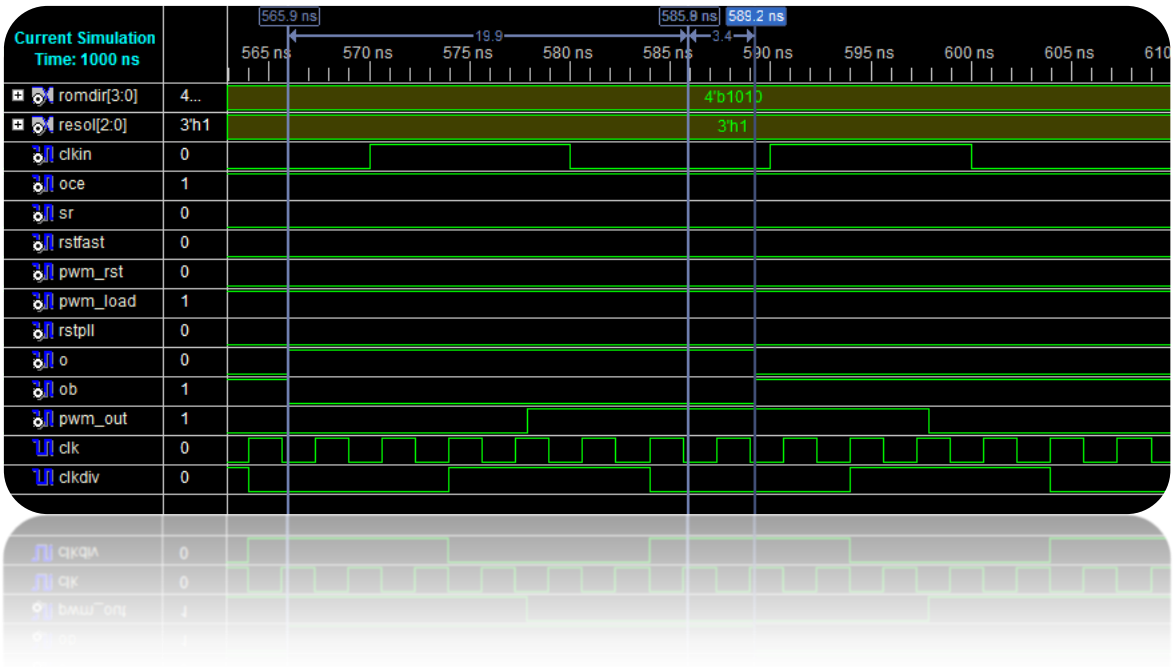



Figura 4.9: Medida de los pulsos *de* resolución.

# 6. IMPLEMENTACIÓN EN EL DISPOSITIVO *FPGA*.

## 6.1. INTRODUCCIÓN

En este proyecto se utilizó la familia del fabricante Xilinx que corresponde a la tecnología XC5VLX110T de VIRTEX-5. La figura 5.1 nos muestra una imagen del dispositivo *FPGA XC5VLX110T*.

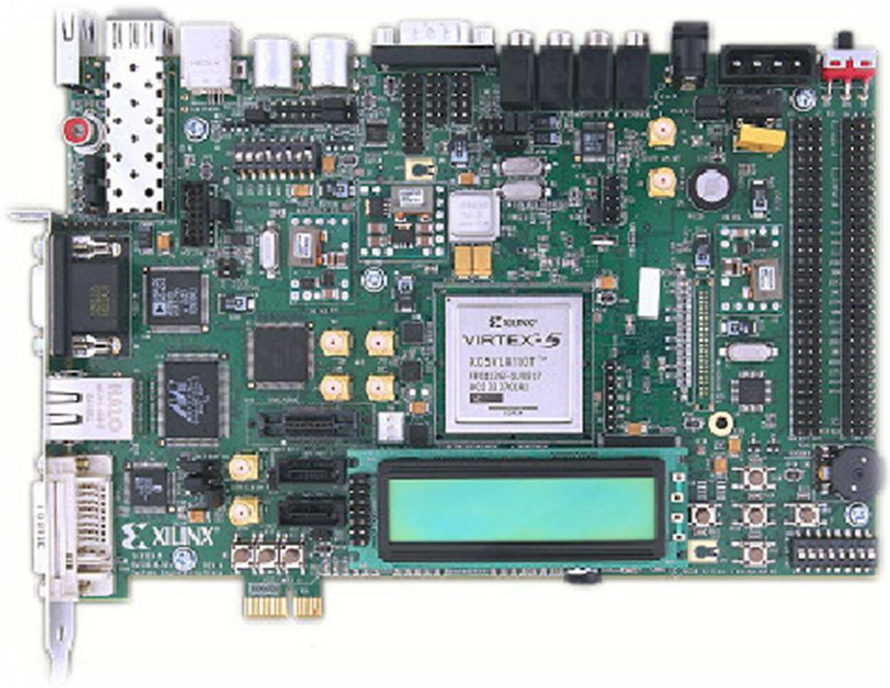



Figura 5.1: Dispositivo *FPGA*.

## 6.2. CARACTERÍSTICAS DE LA FAMILIA XUPV505-LX110T.

El XUPV505-LX110T es una evaluación de propósito general unificada para la enseñanza y la investigación rico en características y que constituye una plataforma de desarrollo con memoria a bordo e interfaces estándar de la industria de la conectividad.

Disciplinas relacionadas con el uso del dispositivo *FPGA*:

- Diseño Digital.
- Los sistemas empotrados.
- Procesamiento digital de las señales.
- Arquitectura de Ordenadores.
- Sistema Operativo.
- La creación de redes.
- Procesamiento de video e imágenes.
- Transceptores en serie de E/S de alta velocidad.

## 6.3. MEDIDAS Y RESULTADOS.

Creamos un programa superior (TOP) el cual contendrá todos los componentes del diseño y el archivo *.UCF* con las limitaciones de tiempo y la configuración de los pines. El módulo TOP es el programa que está preparado para generar la señal que deseamos.

El último paso consiste en programar nuestro diseño en el dispositivo *FPGA* y para ello haremos uso de la herramienta *Generating Programming File*. El proceso *Generate Programming File* genera el archivo *.BIT* con el cual podremos programar la *FPGA* con la herramienta *Manage Configuration Proyect IMPACT*. La herramienta *IMPACT* se muestra en la venta de procesos de la figura 5.2.

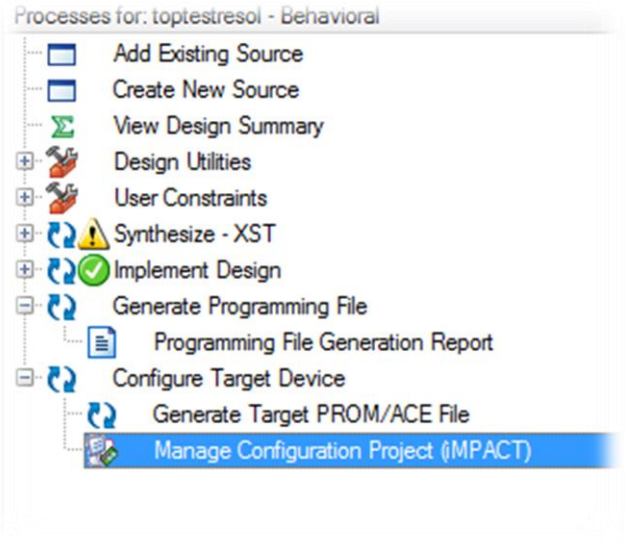



Figura 5.2: Ventana de procesos de *ISE (Generate Programming File)*.

### 6.3.1. CONFIGURACIÓN DE LOS PINES.

En primer lugar configuraremos los pines del dispositivo *FPGA* para indicar por donde deseamos obtener la señal de salida y realizar las medidas con el osciloscopio.

El diseño consta con una salida tipo diferencial y una señal de prueba solamente para la salida PWM por lo tanto configuraremos las salidas con los conectores E/S diferenciales que posee la *FPGA*.

### CONECTORES DIFERENCIALES DE E/S.

La *FPGA* consta de un conector J4 que contienes 16 pares de conectores para señales diferenciales (*LVDS*). EL *VCCIO* de estas señales puede ser establecido a 2.5V y 3.3V con la configuración de puente del conector J20. Además, también podemos utilizar los conectores de expansión *Single-Ended* J6 que contiene 32 conexiones para datos. Todas las señales *Single-Ended* pueden ser configuradas también con valores de 2.5V o 3.3V por el puente J20. Los diferentes pines se pueden ver en la tabla de la figura 5.3.

| J4 Differential Pin Pair | | Schematic Net Name | | FPGA Pin | |
|---|---|---|---|---|---|
| Pos | Neg | Pos | Neg | Pos | Neg |
| 4 | 2 | HDR2_4 | HDR2_2 | L34 | K34 |
| 8 | 6 | HDR2_8 | HDR2_6 | K33 | K32 |
| 12 | 10 | HDR2_12 | HDR2_10 | P32 | N32 |
| 16 | 14 | HDR2_16 | HDR2_14 | T33 | R34 |
| 20 | 18 | HDR2_20 | HDR2_18 | R33 | R32 |
| 24 | 22 | HDR2_24 | HDR2_22 | U33 | T34 |
| 28 | 26 | HDR2_28 | HDR2_26 | U32 | U31 |
| 32 | 30 | HDR2_32 | HDR2_30 | V32 | V33 |
| 36 | 34 | HDR2_36 | HDR2_34 | W34 | V34 |
| 40 | 38 | HDR2_40 | HDR2_38 | Y33 | AA33 |
| 44 | 42 | HDR2_44 | HDR2_42 | AF34 | AE34 |
| 48 | 46 | HDR2_48 | HDR2_46 | AF33 | AE33 |
| 52 | 50 | HDR2_52 | HDR2_50 | AC34 | AD34 |
| 56 | 54 | HDR2_56 | HDR2_54 | AC32 | AB32 |
| 60 | 58 | HDR2_60 | HDR2_58 | AC33 | AB33 |
| 64 | 62 | HDR2_64 | HDR2_62 | AN32 | AP32 |

Figura 5.3: Tabla de conectores diferenciales.

La placa contiene un oscilador de cristal de 100Mhz (X1) alimentado a una fuente de 3.3V y un generador programable de reloj (U8) con el cual podremos generar una variedad de relojes. El generador de reloj programable proporciona los siguientes valores predeterminados de fábrica:

- 25 MHz *to the Ethernet PHY (U16).*
- 14 MHz *to the audio codec (U22).*
- 27 MHz *to the USB Controller (U23).*
- 33 *MHz to the System ACE CF (U2).*
- 33 MHz, 27 MHz, *and a differential 200Mhz clock.*

La siguiente tabla clasifica las conexiones de los osciladores.

| Reference Designator | Clock Name | FPGA Pin | Description |
|---|---|---|---|
| X1 | USER_CLK | AH15 | 100 MHz single-ended |
| U8 | CLK_33MHZ_FPGA | AH17 | 33 MHz single-ended |
| U8 | CLK_27MHZ_FPGA | AG18 | 27 MHz single-ended |
| U8 | CLK_FPGA_P | L19 | 200 MHz differential pair (pos) |
| U8 | CLK_FPGA_N | K19 | 200 MHz differential pair (neg) |

Figura 5.4: Tabla de frecuencias de oscilación.

Seleccionamos los pines del conector J4 y configuramos el archivo *UCF* dejando así preparadas las salidas *LVDS* para realizar las medias necesarias. Se seleccionaron los pines L34 Y K34 para las salidas diferenciales, el pin H32 tipo *single –ended* para realizar la medida en su caso de la salida *PWM* y el pin AH15 para el reloj de 100MHZ.

### CONFIGURACIÓN DEL ARCHIVO *UCF*.

*NET CLKIN LOC = AH15;*

*NET O LOC = L34;*

*NET OB LOC = K34;*

*NET PWM_OUT LOC = H32;*

*NET "CLKIN" TNM_NET = CLKIN;*

*TIMESPEC TS_CLK = PERIOD "CLKIN" 10 ns HIGH 50%.*

## 6.3.2. EQUIPO DE TRABAJO.

Equipo necesario para la implementación y medida del proyecto:

- Ordenador.
- *Sotware ISE de Xilinx*.
- Dispositivo *FPGA* de la familia VIRTEX-5, XUPV5-LX110T.
- Sondas.
- Osciloscopio digital Tektronix TDS7154B.

En la siguiente figura 5.5 se puede ver las conexiones realizadas para la medida de las señales de salida. Se utilizaron dos sondas que fueron aplicadas por los pines L34 y K34 del conector j4para realizar la medida de las señales diferenciales.

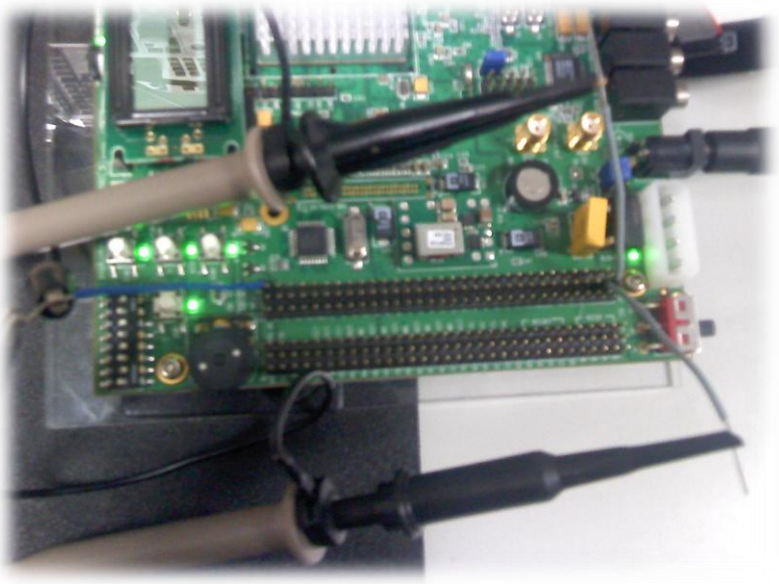


Figura 5.5: Conexiones realizadas para la medida de señal.

La imagen 5.6 nos muestra el osciloscopio digital el cual se encuentra representando una señal deferencial.

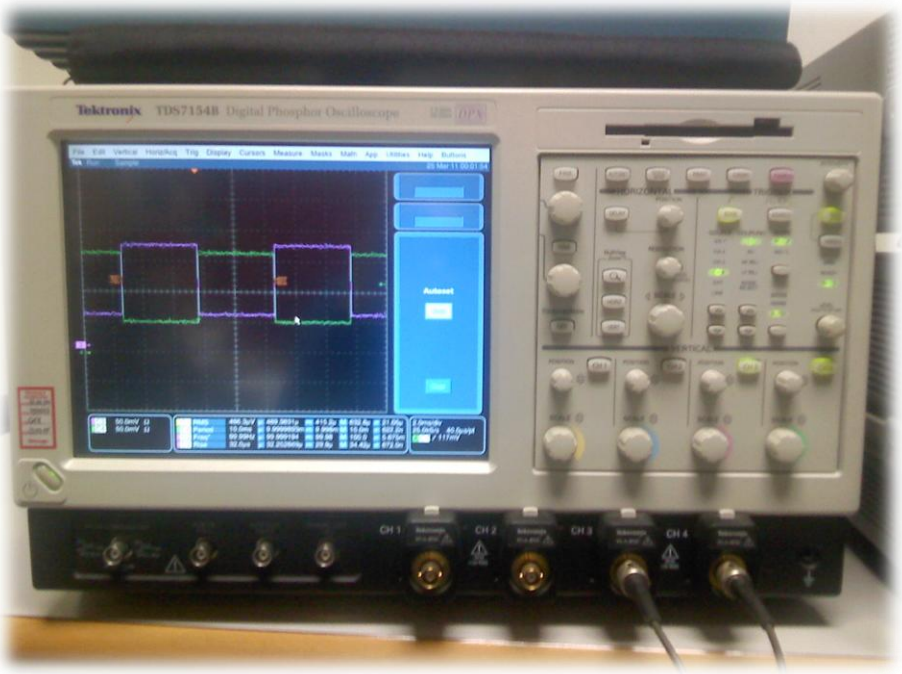



Figura 5.6: Osciloscopio digital Tektronix.

### 6.3.3. MEDIDA DE LA GENERACIÓN DE SEÑALES SIN RESOLUCIÓN.

Para realizar las medidas programaremos mediante el TOP la señal que deseamos obtener por los pines diferenciales. En la primera etapa de medida, verificaremos la generación de pulsos que nos proporciona la etapa *PWM*. Después pasaremos a medir los pulsos de resoluciones y también una señal completa formada por señal *PWM* y señal de  resolución.

En la figura 5.7 podemos apreciar la medida de la señal diferencial para un periodo de 10ms. Se puede ver que hemos utilizado los cursores que nos brinda el osciloscopio digital para realizar dicha medida.

Medida de la señal diferencial de T=10ms.

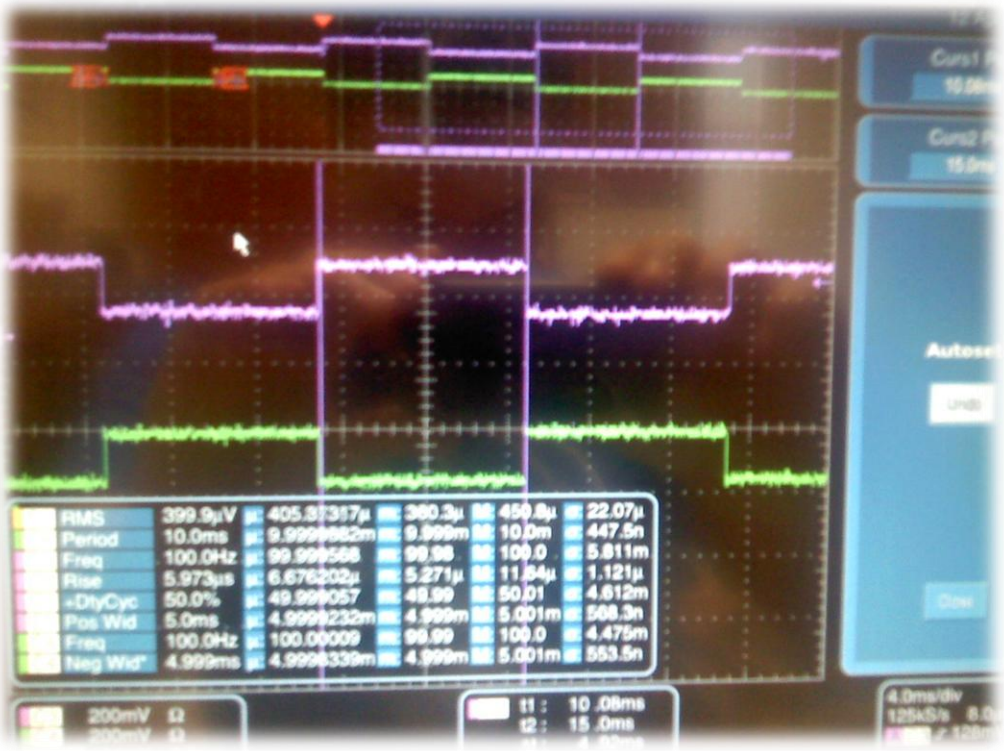



Figura 5.7: Medida de la señal diferencial.

En la primera medida verificamos que la señal de salida que posee una valor T=10ms y f=100Hz. La figura 5.8 nos muestra con detalle el periodo y la frecuencia de la señal extraídos de canal c3 del osciloscopio digital.

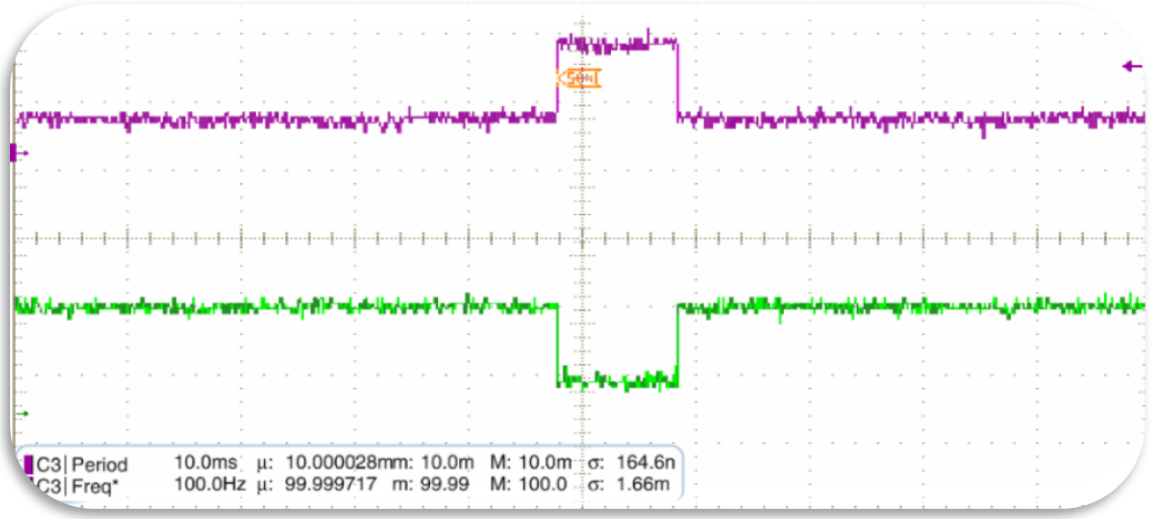



Figura 5.8: Señal diferencial de T=10ms.

### 6.3.4. MEDIDA DE LA RESOLUCIÓN.

La segunda etapa de medida se encargará de verificar los pulsos de resolución añadidos a una señal base generada previamente por el módulo *PWM*. Comprobaremos pulsos de resolución con valores como:

- T1=1.7ns.
- T2=8.5ns.
- T3=10.2ns.

De esta forma podremos ver el mínimo y el máximo valor de resolución. Como señal base generamos una señal de 10ms a la cual le añadimos un valor de resolución de “001” que corresponde con 1,7ns.

La señal extraída por el canal c3 del osciloscopio digital se puede apreciar en la figura 5.9. Con los cursores de medida comprobamos el ancho del pulso que es de 2.2ns cerca de los 1.7ns. Esto se debe a que la señal era muy inestable para medir con precisión debido a que son pulsos muy estrechos del orden de ns(nanosegundos).

Medida del pulso de resolución de T1=1.7ns.

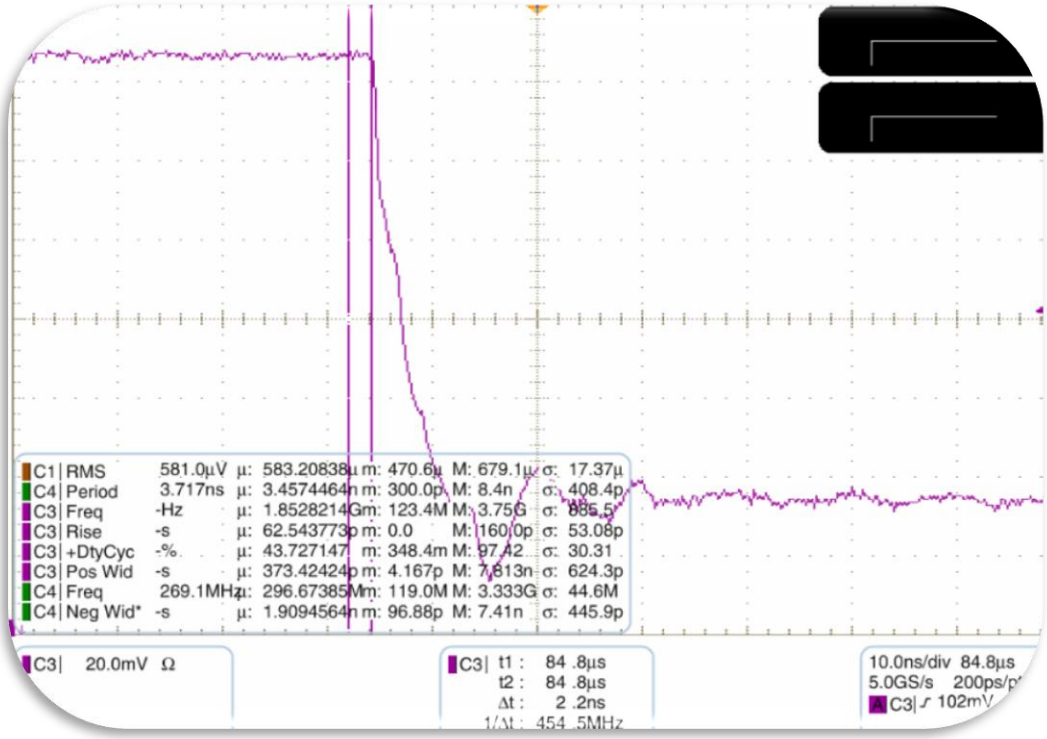



Figura 5.9: Medida del pulso de resolución de T=1.7ns.

La medida de resolución de T=8.5ns se muestra en la figura 5.10. Aquí también observamos la diferencia de tiempos de los cursores. Esto se debe a que el instrumento de medida no posee la suficiente resolución para medir con precisión sin que la señal se distorsione. A pesar de la dificultad para medir pulsos de tan pequeña resolución se realizaron medidas de los periodos de valor mínimo y máximo de resolución.

En esta medida la diferencia era de un ancho 9.2ns cercanos al periodo de valor 8.7ns generado.

Medida del pulso de resolución de T2=8.5ns.

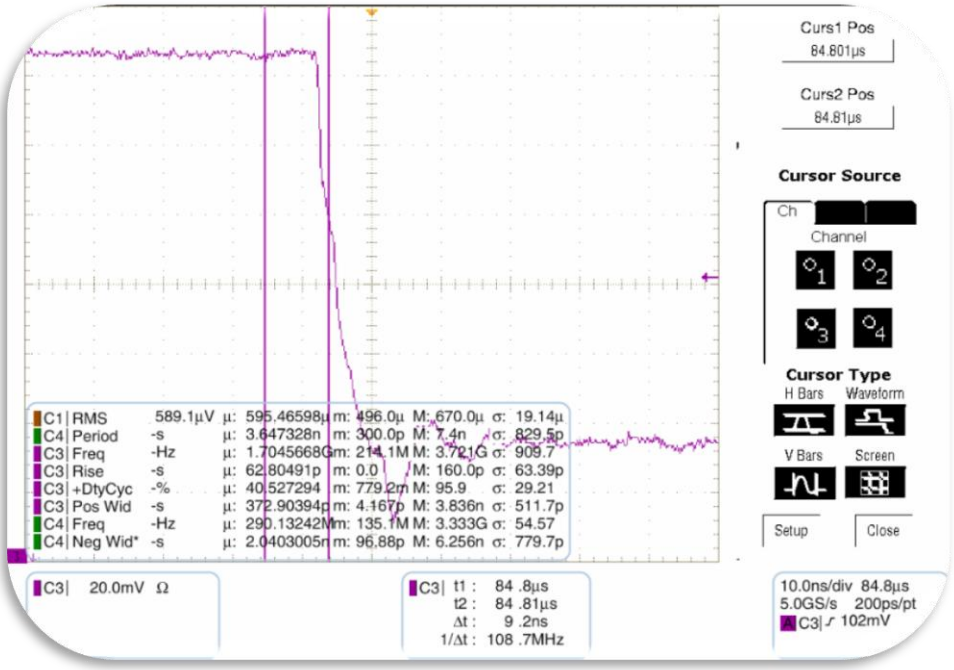



Figura 5.10: Medida del pulso de resolución de T=8.5ns.

La tercera medida correspondiente a la medida de resolución de T=10.2ns se muestra en la figura 6.1. En esta medida se ha podido medir mejor y se ha obtenido una diferencia de de 10.4ns muy cercano a los 10.2ns de ancho generado.

Medida del pulso de resolución de T3=10.2ns.

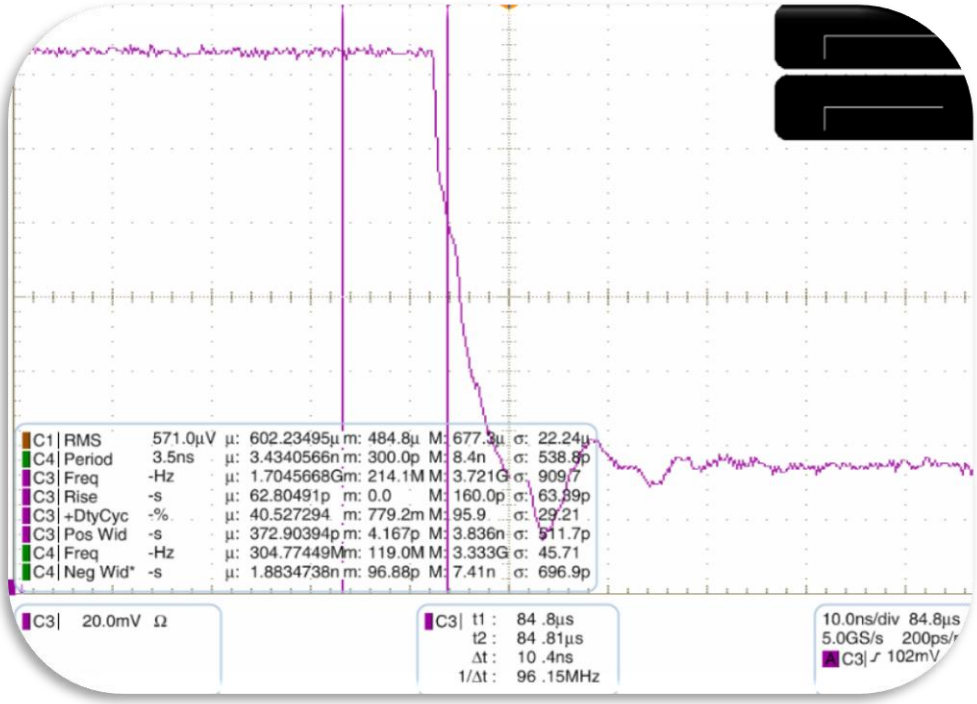



Figura 6.1: Medida del pulso de resolución de T=10.2ns.

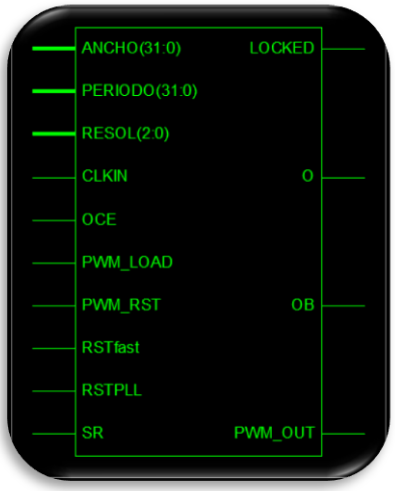
# 7. ESQUEMÁTICOS Y RECURSOS *HARDWARE*.

## 7.1. ESQUEMÁTICOS.

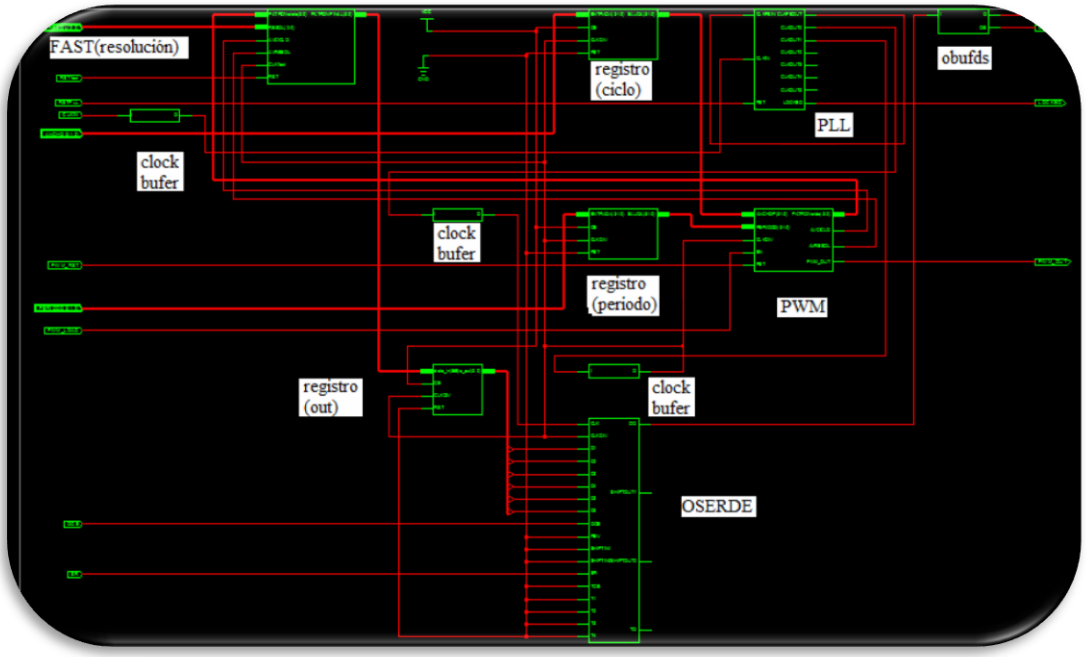
Acontinuación se representará el diseño con los esquemáticos que nos ofrece *ISE de Xilinx*.Los esquemáticos hacen referencia a la memoria, al diseño general sin memoria y con memoria.

### 7.1.1. ESQUEMÁTICO1: INTERFAZ DE ENTRADAS Y SALIDAS.

En este esquemático podemos ver los puertos de entrada y salida de diseño excluyendo la memoria.



En el siguiente esquemático se muestra el interior del diseño excluyendo la memoria. Aquí se pueden distinguir los diferentes componentes utilizados y sus conexiones.

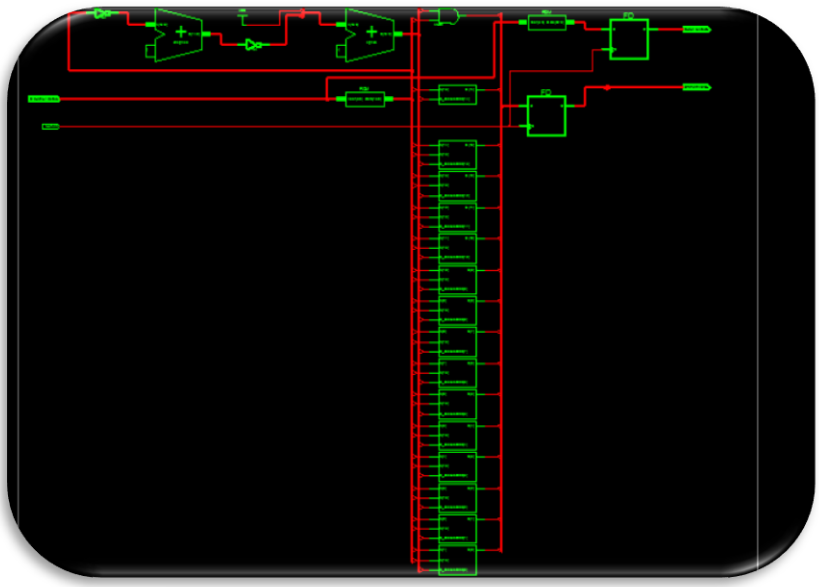



### 7.1.2. ESQUEMÁTICOS DEL BLOQUE DE MEMORIA.

Se ha extraído dos esquemáticos de la memoria, el *rtl schematic 1* que detalla los puertos de entrada y salida y el *rtl shematic 2* que nos muestra su interior.

*RTL SCHEMATIC 1.*

*RTL SCHEMATIC* 2*(*Vista del interior de la memoria*).*



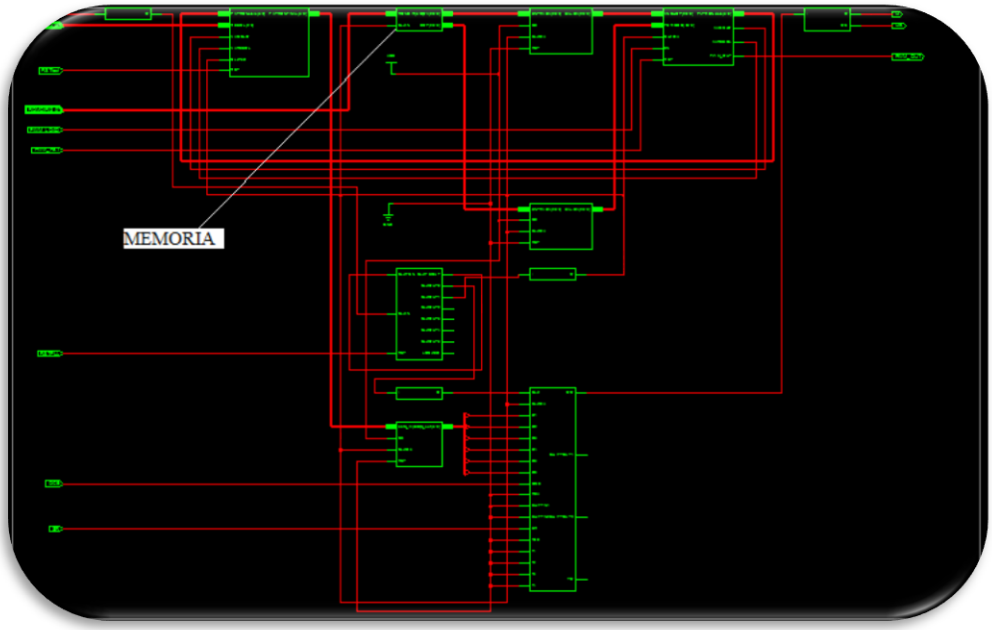
### 7.1.3. ESQUEMÁTICO DEL DISEÑO COMPLETO.

En las siguientes imágenes incluimos el bloque de memoria para completar el diseño.

El primer esquemático representa los puertos de entrada y salida del diseño completo. Y en el segundo esquemático podemos ver todos los componentes y sus conexiones internas.

*RTL SCHEMATIC* 1 del diseño completo.

*RTL SCHEMATIC* 2 del diseño completo. (Vista interna del diseño completo).




## 7.2. RECURSOS HARDWARE.

Tras la ejecución del *place and route* verificamos con el fichero place *and route report* los recursos *hardware* que utiliza el diseño.

El informe contiene la siguiente información:

- Los detalles de los recursos que el dispositivo utiliza en la implementación.
- El resumen también incluye información sobre la configuración del nivel de esfuerzo efectuado para lograr los resultados.
- El informe de reloj resumido que contiene las listas de todos los relojes usados en el diseño. También incluye información sobre el tipo de reloj, tiempos máximo y mínimo, *fonout* y el s*kew*.
- El Resumen de la sincronización que nos muestra los detalles sobre la ejecución del diseño frente a las limitaciones de tiempo.

## RECURSOS DEL DISEÑO.

La siguiente figura hace referencia al *place and route report* y detalla los recursos que utiliza nuestro diseño.

```
Device Utilization Summary:

   Number of BUFGs                          2 out of 32      6%
   Number of External IOBs                 15 out of 480     3%
      Number of LOCed IOBs                  0 out of 15      0%

   Number of External IOBMs                 1 out of 240     1%
      Number of LOCed IOBMs                 0 out of 1       0%

   Number of External IOBSs                 1 out of 240     1%
      Number of LOCed IOBSs                 0 out of 1       0%

   Number of OSERDESs                       1 out of 560     1%
   Number of PLL_ADVs                       1 out of 6      16%
   Number of Slice Registers              114 out of 28800   1%
      Number used as Flip Flops           114
      Number used as Latches                0
      Number used as LatchThrus             0

   Number of Slice LUTS                   225 out of 28800   1%
   Number of Slice LUT-Flip Flop pairs    230 out of 28800   1%
```

Figura 6.2: Recursos hardware.

# 8. CONCLUSIONES.

Con el diseño y la implementación del sintetizador de pulsos se demuestra la flexibilidad y la calidad que posee el dispositivo *FPGA* para obtener un alto nivel de calidad de señal. Por otra parte el lenguaje *VHDL* proporciona muchas facilidades al programador.

Al alcanzar un alto rendimiento con el uso de componentes embebidos hemos podido contemplar los niveles de procesamiento y la capacidad de procesar señales a frecuencias elevadas.

Gracias al lenguaje *VHDL* se diseño una arquitectura con una jerarquía modular obteniendo así la cualidad de ser reutilizada para posteriores diseños como un dispositivo portable.

La idea final era la de crear un instrumento destinado para el testeo de componentes mediante el envío de estímulos. De esta manera se consiguió un generador de pulsos con alto rendimiento con un gran nivel de reconfiguración y con la capacidad de adaptarse a los cambios de señal efectuados.

Dada la actual flexibilidad del equipo de trabajo utilizado el proyecto deja una puerta abierta para posteriores ampliaciones e investigaciones.

En resumen, se ha demostrado que es posible realizar un sistema de control digital con altas prestaciones basado en un dispositivo lógico programable (*FPGA*) es decir se comprobó que podemos diseñar nuestros propios dispositivos por un menor presupuesto.

# 9. ANEXO I.

## 9.1. ESTRUCTURA *VHDL*.

El diseño en *VHDL* consiste en la construcción del diagrama en bloque del sistema. En diseños complejos como en software los programas son generalmente jerárquicos y *VHDL* ofrece un buen marco de trabajo para definir los módulos que integran el sistema y sus interfaces dejando los detalles para pasos posteriores.

Es un lenguaje basado en texto, se puede utilizar cualquier editor para esta tarea, aunque el entorno de los programas de *VHDL* incluye su propio editor de texto. Después que se ha escrito algún código se hace necesario compilarlo. El compilador analiza este código y determina los errores de sintaxis y chequea la compatibilidad entre módulos.

El área de la simulación la cual permite establecer los estímulos a cada modulo y observar su respuesta nos da la posibilidad de crear bancos de prueba que automáticamente aplica entradas y compara las salidas con las respuestas deseadas. La simulación es un paso dentro del proceso de verificación. El propósito de la simulación es verificar que el circuito trabaja como se desea.

Hay dos áreas que verificar:

- Su comportamiento funcional en donde se estudia su comportamiento lógico independiente de consideraciones de tiempo como las demoras en las compuertas.

- Su verificación en el tiempo en donde se incluye las demoras de las compuertas y otras consideraciones de tiempo como los tiempos de establecimiento (*set-up time*) y los tiempos de mantenimiento (*hold-time*).

Después de la verificación se está listo para entrar en la fase final del diseño. La naturaleza y herramientas en esta fase dependen de la tecnología, pero hay tres pasos básicos:

- El primero es la síntesis que convierte la descripción en *VHDL* en un conjunto de componentes que pueden ser realizados en la tecnología seleccionada. Por ejemplo con *PLD* se generan las ecuaciones en suma de productos. En *ASIC* genera una lista de compuertas y un *netlist* que especifica como estas compuertas son interconectadas.

- En el siguiente paso de ajuste donde los componentes se ajustan a la capacidad del dispositivo que se utiliza. Para *PLD* esto significa que acopla las ecuaciones obtenidas con los elementos *AND – OR* que dispone el circuito. Para el caso de *ASIC* se dibujarían las compuertas y se definiría como conectarlas.

- En él último paso se realiza la verificación temporal ya que a esta altura se pueden calcular los elementos parásitos como las capacidades de las conexiones.

Con la simulación comprobaremos la funcionalidad deseada y con la sintetización crearemos un circuito que funciona como modelo. Se hace un resumen gráfico con la figura 6.3.

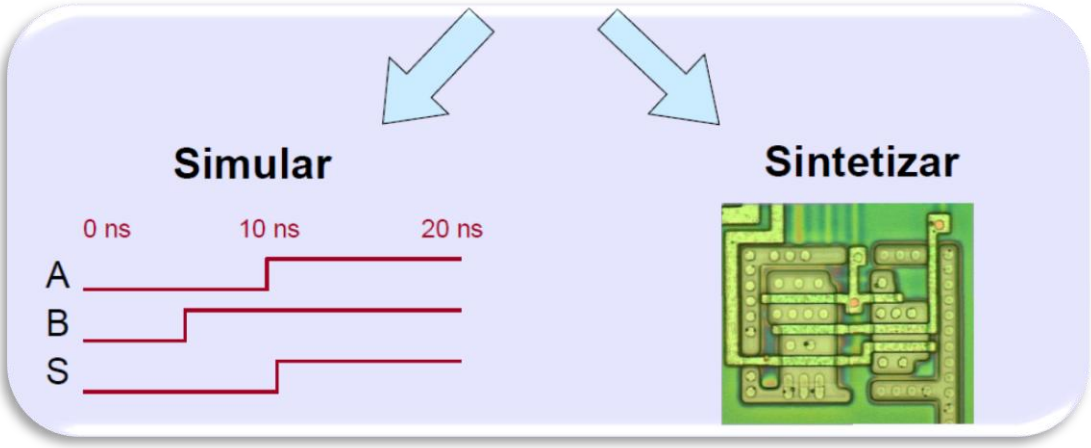



Figura 6.3: Simulación y sintetización.

## 9.2. MODELADO *HARDWARE VHDL.*

### BLOQUE DE ESTRUCTURA.

En este bloque se realiza el diseño, podemos modelar de forma estructural a través de componentes o de forma funcional mediante algoritmos. Siempre se comienza por definir las entradas y salidas en la denominada entidad y después pasamos a definir las funciones que queremos en la estructura.

### BLOQUE DE CONCURRENCIA.

Con *VHDL* tenemos la posibilidad de utilizar un programa llamado proceso el cual se compone de sentencias y podemos programarlo de forma que todo se pueda ejecutar en forma paralela. Para lograr el sincronismo podemos utilizar un elemento denominado señal el cual lo introduce el programador para que el proceso se ejecute cuando exista algún cambio en las señales.

### BLOQUE DE TIEMPO

Algo muy importante es que podemos realizar una simulación en el tiempo. Las estructuras de simulación son diferentes con respecto a las sintaxis de los proyectos normales. La simulación trabaja en forma de eventos entre las entradas, salidas y señales internas de modo que cuando ocurra un cambio en la lista de eventos actuales el simulador calculará las consecuencias.

En general un diseño *VHDL* se compone de un conjunto de bloques o módulos donde cada uno de ellos contiene declaraciones o instrucciones que describen y estructuran el comportamiento del sistema. Dentro de cada módulo se declaran diferentes unidades de diseño como son:

- Entidad.
- Arquitectura.
- Configuración.
- Paquete.
- Biblioteca.

### 9.2.1. ENTIDAD Y ARQUITECTURA.

Las principales unidades de diseño son:

- Entidad: Es el elemento básico del lenguaje *VHDL* que define el nombre de un componente y su interfaz de entrada-salida. La interfaz es definida por un conjunto de puertos mientras que la implementación queda oculta al resto del circuito como un modelo de caja negra.

- Arquitectura: Describe el funcionamiento de la entidad a la que está asociada. Una misma entidad puede contener una o más arquitecturas. Se compone de dos partes: la "declarativa" donde se incluyen las señales, variables y componentes que se va a emplear y el "cuerpo" donde se incluye la implementación de la entidad por medio de instrucciones o sentencias.

La caja negra se visualiza en la figura 6.4.

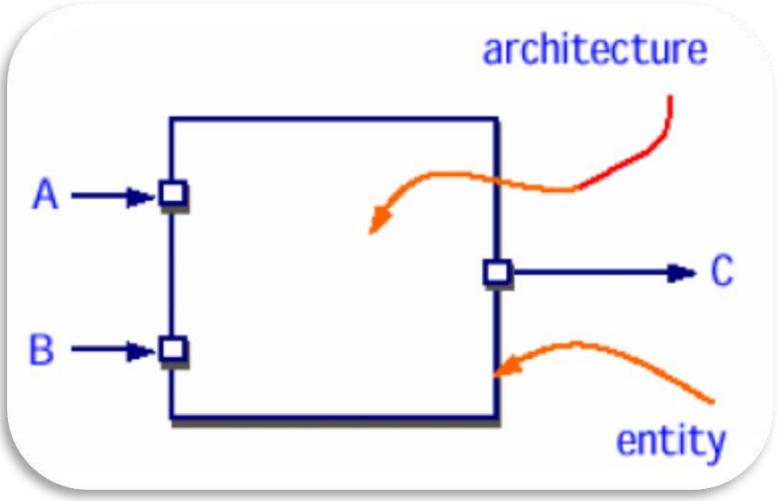



Figura 6.4: Ejemplo de caja negra.

Características de la caja negra:

• En *VHDL* la caja negra se denomina entidad
– La *ENTITY* describe la *E/S* del diseño

• Para describir su funcionamiento se asocia una implementación que se denomina arquitectura
– La *ARCHITECTURE* describe el contenido del diseño.

## DECLARACIÓN DE LA ENTIDAD Y ARQUITECTURA.

Ejemplo de declaración de la caja negra:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY mi_componente IS PORT (
        clk, rst: IN std_logic;
        d: IN std_logic_vector(7 DOWNTO 0);
        q: OUT std_logic_vector(7 DOWNTO 0));
END mi_componente
```

La estructura del diseño es visualizada en la figura 6.5.

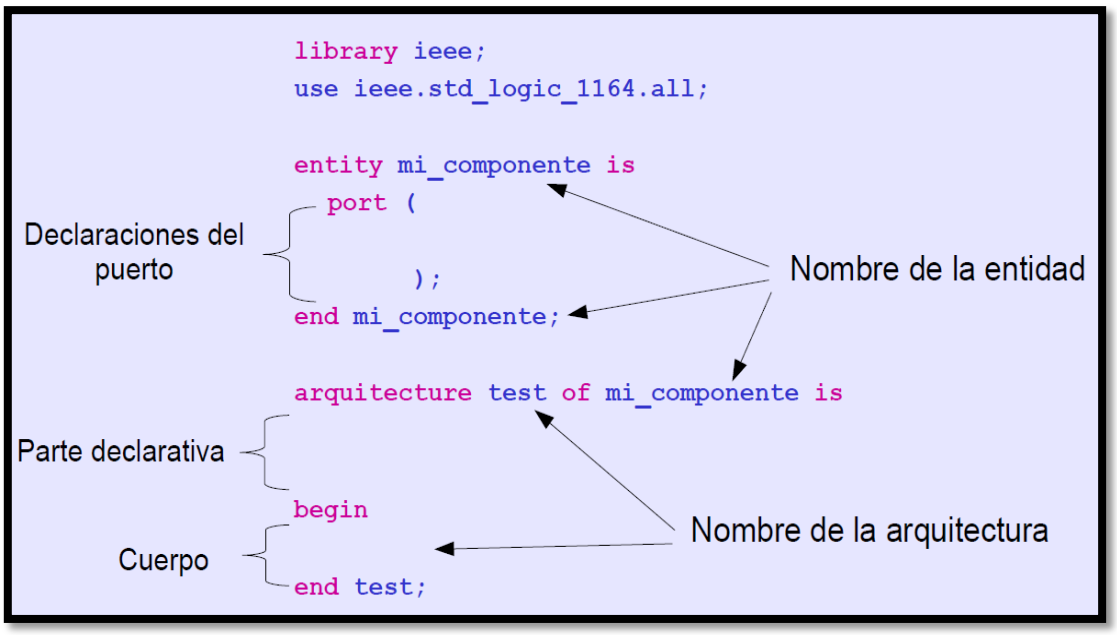



Figura 6.5: Estructura del diseño en *VHDL*.

### 9.2.2. LIBRERIAS.

Una librería consiste en una colección de unidades de diseño analizadas previamente con lo cual se facilita la utilización de estas en nuevos diseño. Para incluir una librería se utiliza la siguiente sintaxis:

*LIBRARY identificador_librería [,identificador_librería ] ;*

La cláusula LIBRARY permite utilizar la librería especificada únicamente para la unidad de diseño en la cual se declara. Una unidad de diseño es una entidad, paquete, arquitectura, o cuerpo de paquete.

Para desarrollar los diferentes programas se utilizó las correspondientes librerías:

- *librerías IEEE use IEEE.STD_LOGIC_1164.ALL;*
  *use IEEE.STD_LOGIC_ARITH.ALL;*
  *use IEEE.STD_LOGIC_UNSIGNED.ALL;*

- *library work; use work.componentes.all*

- *Library UNISIM;*
  *use UNISIM.vcomponents.all.*

La librería *UNISIM* es la que hace referencia a un *core* interno, es decir, a un componente embebido que posee el dispositivo *FPGA*. Facilita y agiliza el diseño ya que permiten tener acceso a estructuras lógicas predeterminadas por el fabricante.

### 9.2.3. TIPOS DE DATOS BÁSICOS.

Un tipo de dato especifica el grupo de valores que un objeto de datos puede tomar así como las operaciones que son permitidas con esos valores. En *VHDL* es sumamente importante el tipo de dato, los objetos de datos no pueden tomar o no se les puede asignar un objeto de datos de otro tipo, y no todas las operaciones se pueden utilizar con los diferentes tipos de datos a menos que se utilicen las librerías adecuadas en las que estén definidas funciones para la conversión de tipos.

Es posible que el usuario defina subtipos y tipos compuestos, modificando los tipos básicos, así como definir tipos particulares con combinaciones de los diferentes tipos. A continuación se discutirán las dos categorías de tipos de datos más utilizadas en síntesis: escalares y compuestos. Los tipos de datos son visualizados en la figura 6.6.

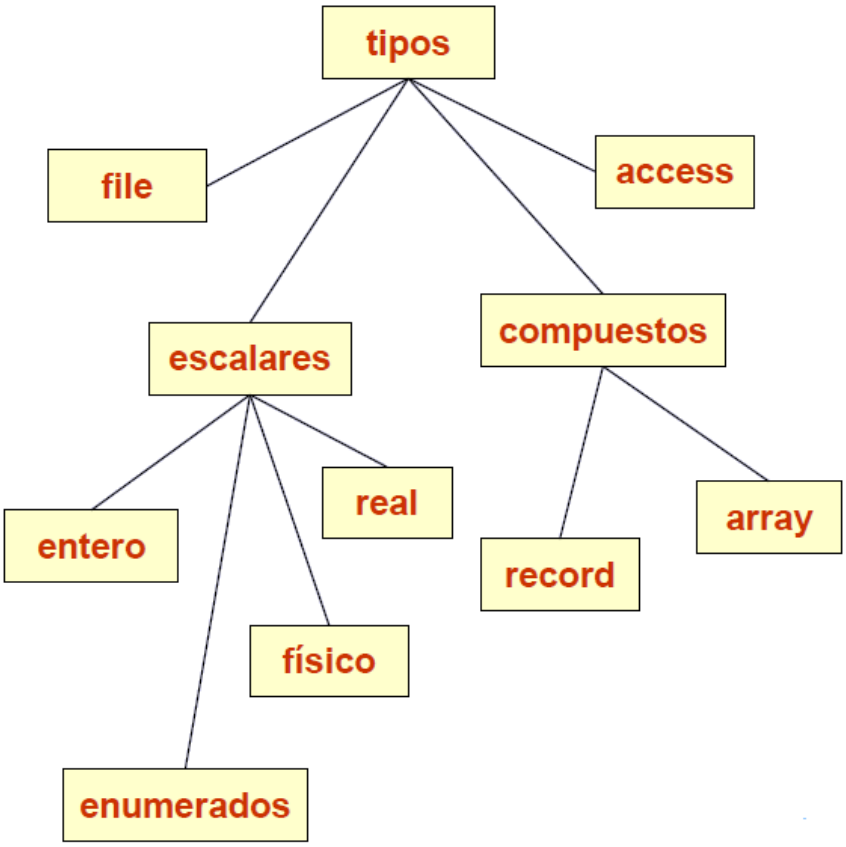



Figura 6.6: Tipos de datos básicos del lenguaje *VHDL*.

Los tipos predefinidos son:

– Escalares: *integer floating point enumerated physical*.
– Compuestos: *array, record*.
– Punteros: *Access*.
– Archivos: *file*.

## *STD_LOGIC Y STD_LOGIC_VECTOR.*

• Definidos en el paquete IEEE.standard_logic_1164
• Son un estándar industrial.
• Los emplearemos siempre para definir los puertos de las entidades.
• Tipo *Std_logic*: valor presente en un cable de 1 bit.
• Tipo *Std_logic_vector*: para definir buses (*array de std_logic*).

## BIT.

Los objetos de este tipo pueden tomar los valores de '0' y '1'. Los valores se representan entre comillas simples, son para indicar que son bits y no números enteros.

## COMPUESTOS.

Un tipo compuesto es un tipo de dato formado con elementos de otros tipos. Existen dos formas de tipos compuestos: *ARRAYS* y *RECORDS*.

## OPERADORES.

Un operador nos permite construir diferentes tipos de expresiones mediante los cuales podemos calcular datos utilizando los diferentes objetos de datos con el tipo de dato que maneja dicho objeto.

En *VHDL* existen distintos operadores de asignación con lo que se transfieren valores de un objeto de datos a otro y operadores de asociación que relacionan un objeto de datos con otro, lo cual no existe en ningún lenguaje de programación de alto nivel.

El uso de los operadores que aquí son expuestos dependerá del software utilizado ya que no es regla que los utilicen todos. Para conocer las operaciones que pueden ser utilizadas así como los paquetes incluidos en el software es recomendable revisar las librerías del programa. De no encontrarse algún operador especial para ser utilizado con algún tipo de dato especifico es necesario sobrecargar los operadores o en ocasiones crearlo. Para poder utilizar la mayoría de estos operadores con los tipos *signed*, *unsigned* y *std_logic_vector* basta con utilizar el paquete donde se encuentran declarados estos tipos porque dentro de los mismos paquetes ya se encuentran varias funciones aritméticas y lógicas para que sean utilizadas con estos tipos.

## SUBTIPOS.

Un subtipo es un "subgrupo" de un tipo predefinido. Los subtipos son útiles para crear tipos de datos con limitaciones sobre tipos mayores.

Declaración de subtipos:

*SUBTYPE identificador IS tipo_base RANGE valor*
*( DOWNTO / TO ) valor ;*

## TIPOS DE OPERADORES:

_Lógicos AND, OR, NAND, NOR, XOR, XNOR, NOT

_Comparación =, /=, <, >, <=, >=

_Adición +, -, &

_Multiplicación *, /, MOD, REM

_Misceláneo abs, **

_Asignación <=, :=

_Asociación =>

_Corrimiento SLL, SRL, SLA, SRA, ROL, ROR.

### 9.2.4. VARIABLES, SEÑALES Y CONSTANTES.

#### VARIABLES Y SEÑALES.

Los objetos de datos de la clase variable son similares a las constantes, con la diferencia que su valor puede ser modificado cuando sea necesario. Las variables en *VHDL* son similares a cualquier tipo de variable de un lenguaje de programación de alto nivel. A las variables también se les puede asignar un valor inicial al momento de ser declaradas. Se utilizan únicamente en los procesos y subprogramas (funciones y procedimientos). Las variables generalmente se utilizan como índices, principalmente en instrucciones de bucle o para tomar valores que permitan modelar componentes.

Un objeto de la clase señal es similar a un objeto de la clase variable con una importante diferencia: las señales si pueden almacenar o pasar valores lógicos mientras que una variable no lo puede hacer. Las señales por lo tanto representan elementos de memoria o conexiones y si pueden ser sintetizadas. La tabla siguiente de la figura 6.7 nos muestra las diferentes señales y variables.

| | **Señales** | **Variables** |
|---|---|---|
| **Sintaxis** | destino <= fuente | destino := fuente |
| **Utilidad** | modelan nodos físicos del circuito | representan almacenamiento local |
| **Visibilidad** | global (comunicación entre procesos) | local (dentro del proceso) |
| **Comportamiento** | se actualizan cuando avanza el tiempo (se suspende el proceso) | se actualizan inmediatamente |

Figura 6.7: Tabla de variables y señales.

### CONSTANTES.

Una constante es un elemento que puede tomar un único valor de un tipo dado. A las constantes se les debe asignar un valor en el momento de la declaración. Una vez que se le ha asignado algún valor éste no puede ser cambiado dentro de la descripción del diseño. Las constantes pueden ser declaradas dentro de las entidades, arquitecturas, procesos o paquetes. Las constantes que se declaren en un paquete pueden ser utilizadas en cualquier descripción en la que se está utilizando dicho paquete. Por otra parte las constantes declaradas dentro de una entidad pueden ser utilizadas por las arquitecturas en las que se está haciendo la descripción de dicha entidad y aquellas constantes que sean declaradas dentro de una arquitectura o proceso son validas únicamente dentro de la estructura correspondiente.

## 9.2.5. SENTENCIAS.

### 9.2.5.1. SENTENCIAS SECUENCIALES.

En la mayoría de los lenguajes de descripción de software todas las sentencias de asignamiento son de naturaleza secuencial. Esto significa que la ejecución del programa se llevara a cabo de arriba a abajo, es decir siguiendo el orden en el que se hayan dispuesto dichas sentencias en el programa, por ello es de vital importancia la disposición de las mismas dentro del código fuente.

*VHDL* lleva a cabo las asignaciones a señales dentro del cuerpo de un proceso (*process*) de forma secuencial, con lo que el orden en el que aparezcan las distintas asignaciones será el tenido en cuenta a la hora de la compilación. Esto hace que cuando utilicemos modelos secuenciales en *VHDL* estos se comporten de forma parecida a cualquier otro lenguaje de programación como Pascal, C, etc.

#### 9.2.5.1.1. SENTENCIA *IF*.

La construcción *if-then-else* es usada para seleccionar un conjunto de sentencias para ser ejecutadas según la evaluación de una condición o conjunto de condiciones cuyo resultado debe ser o true o false.

Su estructura es la siguiente:

*if (condición) then*
*-- haz una cosa;*
*else*
*-- haz otra cosa diferente;*
*end if;*

Si la condición entre paréntesis es verdadera, la sentencia secuencial seguida a la palabra *then* es ejecutada. Si la condición entre paréntesis es falsa, la sentencia secuencial seguida a la palabra *else* es ejecutada. La construcción debe ser cerrada con las palabras *end if*.

La sentencia *if-then-else* puede ser expandida para incluir la sentencia *elsif*, la cual nos permite incluir una segunda condición si no se ha cumplido la primera (la cual tiene prioridad).

Si se da la situación en la cual la primera condición es verdadera, se ejecuta las sentencias que van después del primer *then*. Si no es verdadera la primera condición, se pasa a evaluar la segunda, y de ser esta verdad, ejecuta las sentencias que están a continuación del segundo *then*. Si ninguna de las dos es verdadera, se ejecuta lo que está detrás de la palabra *else*. Nótese que para que se ejecute las sentencias con el nombre "otra cosas diferente", no solo debe ser la segunda condición verdadera, sino que además la primera condición debe ser falsa.

### 9.2.5.1.2. SENTENCIA *CASE*.

La sentencia *case* es usada para especificar una serie de acciones según el valor dado de una señal de selección. Esta sentencia es equivalente a la sentencia *with-select-when* con la salvedad que la sentencia que nos ocupa es secuencial no combinacional.

La estructura es la siguiente:

*case (señal a evaluar) is*

*when (valor 1) => haz una cosa;*
*when (valor 2) => haz otra cosa;*
*...*
*when (último valor) => haz tal cosa;*

*end case;*

En el caso que la señal a evaluar (situada después del *case*) tenga el "valor 1", entonces se ejecuta "una cosa", si tiene el "valor 2", se ejecuta "otra cosa", ... y si tiene el "último valor", se ejecuta "tal cosa". Esta sentencia parece hecha a la medida para crear multiplexores.

La sentencia case también nos permite especificar un rango de valores posibles de la señal de selección, para los cuales hacer una asignación, mediante la palabra reservada *to*.

Como ejemplo veamos dos fragmentos de código que son equivalentes:

```
case control is
when "000"  => d <= a;
when "001"  => d <= a;
when "010"  => d <= a;
when "011"  => d <= b;
when "100"  => d <= b;
when "101"  => d <= b;
when "110"  => d <= b;
when "111"  => d <= c;
when others => d <= null;
end case;
```

### 9.2.5.2. SENTENCIAS CONCURRENTES.

La naturaleza propia de los circuitos eléctricos obliga a *VHDL* a soportar un nuevo tipo de asignación de señales, que nos permita implementar este tipo de operatividad. En ella todas las asignaciones se llevan a cabo en paralelo (al mismo tiempo). En una asignación concurrente la señal que esté a la izquierda de la asignación es evaluada siempre que alguna de las señales de la derecha modifique su valor.

Como ejemplo tenemos las siguientes sentencias de asignación:

```
c <= a and b;
s <= a xor b;
```

## 9.3. FORMAS DE PROGRAMACIÓN.

En *VHDL* la descripción del modelo se puede realizar utilizando las siguientes formas de programación:

- Funcional (*Behavioral*): describe el algoritmo que refleja la funcionalidad o el comportamiento de dicho componente de manera secuencial.

- Flujo de Datos (*Data Flow*): específica los flujos de datos del sistema y la interconexión entre sus componentes. Se trabaja por medio de funciones lógicas, que se ejecutan de forma concurrente.

- Estructural (*Structural*): esta descripción se basa en modelos lógicos ya establecidos como restadores, sumadores, compuertas, entre otros.

## 9.4. PROCESOS.

La sentencia *process* es una de las construcciones típicas de *VHDL* usadas para agrupar algoritmos. Esta sentencia se inicia (de forma opcional) con una etiqueta seguida de dos puntos (:), después del *process* y una lista de variables sensibles. La lista sensible, indica que señales harán que se ejecuta el proceso, es decir, qué variables deben cambiar para que se ejecute el proceso. Dentro de un proceso se encuentran sentencias secuenciales no concurrentes. Esto hace que el orden de las órdenes dentro de un proceso sea importante ya que se ejecuta una después de otra y los posibles cambios que deba haber en las señales alteradas se producen después de evaluar todo el ciclo al completo. Ésta característica define una de las particularidades de *VHDL*.

Un proceso describe el comportamiento de un circuito.

- Cuyo estado puede variar cuando cambian ciertas señales.
- Utilizando construcciones muy expresivas: *if..then..else*, *case*, bucles *for* y *while*, etc…
- Y que además puede declarar variables, procedimientos, etc...
- Las instrucciones dentro del proceso se ejecutan secuencialmente, una detrás de otra, pero sin dar lugar a que avance el tiempo durante su ejecución.
- El tiempo sólo avanza cuando se llega al final del proceso.
- Las señales modelan hilos del circuito, y como tales, sólo pueden cambiar de valor si se deja que avance el tiempo.
- Una arquitectura puede tener tantos procesos como queramos, y todos se van a ejecutar en paralelo

La estructura de proceso es la siguiente:

*process(lista de señales).*
*...*
*parte declarativa (variables, procedimientos, tipos, etc…)*
*...*
*begin*
*...*
*instrucciones que describen el comportamiento*
*...*
*end process;*

## 9.5. BANCO DE PRUEBAS (*TEST BENCH*).

Hay que hacer un banco de pruebas (*test bench*) para cada componente diseñado. La simulación de un componente consiste en:

- Generar unos estímulos.
- Observar los resultados.

Para hacer un banco de pruebas el primer paso es instanciar el diseño que vamos a verificar y sucesivamente desarrollaremos "un proceso o procesos" para generar los estímulos y observaremos el resultado. L forma de trabajo del *test bench* se visualiza en la figura 6.8.

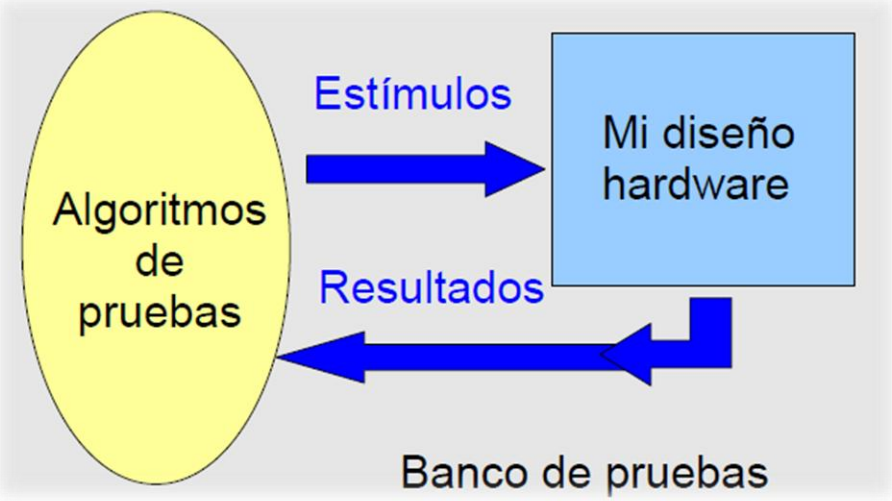



Figura 6.8: Banco de pruebas.

# 10. ANEXO 2.

## 10.1. CÓDIGOS DE PROGRAMACIÓN.

### 10.1.1. PROGRAMAS.

#### 10.1.1.1 MÓDULO *PLL*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
Library UNISIM;
use UNISIM.vcomponents.all;

entity pll1 is
Port (
CLKIN : in  STD_LOGIC;
CLKFBIN : in  STD_LOGIC;
RST  : in  STD_LOGIC;
CLKFBOUT : out  STD_LOGIC;
CLKOUT0 : out  STD_LOGIC;
CLKOUT1 : out  STD_LOGIC;
CLKOUT2 : out  STD_LOGIC;
CLKOUT3 : out  STD_LOGIC;
CLKOUT4 : out  STD_LOGIC;
CLKOUT5 : out  STD_LOGIC;
LOCKED : out  STD_LOGIC);
end pll1;

architecture Behavioral of pll1 is
begin
PLL_BASE_inst : PLL_BASE
generic map (
BANDWIDTH => "OPTIMIZED"
CLKFBOUT_MULT => 6,      -- Multiplication factor for all output clock
CLKFBOUT_PHASE => 0.0,  -- Phase shift (degrees) of all output clocks
CLKIN_PERIOD => 10.0,     -- Clock period (ns) of input clock on CLKIN

CLKOUT0_DIVIDE => 1,   -- Division factor for CLKOUT0 (1 to 128)
CLKOUT0_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT0 (0.01 to 0.99)
CLKOUT0_PHASE => 0.0,   -- Phase shift (degrees) for CLKOUT0 (0.0 to 360.0)
CLKOUT1_DIVIDE => 6,   -- Division factor for CLKOUT1 (1 to 128)
CLKOUT1_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT1 (0.01 to 0.99)
CLKOUT1_PHASE => 0.0,   -- Phase shift (degrees) for CLKOUT1 (0.0 to 360.0)
```

```
CLKOUT2_DIVIDE => 1,   -- Division factor for CLKOUT2 (1 to 128)
CLKOUT2_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT2 (0.01 to 0.99)
CLKOUT2_PHASE => 0.0,   -- Phase shift (degrees) for CLKOUT2 (0.0 to 360.0)
CLKOUT3_DIVIDE => 1,  -- Division factor for CLKOUT3 (1 to 128)
CLKOUT3_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT3 (0.01 to 0.99)
CLKOUT3_PHASE => 0.0,    -- Phase shift (degrees) for CLKOUT3 (0.0 to 360.0)
CLKOUT4_DIVIDE => 1,  -- Division factor for CLKOUT4 (1 to 128)
CLKOUT4_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT4 (0.01 to 0.99)
CLKOUT4_PHASE => 0.0,    -- Phase shift (degrees) for CLKOUT4 (0.0 to 360.0)
CLKOUT5_DIVIDE => 1,       -- Division factor for CLKOUT5 (1 to 128)
CLKOUT5_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT5 (0.01 to 0.99)
CLKOUT5_PHASE => 0.0,      -- Phase shift (degrees) for CLKOUT5 (0.0 to 360.0)

COMPENSATION => "SYSTEM_SYNCHRONOUS",
-- "SYSTEM_SYNCHRNOUS","SOURCE_SYNCHRNOUS","INTERNAL
"EXTERNAL", "DCM2PLL", "PLL2DCM"

DIVCLK_DIVIDE => 1,      -- Division factor for all clocks (1 to 52)
REF_JITTER => 0.100)    - Input reference jitter (0.000 to 0.999 UI%)
port map (

CLKFBOUT => CLKFBOUT,   -- General output feedback signal
CLKOUT0 => CLKOUT0,    -- One of six general clock output signals
CLKOUT1 => CLKOUT1,    -- One of six general clock output signals
CLKOUT2 => CLKOUT2,    -- One of six general clock output signals
CLKOUT3 => CLKOUT3,    -- One of six general clock output signals
CLKOUT4 => CLKOUT4,    -- One of six general clock output signals
CLKOUT5 => CLKOUT5,    -- One of six general clock output signals
LOCKED => LOCKED,         -- Active high PLL lock signal
CLKFBIN => CLKFBIN,        -- Clock feedback input
CLKIN => CLKIN,            -- Clock input
RST => RST                 -- Asynchronous PLL reset
 );
end Behavioral;
```

### 10.1.1.2. MÓDULO *PWM*.

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

entity gpwm is

Port (
CLKDIV : in  STD_LOGIC;
ANCHOP: in  STD_LOGIC_vector(31 downto 0);
PERIODO: in  STD_LOGIC_vector(31 downto 0);
RST : in  STD_LOGIC;
EN : in  STD_LOGIC;
AVCICLO : out  STD_LOGIC;
AVRESOL : out  STD_LOGIC;
PATRONciclo: out  STD_LOGIC_vector(5 downto 0);
PWM_OUT : out  STD_LOGIC);

end gpwm;

architecture Behavioral of gpwm is
type ESTADOS is (S0,S1,S2);
signal S : ESTADOS;
begin

process (CLKDIV)
variable CUENTA: integer:=0;
variable INTANCHOP : integer:=0;
variable INTPERIODO : integer:=0;
begin
if CLKDIV'event and CLKDIV='1' then

if (EN='1') then

case S is
when S0 => INTPERIODO := conv_integer (PERIODO);
CUENTA :=0;

if (RST='1') then
S<=S0;
CUENTA:=0;
AVCICLO<='0';

elsif (RST='0') and (PERIODO > 0)  then
S <= S1;
```

```
PATRONciclo<="111111";

elsif (PERIODO = 0) then
S <= S0;
PATRONciclo<="000000";
AVRESOL<='1';
end if;

when S1 => INTANCHOP := conv_integer (ANCHOP);
CUENTA:=CUENTA+1;

if (CUENTA < INTANCHOP) then
S <= S1;
AVCICLO<='1';
AVRESOL<='0';

elsif (CUENTA = INTANCHOP) then
S<=S2;
AVCICLO<='0';
AVRESOL<='1';
end if;

when S2 =>    INTPERIODO := conv_integer (PERIODO);
CUENTA:=CUENTA+1;

If ( CUENTA < INTPERIODO) then
S <= S2;
AVCICLO<='0';
AVRESOL<='1';

elsif (CUENTA = INTPERIODO)then
S<=S1;
CUENTA:=0;
AVCICLO<='1';
AVRESOL<='0';
end if;

end case;
end if;
end if;
end process;
process2 : process (S)
begin
case S is
when S0 => PWM_OUT<= '0';
when S1 => PWM_OUT<= '1';
when S2 => PWM_OUT<= '0';
end case;
end process process2;
end Behavioral;
```

### 10.1.1.3. MÓDULO *FAST* (RESOLUCIÓN).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity fast is
Port (
PATRONciclo: in  STD_LOGIC_vector(5 downto 0);
AVCICLO: in  STD_LOGIC;
AVRESOL: in  STD_LOGIC;
RESOL : in  STD_LOGIC_vector(2 downto 0);
CLKfast: in  STD_LOGIC;
RST : in  STD_LOGIC;
PATRONFINAL: out  STD_LOGIC_vector(5 downto 0)
);
end fast;

architecture Behavioral of fast is
type ESTADOS is(s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10);
signal s:ESTADOS;
begin

G_REG:process(RST, CLKfast, AVCICLO, AVRESOL)
variable CUENTA: integer:=0;
variable INTRESOL: integer:=0;

begin

if (RST='1')then
s<=s0;
elsif (CLKfast'event and CLKfast='1') then
case s is

when s0 => if(PATRONciclo="111111" and AVCICLO='1')  then s<=s1;
elsif (PATRONciclo="000000" and AVCICLO='0' and AVRESOL='1') then s<=s10;
end if;

when s1=>  if (AVRESOL='1') and (RESOL="000")  then s<=s2;
--sistema de control resolución introducimos el pulso con periodo y ancho

elsif (AVRESOL='1') and (RESOL="001") then s<=s3;
elsif (AVRESOL='1') and (RESOL="010") then s<=s4;
elsif (AVRESOL='1') and (RESOL="011") then s<=s5;
elsif (AVRESOL='1') and (RESOL="100") then s<=s6;
elsif (AVRESOL='1') and (RESOL="101") then s<=s7;
elsif (AVRESOL='1') and (RESOL="110") then s<=s8;
```

```
elsif (AVRESOL='1') and (RESOL="111") then s<=s2;
elsif (AVRESOL='0') then s<=s1; end if;


when s2=> if (AVCICLO='0' and PATRONciclo="000000")  then s<=s10;
--y control de resolución sola para volver a s10(ceros)
elsif (RST='0')  then s<=s9; --000 primera elección de resol(sin resol)
end if;

when s3=> if (AVCICLO='0' and PATRONciclo="000000")  then s<=s10;
elsif (RST='0')  then s<=s9;
end if;

when s4=> if (AVCICLO='0' and PATRONciclo="000000")  then s<=s10;
elsif (RST='0')  then s<=s9;
end if;
when s5=>if (AVCICLO='0' and PATRONciclo="000000")  then s<=s10;
elsif (RST='0')  then s<=s9;
end if;

when s6=> if (AVCICLO='0' and PATRONciclo="000000")  then s<=s10;
elsif (RST='0')  then s<=s9;
end if;

when s7=> if (AVCICLO='0' and PATRONciclo="000000")  then s<=s10;
elsif (RST='0')  then s<=s9;
end if;

when s8=> if (AVCICLO='0' and PATRONciclo="000000")  then s<=s10;
elsif (RST='0')  then s<=s9;
end if;

when s9=> if (AVCICLO='1')  then s<=s1;
--espero el comienzo del nuevo pulso
elsif (RST='0') then s<=s0;
end if;

when s10=> if (AVRESOL='1') and (RESOL="000")  then s<=s2;
----sistema de control
elsif (AVRESOL='1') and (RESOL="001") then s<=s3;
elsif (AVRESOL='1') and (RESOL="010") then s<=s4;
elsif (AVRESOL='1') and (RESOL="011") then s<=s5;
elsif (AVRESOL='1') and (RESOL="100") then s<=s6;
elsif (AVRESOL='1') and (RESOL="101") then s<=s7;
elsif (AVRESOL='1') and (RESOL="110") then s<=s8;
elsif (AVRESOL='1') and (RESOL="111") then s<=s2;
elsif (AVRESOL='0') then s<=s1;
end if;

end case;
```

```
end if;
end process G_REG;

H:process(s,PATRONciclo)
 begin

case s is
when s0 =>PATRONFINAL<="000000";
when s1 =>PATRONFINAL<="111111";---decido donde ir
when s2 =>PATRONFINAL<="000000";------principio datos resol 000
when s3 =>PATRONFINAL<="000001";  --001
when s4 =>PATRONFINAL<="000011";  --010
when s5 =>PATRONFINAL<="000111";  --011
when s6 =>PATRONFINAL<="001111";  --100
when s7 =>PATRONFINAL<="011111";  --101
when s8 =>PATRONFINAL<="111111";-----fin datos resol  110
when s9 =>PATRONFINAL<="000000";
when s10 =>PATRONFINAL<="000000";-----opcion solo resol
end case;
end process H;
end Behavioral;
```

### 10.1.2. PROGRAMACIÓN DE LOS *BUFFERS*.

#### 10.1.2.1. IBUFG.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
Library UNISIM;
use UNISIM.vcomponents.all;

entity ibufg1 is
Port (     I : in  STD_LOGIC;
          O : out  STD_LOGIC);
end ibufg1;

architecture Behavioral of ibufg1 is
begin
 IBUFG_inst : IBUFG
   generic map (
IBUF_DELAY_VALUE => "0", -- Specify the amount of added input   delay for buffer
IOSTANDARD => "DEFAULT")
   port map (
      O => O, -- Clock buffer output
      I => I  -- Clock buffer input (connect directly to top-level port)
   );
end Behavioral;
```

### 10.1.2.2. BUFG1.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
Library UNISIM;
use UNISIM.vcomponents.all;

entity bufg1 is
Port (
I : in  STD_LOGIC;
O : out  STD_LOGIC);
end bufg1;

architecture Behavioral of bufg1 is
begin
BUFG_inst : BUFG
port map (
O => O,     -- Clock buffer output
I => I      -- Clock buffer input
);
end Behavioral;
```

### 10.1.2.3 BUFG2.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
Library UNISIM;
use UNISIM.vcomponents.all;

entity bufg2 is
Port (
I : in  STD_LOGIC;
O : out  STD_LOGIC);
end bufg2;
architecture Behavioral of bufg2 is

begin
BUFG_inst : BUFG
port map (
O => O,     -- Clock buffer output
I => I      -- Clock buffer input
);
end Behavioral;
```

### 10.1.3. PROGRAMACIÓN DE LOS REGISTROS.

#### 10.1.3.1. REGISTRO FFIN.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
Library UNISIM;
use UNISIM.vcomponents.all;

entity ffin is
Port  (
ENTRADA: in std_logic_vector(29 downto 0);
CLKDIV, CE, RST : in std_logic;
SALIDA: out std_logic_vector(29 downto 0)
);
end ffin;

architecture Behavioral of ffin is
begin

process(CLKDIV,RST)
begin
if (RST='1') then
SALIDA<="00000000000000000000000000000000";

elsif(CLKDIV'event and CLKDIV='1') then
-- esto detecta las transiciones de subida

if (CE ='0') then-- output enable
SALIDA <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
else
SALIDA <= ENTRADA;
end if;
end if;
end process;
end Behavioral;
```

### 10.1.3.2. REGISTRO FFINPERIODO.

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

entity ffinperiodo is

Port  (
ENTRADA: in std_logic_vector(29 downto 0);
CLKDIV, CE, RST : in std_logic;
SALIDA: out std_logic_vector(29 downto 0)
);
end ffinperiodo;

architecture Behavioral of ffinperiodo is

begin
process(CLKDIV,RST)
begin
if (RST='1') then
SALIDA<="00000000000000000000000000000000";

elsif(CLKDIV'event and CLKDIV='1') then
-- esto detecta las transiciones de subida
if (CE ='0') then-- output enable
SALIDA <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
else
SALIDA <= ENTRADA;
end if;
end if;
end process;
end Behavioral;
```

### 10.1.3.3. REGISTRO FFOUT.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
Library UNISIM;
use UNISIM.vcomponents.all;

entity ffout is
Port (
CE,CLKDIV,RST : in std_logic;
data_in : in std_logic_vector (5 downto 0);-- in D
data_out: out std_logic_vector (5 downto 0)-- out Q
);
end ffout;

architecture Behavioral of ffout is
begin
process(CLKDIV,RST)
begin

if (RST='1') then
data_out<="000000";
elsif(CLKDIV'event and CLKDIV='1') then
-- esto detecta las transiciones de subida

if (CE ='0') then-- output enable
data_out <= "ZZZZZZ";
else
data_out <= data_in;
end if;
end if;
end process;
end Behavioral;
```

### 10.1.4. BLOQUE DE MEMORIA.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ieee.numeric_std.ALL;

entity memoria is
port (
CLKIN : in std_logic;
ROMDIR : in std_logic_vector(3 downto 0);
DOP : out std_logic_vector(29 downto 0);
DOA : out std_logic_vector(29 downto 0)
);
end memoria;

architecture Behavioral of memoria is
type ram_type is array (0 to 15) of std_logic_vector (29 downto 0);

constant ROM :
ram_type:=("000000000000000000000000000000","000000000111101000010010000000","000000000011110100001001000000","000000000000110000110101000000","000000000000011000011010100000","000000000000000010011100010000","000000000000000000001111101000","000000000000000000000011001000","000000000000000000000001100100",
"000000000000000000000000001010","000000000000000000000000000010","000000000000000000000000000001","000000000000000000000000000000","000000000000000000000000000000","000000000000000000000000000000");

subtype sINT is integer range -2**15 to 2**30-1;
signal S,M: sINT:=0;
signal AN:std_logic_vector(29 downto 0);
begin

process (CLKIN)
begin
if CLKIN'event and CLKIN = '1' then
DOP<=ROM(CONV_INTEGER(ROMDIR));
DOA<=AN;
end if;
end process;

M <=conv_integer(ROM(CONV_INTEGER(ROMDIR)));
S<= M/2;
AN <= CONV_STD_LOGIC_VECTOR(S,30);
end behavioral;
```

### 10.1.5. MÓDULO DE CONEXIÓN DE LOS COMPONENTES.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library work;
use work.componentes.all;
entity testresolmemo is

Port (
ROMDIR : in  STD_LOGIC_VECTOR(3 downto 0);
RESOL: in  STD_LOGIC_VECTOR(2 downto 0);
O : out  STD_LOGIC;
OB : out  STD_LOGIC;
CLKIN: in  STD_LOGIC;
OCE : in  STD_LOGIC;
SR : in  STD_LOGIC;
RSTfast : in  STD_LOGIC;
PWM_RST : in  STD_LOGIC;
PWM_LOAD : in  STD_LOGIC;
PWM_OUT: out  STD_LOGIC;
RSTPLL : in  STD_LOGIC
);

end testresolmemo;

architecture Behavioral of testresolmemo is
signal CLKFBIOUT:std_logic;
signal OUT0:std_logic;
signal OUT1:std_logic;
signal CLKFAST:std_logic;
signal CLKDIV:std_logic;
signal CLKINPLL:std_logic;
signal OQ:std_logic;
signal DOP:std_logic_vector(29 downto 0); --PER
signal DOA:std_logic_vector(29 downto 0);--ANCHO
signal AVCICLO:std_logic;
signal AVRESOL:std_logic;
signal PATR:std_logic_vector(5 downto 0);
signal PATRON:std_logic_vector(5 downto 0);
signal PATRONciclo:std_logic_vector(5 downto 0);
signal SAL_ANCHO: std_logic_vector(29 downto 0);
signal SAL_PER:std_logic_vector(29 downto 0);
signal LOCKED:std_logic;
```

```
begin

u0:ibufg1 port map(CLKIN, CLKINPLL);
u1:pll1 port map(CLKINPLL, CLKFBIOUT, RSTPLL, CLKFBIOUT, OUT0, OUT1,
open, open, open, open, LOCKED);
u2:bufg1 port map(OUT0, CLKFAST);
u3:bufg2 port map(OUT1, CLKDIV);
u4:memoria port map(CLKDIV, ROMDIR, DOP, DOA);
u5:ffin port map(DOA, CLKDIV, '1', '0' , SAL_ANCHO);
--ENTRADA, CLKDIV, CE, RST, SALIDA
u6:ffinperiodo port map(DOP, CLKDIV, '1', '0' , SAL_PER);
--ENTRADA, CLKDIV, CE, RST, SALIDA
u7:gpwm port map(CLKDIV, SAL_ANCHO, SAL_PER, PWM_RST,
PWM_LOAD, AVCICLO, AVRESOL, PATRONciclo, PWM_OUT);
u8:fast port map(PATRONciclo, AVCICLO, AVRESOL, RESOL, CLKDIV, RSTfast,
PATRON);
u9:ffout port map('1', CLKDIV, '0', PATRON, PATR);
u10:oserde port map(OQ, open, open, open, CLKFAST, CLKDIV, PATR(0), PATR(1),
PATR(2), PATR(3), PATR(4), PATR(5), OCE, '0', '0', '0', SR, '0', '0', '0', '0', '0');
u11:obufds1 port map(O, OB, OQ);
end Behavioral;
```

### 10.1.6. TOP.

El módulo top es el que contiene todos los componentes y el archivo *UCF* con las limitaciones de tiempo y configuración de los pines. El TOP es el programa preparado para la implementación del diseño en el dispositivo *FPGA.*

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

entity toptestresol is
Port (
CLKIN: in  STD_LOGIC;
O : out  STD_LOGIC;
OB : out  STD_LOGIC;
PWM_OUT: out  STD_LOGIC----------PWM
);
end      toptestresol;
architecture Behavioral of toptestresol is

component testresol
Port (
O : out  STD_LOGIC;
OB : out  STD_LOGIC;
CLKIN: in  STD_LOGIC;
OCE : in  STD_LOGIC;
SR : in  STD_LOGIC;
RESOL : in  STD_LOGIC_vector( 2 downto 0);
RSTfast : in  STD_LOGIC;
PWM_RST : in  STD_LOGIC;----------PWM
PWM_LOAD : in  STD_LOGIC;----------PWM
PWM_OUT: out  STD_LOGIC;----------PWM
RSTPLL : in  STD_LOGIC;----------Pll
ANCHO: in std_logic_vector(29 downto 0);
PERIODO: in std_logic_vector(29 downto 0) ;
end component;

begin
u1:testresol port map(O, OB, CLKIN, '1', '0', "001", '0', '0', '1', PWM_OUT, '0',
"00000000000000000000000000000001", "00000000000000000000000000000010");
end Behavioral;
```

# 11. BIBLIOGRAFIA.


[1] Jhonny Posada Contreras. (2005*). Introducción a las técnicas de modulación, 2005.* México. (Universidad autónoma de México) <http://redalyc.uaemex.mx/src/inicio/ArtPdfRed.jsp?iCve=47802507>

[2] Actel Corporation. *Using FPGA for digital PLL aplications.* (1996*).* <http://www.actel.com/documents/DigitalPLL_AN.pdf > (abril de 1996).

[3] Nasa office of logic design. *FPGA high speed and signal quality*. <http://klabs.org/richcontent/fpga_content/pages/notes/high_speed_signal_quality.htm> (3 de febrero de 2010).

[4] Instituto de investigación y desarrollo de telecomunicaciones. *Sintetizador Digital Directo utilizando Difuminado de Fase.* <http://www.lacetel.cu/Productos.htm > (29 de abril de 2011).

[5] Bryan H. Fletche. (2005) *FPGA embedded processors.* San Diego California: Memec. <http://www.xilinx.com/products/design_resources/proc_central/resource/ETP-367paper.pdf>

[6] Hube Pages. *Embedded systems development board: FPGA vs Micro controllers.* (2011). <http://hubpages.com/hub/Embedded-Systems-Choosing-the-right-development-board>. (2011).

[7] Scott Hauck, Member, IEEE, Matthew M. Hosler, and Thomas W. Fry. (2000). *High performance carry chance for FPGA.* <http://home.engineering.iastate.edu/~zambreno/classes/cpre583/documents/HauHos00A.pdf > (2 de abril del 2000).

[8] A. Mansouri, A. Ahaitouf, and F. Abdi. (2009). *An efficient architecture and FPGA implementation of high speed for Wavelet filter*. <http://paper.ijcsns.org/07_book/200903/20090307.pdf > (3 de marzo de 2009)

[9] IEEE Std. 1076, 2000 Edition, "*IEEE Standard VHDL Language Reference Manual*".

[10] *Xilinx university program XUPV5-LX110T development system.* <http://www.xilinx.com/univ/xupv5-lx110t.htm>

[11] EETIMES DESIGN. CHRISTIAN WEBER, JINJIN HE, LIZHI CHARLIE ZHONG, AND HUAPING LIU. *HIGH SPEED SERDES.*(2011). <HTTP://WWW.EETIMES.COM/DESIGN/EMBEDDED/4212312/MULTI BAND-ARCHITECTURE-FOR-HIGH-SPEED-SERDES>

[12] Stauffer, D.R., Mechler, J.T., Sorna, M.A., Dramstad, K., Ogilvie, C.R., Mohammad, A., Rockrohr, J.D. *High speed Serdes devices and applications.* XIV, 490 p. 20 illus. (2009). <http://www.springer.com/engineering/circuits+%26+systems/book/978-0-387-79833-2>

[13] *Serdes FPGA system simulation using the Xilinx design kit.* <http://www.ansoft.com/si/pdf/SERDES_FPGA_system_simulation_usingthe_Xilinx_design_kit_FINAL.pdf>

[14] *Designing with VHDL*. <http://www.xilinx.com/training/languages/designing-with-vhdl.htm>

[15] Fernando Pardo Carpio. (2003). *VHDL - Lenguaje para descripción y modelado de circuitos*. Madrid: Ra-Ma, Librería y Editorial Microinformática.

[16] *Virtex-5 FPGA datasheet: dc and switching characteristics*. <http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf>

[17] *Virtex-5 Family Overview*. <http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf>

[18] *Virtex-5 FPGA user guide.* <http://www.xilinx.com/support/documentation/user_guides/ug190.pdf>

# 12. GLOSARIO.

| | |
|---|---|
| FPGA | *Field Programmable Gate Array.* |
| VHDL | *Very High Speed Integrated Circuit.* |
| PAL | *Programmable Array Logic.* |
| CPLD | *Complex Programmable Logic Device.* |
| ASIC | *Application Specific Integrated Circuit.* |
| ASMBL | *Advanced Silicon Modular Block.* |
| FIFO | *First In, First Out.* |
| RAM | *Ramdom Acces Memory.* |
| DCM | *Digital Clock Manager.* |
| PLL | *Phase-locked loop.* |
| MAC | *Media Access Controllers.* |
| DARPA | *Defense Advanced Research Projects Agency.* |
| IEEE | *Institute of Electrical and Electronics Engineer.* |
| PWM | *Pulse Width Modulation.* |
| SSI | *Small Scale Integration.* |
| MSI | *Médium Scale Integration* |
| VLSI | *Very Large Scale Integration.* |
| USB | *Universal Serial Bus.* |
| CMT | *Clock Management Tile.* |
| PFD | *Phase-Frequency Detector.* |
| LF | *Loop Filter.* |
| VCO | *voltage-controlled oscillator.* |
| CP | *Charge pump.* |
| DCI | *Digitally Controlled Impedance.* |

| | |
|---|---|
| IOB | *Imput/Ouput Block.* |
| LVDS | *Low-voltage differential signaling.* |
| CLB | *Configurable Logic Block.* |
| SDR | *Single Data Rate.* |
| DDR | *Double Data Rate.* |
| XST | *Xilinx Synthesis Technology.* |
| NCD | *Native Circuit Description.* |
| NGD | *Native generic database.* |
| HDL | *Hardware Description Language.* |
| FSM | *Finite State Machines.* |
| XCF | *XST Constraints File.* |
| ASCII | *American Standard Code for Information Interchange.* |
| UCF | *User Constraints File.* |
| SDF | *Standard Delay Format.* |