epsc Escola Politècnica Superior de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL DE FI DE CARRERA

TÍTULO: Diseño e implementación con FPGA de un demodulador para comunicaciones digitales.

AUTOR: Juan Antonio Guerrero Balmori

DIRECTOR: Gabriel Montoro López

FECHA: 17 de Julio del 2006

Título: Diseño e implementación con FPGA de un demodulador para comunicaciones digitales.

Autor: Juan Antonio Guerrero Balmori

Director: Gabriel Montoro López

Fecha: 17 de Julio del 2006

Resumen

El titulo del proyecto es "Diseño e implementación con FPGA de una demodulador para comunicaciones digitales" y como su nombre indica su objetivo básico es el diseño e implementación de un demodulador que se ajuste a algún tipo de comunicación digital. Pero además, este proyecto lo que pretende es que el estudiante adquiera conocimientos sobre diseño de dispositivos lógicos programables, y sobretodo familiarizarse con el uso de la herramienta que se utiliza para trabajar con estos dispositivos, el código VHDL.

De esta manera, además de refrescar conocimientos sobre demodulación, filtrado, muestreo y electrónica digital se han aprendido otros nuevos como son: la programación en VHDL y trabajar con dispositivos programables como es la FPGAs.

Este proyecto esta formado por cuatro capítulos en los que: se hace una pequeña introducción a los PLD, se explican las bases teóricas para el diseño del demodulador y se muestra como se ha implementado. A continuación, resumiremos estos capítulos.

En el primer capitulo haremos una breve explicación de lo que es un PLD (*Programable Logia Device*) y de los distintos tipos que existen. Este capitulo no pretende dar información sobre los PLD para después poder elegir el modelo que mas nos convenga (ya que el proyecto ya cuenta con una FPGA determinada), sino que da información a modo de introducción para poder ver las diferencias y ventajas que tendrá nuestra FPGA respecto a otros dispositivos PLD, además de dar a conocer, a quien no lo sepa, lo que es un PLD.

En el segundo capitulo veremos los distintos formatos que existen para trabajar con números binarios con signo, y los métodos para pasar de un formato a otro.

Para poder entender el funcionamiento de parte del código que forma el demodulador digital, hemos de conocer como funcionan algunas de las operaciones básicas de la aritmética binaria. Por lo tanto, en este capitulo

también veremos como funcionan las operaciones aritméticas que hemos utilizado.

En el tercer capitulo haremos una presentación de los distintos bloques que formarán el demodulador. De es esta manera, veremos su función dentro del circuito y sus condiciones de diseño a modo de introducción al siguiente capitulo, en el cual se usaran lo conceptos de diseño aquí visto para realizar la implementación.

En primer lugar haremos una introducción al concepto de IF-sampling y down-converter en los que se basará el funcionamiento del demodulador. Luego veremos el esquema de bloques de los distintos elementos que compondrán el demodulador.

Una vez visto el diagrama de bloques comenzaremos explicando como funciona el mezclador complejo, el cual se encargará de multiplicar la señal de entrada por un coseno y un seno para extraer las componentes en fase y cuadratura respectivamente. Después analizaremos como funciona la DDS (Direct Digital Syntetizer) que es la encargada de genera las dos formas de onda anteriores para pasárselas al mezclador. Estos dos bloques constituirán lo que se denomina DDC (Digital Down Converter) que será la primera parte del demodulador.

Una vez visto el DDC se hará una pequeña introducción del bloque filtrado, para pasar luego a explicar mas detalladamente como funciona cada uno de los filtros. Primero explicaremos como funciona el filtro CIC, el cual se encargará de hacer un diezmado de las componentes en fase y en cuadratura para bajar su frecuencia y así reducir la carga computacional del siguiente filtro. Esta parte también influirá en la parte de IF-sampling.

El siguiente y ultimo filtro que se explicará en la parte de filtrado, es el filtro FIR. Este filtro se encargará de realizar un filtrado paso-bajos de las componentes en fase y cuadratura ya diezmadas, de manera que solo quede la banda útil, es decir la banda base, eliminando las interferencias producidas por las imágenes generadas en la conversión A/D y el filtro CIC.

Para acabar este capitulo, nos faltará ver el ultimo de los bloques que conformarán nuestro demodulador, es decir, el decisor de QAM de 16 símbolos o 16QAM. Este bloque es muy sencillo ya que, dependiendo del valor que tengan las componentes I y Q, ya diezmadas y filtradas, la salida tendrá un valor u otro, el cual dependerá de cómo esta distribuida la constelación de la modulación.

En el cuarto y ultimo capitulo se realiza la implementación del demodulador, es decir, se explicarán los siguientes puntos:

- Numero de entradas y salidas.
- Numero de bits de cada una de las entradas y salidas.
- Calculo de factores de diseño.
- Funcionamiento del código (basándose en lo visto en el capitulo

anterior).

A lo largo del diseño del demodulador digital se tienen siempre en mente consideraciones de:

- Costo de implementación (número de compuertas lógicas).
- Requerimientos de tiempo de computación.
- Rango dinámico de la representación o numero de bits.
- Relación señal a ruido de las señales digitales.

Un dato importante a tener en cuenta, a la hora de tomar estas consideraciones, es el modelo de FPGA que vamos a utilizar. Se trata de una Virtex 4 modelo XC4VSX35 y encapsulado FFG668.

Todo el proceso de implementación se ha realizado con el programa ISE 7.1 proporcionado por la misma Xilinx. Este programa nos permite, mediante código VHDL, implementar los distintos bloques que componen un nuestro dispositivo, permitiéndonos sintetizar el código comprobando en todo momento que este pueda funcionar en la FPGA. Además, incorpora una herramienta llamada Place & Route que nos permite asignar las entradas y salidas de nuestro código a los distintos pines que tenga la FPGA que utilicemos.

De esta manera a lo largo de la implementación también veremos el coste que van a tener los distintos bloques del demodulador a la hora de consumir los recursos de la FPGA.

Por ultimo, antes de los anexos, encontraremos el apartado de referencias. En este apartado se mostrarán tanto los libros que se han utilizado como las direcciones de las paginas Web de las cuales se ha extraído información para realizar el proyecto.

Y ya para acabar, y como complemento al proyecto, encontramos los anexos. En ellos encontraremos el código programado de cada uno de los bloques y librerías que forman nuestro demodulador, además, encontraremos información sobre la FPGA utilizada (virtex 4), extraída de los data-sheet de Xilinx.

Title: Demodulator for digital communications design and implementation with FPGA

Author: Juan Antonio Guerrero Balmori

Director: Gabriel Montoro López

Date: July, 17th 2006

Overview

The title of the project is "Demodulator for digital comunications design and implementation with FPGA" and its basic objective is the design and implementation of a demodulator which that perform some digital comunication type. But this project also pretends student takes knowledge about programable logic devices design and to familiarize with the use of a tools that is used to work with this devices, the VHDL code. In this way, besides to remember knowledge of demodulation, filtered, sampling and digital electronic, have known another new like VHDL programation and to work with programable devices as FPGAs.

This project is formed by 4 chapters in which are made a little introduction of PLD, explain the theoric bases of demodulation design and is shown how is implemented. Next we'll summarize these chapters.

In chapter 1 are made a short explanation about PLD (Program Logic Device) and the different types that exist. This chapter not pretend to tell information about PLD to choose model more suitable (the project has a fixed FPGA) but tell information like a introduction to see the differences and advantages of our FPGA with regard to other PLD devices, besides to know what is a PLD, no-one to know.

In chapter 2 will see the different formats that exists to work with binary numbers with sign and methods to pass at other formats.

To know all about function of a part of code that forms digital demodulator we must know how to work some of basic operations of binary arithmetic. So in this chapter will see too how to work arithmetic operations used.

In chapter 3 will make a presentation of the different blocks that will perform the demodulator. In this way we will see its function inside circuit and the design conditions like introductions to the next chapter in which will use the concepts of design to realize the implementation.

First will make an introduction to IF-sampling and down converter concept in

which based the function of demodulator. Then, will see the block diagram of different elements that perform the demodulator.

Once seen the block diagram, will tell how to run the complex mixed. This will multiply the input signal by a cosine and sine to take out the in phase and cuadrature components respectively. After that will analyze how to work DDS (Direct Digital Syntetyzer) that takes charge of to generate the two wave forms previous to pass at mixer. This two blocks will make up a DDC (Digital Down Converter) that will be the first part of demodulator.

After seen DDC will make a little introduction of filtering block for pass to explain in detail how work CIC filter, which will take care of make a decimate of the in phase and a cuadrature components to low its rate and, in this way, to reduce computational charge of the next filter. This part also will affect in IF-sampling part.

The last filter that will be telled in the filtered part, is the FIR filter. This filter will take care of realize a low-pass filtering of the in phase and cuadrature decimated components in order to that only remain base band, removing the interference produced by the images of A/D conversion and CIC filter.

In order to finish this chapter, we must see the last of the blocks that will perform our demodulator, this is the 16 symbol QAM or 16QAM decisor. This block is very simply because depending of the value of I and Q components, decimated and filtered, the exit will have a value or another. This value will depends how is distributed the modulation constellation.

In the 4th and last chapter is realized the demodulator implementation. The next points are explained:

- Number of inputs and outputs.
- Number of bits of each inputs and outputs.
- Design factors calculation.
- Function of code (is based in the chapter 3).

In the design of digital demodulator always remember the next considerations:

- Cost of implementation (number of logic gates).
- Request of computation time.
- Dynamic rank of representation or number of bits.
- SNR of digital signals.

An important fact when we take this considerations is FPGA model that we used to. This is a virtex 4 XC4VSX35 model with an FFG668 package.

All the process of implementation is realized with ISE 7.1 program supplied by Xilinx. This program allows, with VHDL code, to implement the different blocks or our device. This permit sintetize the code checking all time that this can run in FPGA. Besides incorporate a tool called Place & Route. This tool allows us to assign the inputs and outputs of our code to different pins of FPGA that we

used.

In this way, all along of implementation also we'll see the cost of the different blocks of the demodulator in time to consume the FPGA resorts.

In the end, before the annexes, we will find the references apart. In this apart will show the books that we used and the address of web pages where are found information to make the project.

With project complement there are the annexes. In the annexes we'll found the programmed code of each block and libraries that form our demodulator. Besides we'll found information about FPGA used (virtex 4), take out of Xilinx data-sheet.

ÍNDICE

INTRODUCCIÓN 1

CAPITULO 1- INTRODUCCIÓN A LOS PLDS 3

1.1. Introducción 3

1.2. Estructura básica de un PLD 3
1.2.1. ASIC 3
1.2.2. PROM 4
1.2.3. PAL 4
1.2.4. GAL 4
1.2.5. PLA 4
1.2.6. PLDs complejos (FPGA) 5

CAPITULO 2 - FORMATO Y ARITMÉTICA BINARIA 7

2.1. Introducción 7

2.2. Números con signo 7
2.2.1. El bit de signo 7
2.2.2. Sistema signo-magnitud 7
2.2.3. Sistema del complemento a 1 8
2.2.4. Obtención de un complemento a 1 de un número binario 8
2.2.5. Sistema de complemento a 2 9
2.2.6. Obtención del complemento a 2 de un número binario 9
2.2.7. Rango de representación de los números con signo 10

2.3. Conversión decimal-binario 10

2.4. Aritmética binaria 11
2.4.1. Suma binaria 11
2.4.2. Suma de números con signo 13
2.4.3. Multiplicación binaria 15
2.4.4. Multiplicación de números con signo 15

2.5. CONCLUSIONES 18

CAPITULO 3 – DISEÑO TEÓRICO 21

3.1. Introducción 21

3.2. Down conversion y IF-sampling 21

3.3. Diagrama de bloques 22

3.4. Mezclador complejo 23

3.5. DDS 24
3.5.1. Frecuencia de salida 26
3.5.2. Resolución en frecuencia 26
3.5.3. Incremento de fase 26

3.5.4. Pureza espectral 26

3.6. Filtrado 27

3.7. Filtro CIC 28
3.7.1. Estructura y diseño 29
3.7.2. Características frecuenciales 31
3.7.3. AUMENTO DE BITS 34

3.8. FIR 34

3.9. Decisor 16QAM 36

CAPITULO 4 – IMPLEMENTACIÓN EN FPGA 39

4.1. Introducción 39

4.2. Mezclador 40

4.3. DDS 41

4.4. CIC 44

4.5. FIR 46

4.6. Decisor 16QAM 48

4.7. Demodulador 49

CONCLUSIONES 51

REFERENCIAS 53

ANEXO 1 – CÓDIGO DEL MEZCLADOR 55

ANEXO 2 – CÓDIGO DE LA DDS 57

ANEXO 3 – CÓDIGO DEL FILTRO CIC 61

ANEXO 4 - CÓDIGO DEL FILTRO FIR 65

ANEXO 5 – CÓDIGO DEL DECISOR DE 16QAM 69

ANEXO 6 – CÓDIGO DE LA LIBRERÍA DE FUNCIONES 71

ANEXO 7 – CÓDIGO DEL DEMODULADOR 77

ANEXO 8 – DIAGRAMA DE BLOQUES DEL DEMODULADOR 81

ANEXO 9 – COMPARATIVAS VIRTEX 4 83

INTRODUCCIÓN

El titulo del proyecto es "Diseño e implementación con FPGA de una demodulador para comunicaciones digitales" y como su nombre indica su objetivo básico es el diseño e implementación de un demodulador que se ajuste a algún tipo de comunicación digital. Pero además, este proyecto lo que pretende es que el estudiante adquiera conocimientos sobre diseño de dispositivos lógicos programables, y sobretodo familiarizarse con el uso de la herramienta que se utiliza para trabajar con estos dispositivos, el código VHDL[1].

De esta manera, además de refrescar conocimientos sobre demodulación, filtrado, muestreo y electrónica digital se han aprendido otros nuevos como son: la programación en VHDL y trabajar con dispositivos programables como es la FPGAs.

Este proyecto esta formado por cuatro capítulos en los que: se hace una pequeña introducción a los PLD, se explican las bases teóricas para el diseño del demodulador y se muestra como se ha implementado. A continuación, resumiremos estos capítulos.

Capitulo 1- Introducción a los PLDs

1.1. Introducción

En este capitulo [2] haremos una breve explicación de lo que es un PLD (*Programable Logia Device*) y de los distintos tipos que existen. Este capitulo no pretende dar información sobre los PLD para después poder elegir el modelo que mas nos convenga (ya que el proyecto ya cuenta con una FPGA determinada), sino que da información a modo de introducción para poder ver las diferencias y ventajas que tendrá nuestra FPGA respecto a otros dispositivos PLD, además de dar a conocer, a quien no lo sepa, lo que es un PLD.

1.2. Estructura básica de un PLD

Los dispositivos lógicos programables, como su propio nombre implica, son una familia de componentes que contienen conjuntos de elementos lógicos (AND, OR, NOT, LATCH, FLIP-FLOP) que pueden configurarse en cualquier función lógica que el usuario desee y que el componente soporte.

Un dispositivo programable por el usuario es aquel que contiene una arquitectura general pre-definida en la que el usuario puede programar el diseño final del dispositivo empleando un conjunto de herramientas de desarrollo. Las arquitecturas generales pueden variar pero normalmente consisten en una o más matrices de puertas AND y OR para implementar funciones lógicas. Muchos dispositivos también contienen combinaciones de flip-flops y latches que pueden usarse como elementos de almacenaje para entrada y salida de un dispositivo. Los dispositivos más complejos contienen macrocélulas. Las macrocélulas permite al usuario configurar el tipo de entradas y salidas necesarias en el diseño

Hay varias clases de dispositivos lógicos programables: ASICs, FPGAs, PLAs, PROMs, PALs, GALs, y PLDs complejos.

1.2.1. ASIC

ASIC significa Circuitos Integrados de Aplicación Específica y son dispositivos definibles por el usuario. Los ASICs, al contrario que otros dispositivos, pueden contener funciones analógicas, digitales, y combinaciones de ambas. En general, son programables mediante máscara y no programables por el usuario. Esto significa que los fabricantes configurarán el dispositivo según las especificaciones del usuario. Se usan para combinar una gran cantidad de funciones lógicas en un dispositivo. Sin embargo, estos dispositivos tienen un

costo inicial alto, por lo tanto se usan principalmente cuando es necesario una gran cantidad.

1.2.2. PROM

Las PROM son memorias programables de sólo lectura. Aunque el nombre no implica la lógica programable, las PROM, son de hecho lógicas. La arquitectura de la mayoría de las PROM consiste generalmente en un número fijo de términos AND que alimenta una matriz programable OR. Se usan principalmente para decodificar las combinaciones de entrada en funciones de salida.

1.2.3. PAL

Las PAL son dispositivos de matriz programable. La arquitectura interna consiste en términos AND programables que alimentan términos OR fijos. Todas las entradas a la matriz pueden ser combinadas mediante AND entre si, pero los términos AND específicos se dedican a términos OR específicos. Las PAL tienen una arquitectura muy popular y son probablemente el tipo de dispositivo programable por usuario más empleado. Si un dispositivo contiene macrocélulas, comúnmente tendrá una arquitectura PAL. Las macrocélulas típicas pueden programarse como entradas, salidas, o entrada/salida (e/s) usando una habilitación tri-estado. Normalmente tienen registros de salida que pueden usarse o no conjuntamente con el pin de e/s asociado. Otras macrocélulas tienen más de un registro, varios tipos de retroalimentación en las matrices, y ocasionalmente realimentación entre macrocélulas.

1.2.4. GAL

Las GAL son dispositivos de matriz lógica genérica. Están diseñados para emular muchas PAL pensadas para el uso de macrocélulas. Si un usuario tiene un diseño que se implementa usando varias PAL comunes, puede configurarlas en la misma GAL para emular cada de uno de los otros dispositivos. Esto reducirá el número de dispositivos diferentes en existencia y aumenta la cantidad comprada. Comúnmente, una cantidad grande del mismo dispositivo debería rebajar el costo individual del dispositivo. Estos dispositivos también son eléctricamente borrables, lo que los hace muy útiles para los ingenieros de diseño.

1.2.5. PLA

Las PLA son matrices lógicas programables. Estos dispositivos contienen ambos términos AND y OR programables lo que permite a cualquier término

AND alimentar cualquier término OR. Las PLA probablemente tienen la mayor flexibilidad frente a otros dispositivos con respecto a la lógica funcional. Normalmente poseen realimentación desde la matriz OR hacia la matriz AND que puede usarse para implementar máquinas de estado asíncronas. La mayoría de las máquinas de estado, sin embargo, se implementan como máquinas síncronas. Con esta perspectiva, los fabricantes crearon un tipo de PLA denominado Secuencial (*Sequencer*) que posee registros de realimentación desde la salida de la matriz OR hacia la matriz AND.

1.2.6. PLDs complejos (FPGA)

Los PLDs complejos son lo que el nombre implica, Dispositivos Complejos de Lógica Programable. Se consideran PAL muy grandes que tienen algunas características de las PLA. La arquitectura básica es muy parecida a la PAL con la capacidad para aumentar la cantidad de términos AND para cualquier término OR fijo. Esto se puede realizar quitando términos AND adyacentes o empleando términos AND desde una matriz expandida. Esto permite que cualquier diseño pueda ser implementado dentro de estos dispositivos.
FPGA

Las FPGA son Campos de Matrices de Puertas Programables. Simplemente son matrices de puertas eléctricamente programables que contienen múltiples niveles de lógica. Las FPGA se caracterizan por altas densidades de puerta, alto rendimiento, un número grande de entradas y salidas definibles por el usuario, un esquema de interconexión flexible, y un entorno de diseño similar al de matriz de puertas. No están limitadas a la típica matriz AND-OR. Por contra, contienen una matriz interna configurable de relojes lógicos (CLBs) y un anillo de circunvalación de bloques de e/s (IOBs).

Cada CLB contiene lógica programable combinacional y registros de almacenamiento. La sección de lógica combinacional es capaz de implementar cualquier función booleana de sus variables de entrada.

Cada IOB puede programarse independientemente para ser una entrada, y salida con control tri-estado o un pin bidireccional. También contiene flip-flops que pueden usarse como buffers de entrada y salida. Los recursos de interconexión son una red de líneas que corren horizontalmente y verticalmente las filas y columnas entre el CLBS.

Los interruptores programables conectan las entradas y salidas de IOBS y CLBS a líneas cercanas. Las líneas largas recorren la anchura o longitud entera del dispositivo, estableciendo intercambios para proporcionar una distribución de señales críticas con la mínima demora o distorsión.

Los diseñadores que usan FPGAs pueden definir funciones lógicas en un circuito y revisar estas funciones como sea necesario. Así, las FPGAs pueden diseñarse y verificarse en unos días, a diferencia de las varias semanas necesarias para las matrices de puerta programables.

Capitulo 2 - FORMATO Y ARITMÉTICA BINARIA

2.1. Introducción

En este capitulo [3] veremos los distintos formatos que existen para trabajar con números binarios con signo, y los métodos para pasar de un formato a otro. Además, analizaremos las distintas operaciones aritméticas binarias que hemos utilizado para desarrollar parte del código.

2.2. Números con signo

Un número binario con signo queda determinado por su magnitud y su signo. El signo indica si un número es positivo o negativo, y la magnitud es el valor del número. Existen tres formatos binarios para representar los números enteros con signo: signo-magnitud, complemento a 1 y complemento a 2, de los cuales solo utilizaremos el primero y el tercero.

2.2.1. El bit de signo

El bit más a la izquierda en un número binario con signo es el bit de signo, que indica si el número es positivo o negativo. Se utiliza un 0 para el signo positivo y un 1 para el signo negativo.

2.2.2. Sistema signo-magnitud

Cuando un número binario con signo se representa en formato signo-magnitud, el bit más a la izquierda es el bit de signo y los bits restantes son los bits de magnitud. Los bits de magnitud son el número binario real (no complementado) tanto para los números positivos como para los negativos.

Por ejemplo, el numero decimal +25 se expresa, utilizando signo-magnitud, como un número binario con signo de 8 bits como se muestra en la **Fig. 2.1**.

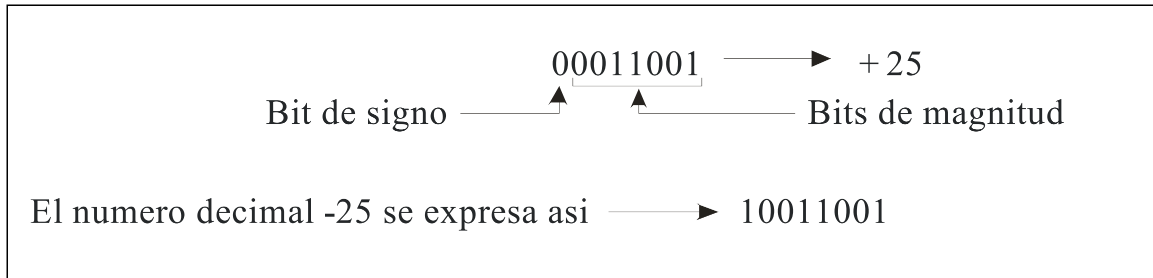


Fig. 2.1 Sistema de números binarios negativos signo-magnitud

Obsérvese que la única diferencia entre +25 y -25 es el bit de signo, ya que los bits de magnitud representan el binario real tanto para números positivos como negativos.

En el sistema signo-magnitud, un número negativo tiene los bits de magnitud que el correspondiente número positivo, pero el bit de signo es un 1 en lugar de un cero.

2.2.3. Sistema del complemento a 1

Los números positivos en el sistema del complemento a 1 se representan de la misma forma que los números positivos en el formato signo-magnitud. Sin embargo, los números negativos son el complemento a 1 del correspondiente número positivo.

2.2.4. Obtención de un complemento a 1 de un número binario

El complemento a 1 de un número binario se obtiene cambiando todos los 1s por 0s, y todos los 0s por 1s, como se ilustra a continuación:

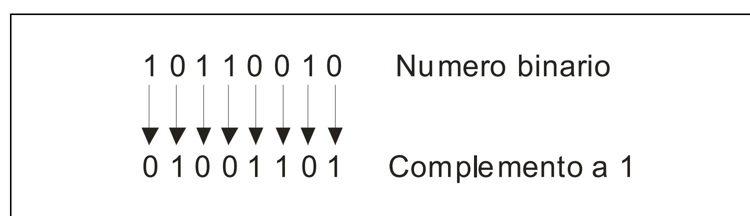


Fig. 2.2 Conversión a complemento a 1

Por ejemplo, usando ocho bits, el número decimal -25 se expresa como el complemento a 1 de +25 (00011001), es decir, 11100110.

La forma más sencilla de obtener el complemento a 1 de un número binario mediante un circuito digital es utilizar inversores en paralelo (circuitos NOT), como muestra la **Fig. 2.3** para un número binario de 8 bits.

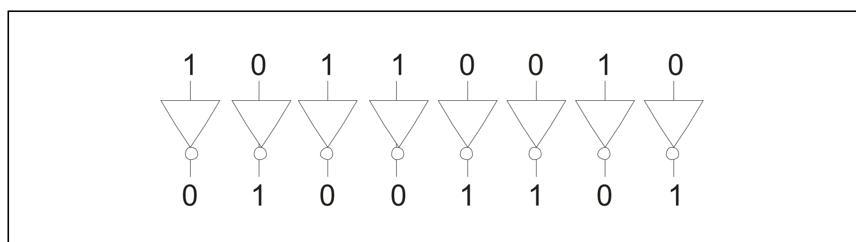


Fig. 2.3 Conversión a complemento a 1

2.2.5. Sistema de complemento a 2

Los números positivos en el sistema del complemento a 2 se representan de la misma forma que en los sistemas de complemento a 1 y de signo-magnitud. Los números negativos son el complemento a 2 del correspondiente número positivo. De nuevo, usando ocho bits, tomamos -25 y lo expresamos como el complemento a 2 de +25 (00011001), es decir, 11100111.

2.2.6. Obtención del complemento a 2 de un número binario

El complemento a 2 de un número binario se obtiene sumando 1 al LSB del complemento a 1.

$$\text{Complemento a 2} = \text{Complemento a 1} + 1 \qquad \textbf{(2.1)}$$

Un método alternativo para obtener el complemento a 2 de un número binario es el siguiente:

- Paso 1. Se empieza por la derecho con el LSB y se escriben los bits como están hasta encontrar el primer 1, incluido éste.

- Paso 2. Se calcula el complemento a 1 de bits restantes.

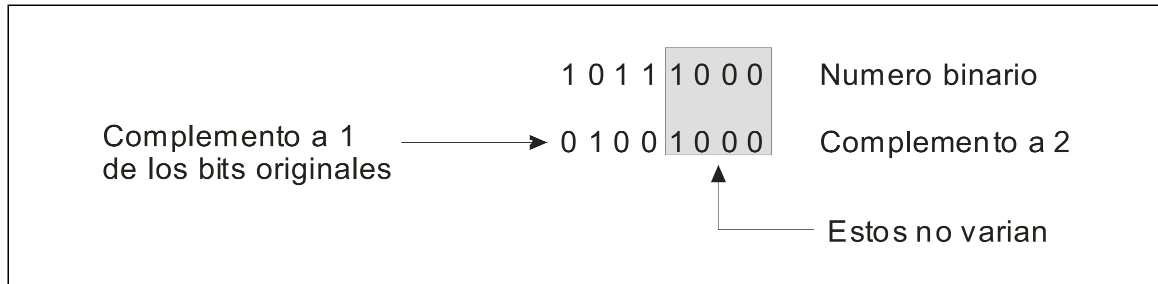


Fig. 2.4 Conversión a complemento a 2

Para convertir un número en complemento a 1 o en complemento a 2 al formato binario real (no complementado) se usan los dos mismo procedimientos que acabamos de describir. Para convertir el complemento a 1 al binario real, se invierten todos los bits. Para convertir el complemento a 2 al binario real, primero se obtienen el complemento a 1 y se suma 1 al resultado obtenido.

2.2.7. Rango de representación de los números con signo

Para los ejemplos hemos utilizado números de 8 bits, lo cual nos permite representar 256 números diferentes. La fórmula par calcular el número de combinaciones diferentes de *n* bits es la siguiente:

$$\text{Nº total de combinaciones} = 2^n \qquad \textbf{(2.2)}$$

Para los números con signo, el rango de valores para números de n bits es:

$$\text{de } -\left(2^{n-1}\right) \text{ a } +\left(2^{n-1}-1\right) \qquad \textbf{(2.3)}$$

Habiendo en cada caso un bit de signo y *n -1* bits de magnitud. De esta manera, con 8 bits, se puede contar desde -128 hasta +127.

2.3. Conversión decimal-binario

Existen dos métodos para realizar la conversión de un número decimal a binario [ ], el primero es el método de la suma de pesos y el segundo es el método de la división sucesiva por 2. El método que utilizaremos y, por lo

tanto, veremos a continuación es el método de la división sucesiva por 2, ya que es un método sistemático que nos vendrá mejor a la hora de programarlo.

Comenzaremos con el ejemplo de la **Fig. 2.5**, en el cual vemos que, para convertir a binario el número 12, comenzamos dividiendo 12 entre 2. Luego cada cociente resultante se divide por 2 hasta que se obtiene un cociente cuya parte entera es 0. Los restos generados en cada división forman el número binario.

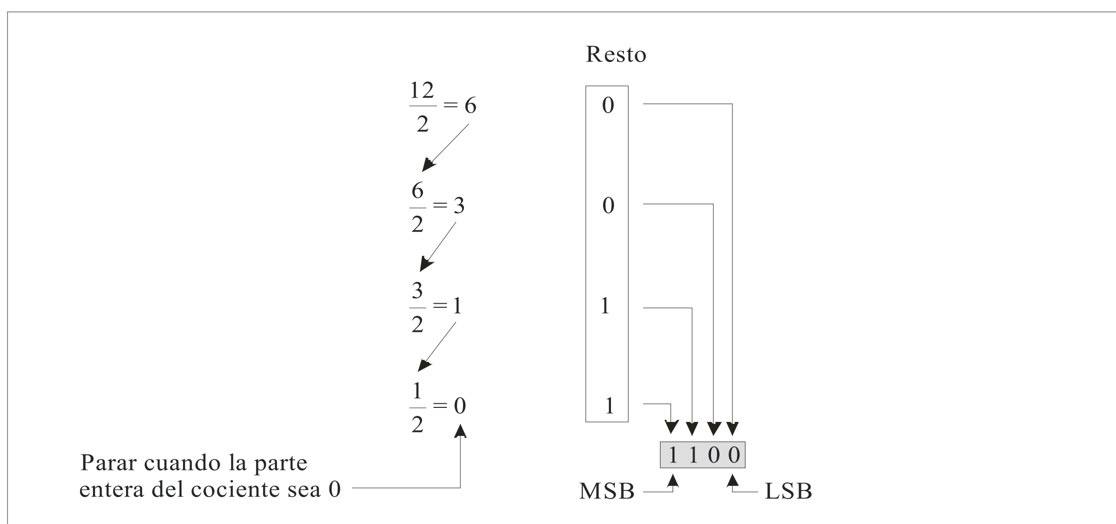


Fig. 2.5 Método de la división sucesiva por 2

El primer resto es el bit menos significativo (LSB) del número binario, y el último resto es el bit mas significativo (MSB).

Por último, faltaría poner el bit de signo correspondiente delante del bit de magnitud mas significativo.

2.4. Aritmética binaria

Para poder entender el funcionamiento de parte del código que forma el demodulador digital, hemos de conocer como funcionan algunas de las operaciones básicas de la aritmética binaria [ ]. Por lo tanto, en este apartado veremos como funcionan las operaciones aritméticas que hemos utilizado.

2.4.1. Suma binaria

Las cuatro reglas básicas para sumar dígitos binarios son:

$0+0=0$	Suma 0 con acarreo 0
$0+1=1$	Suma 1 con acarreo 0
$1+0=1$	Suma 1 con acarreo 0
$1+1=10$	Suma 0 con acarreo 1

Fig. 2.6 Bases de la suma binaria

Se puede observar que las tres primeras reglas dan lugar a un resultado de un único bit, y la cuarta regla, la suma de dos unos, da lugar a 2 en binario (10). Cuando se suman números binarios, teniendo en cuenta la última regla se obtiene en la columna dada la suma 0 y un acarreo de 1 que pasa a la siguiente columna de la izquierda, tal y como se muestra en la siguiente suma de la **Fig. 2.7**.

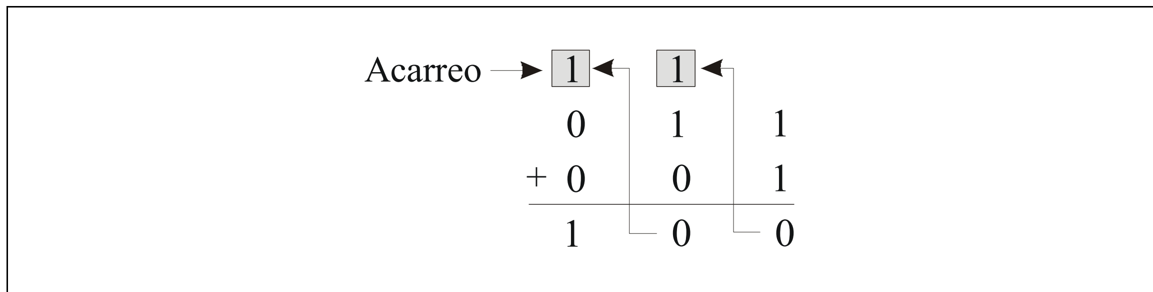


Fig. 2.7 Ejemplo de suma binaria

Podemos ver en la columna de la derecha 1+1=0 con acarreo 1, que pasa a la siguiente columna de la izquierda. En la columna central, 1+1+0=0 con acarreo 1, que pasa a la siguiente columna de la izquierda. Y en la columna de la izquierda, 1+0+0=1.

Cuando existe un acarreo igual a 1, se produce una situación en la que se deben sumar tres bits (un bit de cada uno de los números y un bit de acarreo), tal como se muestra en la **Fig. 2.8**.

Bits de Acarreo

$1+0+0=01$	Suma 1 con acarreo 0
$1+1+0=10$	Suma 0 con acarreo 1
$1+0+1=10$	Suma 0 con acarreo 1
$1+1+1=11$	Suma 1 con acarreo 1

Fig. 2.8 Ejemplo de suma binaria con acarreo

2.4.2. Suma de números con signo

Los dos números en una suma se denominan sumandos. El resultado de la suma. Cuando se suman dos números binarios con signo pueden producirse cuatro casos:

1. Ambos números son positivos.
2. El número positivo es mayor que el negativo en valor absoluto.
3. El número negativo es mayor que el positivo en valor absoluto.
4. Ambos números son negativos.

Veamos caso por caso utilizando números con signo de 8 bits como ejemplos. Como referencia se presentan los números decimales equivalentes.

- Ambos números son positivos:

$$\begin{array}{r} 00000111 \\ +\ 00000100 \\ \hline 00001011 \end{array} \qquad \begin{array}{r} 7 \\ +\ 4 \\ \hline 11 \end{array}$$

Fig. 2.9 Suma de dos números positivos

La suma es positiva y, por tanto, el resultado es un número binario real (no complementado).

- El número positivo es mayor que el número negativo en valor absoluto:

En el caso de que haya un numero negativo, a la hora de hacer la suma, este deberá estar en complemento a 2. De esta manera realizaremos la suma sin tener en cuenta el bit de acarreo final.

$$\text{Acarreo que se descarta} \rightarrow \boxed{1}\ \begin{array}{r} 00001111 \\ +\ 11111010 \\ \hline 00001001 \end{array} \qquad \begin{array}{r} 15 \\ +\ -6 \\ \hline 9 \end{array}$$

Fig. 2.10 Suma de dos números de diferente signo

La suma es positiva y, por tanto, el resultado es un número binario real (no complementado).

- El número negativo es mayor que el número positivo en valor absoluto:

Al igual que en el caso anterior, el numero negativo deberá estar complementado a 2.

$$\begin{array}{r} 00010000 \\ +\ 11101000 \\ \hline 11111000 \end{array} \qquad \begin{array}{r} 16 \\ +\ -24 \\ \hline -8 \end{array}$$

Fig. 2.11 Suma de dos números de diferente signo

La suma es negativa y, por tanto, el resultado está en complemento a 2. De esta manera si lo queremos en otro formato deberemos convertirlo.

- Ambos números son negativos:

En el caso de tener dos números negativos los sumaremos como si fueran positivos, es decir, sin tener en cuenta el bit de signo. Una vez sumados, colocaremos el bit de signo en su posición.

Condición de desbordamiento (overflow): Cuando se suman dos números y el número de bits requerido para representar la suma excede al número de bits de los dos números, se produce un desbordamiento, que se indica mediante un bit de signo incorrecto. Un desbordamiento se puede producir sólo cuando ambos números son positivos o negativos. El siguiente ejemplo con números de 8 bits ilustrará esta condición.

$$\begin{array}{r} 01111101 \\ +\ 00111010 \\ \hline 10110111 \end{array} \qquad \begin{array}{r} 125 \\ +\ 58 \\ \hline 183 \end{array}$$

Signo incorrecto
Magnitud incorrecta

Fig. 2.12 Ejemplo de desbordamiento

En este ejemplo, la suma, 183 requiere ocho bits de magnitud. Puesto que los números tienen siete bits de magnitud (un bit es el bit de signo), se produce un acarreo en el bit de signo que da lugar a la indicación de desbordamiento.

Para evitar el desbordamiento basta con añadir un bit de magnitud más al resultado.

Suma de números de dos en dos: Esto se puede conseguir sumando los dos primeros números, luego se suma el tercer número a la suma de los dos primeros, después se suma el cuarto número al resultado anterior, y así sucesivamente. Así es como las computadoras suman cadenas de números.

2.4.3. Multiplicación binaria

Las cuatro reglas básicas de la multiplicación de bits son las siguientes:

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

Fig. 2.13 Ejemplo de suma binaria con acarreo

La multiplicación con números binarios se realiza de la misma forma que con números decimales. Se realizan los productos parciales, desplazando cada producto parcial una posición a la izquierda, y luego se suman dichos productos.

2.4.4. Multiplicación de números con signo

Los números en una multiplicación se denominan multiplicando, multiplicador y producto. La siguiente multiplicación decimal ilustra estos términos:

8	Multiplicando
x 3	Multiplicador
24	Producto

Fig. 2.14 Multiplicación decimal

La operación de la multiplicación en muchas computadoras se realiza utilizando la suma. La suma directa y los productos parciales son dos métodos básicos para realizar la multiplicación utilizando la suma. En el método de la suma directa, se suma el multiplicando en número de veces igual al multiplicador. En el ejemplo decimal anterior (3 x 8), se suma tres veces el multiplicando: 8 + 8 + 8= 24. La desventaja de este método es que será muy largo cuando el multiplicador sea un número grande. Si, por ejemplo, se multiplica 75 x 350, se debe sumar 75 veces el número 350.

El método de los productos parciales es quizá el más común, ya que es la forma de multiplicar manualmente. El multiplicando se multiplica por cada dígito del multiplicador, empezando por el dígito menos significativo. El resultado de la multiplicación del multiplicando por un dígito del multiplicador se denomina producto parcial. Cada producto parcial se desplaza una posición a la izquierda y, cuando se han obtenido todos los productos parciales, se suman para obtener el producto final.

El signo del producto de una multiplicación depende de los signos del multiplicando y del multiplicador, de acuerdo con las siguientes reglas:

- Si son del mismo signo, el producto es positivo.
- Si son de signo diferente, el producto es negativo.

Cuando dos números binarios se multiplican, ambos números deben estar en formato binario real (no complementado). El método de la suma directa se ilustra en la **Fig. 2.15** sumando los números binarios de dos en dos.

```
   01001101   Primera vez
+  01001101   Segunda vez
-----------
   10011010   Suma parcial
+  01001101   Tercera vez
-----------
   11100111   Suma parcial
+  01001101   Cuarta Vez
-----------
  100110100   Producto
```

Fig. 2.15 Método de la suma directa

Ahora vamos a ver el método de los productos parciales para la multiplicación binaria. Los pasos básicos del procedimiento son los siguientes:

- Paso 1. Determinar si los signos del multiplicando y del multiplicador son iguales o diferentes. Así se determina el signo que tendrá el producto.

- Paso 2. Poner cualquier número negativo en formato real (no complementado). Puesto que la mayoría de las computadoras almacenan los números negativos en complemento de a 2, se requiere la operación de complemento a 2 para obtener el número negativo en formato real.

- Paso 3. Empezar por el bit del multiplicador menos significativo y generar los productos parciales. Cuando el bit desmultiplicador es 1, el producto parcial es igual al multiplicando. Cuando el bit del multiplicador es 0, el producto parcial es cero. Cada sucesivo producto parcial debe desplazarse un bit a la izquierda.

- Paso 4. Sumar cada producto parcial a la suma de los productos parciales anteriores para obtener el producto final.

- Paso 5. Si el bit de signo que se había determinado en el paso 1 es negativo, colocamos un 1 en la posición más a la izquierda. Si es positivo, añadiremos un 0.

1010011	Multiplicando
x 0111011	Multiplicador
1010011	Primer producto parcial
+ 1010011	Segundo producto parcial
11111001	Suma del primero y el segundo
+ 0000000	Tercer producto parcial
011111001	Suma
+ 1010011	Cuarto producto parcial
1110010001	Suma
+ 1010011	Quinto producto parcial
100011000001	Suma
+ 1010011	Sexto producto parcial
1001100100001	Suma
+ 0000000	Septimo producto parcial
1001100100001	Producto final

Fig. 2.16 Método de los productos parciales

2.5. CONCLUSIONES

En resumen, el formato que utilizaremos en el diseño es el de signo-magnitud. Esto se debe a que la operación básica de nuestro diseño, y la que mas se va a utilizar, es la multiplicación binaria, la cual se realiza en formato signo-magnitud.

De esta manera, cuando tengamos que utilizar otras operaciones que utilizan el complemento a 2, como la suma, tendremos que hacer una conversión previa de los números negativos a este formato. Además, si el conversor A/D que utilicemos a la entrada utiliza el complemento a 2 como formato de salida, tendremos que añadir una fase previa en nuestro diseño que haga la conversión a signo-magnitud.

Esto hará que en ciertas partes del diseño se consuman más recursos, pero si trabajáramos en complemento a 2 este consumo sería aun mayor, ya que en cada multiplicación que hiciéramos tendríamos que hacer la conversión a signo-magnitud.

CAPITULO 3 – DISEÑO TEÓRICO

3.1. Introducción

En este capitulo [4, 5, 6, 7, 8] haremos una presentación de los distintos bloques que formarán el demodulador. De es esta manera, veremos su función dentro del circuito y sus condiciones de diseño a modo de introducción al siguiente capitulo, en el cual se usaran lo conceptos de diseño aquí visto para realizar la implementación.

3.2. Down conversion y IF-sampling

Antes de pasar por nuestro demodulador, la señal recibida, que esta en frecuencia intermedia, ha de ser convertida a formato digital. Para esto, se ha de utilizar un conversor A/D que haga el muestreo la señal de banda limitada centrada en una frecuencia intermedia FI, lo que se conoce como IF-sampling. Este muestreo A/D generará unas imágenes de la banda de interés que quedarán centradas en nF_S, donde n = 1, 2, 3, 4, etc., siendo la frecuencia imagen más baja la que nos interese.

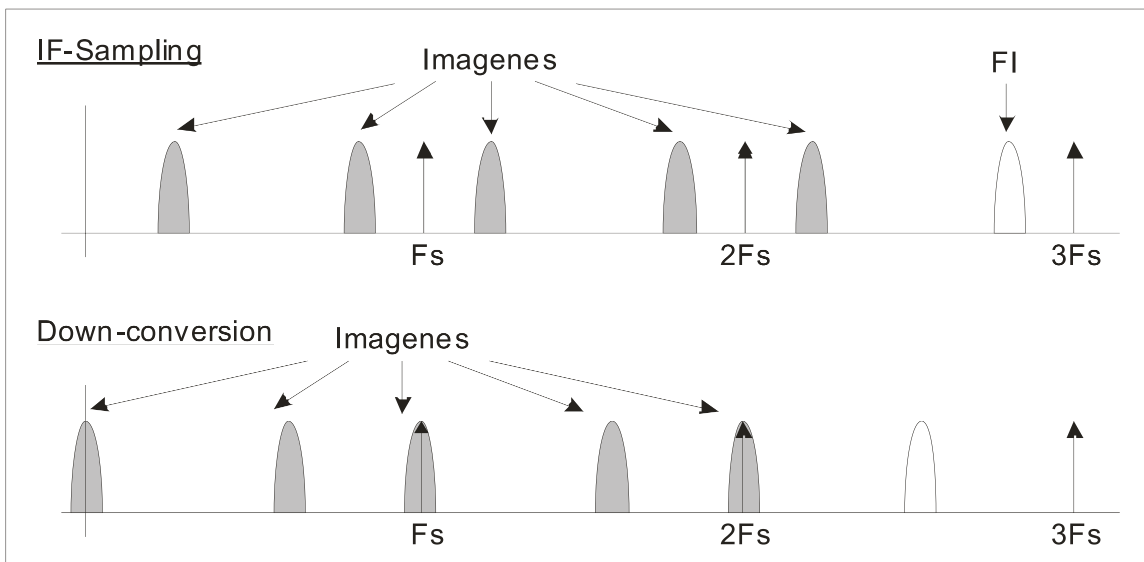


Fig. 3.1 Ejemplo de IF-sampling y down-conversion

En la **Fig. 3.1** se muestra como queda el espectro después de aplicar IF-sampling y down conversion. En este ejemplo, el conversor A/D trabaja a una frecuencia de muestreo de F_S. Esta frecuencia esta por debajo de la señal de interés, por lo tanto deberemos ir con cuido a la hora de elegir el conversor A/D

ya que este deberá estar habilitado con la función de IF-sampling apropiada para nuestra aplicación.

En el segundo espectro, también podemos ver el resultado de aplicar down-conversion, también conocida como desplazamiento en frecuencia. Esto se consigue mezclando, o multiplicando, las muestras generadas por el conversor A/D por la salida de un generador de señales sinusoidales o DDS. Este proceso se realizará en la primera parte de nuestro demodulador.

De esta manera, mediante el IF-sampling y la down-conversion trasladaremos la señal de interés situada en FI a banda base para así poder demodularla posteriormente.

3.3. Diagrama de bloques

Nuestro circuito se puede dividir en tres partes: la primera el desplazador en frecuencia digital o *Digital Down Converter* (DDC), la segunda la parte de filtrado, y por ultimo el decidor de 16QAM.

Fig. 3.2 Esquema básico del demodulador

La primera parte, como hemos visto en el apartado anterior, estará compuesta por un mezclador y una DDS, que desplazarán a banda base la señal de interés. La segunda estará compuesta de un filtro CIC y un filtro FIR, que se encargará de limpiar el espectro dejando solamente la banda de interés. Y por ultimo estará el decisor de 16QAM que traducirá la información obtenida. Esta ultima parte se ha introducido al azar para comprobar el funcionamiento de demodulador, pero posteriormente se le puede añadir cualquier otro tipo de modulación.

Por lo tanto, el diagrama de bloques es el siguiente:

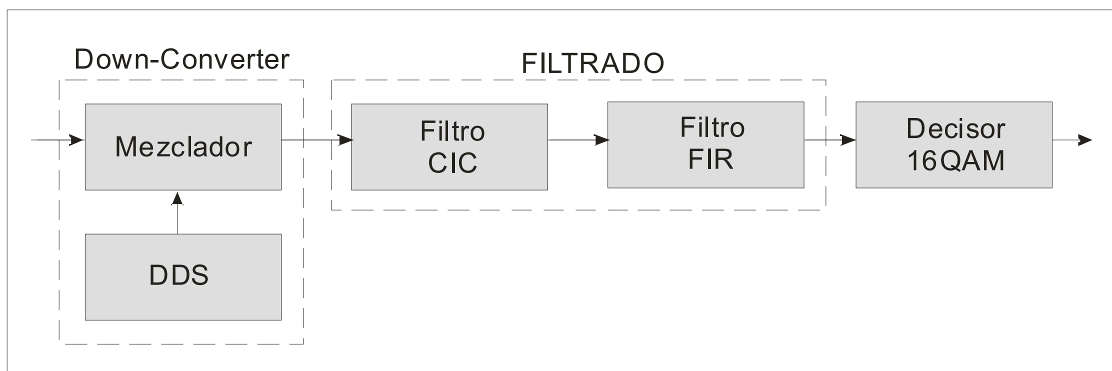


Fig. 3.3 Diagrama de bloques del demodulador

A continuación, veremos la como funciona la primera parte del diseño, es decir, el mezclador y la DDS.

3.4. Mezclador complejo

Según la aplicación los mezcladores se pueden clasificar como:

- UP-CONVERTERS: Se utilizan en transmisión para subir en frecuencia la señal que tenemos en banda base a la frecuencia intermedia FI de la señal de radiofrecuencia. Matemáticamente corresponde a tomar como señal de salida la frecuencia suma.

- DOWN-CONVETERS: Se utilizan en recepción para bajar en frecuencia la señal la señal que esta en FI para situarla en banda base. Matemáticamente corresponde a tomar como señal de salida la frecuencia diferencia.

En nuestro caso lo que necesitaremos es un down-converter, ya que la primera etapa del demodulador se encarga de acondicionar la señal de entrada de tal manera que consigamos dos cosas: la primera es desplazar la frecuencia de interés a banda base y la segunda descomponerla en señales I y Q, en fase y en cuadratura (*in phase*, *quadrature phase*). Estos dos requisitos se logran multiplicando la señal de entrada por dos sinusoides a frecuencia de portadora f_c (*carrier frequency*), con 90º de desfase relativo entre ellas.

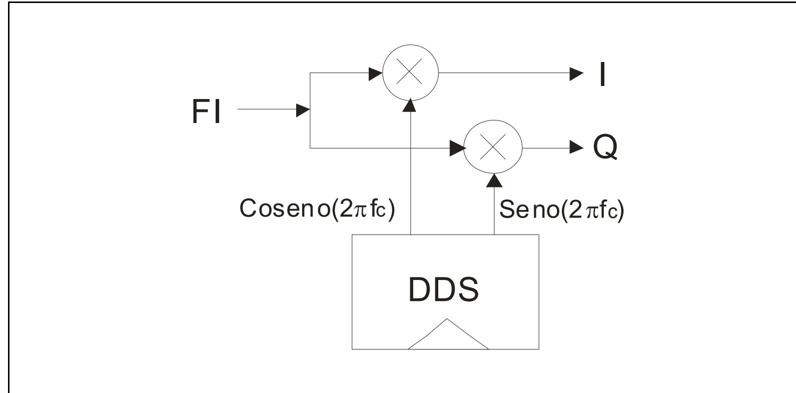


Fig. 3.4 Diagrama de bloques del mezclador

Para ello utilizaremos un mezclador complejo, el cual se encargará de tomar la señal de entrada y multiplicarla por un coseno y un seno, generados por la DDS, obteniendo así la señal en fase I y la señal en cuadratura Q respectivamente. Esto se debe a que cualquier señal paso-banda se puede definir de la siguiente manera:

$$v(t) = I(t)\cos(\omega_0 t) - Q(t)sen(\omega_0 t) \quad \textbf{(3.1)}$$

Por lo tanto, tras su correspondiente desarrollo matemático se deduce que:

$$I(t) = v(t) \times \cos(\omega_0 t) \quad \textbf{(3.2)}$$

$$Q(t) = v(t) \times sen(\omega_0 t) \quad \textbf{(3.3)}$$

El siguiente bloque explica como generaremos estas dos formas de onda.

3.5. DDS

Un método común para generar digitalmente los valores de una señal sinusoidal es usar una tabla de consulta o LUT (look-up table). Ésta almacena las muestras de una sinusoide, las cuales se irán extrayendo a partir de un integrador digital que generará una fase, la cual nos direccionará a una posición de la tabla donde estará almacenado el valor deseado. De esta manera, dependiendo de los parámetros del sistema, podemos hacer que la DDS genere la frecuencia de salida que nosotros queramos, además de poder controlar la supresión de espúmeos de la onda generada.

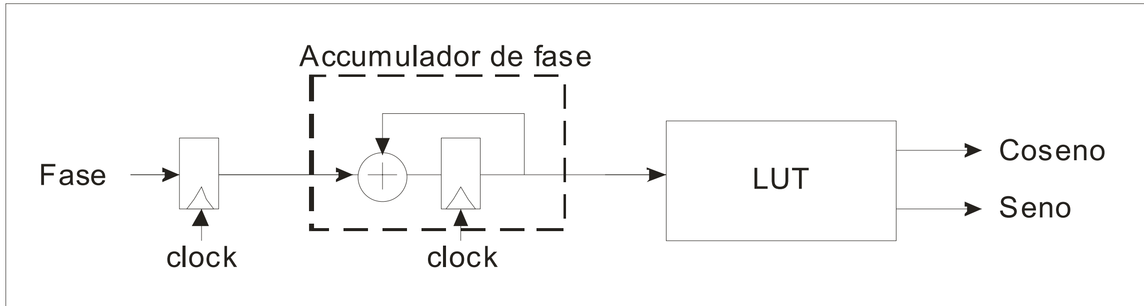


Fig. 3.5 Esquema de la DDS

Como podemos ver en la **Fig. 3.5** la DDS está compuesta por un integrador digital o acumulador de fase que controla la LUT en la que están guardados los distintos valores de un periodo de una señal sinusoidal.

La LUT, tradicionalmente, almacenaba las muestras uniformemente espaciadas de la forma de onda de un coseno y un seno. Estas muestras representaban un único periodo completo de longitud $N = 2^{B_{\Theta(n)}}$. Hoy en día, aprovechando la simetría de los distintos cuadrantes de una forma de onda sinusoidal, lo que se hace es almacenar solamente el primer cuadrante de una sinusoide y, a partir de éste, generaremos los valores del seno y el coseno. Esta forma de implementación hará mucho más eficiente nuestro diseño, ya que reduciremos considerablemente la memoria utilizada en la FPGA, haciendo que la N almacenada sea 4 veces menor y sólo se guarde un único cuadrante de las dos formas de onda.

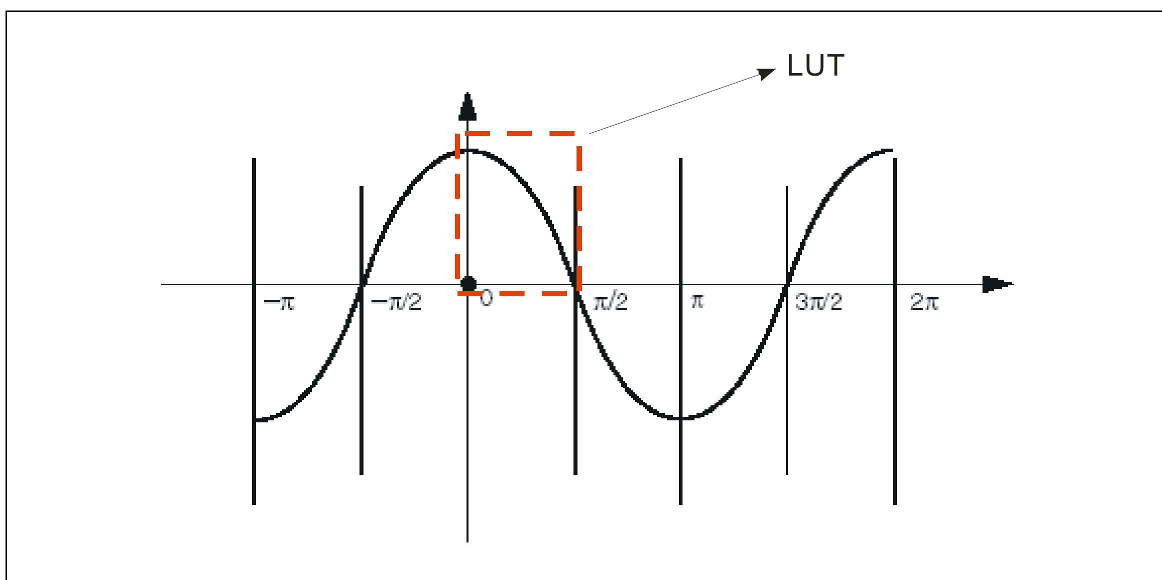


Fig. 3.6 Parte de la onda almacenada en la LUT

3.5.1. Frecuencia de salida

La frecuencia de salida f_{out} de la señal generada por la DDS dependerá de la frecuencia de reloj del sistema f_{clk}, el número de bits en el acumulador de fase $B_{\Theta(n)}$ y el valor del incremento de fase $\Delta\theta$. Para obtenerla utilizaremos la siguiente fórmula:

$$f_{out} = \frac{f_{clk}\Delta\theta}{2^{B_{\Theta(n)}}} \quad [Hz] \tag{3.4}$$

Ésta nos dará la frecuencia de salida en hertzios.

3.5.2. Resolución en frecuencia

La resolución en frecuencia Δf del sintetizador dependerá de la frecuencia de reloj y del número de bits $B_{\theta(n)}$ empleados en el acumulador de fase. La resolución en frecuencia quedará determinada a partir de la siguiente fórmula:

$$\Delta f = \frac{f_{clk}}{2^{B_{\theta(n)}}} \quad [Hz] \tag{3.5}$$

3.5.3. Incremento de fase

El incremento de fase $\Delta\theta$ es un valor sin signo, el cual define la frecuencia de salida del sintetizador. El valor de incremento de fase necesario para conseguir una frecuencia f_{out} en hertzios queda definido en la siguiente fórmula:

$$\Delta\theta = \frac{f_{out} 2^{B_{\Theta(n)}}}{f_{clk}} \tag{3.6}$$

3.5.4. Pureza espectral

La exactitud a la hora de generar la señal de salida afecta directamente al funcionamiento, en cuanto a ruido se refiere, del DDC. El ruido introducido por la DDS es causado por los errores de amplitud y de fase, que se manifiestan

mediante la reducción de la relación señal a ruido (SNR) y que se reduce el rango dinámico libre de espurios (SFDR) respectivamente. Por lo tanto, hay que tener en cuenta que cada bit adicional de fase que se añada producirá una mejora del SFDR de aproximadamente 6dB y el incremento extra en la resolución en amplitud mejora de la SNR en aproximadamente 6dB.

De esta manera, la fórmula para decidir el número de bits que representará los valores del seno y el coseno será la siguiente:

$$B_{\Theta(n)} = \frac{S}{6} \tag{3.7}$$

Donde S es la supresión de espureos mínima que deberá tener la señal de salida de la DDS.

La formula que nos ayudará a decidir el tamaño de la LUT es la siguiente:

$$B_{out} = \frac{SNR}{6} \tag{3.8}$$

Donde SNR será la relación señal a ruido mínima que queremos que tenga esta señal.

La formula **(3.8)** corresponde a su vez con el tamaño que tendrá la señal de salida de la DDS.

3.6. Filtrado

Una vez tenemos la banda de interés en banda base, tendremos que filtrarla para eliminar interferencias. La parte de filtrado esta compuesta por dos filtros: el primero es un filtro CIC y el segundo será un filtro FIR.

El filtro CIC, además de realizar un prefiltrado de la señal, reducirá la frecuencia de muestreo mediante un diezmado de las muestras. Este diezmado se hace para reducir la carga computacional en el siguiente filtro, el cual es un simple filtro paso-bajo que dejará solamente la banda de interés.

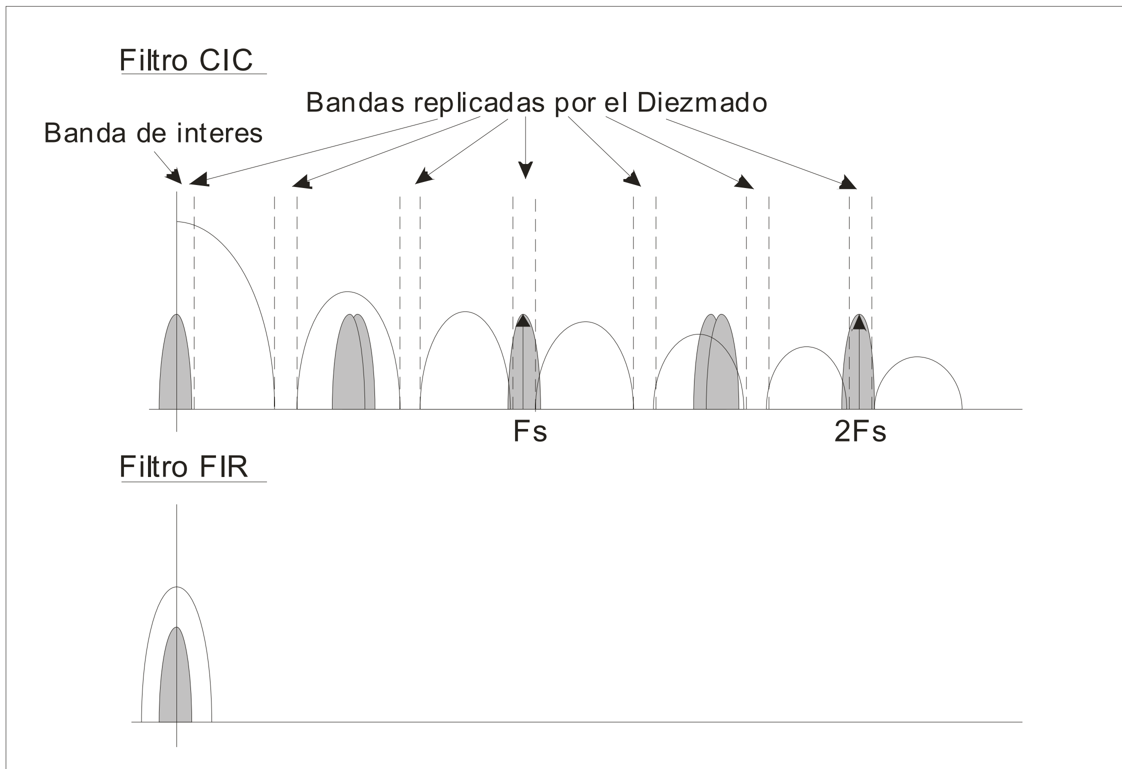


Fig. 3.7 Ejemplo del bloque de filtrado

En la **Fig. 3.7** se muestra el espectro después de pasar por filtro CIC y el filtro FIR. Como podemos ver, el filtro CIC añadirá más imágenes, a parte de las ya añadidas por el conversor A/D, que añadirán interferencias a la señales de interés. Pero a su vez, se añadirán imágenes constructivas que aumentarán la ganancia de la banda de interés. El filtro FIR tendrá una respuesta paso-bajos que atenuará todo lo que esté fuera de la banda de interés.

A continuación, explicaremos más detalladamente como funcionan estos filtros.

3.7. Filtro CIC

La señal mezclada tiene que ser filtrada para conservar la parte del espectro que contiene la banda de interés y eliminar todo lo que queda fuera de dicha banda (que constituiría ruido e interferencia). El filtro tiene que ser normalmente un filtro de banda estrecha con un orden elevado para eliminar todas las componentes indeseadas en la medida de lo posible, por lo que tendremos que llegar a un compromiso entre la relación SNR y el orden del filtro requerido.

La necesidad de un filtro de orden elevado se traduce en un filtro costoso si se realiza a la tasa de muestreo de la señal de entrada. En lugar de esto, podemos utilizar una aproximación multi-tasa en la cual primero se diezme la señal a una tasa de muestreo mucho más baja utilizando un filtro que requiera menor carga computacional. Una vez hecho esto, la señal se filtrará de manera

que el filtro que trabaje a una tasa de muestreo mucho más reducida, por lo que los requisitos son menos estrictos y, por tanto, el orden del filtro será menor.

El filtro con el cual podemos conseguir este diezmado es el filtro CIC (*Cascaded Comb Integrator*).

3.7.1. Estructura y diseño

Los dos componentes básicos de un filtro CIC son un integrador y un filtro de peine(o *comb filter*). Un integrador es simplemente un filtro IIR unipolar con coeficiente de retroalimentación unitario:

$$y[n] = y[n-1] + x[n] \tag{3.9}$$

Este sistema es conocido también como un acumulador. La función de transferencia para un integrador en el plano z es:

$$H_I(z) = \frac{1}{1-z^{-1}} \tag{3.10}$$

La respuesta impulsional es básicamente un filtro paso-bajo con rolloff de -20 dB por década (-6 dB por octava), pero con ganancia infinita en continua (DC). Esto es debido al polo único en z = 1, lo que hace que la salida pueda aumentar sin limite para una entrada sin limite. En otras palabras, un solo integrador por él mismo es inestable.

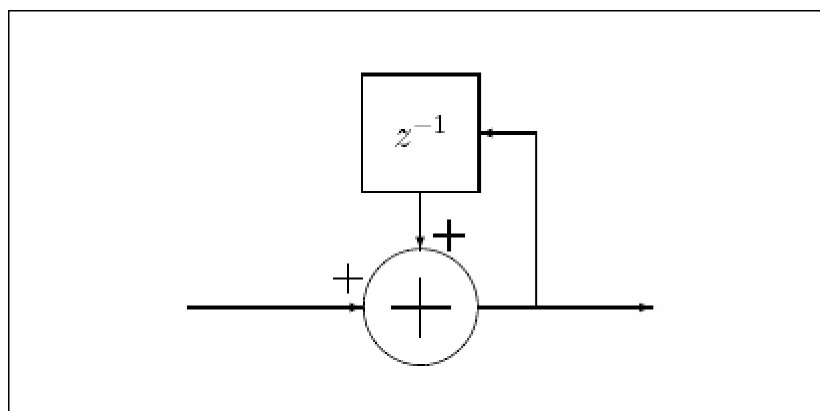


Fig. 3.8 Diagrama de bloques de un integrador

Un filtro peine trabajando a una frecuencia de muestreo alta f_s con una variación de frecuencia de R. Es un filtro FIR asimétrico descrito por la siguiente ecuación:

$$y[n] = x[n] - x[n - RM] \quad \textbf{(3.11)}$$

En esta ecuación, M es un parámetro de diseño y es llamado el *retardo* diferencial (*diffrential delay*). M puede ser cualquier número entero positivo, pero está normalmente limitado a 1 o 2. La correspondiente función de transferencia del filtro de peine a f_s es la siguiente:

$$H_C(z) = 1 - z^{-RM} \quad \textbf{(3.12)}$$

Cuando R = 1 y M = 1, la respuesta impulsional es una función de paso-alto con una ganancia de 20 dB por decada (6 dB por octava), después de todo, es el inverso de un integrador. Cuando RM $\neq 1$, entonces la respuesta impulsional adopta la forma de un coseno alzado con RM ciclos de 0 a 2 π.

Para construir un filtro CIC, debemos unir (o encadenar de la salida a la entrada) N integradores seguidos de N filtros de peine. Esto debería ser suficiente para que el filtro CIC funcione, pero para simplificar la señal de salida se le añade una variación de frecuencia. Por lo tanto, una vez hayamos incorporado la variación de frecuencia, los filtros de peine quedarán descritos por la siguiente ecuación:

$$y[n] = x[n] - x[n - M] \quad \textbf{(3.13)}$$

En la cual hemos variado la frecuencia de muestreo a una R veces mas baja, es decir f_s/R. Con esto lograremos tres cosas. Primero, hemos reducido la frecuencia de la mitad del filtro y por tanto aumentado la eficiencia. Segundo, hemos reducido el número de elementos retardantes necesarios en la sección de los filtros de peine. Tercero, y más importante, la estructura del integrador y la del filtro de peine son ahora independientes al cambio de frecuencia. Esto significa que podemos diseñar un filtro CIC con un cambio de frecuencia programable y mantener la misma estructura del filtro.

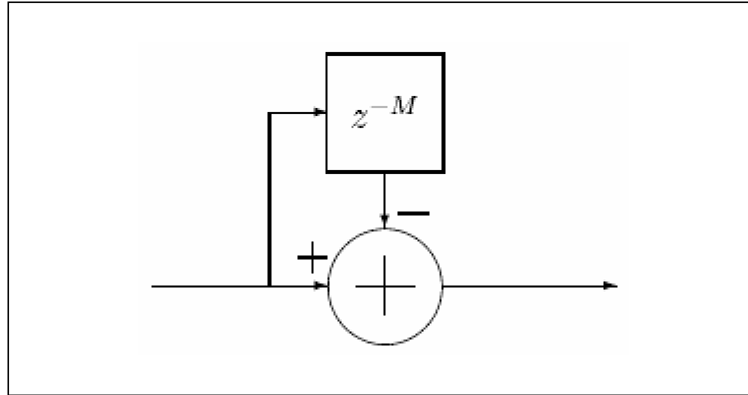


Fig. 3.9 Diagrama de bloques de un filtro de peine

Resumiendo, un diezmador CIC debería tener N integradores en cascada sincronizados a una frecuencia f_s, seguido por un cambio de frecuencia de factor R, y, por ultimo, N filtros de peine en cascada sincronizados a una frecuencia f_s/R.

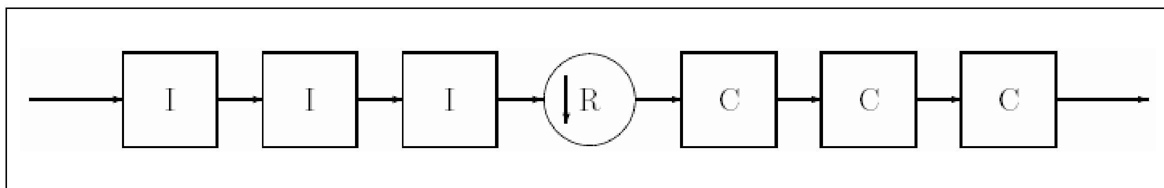


Fig. 3.10 Diagrama de bloques de un filtro CIC Diezmador

3.7.2. Características frecuenciales

La función de transferencia para un filtro CIC a una frecuencia f_s es:

$$H(z) = H_I^N(z) H_C^N(z) = \frac{\left(1 - z^{-RM}\right)^N}{\left(1 - z^{-1}\right)^N} = \left(\sum_{k=0}^{RM-1} z^{-k} \right)^N \qquad \textbf{(3.14)}$$

Esta ecuación muestra que incluso considerando que un filtro CIC tiene integradores, los cuales por sí mismos tienen una respuesta impulsional infinita, dicho filtro es equivalente a N filtros FIR, teniendo cada uno una respuesta impulsional rectangular. Como todos los coeficientes de estos filtros FIR son unitarios, y por lo tanto simétricos, un filtro CIC también tiene una fase lineal y un retardo de grupo constante.

La magnitud de la respuesta impulsional a la salida del filtro es la siguiente:

$$\left|H(f)\right| = \left|\frac{\sin \pi M f}{\sin \frac{\pi f}{R}}\right|^{N} \tag{3.15}$$

Si utilizamos la aproximación $\sin x \approx x$ para una x pequeña y algo de álgebra, podemos aproximar esta función para R grande como:

$$\left|H(f)\right| \approx \left|RM \frac{\sin \pi M f}{\pi M f}\right|^{N} \quad para\ 0 \leq f \leq \frac{1}{M} \tag{3.16}$$

Podemos apreciar unas cuantas cosas acerca de la respuesta. Una es que el espectro de salida tiene nulos en múltiplos de $f = 1/M$. Además, la zona alrededor de la anulación es donde ocurre el aliasing. Si definimos la frecuencia de corte f_c como el límite de la banda de paso útil, entonces la zona de aliasing estará en:

$$\left(i - f_c\right) \leq f \leq \left(i + f_c\right) \tag{3.17}$$

para $f \leq 1/2$ y $i = 1, 2, ..., \left[R/2\right]$. Si $f_c \leq M/2$, entonces el máximo de estos sucederá en el limite inferior de la primera banda, $1 - f_c$. El sistema diseñado debe tener esto en cuenta, y ajustar R, M y N según sea conveniente.

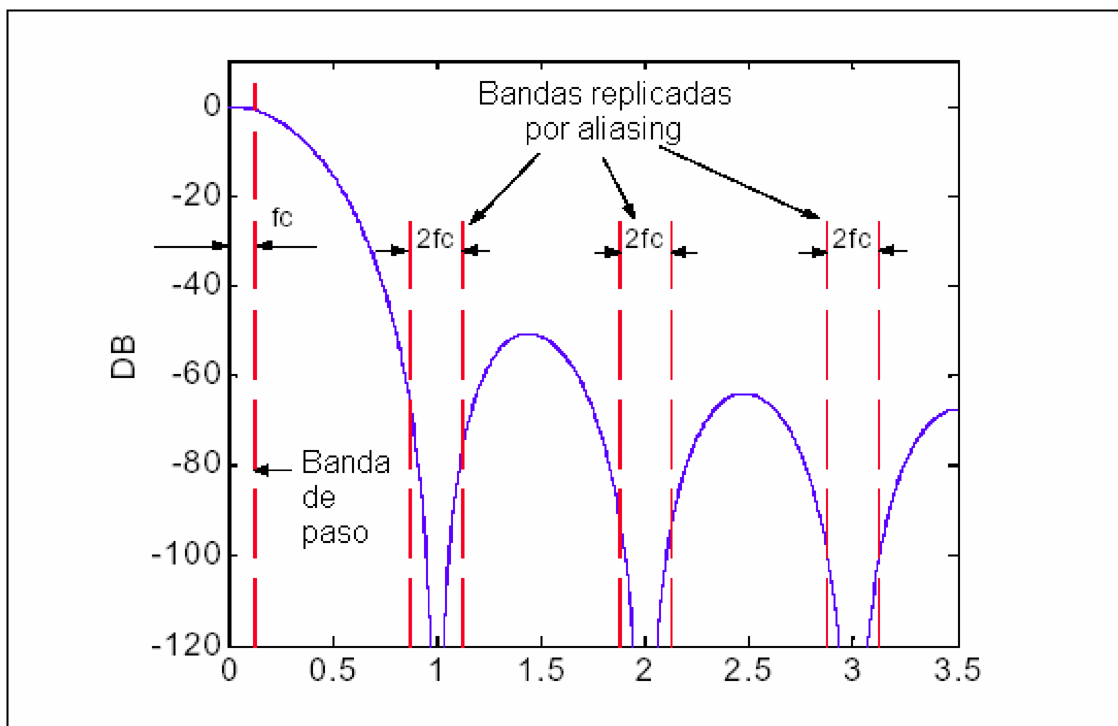


Fig. 3.11 Espectro de salida para M=1, N=4 y R=7

Otra cosa que podemos advertir es que la atenuación de la banda de paso dependerá del número de etapas N. Por lo tanto, si se incrementa el número de etapas mejoramos el rechazo imágenes, aunque esto también incrementa las perdidas en la banda de paso.

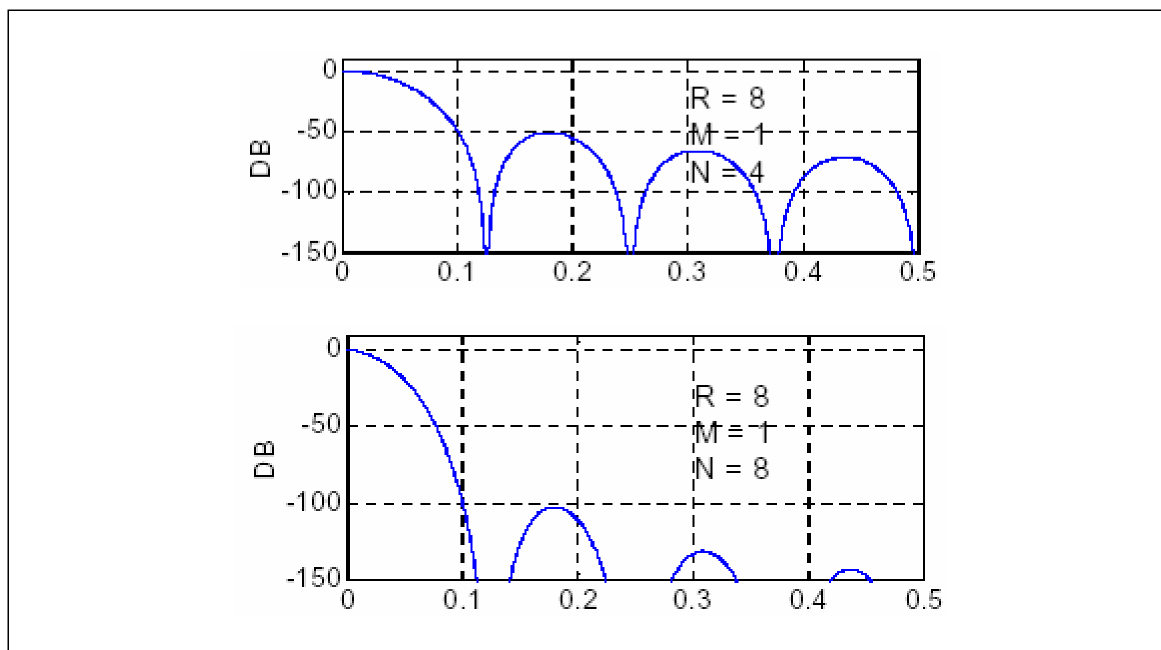


Fig. 3.12 Comparación de ganancias usando N=4 y N=8

También podemos ver que el aumento de la ganancia del filtro en DC esta relacionado con la variación de frecuencia.

3.7.3. AUMENTO DE BITS

Para los CIC diezmadores, la ganancia G a la salida del final de la sección de filtros de peine es:

$$G = \left(RM\right)^{N} \quad \textbf{(3.18)}$$

Teniendo en cuenta que trabajamos con números con signo, podemos usar este resultado para calcular el número de bits requeridos para el último filtro de peine debido al aumento de bits. Si Bin es el número de bits a la entrada, entonces el número de bits a la salida, Bout, será:

$$B_{out} = \left\lceil N \log_2 RM + B_{in} \right\rceil \quad \textbf{(3.19)}$$

También puede ser que se necesiten Bout bits para cada integrador y filtro de peine. La entrada necesita estar señalada extendida a Bout bits, pero LSB puede ser truncado o redondeado en etapas posteriores.

3.8. FIR

Una vez se ha diezmado la señal en banda, el siguiente paso será eliminar las interferencias producidas por las imágenes generadas en la conversión A/D y el filtro CIC. Para esto bastará con utilizar un filtrado paso-bajo que atenuará todas las frecuencias que estén fuera de la banda de interés.

Esto se conseguirá mediante la utilización de un filtro FIR (*Finite Impulse Response*) o Respuesta finita al impulso. Se trata de un filtro en el que, como su nombre indica, si la entrada es un impulso, la salida tendrá un número finito de términos no nulos que son los coeficientes que configuran el tipo de filtrado. Por lo tanto, si variamos estos coeficientes, los cuales pueden cambiarse en cualquier momento, puede generarse cualquier tipo de filtrado.

Para obtener la salida sólo se utilizan entradas actuales y anteriores, lo que hará que estos filtros tengan todos los polos en el origen, por lo que su comportamiento será estable. Su expresión en el dominio n es:

$$y_n = \sum_{k=0}^{N-1} b_k x\left(n-k\right) \quad \textbf{(3.20)}$$

En la expresión anterior N es el orden del filtro, que también coincide con el número de términos no nulos y con el número de coeficientes del filtro. Los coeficientes son b_k.

La salida es básicamente la convolución de la señal de entrada x(n) con la respuesta impulsional h(n), por lo que se puede expresar de esta otra forma:

$$y_n = \sum_{k=0}^{N-1} h_k x_{n-k} \quad \textbf{(3.21)}$$

Aplicando la transformada Z a la expresión anterior:

$$H(z) = \sum_{k=0}^{N-1} h_k z^{-k} = h_0 + h_1 z^{-1} + \ldots + h_{N-1} z^{-\left(N-1\right)} \quad \textbf{(3.22)}$$

Por lo tanto, para generar el filtrado bastará con almacenar, mediante retardos, tantas entradas sucesivas como coeficientes (orden) tenga el filtro, de manera que en cada ciclo se pueda realizar la convolución de estas entradas con la respuesta impulsional de filtro, mediante la multiplicación de cada una de las entradas retardadas por su correspondiente coeficiente. El filtro tendrá entonces la siguiente estructura:

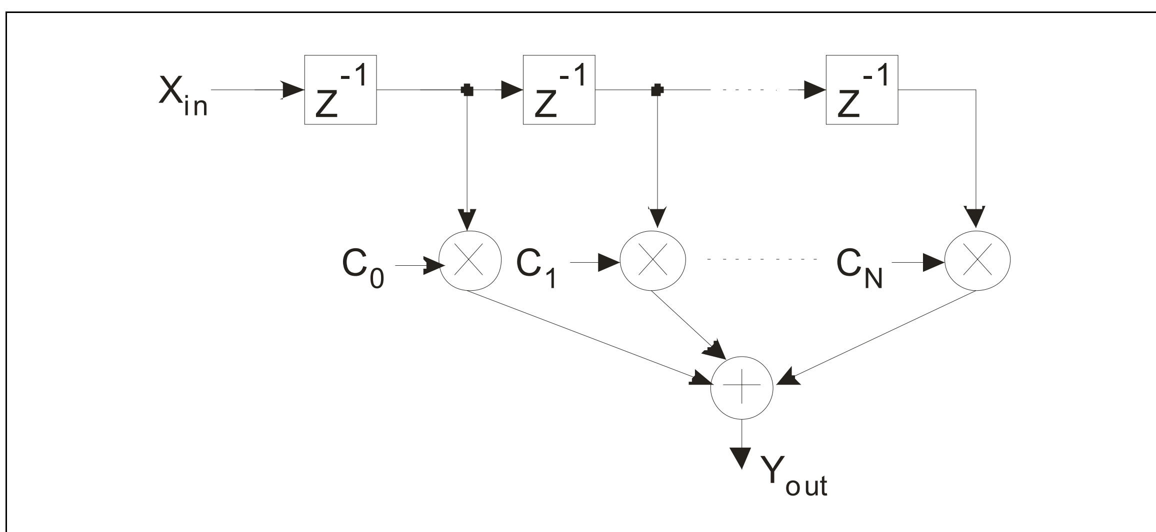


Fig. 3.13 Esquema de bloques del filtro FIR

Los filtros FIR tienen la gran ventaja de que pueden diseñarse para ser de fase lineal, lo cual hace que presenten ciertas propiedades en la simetría de los coeficientes. Además son siempre estables.

Por contra también tienen la desventaja de necesitar un orden mayor respecto a los filtros IIR para cumplir las mismas características. Esto se traduce en un mayor gasto computacional.

3.9. Decisor 16QAM

Una vez hayamos separado y filtrado las componentes en fase y cuadratura podremos interpretar la información que estamos recibiendo, la cual dependerá de la modulación que estemos utilizando. A modo de ejemplo, hemos diseñado un decisor para una modulación QAM de 16 símbolos o 16QAM.

Este bloque es muy sencillo ya que, dependiendo del valor que tengan las componentes I y Q, la salida tendrá un valor u otro, el cual dependerá de cómo esta distribuida la constelación. La constelación que se ha utilizado corresponde con la del estándar V.22 bis y V.32*uncoded* y es el siguiente:

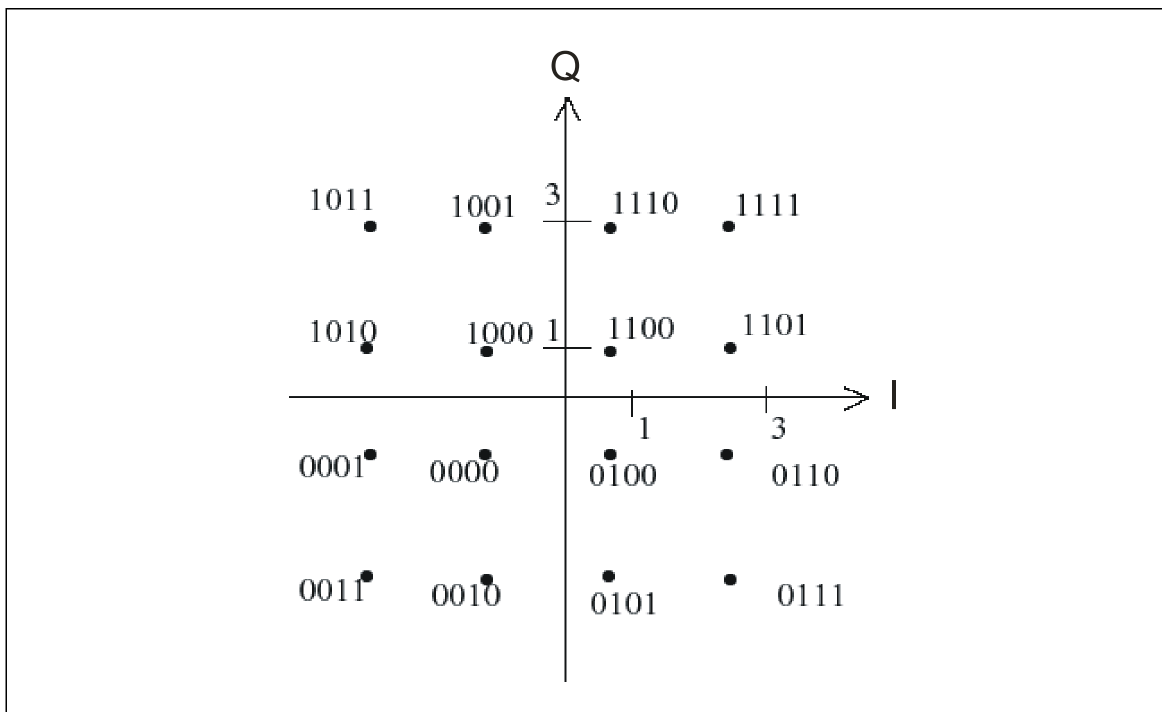


Fig. 3.14 Constelación para 16QAM

Como podemos ver en la **Fig. 3.14**, a cada combinación de valores IQ se le asigna un código, hasta un total de 16 posibles combinaciones, los cuales se

mostrara a la salida del decisor en relación a las entradas que haya en ese momento.

Este tipo de modulación suele representar sus valores en una escala de 0 a 5v de salida. En nuestro caso, al trabajar con números binarios con signo, nuestro rango de representación irá de 0 a 2^{n-1}-1 para los números positivos y de -1 a -2^{n-1} para los números negativos, por lo que la correspondencia con los símbolos será la siguiente:

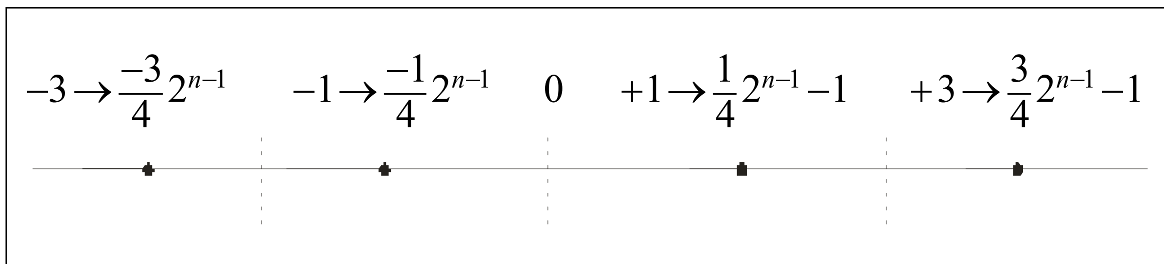


Fig. 3.15 Valor binario de los símbolos

De esta manera, los umbrales de decisión de los símbolos estará justo en medio de estos cuatro valores es decir en $-\frac{1}{2}2^{n-1}$, 0 y $\frac{1}{2}2^{n-1}-1$.

CAPITULO 4 – IMPLEMENTACIÓN EN FPGA

4.1. Introducción

En este capitulo [4, 5, 6, 7, 8] se realiza la implementación del demodulador, es decir, se explicarán los siguientes puntos:

- Numero de entradas y salidas.
- Numero de bits de cada una de las entradas y salidas.
- Calculo de factores de diseño.
- Funcionamiento del código (basándose en lo visto en el capitulo anterior).

A lo largo del diseño del demodulador digital se tienen siempre en mente consideraciones de:

- Costo de implementación (número de compuertas lógicas).
- Requerimientos de tiempo de computación.
- Rango dinámico de la representación o numero de bits.
- Relación señal a ruido de las señales digitales.

Un dato importante a tener en cuenta, a la hora de tomar estas consideraciones, es el modelo de FPGA que vamos a utilizar. Se trata de una Virtex 4 modelo XC4VSX35 y encapsulado FFG668.

Todo el proceso de implementación se ha realizado con el programa ISE 7.1 proporcionado por la misma Xilinx. Este programa nos permite, mediante código VHDL, implementar los distintos bloques que componen un nuestro dispositivo, permitiéndonos sintetizar el código comprobando en todo momento que este pueda funcionar en la FPGA. Además, incorpora una herramienta llamada Place & Route que nos permite asignar las entradas y salidas de nuestro código a los distintos pines que tenga la FPGA que utilicemos.

De esta manera a lo largo de la implementación también veremos el coste que van a tener los distintos bloques del demodulador a la hora de consumir los recursos de la FPGA.

4.2. Mezclador

Como hemos visto en el capitulo anterior, este bloque se encargará de multiplicar la entrada FI por las dos señales generadas por el bloque de la DDS, dando dos resultados: las componentes en fase y la componente en cuadratura. Para ello, realizaremos dos multiplicaciones digitales de las tres señales obteniendo así las dos componentes.

La salida de un multiplicador digital tiene un número de bits igual a la suma de los bits de las dos señales de entrada. Esto hará que, cada vez que se realice una multiplicación, aumentemos la carga computacional y el consumo de recursos de los bloques posteriores. Una solución a este problema es truncar la salida de los dos multiplicadores, descartando los bits menos significativos quedándonos solamente con el mismo número de bits que la señal de entrada.

Por lo tanto, el mezclador incorporará un truncador del número bits, quedando el diagrama de bloques de la siguiente manera:

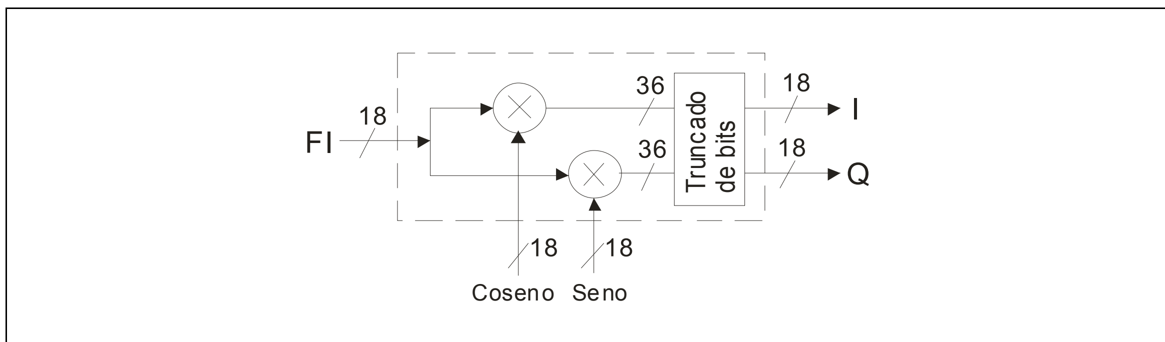


Fig. 4.1 Diagrama de bloques del mezclador

El bloque del mezclador será el siguiente:

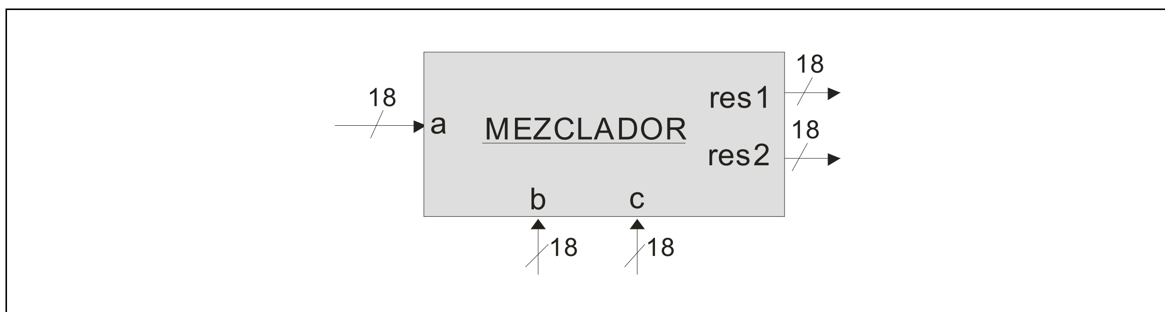


Fig. 4.2 Bloque del mezclador

Como se puede apreciar, tanto a la salida I y Q (res1 y res2) como a la entrada FI (a) se les a dado el mismo número de bits que a la señales de seno y el

coseno (b y c). Esto es porque la salida de la DDS, como veremos en el siguiente apartado, tiene un número de bits fijo que depende de las condiciones de diseño de esta, y por lo tanto es más fácil hacer que las otras entradas y salidas dependan de esta. En el caso de que la entrada FI no coincidiera en el número de bits, tendríamos que incluir un sencilla etapa previa en la que modificaríamos este valor.

El código final del mezclador se puede ver en el anexo 1. Podemos ver en el código que hemos utilizado una función llamada “multiplicar”. Esta función ha sido creada por nosotros y realiza la multiplicación de dos números binarios en formato signo-magnitud, tal y como se explica en el capitulo 2. Hemos tenido que crearla ya que la función que proporciona VHDL para multiplicar números no reconoce números con signo. Podemos encontrar el código de esta función en el anexo 6.

En el siguiente grafico se puede ver el consumo de recursos del bloque:

```
Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                     666  out of  15360     4%
 Number of Slice Flip Flops:            36  out of  30720     0%
 Number of 4 input LUTs:              1158  out of  30720     3%
 Number of bonded IOBs:                 91  out of    450    20%
 Number of GCLKs:                        1  out of     32     3%
```

Fig. 4.3 Consumo de recursos del bloque

4.3. DDS

La función de este bloque es la generar un coseno y un seno de una determinada frecuencia. Para ello lo que se hace es almacenar las muestras de periodo en un tabla de consulta (LUT) y a partir de una fase acumulada se extraen las dos señales a la frecuencia deseada. Como hemos visto en el capitulo 2, lo que haremos es guardar un único cuadrante de un periodo de una de las dos formas de onda y a partir de este generaremos las señales completas.

Antes de nada hemos de saber lo que nos va a ocupar la tabla. Para ello debemos tener en cuenta que la SFDR dependerá del tamaño de esta tabla. Por lo tanto, si marcamos una SFDR de S = 60 dB, tenemos que el numero de bits que formarán la LUT serán:

$$B_{\Theta(n)} = \frac{S}{6} = \frac{60}{6} = 10 bits \qquad \textbf{(4.1)}$$

Entonces, teniendo en cuenta que la tabla almacenada será cuatro veces menor, tendremos que restarle 2 bits al tamaño almacenado, quedando almacenadas finalmente un total de 256 muestras del primer cuadrante de un coseno.

Otro parámetro de diseño es el numero de bits que tendrá cada una de las muestras. Para ello debemos tener en cuenta que la SNR dependerá del tamaño de esta tabla, por lo tanto, si marcamos una SNR = 108 dB, tenemos que el numero de bits que formarán la LUT serán:

$$B_{out} = \frac{SNR}{6} = \frac{108}{6} = 18bits \tag{4.2}$$

Por lo tanto, a la hora de generar las 2 señales tendremos que tener en cuenta dos factores, uno es el signo de cada una de las señales y la otra es la dirección en la que se han de leer los valores de la LUT.

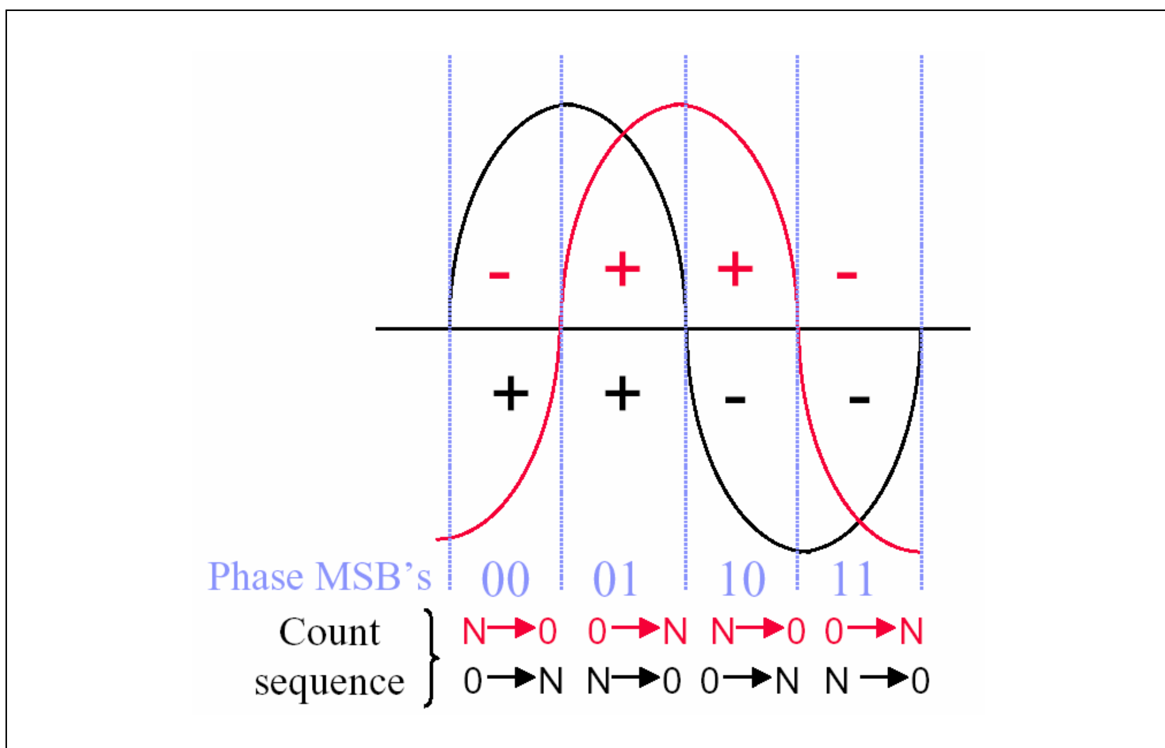


Fig. 4.4 MSB del incremento de fase

En la **Fig. 4.4** podemos ver que la dirección de lectura se irá alternando al final de cada cuadrante y que siempre será la inversa para las dos señales. De esta manera para una señal la dirección será la fase acumulada y para la otra N menos la fase acumulada. En cambio, el signo cambiará cada dos cuadrantes, coincidiendo con el cambio de dirección de abajo hacia arriba.

Por lo tanto, ya sabemos como generar las dos señales, ahora hemos de saber como seleccionar las muestras para generar una determinada frecuencia.

Si nos fijamos en la formula **(3.4)** la frecuencia de salida de la DDS dependerá de la frecuencia de reloj que estemos utilizando, del numero de bits que forma la LUT y además de una tercera variable. Esta variable es el incremento de fase y será un parámetro que se le introducirá a la DDS a través de una nueva entrada. Para calcular el valor de esta fase se utilizará la formula **(3.6).** El valor de este incremento estará ligado al tamaño de la tabla por lo que su numero de bits de la variable de entrada será de 8.

Por lo tanto, el bloque de la DDS será el siguiente:

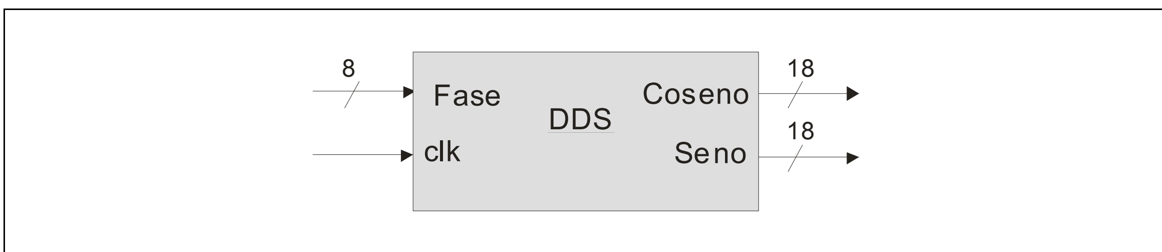


Fig. 4.5 Bloque de la DDS

Y el diagrama de bloques interno quedará así:

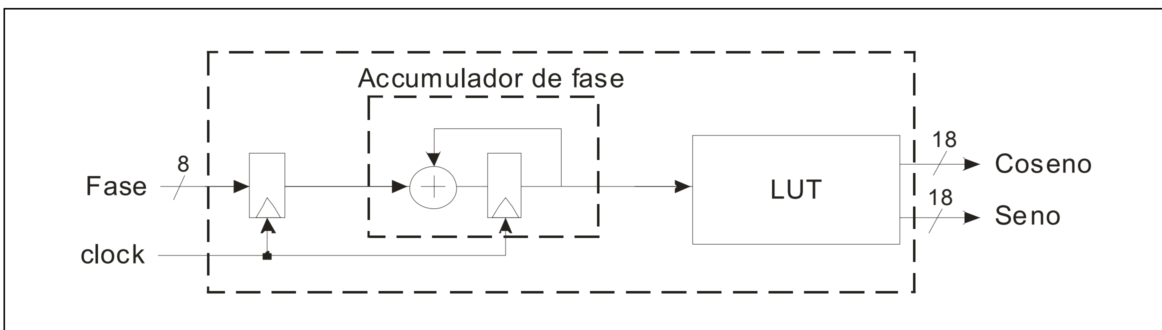


Fig. 4.6 Diagrama de bloques interno de la DDS

En el siguiente grafico se puede ver el consumo de recursos del bloque:

```
Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                      385  out of  15360     2%
 Number of Slice Flip Flops:             92  out of  30720     0%
 Number of 4 input LUTs:                626  out of  30720     2%
 Number of bonded IOBs:                  46  out of    450    10%
 Number of GCLKs:                         1  out of     32     3%
```

Fig. 4.7 Consumo de recursos del bloque

El código de la DDS se puede ver en el anexo 2.

4.4. CIC

Este bloque tiene la función de realizar un diezmado de la frecuencia de la señal de entrada en función del valor de la variable de entrada R. Esta R es un parámetro de diseño que podrá configurarse externamente, pudiendo variarse desde 2 hasta 64, ya que la entrada es de 6 bits.

Además de la R, hay dos parámetros más que se han fijado y que no podrán variarse, la M y la N. M será igual a 1, haciendo de esta manera que la replicas del espectro se produzcan cada f_s/R. N será igual a 4, de manera que la banda útil no se verá muy alterada, aunque esto hará que el resto del espectro no útil no sufra mucha atenuación, pero de esto se encargará el siguiente filtro.

Una vez definido esto, el siguiente paso será decidir el tamaño de los bits de salida. Para ello utilizaremos la formula **(3.19)** que dependerá de los valores ya fijados y de la R, que al ser variable hará que haya un valor máximo y uno mínimo que hará que tengamos que tomar una decisión.

$$B_{out\max} = \left\lceil N\log_2 64M + B_{in} \right\rceil = \left\lceil 4\log_2 64 + 18 \right\rceil = 42 bits \qquad \textbf{(4.3)}$$

$$B_{out\min} = \left\lceil N\log_2 2M + B_{in} \right\rceil = \left\lceil 4\log_2 2 + 18 \right\rceil = 22 bits \qquad \textbf{(4.4)}$$

Por lo tanto, si tenemos en cuenta que al trabajar con números con signo los bits no engordarán tanto y es mas difícil que se produzca desbordamiento, lo que haremos es aumentar un bit por cada suma que se produzca dentro del filtro, para las cuales utilizaremos la función “sumar”, que podemos encontrar en el anexo 6, la cual se encarga de sumar dos números binarios con signo. De manera que el numero final de los bits de salida será de 26, aunque este valor

se truncará a la salida para quitarle carga al filtro FIR dejando un valor final de 18 bits.

El diagrama de bloques final será el siguiente:

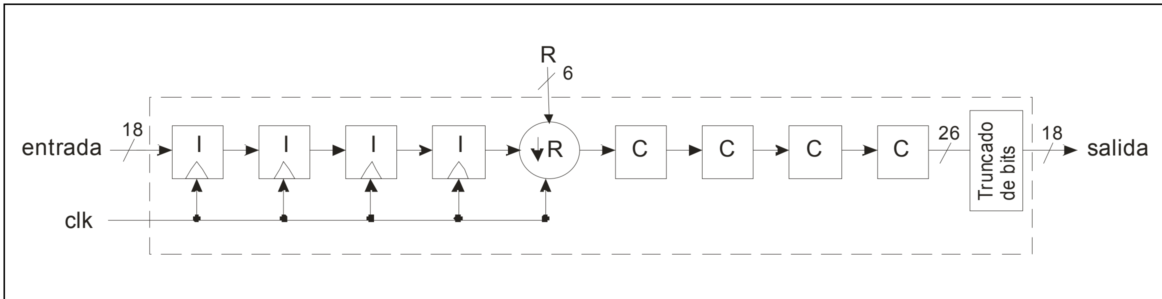


Fig. 4.8 Diagrama de bloques interno del filtro CIC

El bloque del filtro CIC quedará así:

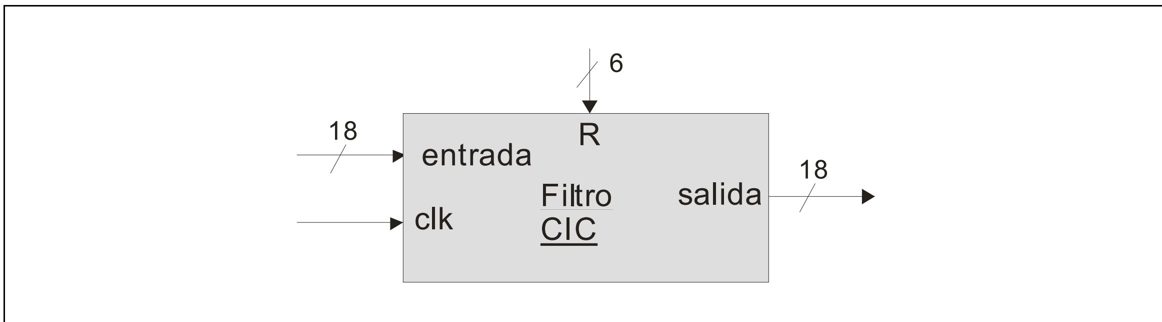


Fig. 4.9 Bloque del filtro CIC

En el siguiente grafico se puede ver el consumo de recursos del bloque:

```
Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                    3216  out of  15360    20%
 Number of Slice Flip Flops:           276  out of  30720     0%
 Number of 4 input LUTs:              6262  out of  30720    20%
 Number of bonded IOBs:                 43  out of    450     9%
 Number of GCLKs:                        1  out of     32     3%
```

Fig. 4.10 Consumo de recursos del bloque

El código del filtro CIC se puede ver en el anexo 3.

4.5. FIR

Como hemos visto en el capitulo anterior, el filtro FIR hará la convolución de la señal de entrada con la respuesta impulsional del filtro para conseguir un filtrado paso-bajos. En nuestra implementación, esta respuesta impulsional se introducirá en una fase previa al funcionamiento del filtro mediante dos variables de entrada. La primera variable "coef" se encargará de introducir los coeficientes y la segunda "sel" de seleccionar en que posición se almacenarán, tal como se ve en la siguiente tabla:

Tabla 4.1. Asignación de valores de la variable "sel"

Sel	coeficiente	Filtrado
0000	C0	OFF
0001	C1	
0010	C2	
0011	C3	
0100	C4	
0101	C5	
0110	C6	
0111	C7	
1111	--	ON

Además, como se puede ver en la tabla, la variable "sel" tendrá un valor definido el desactivará el almacenamiento de coeficiente y pondrá en marcha el FIR. De esta manera, mediante la creación de una interfaz de usuario podrá configurarse los coeficientes del FIR, y una vez configurados activar el filtrado.

El filtro tendrá un total de ocho coeficientes, ya que si lo implementamos a 16 coeficientes nos quedamos sin recursos. Por lo tanto, la variable "sel", como hemos visto en la **tabla 4.1**, tendrá 4 bits.

Los coeficientes tendrán un tamaño de 11 bits, ya que, como estarán comprendidos en -1 y 1, con el rango de valores de -1024 a 1023 tendremos suficiente para representar los coeficientes. Por lo tanto, si los bits de entrada son 18, después de realizar los productos y sumarlos todos, los bits de salida serán 31. Esto se debe a que al multiplicar sumaremos los bits de entrada con los de los coeficientes y por cada par de sumas se le añadirá un bit para evitar desbordamientos. Esto último se consigue gracias a una función llamada "sumar2" ", que podemos encontrar en el anexo 6, la cual se encarga de sumar dos números binarios evitando que se produzca desbordamiento.

En definitiva, el bloque del filtro FIR quedará así:

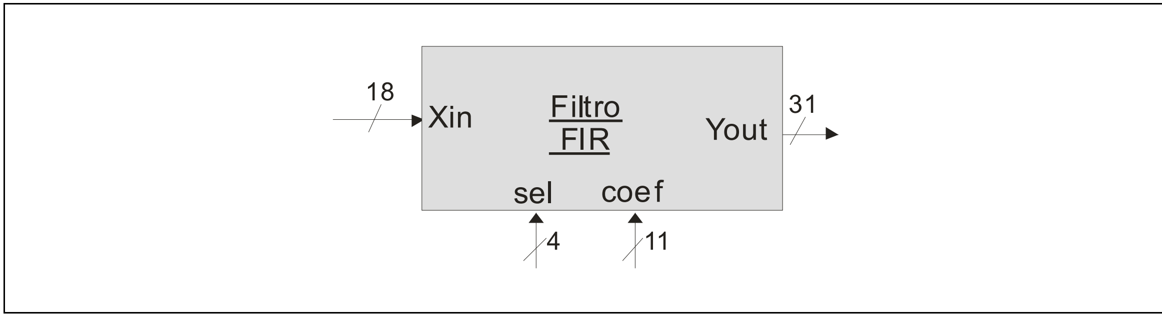


Fig. 4.11 Bloque del filtro FIR

Como podemos ver, este bloque no estará controlado por la frecuencia de clock, sino que estará sincronizado con la frecuencia de salida de la señal diezmada por el filtro CIC.

El diagrama de bloques será el siguiente:

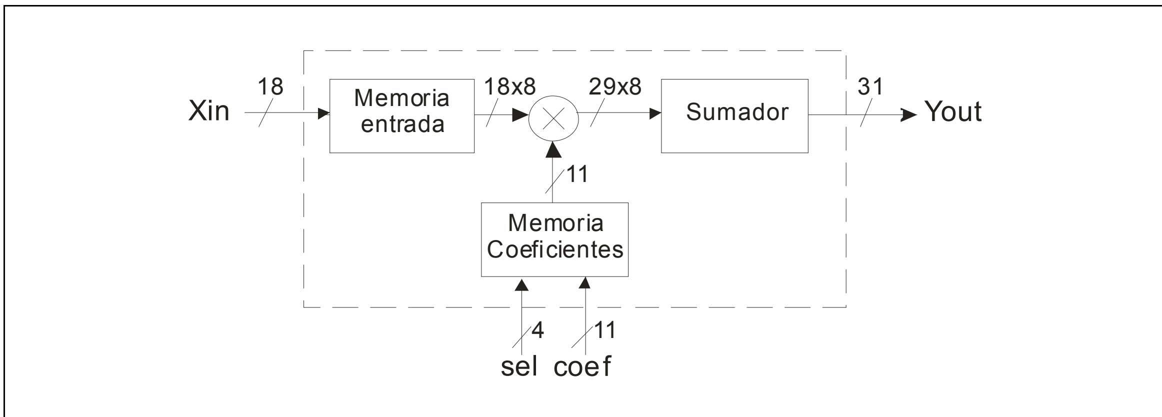


Fig. 4.12 Diagrama de bloques interno del filtro FIR

En el siguiente grafico se puede ver el consumo de recursos del bloque:

```
Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                    3115  out of  15360    20%
 Number of Slice Flip Flops:           347  out of  30720     1%
 Number of 4 input LUTs:              5528  out of  30720    17%
 Number of bonded IOBs:                 64  out of    450    14%
 Number of GCLKs:                        2  out of     32     6%
```

Fig. 4.13 Consumo de recursos del bloque

El código del filtro FIR se puede ver en el anexo 4.

4.6. Decisor 16QAM

El decisor de 16QAM, como su nombre indica, se encargará de decidir con que símbolo de la constelación se corresponden los valores de la componente en fase y cuadratura. Para realizar estas decisiones bastará con utilizar la función *if* utilizando como condición el valor de los umbrales de decisión definidos en el capitulo **3.9**. Teniendo en cuenta que la salida del filtro FIR es de 31 bits los umbrales serán 0 y los siguientes valores:

$$+2 \rightarrow \frac{1}{2}\left(2^{n-1}-1\right)=\frac{1}{2}\left(2^{30}-1\right)=536870911 \qquad \textbf{(4.5)}$$

$$-2 \rightarrow \frac{1}{2}2^{n-1}=\frac{1}{2}2^{30}=-536870812 \qquad \textbf{(4.6)}$$

Una vez tenemos el número bits de entrada y el valor de los umbrales de decisión, nos faltará saber cual es el número de bits a la salida. Esto es fácil, ya que estamos trabajando con un 16QAM la cual tiene 16 símbolos distintos, por lo que tendrá 16 posibles valores de salida. Por lo tanto, la salida será de 4 bits, quedando el bloque de la siguiente manera:

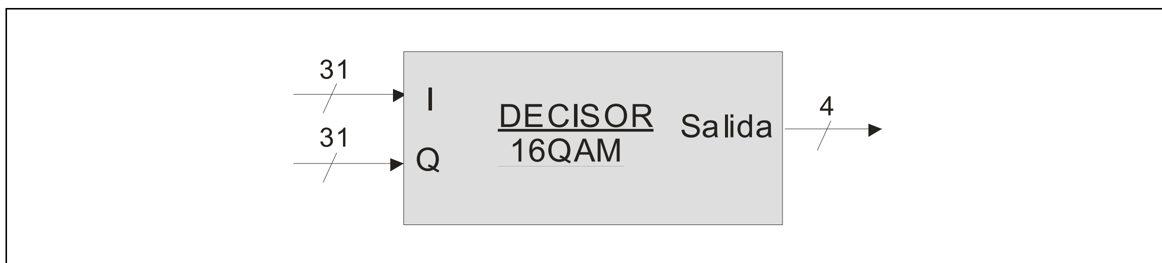


Fig. 4.14 Bloque del decisor 16QAM

En el siguiente grafico se puede ver el consumo de recursos del bloque:

```
Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                       43  out of  15360     0%
 Number of Slice Flip Flops:              4  out of  30720     0%
 Number of 4 input LUTs:                 73  out of  30720     0%
 Number of bonded IOBs:                  66  out of    450    14%
```

Fig. 4.15 Consumo de recursos del bloque

El código puede encontrarse en el anexo 5.

4.7. Demodulador

Por ultimo, solo faltará realizar la interconexión de todos los bloques. Esto se realiza mediante la conexión de todos los puertos en un ultimo dispositivo, creando de esta manera un mapa de puertos que relacione las entradas y salidas de los bloques anteriores. Por lo tanto, el diagrama de bloques final puede verse en el anexo 8.

Finalmente, el demodulador consumirá los siguientes recursos de la FPGA:

```
Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                    10440  out of  15360    67%
 Number of Slice Flip Flops:           1054  out of  30720     3%
 Number of 4 input LUTs:              18999  out of  30720    61%
 Number of bonded IOBs:                  53  out of    450    11%
 Number of GCLKs:                         3  out of     32     9%
```

Fig. 4.16 Consumo de recursos del bloque

El codigo del demodulador se puede ver en el anexo 7.

CONCLUSIONES

Aunque algunas de las partes del demodulador, como por ejemplo el filtro FIR, han quedado algo limitadas, el diseño e implementación se han logrado satisfactoriamente. Ademas, los conceptos principales que se tenian que aprender y refrescar a la hora de realizar el proyecto, los cuales se enumeran en la introducción de este, se han asimilado y aplicado correctamente.

La parte que más tiempo ha llevado implementar y que más problemas ha dado a sido la DDS, pero finalmente se ha conseguido que funcione correctamente aunque la resolución frecuencial no sea muy precisa. Esto se debe a que, para hacerla mas precisa, se ha de tener una tabla de consulta mucho mas grande y, si el proceso de convertir el valor de las muestras a formato binaria ya ha sido pesado para 256 muestras que hemos utilizado, el aumentar este valor en el doble o más sería aun más tedioso. Para una posible futura mejora del codigo, podría pensarse en generar un codigo en C++, Matlab o cualquier otro lenguaje que automatice este proceso y genere los números binarios en un formato que entienda el codigo VHDL.

Por lo que respecta al resto de bloques su funcionamiento es correcto, aunque un poco limitado debido a las dimensiones de la FPGA. Quiza si se pulieran más las funciones más utilizadas se conseguiria reducir el consumo de recursos de manera que se puedan ampliar estos bloques. La verdad es que se ha dedicado más tiempo a hacer que el codigo funcione que a pulir ciertos aspectos que harían que la carga computacional y el consumo de recursos se viera reducido.

REFERENCIAS

[1] Curso “Lenguajes de descripción Hardware VHDL”, Universidad del Pais Vasco (UPV), http://www.ehu.es/~jtpolagi/inicio2.htm

[2] Stephen Brown, Zvonko Vranesic, *Fundamentals of Digital Logic with VHDL Design*, Mc Graw Hill.

[3] Thomas L. Floyd, *Fundamentos de sistemas digitales*, Prentice Hall, 7ª edición 2000.

[4] www.xilinx.com

[5] www.altera.com

[6] Ronald E. Crochiere and Lawrence R. Rabiner, Multirate Digital Signal Processing. Pretice-Hall Signal Processing Series, Prentice Hall, 1983.

[7] L. Cordesses, “Direct Digital Synthesis: A Tool for Periodic Wave Generation (Part 1)”, IEEE Signal Processing Magazine, pp. 110-117, Sep. 2004

[8] A. Peled and B. Liu, “A new hardware realization of Digital Filters”, IEEE Trans. on Acoustic Speech and Signal Processing, nº 22, pp. 152 -462, 1974

ANEXO 1 – CÓDIGO DEL MEZCLADOR

```
--------------------------------------------------------------------------------
-- Company:     EPSC
-- Engineer:     Juan Antonio Guerrero Balmori
--
-- Create Date:    17:23:15 04/21/06
-- Design Name:    Multiplicador binario
-- Module Name:    DDS - ddsarch
-- Project Name:     Diseño e implementación con FPGA de un demodulador para
--              comunicaciones digitales.
-- Target Device:     Xilinx Virtex 4
-- Tool versions:              ISE 7.1
--
--------------------------------------------------------------------------------

-- Incluimos las librerías que vamos a utilizar
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.DDCpack.all;

-- Definimos la entidad
entity multip is
   Port ( a : in std_logic_vector(17 downto 0);
          b : in std_logic_vector(17 downto 0);
          c : in std_logic_vector(17 downto 0);
          clk : in std_logic;
                      res1 : out std_logic_vector(17 downto 0);
          res2 : out std_logic_vector(17 downto 0));
end multip;

-- Definimos la arquitectura
architecture multiarch of multip is

begin
        process (clk)
        begin
                if (clk'event and clk = '1') then
                        if (b(0) = 'U') or (a(0)= 'U') or (c(0) = 'U') then
                                res1 <= (others => '0');
                                res2 <= (others => '0');
                        else
                                res1 <= multiplicar (a,b)(34 downto 17);
                                res2 <= multiplicar (a,c)(34 downto 17);
                        end if;
```

```
        end if;
    end process;
end multiarch;
```

ANEXO 2 – CÓDIGO DE LA DDS

```
--------------------------------------------------------------------------------
-- Company:        EPSC
-- Engineer:        Juan Antonio Guerrero Balmori
--
-- Create Date:    17:23:15 04/21/06
-- Design Name:    Direct Digital Syntetizer
-- Module Name:    DDS - ddsarch
-- Project Name:    Diseño e implementación con FPGA de un demodulador para
--                comunicaciones digitales.
-- Target Device:          Xilinx Virtex 4
-- Tool versions:          ISE 7.1
--
--------------------------------------------------------------------------------
-- Incluimos las librerías que vamos a utilizar
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.DDCpack.all;

-- Definimos la entidad
entity DDS is
   Port ( clk : in std_logic;
          fase : in std_logic_vector(8 downto 0);
          seno : out std_logic_vector(17 downto 0);
          coseno : out std_logic_vector(17 downto 0));
end DDS;

-- Definimos la arquitectura
architecture DDSarch of DDS is
        -- Declaramos un nuevo tipo de variable donde guardaremos la LUT
        type std_vector_vector is array (0 to 255) of std_logic_vector(16 downto 0);

begin
        process (clk)
                -- Constante que guarda el valor del primer cuadrante de un coseno.
                constant     LUT:     std_vector_vector     :=     ("11000011010100000",
        "11000011010011110",     "11000011010011000",     "11000011010001111",
        "11000011010000010",     "11000011001110001",     "11000011001011100",
        "11000011001000011",     "11000011000100111",     "11000011000000110",
        "11000010111100010",     "11000010110111011",     "11000010110001111",
        "11000010101100000",     "11000010100101100",     "11000010011110101",
        "11000010010111011",     "11000010001111100",     "11000010000111010",
        "11000001111110100",     "11000001110101010",     "11000001101011100",
        "11000001100001011",     "11000001010110110",     "11000001001011101",
        "11000001000000001",     "11000000110100000",     "11000000100111100",
        "11000000011010100",     "11000000001101001",     "10111111111111001",
        "10111111110000110",     "10111111100001111",     "10111111010010101",
        "10111111000010111",     "10111110110010101",     "10111110100001111",
```

```
"1011111001000110",   "1011110111111001",   "1011110110110100 0",
```

```
"0011111010100 0111",
```

```
			-- Modificamos la posición en el vector en función de la dirección
			if counter = '1' then
				int := int + faseint;
			else
				int := int - faseint;
			end if;

			-- Corregimos los errores de incremento.
			if int >	255 then -- Si nos pasamos del limite del vector...
				int := 255 - (int - 255); -- ...corregimos el valor...
				counter := '0'; -- ...cambiamos la dirección...
				-- ...y cambiamos el signo del coseno
				if signocos = '0' then
					signocos := '1';
				else
					signocos := '0';
				end if;
			else
				if int <	0 then -- Si nos pasamos del limite del vector...
					int := 0 - (int);	-- ...corregimos el valor...
					counter := '1'; -- ...cambiamos la dirección...
					-- ... y cambiamos el signo del seno
					if signosin = '0' then
						signosin := '1';
					else
						signosin := '0';
					end if;
				end if;
			end if;

		end if;
	end process;

end DDSarch;
```

ANEXO 3 – CÓDIGO DEL FILTRO CIC

```
--------------------------------------------------------------------------------
-- Company:       EPSC
-- Engineer:       Juan Antonio Guerrero Balmori
--
-- Create Date:    17:23:15 04/21/06
-- Design Name:    Filtro CIC
-- Module Name:    CIC - CICarch
-- Project Name:   Diseño e implementación con FPGA de un demodulador para
--                 comunicaciones digitales.
-- Target Device:       Xilinx Virtex 4
-- Tool versions:       ISE 7.1
--
--------------------------------------------------------------------------------

-- Incluimos las librerias que vamos a utilizar
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.DDCpack.all;

-- Definimos la entidad
entity CIC is
   Port ( entrada : in std_logic_vector(17 downto 0);
                  clk: in std_logic;
                  R: in std_logic_vector(5 downto 0);
        salida : out std_logic_vector(17 downto 0));
end CIC;

-- Definimos la arquitectura
architecture CICarch of CIC is

begin
        process (clk)

                variable i: std_logic_vector(14 downto 0) := (others=>'0');
                variable entradatemp: std_logic_vector(25 downto 0) := (others=>'0');
                variable salidaant: std_logic_vector(25 downto 0) := (others=>'0');
                variable Iout1: std_logic_vector(25 downto 0) := (others=>'0');
                variable Iant1: std_logic_vector(25 downto 0) := (others=>'0');
                variable Iout2: std_logic_vector(25 downto 0) := (others=>'0');
                variable Iant2: std_logic_vector(25 downto 0) := (others=>'0');
                variable Iout3: std_logic_vector(25 downto 0) := (others=>'0');
                variable Iant3: std_logic_vector(25 downto 0) := (others=>'0');
                variable Iout4: std_logic_vector(25 downto 0) := (others=>'0');
                variable Iant4: std_logic_vector(25 downto 0) := (others=>'0');
                variable Cout1: std_logic_vector(25 downto 0) := (others=>'0');
                variable Cant1: std_logic_vector(25 downto 0) := (others=>'0');
                variable Cout2: std_logic_vector(25 downto 0) := (others=>'0');
                variable Cant2: std_logic_vector(25 downto 0) := (others=>'0');
```

```
		variable Cout3: std_logic_vector(25 downto 0) := (others=>'0');
		variable Cant3: std_logic_vector(25 downto 0) := (others=>'0');
		variable Cout4: std_logic_vector(25 downto 0) := (others=>'0');
		variable Cant4: std_logic_vector(25 downto 0) := (others=>'0');

	begin

		if (clk'event and clk = '1') then
			if (entrada(0) = 'U') then
				salida <= (others => '0');
			else
				entradatemp(16 downto 0) := entrada(entrada'length-2
										downto 0);
				entradatemp(entradatemp'length-1) :=
								entrada(entrada'length-1);
				-- Primer integrador
				Iout1 := sumar (entradatemp,Iant1);
				Iant1 := Iout1;
				-- Segundo integrador
				Iout2 := sumar (Iout1,Iant2);
				Iant2 := Iout2;
				-- Tercer integrador
				Iout3 := sumar (Iout2,Iant3);
				Iant3 := Iout3;
				-- Cuarto integrador
				Iout4 := sumar (Iout3,Iant4);
				Iant4 := Iout4;
				if	(i = 0)		then
					-- Primer Comb
					Cout1 := sumar (Iout4,Cant1);
					Cant1 := Iout4;
					Cant1(Cant1'length-1) := not Iout3(Iout3'length-1);
					-- Segundo Comb
					Cout2 := sumar (Cout1,Cant2);
					Cant2 := Cout1;
					Cant2(Cant2'length-1) := not Cout1(Cout1'length-1);
					-- Tercer Comb
					Cout3 := sumar (Cout2,Cant3);
					Cant3 := Cout2;
					Cant3(Cant3'length-1) := not Cout2(Cout2'length-1);
					-- Cuarto Comb
					Cout4 := sumar (Cout3,Cant4);
					Cant4 := Cout3;
					Cant4(Cant4'length-1) := not Cout3(Cout3'length-1);

					salida <= Cout4(Cout4'length-1 downto Cout4'length-18);
					salidaant := Cout4;
				else
					salida <= salidaant(salidaant'length-1 downto
								salidaant'length-18);
				end if;

				i := i + "00000000000001";
				if (i = R) then
					i := (others=>'0');
```

```
                                        end if;
                              end if;
                    end if;
          end process;
end CICarch;
```

ANEXO 4 - CÓDIGO DEL FILTRO FIR

```
----------------------------------------------------------------------------------
-- Company:        EPSC
-- Engineer:        Juan Antonio Guerrero Balmori
--
-- Create Date:    17:23:15 04/21/06
-- Design Name:    Filtro FIR
-- Module Name:    FIR - FIRarch
-- Project Name:    Diseño e implementación con FPGA de un demodulador para
--                comunicaciones digitales.
-- Target Device:         Xilinx Virtex 4
-- Tool versions:          ISE 7.1
--
----------------------------------------------------------------------------------
-- Incluimos las librerias que vamos a utilizar
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.DDCpack.all;

-- Definimos la entidad
entity FIR is
   Port ( Xin : in std_logic_vector(17 downto 0);
                        sel : in std_logic_vector(3 downto 0);-- Selección de posición de los
coef.
                        coef : in std_logic_vector(10 downto 0);-- Entrada de coeficientes
             Yout : out std_logic_vector(30 downto 0));
end FIR;


-- Definimos la arquitectura
architecture FIRarch of FIR is

begin

        process(Xin,sel)
                -- Definimos las variables globales
                variable C0 : std_logic_vector(10 downto 0):= "00000000001";
                variable C1 : std_logic_vector(10 downto 0):= (others => '0');
                variable C2 : std_logic_vector(10 downto 0):= (others => '0');
                variable C3 : std_logic_vector(10 downto 0):= (others => '0');
                variable C4 : std_logic_vector(10 downto 0):= (others => '0');
                variable C5 : std_logic_vector(10 downto 0):= (others => '0');
                variable C6 : std_logic_vector(10 downto 0):= (others => '0');
                variable C7 : std_logic_vector(10 downto 0):= (others => '0');

                variable Z0 : std_logic_vector(17 downto 0) := (others => '0');
                variable Z1 : std_logic_vector(17 downto 0) := (others => '0');
                variable Z2 : std_logic_vector(17 downto 0) := (others => '0');
                variable Z3 : std_logic_vector(17 downto 0) := (others => '0');
```

```
        variable Z4 : std_logic_vector(17 downto 0) := (others => '0');
        variable Z5 : std_logic_vector(17 downto 0) := (others => '0');
        variable Z6 : std_logic_vector(17 downto 0) := (others => '0');
        variable Z7 : std_logic_vector(17 downto 0) := (others => '0');

        variable p0 : std_logic_vector(27 downto 0) := (others => '0');
        variable p1 : std_logic_vector(27 downto 0) := (others => '0');
        variable p2 : std_logic_vector(27 downto 0) := (others => '0');
        variable p3 : std_logic_vector(27 downto 0) := (others => '0');
        variable p4 : std_logic_vector(27 downto 0) := (others => '0');
        variable p5 : std_logic_vector(27 downto 0) := (others => '0');
        variable p6 : std_logic_vector(27 downto 0) := (others => '0');
        variable p7 : std_logic_vector(27 downto 0) := (others => '0');

        variable s0 : std_logic_vector(28 downto 0) := (others => '0');
        variable s1 : std_logic_vector(28 downto 0) := (others => '0');
        variable s2 : std_logic_vector(28 downto 0) := (others => '0');
        variable s3 : std_logic_vector(28 downto 0) := (others => '0');

    begin

        if sel /= "1111"    then
            case sel    is
                when ("0000")=> C0 := coef;
                when ("0001")=> C1 := coef;
                when ("0010")=> C2 := coef;
                when ("0011")=> C3 := coef;
                when ("0100")=> C4 := coef;
                when ("0101")=> C5 := coef;
                when ("0110")=> C6 := coef;
                when ("0111")=> C7 := coef;
                when others => Yout <= (others=>'1');
            end case;
        else
            if (Xin(0) = 'U') then
                Yout <= (others => 'U');
            else
                -- Desplazamos los valores
                Z7 := Z6;
                Z6 := Z5;
                Z5 := Z4;
                Z4 := Z3;
                Z3 := Z2;
                Z2 := Z1;
                Z1 := Z0;
                Z0 := Xin;

                -- Multiplicamos por los coeficientes
                p0 := multiplicar(C0,Z0);
                p1 := multiplicar(C1,Z1);
                p2 := multiplicar(C2,Z2);
                p3 := multiplicar(C3,Z3);
                p4 := multiplicar(C4,Z4);
                p5 := multiplicar(C5,Z5);
                p6 := multiplicar(C6,Z6);
```

```
                    p7 := multiplicar(C7,Z7);

                    -- y los sumamos
                    s0 := sumar2(p0,p1);
                    s1 := sumar2(p2,p3);
                    s2 := sumar2(p4,p5);
                    s3 := sumar2(p6,p7);

                    Yout <= sumar2(sumar2(s0,s1),
                                   sumar2(s2,s3));
               end if;
          end if;

     end process;

end FIRarch;
```

ANEXO 5 – CÓDIGO DEL DECISOR DE 16QAM

```
--------------------------------------------------------------------------------
-- Company:       EPSC
-- Engineer:       Juan Antonio Guerrero Balmori
--
-- Create Date:    17:23:15 04/21/06
-- Design Name:    Decisor 16QAM
-- Module Name:    QAM16 - QAMarch
-- Project Name:    Diseño e implementación con FPGA de un demodulador para
--                 comunicaciones digitales.
-- Target Device:         Xilinx Virtex 4
-- Tool versions:         ISE 7.1
--
--------------------------------------------------------------------------------

-- Incluimos las librerias que vamos a utilizar
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

-- Definimos la entidad
entity QAM16 is
   Port ( I : in std_logic_vector(30 downto 0);
          Q : in std_logic_vector(30 downto 0);
          salida : out std_logic_vector(3 downto 0));
end QAM16;

-- Definimos la arquitectura
architecture QAMarch of QAM16 is

begin
        process (I,Q)

        begin
                if (I(0) /= 'U')then
                        if(I < -536870912)then
                                if(Q < -536870912)then
                                        salida<="0000"; --3
                                elsif ((Q > -536870912) and (Q<=0))then
                                        salida<="0010"; --1
                                elsif ((Q>0) and (Q<536870911))then
                                        salida<="1000"; --10
                                elsif (Q>=536870911)then
                                        salida<="1001"; --11
                                end if;
                        elsif((I > -536870912) and (I<=0))then
                                if(Q<-536870912)then
                                        salida<="0001"; --2
                                elsif ((Q > -536870912) and (Q<=0))then
                                        salida<="0011"; --0
```

```
				elsif ((Q>0) and (Q<536870911))then
					salida<="1010"; --8
				elsif (Q>=536870911)then
					salida<="1011"; --9
				end if;
			elsif((I>0) and (I<536870911))then
				if(Q<-536870912)then
					salida<="0100"; --5
				elsif ((Q > -536870912) and (Q<=0))then
					salida<="0101"; --4
				elsif ((Q>0) and (Q<536870911))then
					salida<="1100"; --12
				elsif (Q>=536870911)then
					salida<="1110"; --14
				end if;
			elsif(I>=536870911)then
				if(Q<-536870912)then
					salida<="0110"; --7
				elsif ((Q > -536870912) and (Q<=0))then
					salida<="0111"; --6
				elsif ((Q>0) and (Q<536870911))then
					salida<="1101"; --13
				elsif (Q>=536870911)then
					salida<="1111"; --15
				end if;
			end if;
		else
			salida <= "UUUU";
		end if;
	end process;

end QAMarch;
```

ANEXO 6 – CÓDIGO DE LA LIBRERÍA DE FUNCIONES

```
--      Package File Template
--
--      Purpose: This package defines supplemental types, subtypes,
--               constants, and functions


library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

package DDCpack is

        function ca2 (a : std_logic_vector)
                                return std_logic_vector;
        function multiplicar (a: std_logic_vector;
                                b: std_logic_vector)
                                return std_logic_vector;
        function sumar (a,b: std_logic_vector)
                                return std_logic_vector;
        function sumar2 (a,b: std_logic_vector)
                                return std_logic_vector;

        function bintoint (a: std_logic_vector;
                                signo: std_logic)
                                return integer;
        function inttobin (a: integer;
                                tam: integer)
                                return std_logic_vector;
end DDCpack;


package body DDCpack is

-- Función que devuelve el complemento a 2 de un numero binario.
        function ca2 (a : std_logic_vector)
                        return std_logic_vector is

                variable restemp: std_logic_vector((a'length-1) downto 0):= (others => '0');
                variable carry: std_logic := '1';

        begin
                for j in 0 to (a'length-1) loop -- Recorremos los valores del numero
                        restemp(j) := not a(j) xor carry; -- Hallamos el complemento a 1 y
                                                        -- le sumamos 1 al mismo tiempo
                        if (a(j)='1' and carry ='1')then
                                carry := '0';
                        end if;
                end loop;
                return restemp;
```

```
	end ca2;

-- Función que devuelve el producto de dos vectores binarios.
	function multiplicar (a : std_logic_vector;
						b : std_logic_vector)
						return std_logic_vector is

		variable temp: std_logic_vector((a'length + b'length-3) downto 0):=
										(others => '0');
		variable restemp: std_logic_vector((a'length + b'length-2) downto 0):=
										(others => '0');
		variable restemp2: std_logic_vector((a'length + b'length-3) downto 0):=
										(others => '0');
		variable carry: std_logic := '0';

	begin
		restemp:= (others => '0');

		for i in 0 to a'length-2 loop -- Iniciamos un bucle para recorrer los valores de
										a.
			--Inicializamos la variable que guardarán el vector b en la posición
			--correspondiente para sumarlo
			temp := (others => '0');
			if a(i)='1' then -- Si a es 1 ...
				--...copiamos b en la posición que le toca
				temp ((i + b'length-2) downto i):= b((b'length-2) downto 0);
				-- Guardamos el resultado temporal de la suma anterior de a*b
				restemp2 := restemp((a'length + b'length-3) downto 0);

				for j in 0 to (a'length + b'length-3) loop -- Recorremos todas las
							posiciones de los vectores temporales
					-- realizamos la suma de a*b y a*c
					restemp(j) := restemp2(j) xor temp(j)xor carry;
					-- Si hay dos o mas 1 en la suma el carry será 1...
					if ((restemp2(j)='1' and temp(j)='1')
						or(restemp2(j)='1' and carry='1')
						or(temp(j)='1' and carry='1')) then
						carry := '1';
					-- ...sino será 0.
					else
						carry := '0';
					end if;
				end loop;
				--Inicializamos los carry para la siguiente suma
				carry := '0';
			end if;
		end loop;
		-- Ponemos el signo
		if (restemp = 0) then
			restemp(a'length + b'length-2):= '0';
		else
			restemp(a'length + b'length-2):= a(a'length-1) xor b(b'length-1);
		end if;
		return restemp;
	end multiplicar;
```

```
-- Función que devuelve la suma de dos vectores binarios.
	function sumar (a : std_logic_vector;
					b : std_logic_vector)
					return std_logic_vector is

		variable restemp: std_logic_vector((a'length-1) downto 0):= (others => '0');
		variable carry: std_logic := '0';
		variable neginv: std_logic_vector((a'length-1) downto 0):= (others => '1');
		constant resta: std_logic_vector((a'length-2) downto 0):= (others => '1');
	begin

		if (a(a'length-1) = b(b'length-1)) then  -- Si los 2 tienen el mismo signo
			for i in 0 to restemp'length-2 loop
				restemp(i):= a(i) xor b(i) xor carry; -- sumamos
				-- Actualizamos Carry
				if ((a(i)='1' and b(i)='1') or (a(i)='1' and carry='1')
					 or (carry='1' and b(i)='1'))then
					carry := '1';
				else
					carry := '0';
				end if;
			end loop;
			-- Ponemos el signo
			restemp(restemp'length-1):= a(a'length-1);
		else -- Si son de signos diferentes
			carry := '1';
			if (a(a'length-1)='1') then
				neginv(neginv'length-2 downto 0) := not a(a'length-2 downto 0);
				for i in 0 to restemp'length-2 loop
					restemp(i):= neginv(i) xor b(i) xor carry; -- sumamos
					-- Actualizamos Carry
					if ((neginv(i)='1' and b(i)='1') or (neginv(i)='1' and carry='1')
						 or (carry='1' and b(i)='1'))then
						carry := '1';
					else
						carry := '0';
					end if;
				end loop;
				-- Ponemos el signo
				if (a - ('0'& resta))> b then
					restemp(restemp'length-2 downto 0) :=
								ca2(restemp(restemp'length-2 downto 0));
					restemp(restemp'length-1):= a(a'length-1);
				else
					restemp(restemp'length-1):= b(b'length-1);
				end if;
			else
				neginv(neginv'length-2 downto 0) := not b(b'length-2 downto 0);
				for i in 0 to restemp'length-2 loop
					restemp(i):= a(i) xor neginv(i) xor carry; -- sumamos
					-- Actualizamos Carry
					if ((a(i)='1' and neginv(i)='1') or (a(i)='1' and carry='1')
						 or (carry='1' and neginv(i)='1'))then
						carry := '1';
```

```
                    else
                        carry := '0';
                    end if;
                end loop;
                -- Ponemos el signo
                if (b - ('0'& resta))> a then
                    restemp(restemp'length-2 downto 0) :=
                                ca2(restemp(restemp'length-2 downto 0));
                    restemp(restemp'length-1):= b(b'length-1);
                else
                    restemp(restemp'length-1):= a(a'length-1);
                end if;
            end if;
        end if;
        return restemp;
    end sumar;

-- Función que devuelve la suma de dos vectores binarios evitando desbordamiento.
    function sumar2 (a : std_logic_vector;
                         b : std_logic_vector)
                         return std_logic_vector is

        variable restemp: std_logic_vector((a'length) downto 0):= (others => '0');
        variable carry: std_logic := '0';
        variable neginv: std_logic_vector((a'length-1) downto 0):= (others => '1');
        constant resta: std_logic_vector((a'length-2) downto 0):= (others => '1');

    begin

        if (a(a'length-1) = b(b'length-1)) then  -- Si los 2 tienen el mismo signo
            for i in 0 to restemp'length-3 loop
                restemp(i):= a(i) xor b(i) xor carry; -- sumamos
                -- Actualizamos Carry
                if ((a(i)='1' and b(i)='1') or (a(i)='1' and carry='1')
                     or (carry='1' and b(i)='1'))then
                    carry := '1';
                else
                    carry := '0';
                end if;
            end loop;
            -- Ponemos el digito de overflow
            restemp(restemp'length-2) := carry;
            -- Ponemos el signo
            restemp(restemp'length-1):= a(a'length-1);
        else -- Si son de signos diferentes
            carry := '1';
            if (a(a'length-1)='1') then
                neginv(neginv'length-2 downto 0) := not a(a'length-2 downto 0);
                for i in 0 to restemp'length-3 loop
                    restemp(i):= neginv(i) xor b(i) xor carry; -- sumamos
                    -- Actualizamos Carry
                    if ((neginv(i)='1' and b(i)='1') or (neginv(i)='1' and carry='1')
                         or (carry='1' and b(i)='1'))then
                        carry := '1';
                    else
```

```
                                carry := '0';
                        end if;
                end loop;
                -- Ponemos el digito de overflow
                restemp(restemp'length-2) := '0';
                -- Ponemos el signo
                if (a - ('0'& resta))> b then
                        restemp(restemp'length-3 downto 0) :=
                                        ca2(restemp(restemp'length-3 downto 0));
                        restemp(restemp'length-1):= a(a'length-1);
                else
                        restemp(restemp'length-1):= b(b'length-1);
                end if;
            else
                neginv(neginv'length-2 downto 0) := not b(b'length-2 downto 0);
                for i in 0 to restemp'length-3 loop
                        restemp(i):= a(i) xor neginv(i) xor carry; -- sumamos
                        -- Actualizamos Carry
                        if ((a(i)='1' and neginv(i)='1') or (a(i)='1' and carry='1')
                                 or (carry='1' and neginv(i)='1'))then
                                carry := '1';
                        else
                                carry := '0';
                        end if;
                end loop;
                -- Ponemos el digito de overflow
                restemp(restemp'length-2) := '0';
                -- Ponemos el signo
                if (b - ('0'& resta))> a then
                        restemp(restemp'length-3 downto 0) :=
                                        ca2(restemp(restemp'length-3 downto 0));
                        restemp(restemp'length-1):= b(b'length-1);
                else
                        restemp(restemp'length-1):= a(a'length-1);
                end if;
            end if;
        end if;
        return restemp;
    end sumar2;

-- Función que convierte un binario en entero. 1 con signo/0 sin signo
    function bintoint (a: std_logic_vector;
                                    signo: std_logic)
                                            return integer is
        variable res: integer := 0;
        constant i: integer := a'length-1;
    begin
        for j in 0 to (a'length-2) loop
            if a(j)='1' then -- Si el bit es un 1...
                res := res + 2**j; --...sumamos 2 elevado a la posición del bit
            end if;
        end loop;

        if a(a'length-1)='1'        then -- En caso de ser 1
            if signo = '1' then --Si es con signo...
```

```
                                res := res * (-1); -- ...le cambiamos el signo
                        else -- Si es sin signo
                                res := res + 2**(i);--...sumamos 2 elevado a la posición del bit
                        end if;
                end if;
                return res;
        end bintoint;

-- Función que convierte un entero en binario.
        function inttobin (a: integer;
                                tam: integer)
                                return std_logic_vector is
                variable res: std_logic_vector((tam-1) downto 0);
                variable int, i: integer;
        begin
                res := (others => '0');
                int := a;
                i := 0;
                while ((int /= 0) and (i<res'length-1)) loop
                        if (int REM 2 = 1)or(int REM 2 = -1) then
                                res(i) := '1';
                        end if;
                        int := int/2;
                        i := i + 1;
                end loop;
                if(a < 0) then
                        res(res'length-1):='1';
                end if;
                return res;
        end inttobin;

end DDCpack;
```

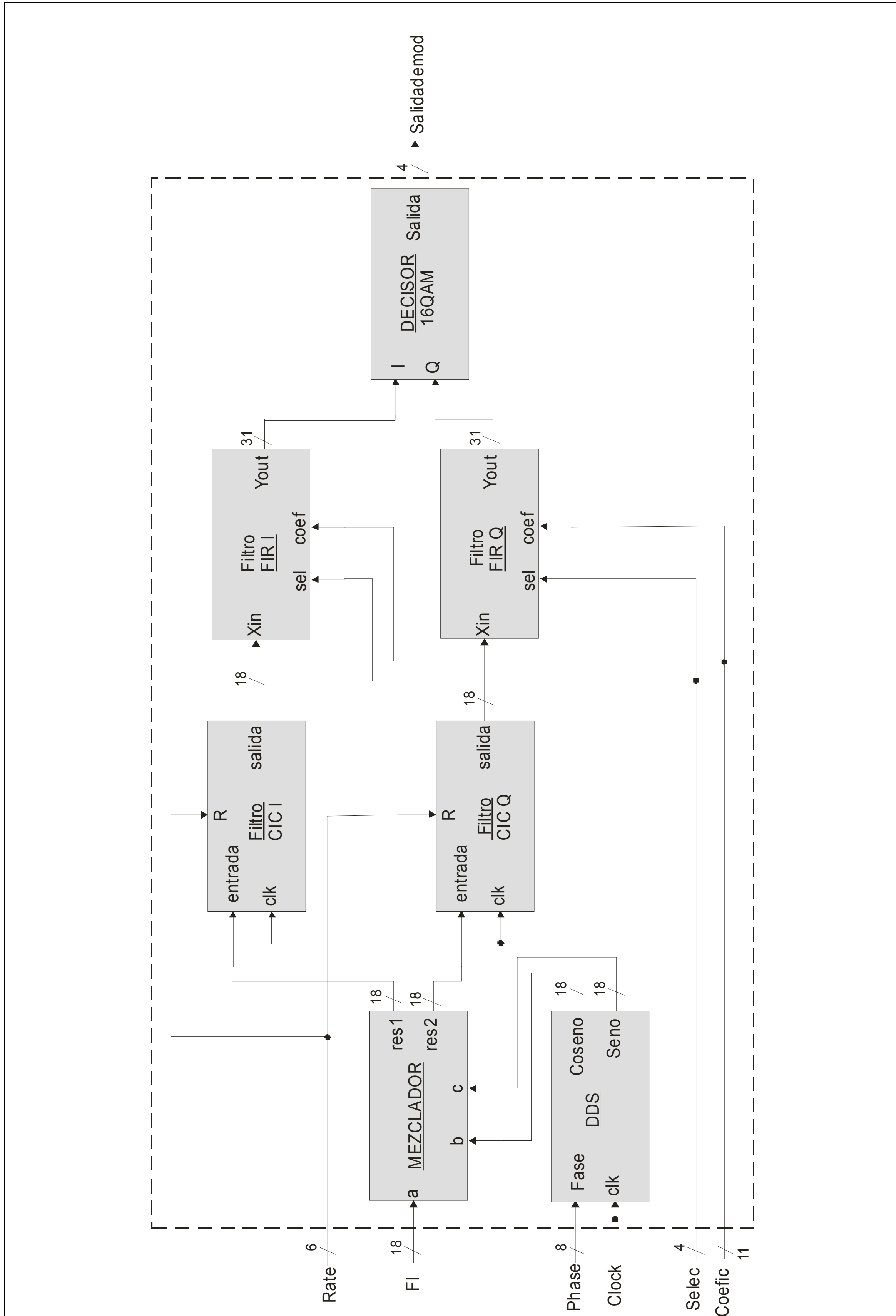
ANEXO 7 – CÓDIGO DEL DEMODULADOR

```
--------------------------------------------------------------------------------
-- Company:      EPSC
-- Engineer:       Juan Antonio Guerrero Balmori
--
-- Create Date:    17:23:15 04/21/06
-- Design Name:    Demodulador
-- Module Name:    demod - demodarch
-- Project Name:      Diseño e implementación con FPGA de un demodulador
para
--                comunicaciones digitales.
-- Target Device:      Xilinx Virtex 4
-- Tool versions:               ISE 7.1
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.ddcpack.all;
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity demod is
   Port ( FI : in std_logic_vector(17 downto 0);
                        clock : in std_logic;

                        -- Parametros de control de la DDS
                        Phase : in std_logic_vector(8 downto 0);

                        -- Parametros de control del filtro CIC
                        Rate : in std_logic_vector(5 downto 0);

                        -- Parametros de control del filtro FIR
                        Selec : in std_logic_vector(3 downto 0);
                        coefic : in std_logic_vector(10 downto 0);

                        -- Salida demodulada en formato 16QAM
                        salidademod : out std_logic_vector(3 downto 0));
end demod;

architecture demodarch of demod is

        --Declaramos los distintos componentes de la DDC.
        component DDS --Declaramos la dds.
```

```
            Port ( clk : in std_logic;
             fase : in std_logic_vector(8 downto 0);
             seno : out std_logic_vector(17 downto 0);
             coseno : out std_logic_vector(17 downto 0));
      end component;

      component multip --Declaramos el multiplicador.
            Port ( a : in std_logic_vector(17 downto 0);
             b : in std_logic_vector(17 downto 0);
             c : in std_logic_vector(17 downto 0);
             clk : in std_logic;
                        res1 : out std_logic_vector(17 downto 0);
             res2 : out std_logic_vector(17 downto 0));
      end component;

      component CIC --Declaramos el filtro CIC.
            Port ( entrada : in std_logic_vector(17 downto 0);
                     clk: in std_logic;
                     R: in std_logic_vector(5 downto 0);
         salida : out std_logic_vector(17 downto 0));
      end component;

      component FIR --Declaramos    el filtro FIR.
      Port ( Xin : in std_logic_vector(17 downto 0);
                     sel : in std_logic_vector(3 downto 0);
                     coef : in std_logic_vector(10 downto 0);
         Yout : out std_logic_vector(30 downto 0));
      end component;

      component QAM16 --Declaramos      el filtro FIR.
      Port ( I : in std_logic_vector(30 downto 0);
         Q : in std_logic_vector(30 downto 0);
         salida : out std_logic_vector(3 downto 0));
      end component;

      signal buffer1: std_logic_vector(17 downto 0);
      signal buffer2: std_logic_vector(17 downto 0);
      signal buffer3: std_logic_vector(17 downto 0);
      signal buffer4: std_logic_vector(17 downto 0);
      signal buffer5: std_logic_vector(17 downto 0);
      signal buffer6: std_logic_vector(17 downto 0);
      signal Iout : std_logic_vector(30 downto 0);
  signal Qout : std_logic_vector(30 downto 0);

begin

      --Realizamos la asignación de puertos.
      dds1: dds
                  port map (
                 clk => clock,
```

```
                fase => Phase,
                        coseno => buffer1,
                seno => buffer2);

      multip1: multip
                 Port map( a => FI,
              b => buffer1,
              c => buffer2,
              clk => clock,
                          res1 => buffer3,
              res2 => buffer4);

      CICI: CIC
                 Port map( entrada => buffer3,
                    clk => clock,
                    R => Rate,
         salida => buffer5);

      CICQ: CIC
                 Port map( entrada => buffer4,
                    clk => clock,
                    R => Rate,
         salida => buffer6);

      FIRI: FIR --Declaramos     el filtro FIR.
      Port map( Xin => buffer5,
                        sel => selec,
                        coef => coefic,
         Yout => Iout);

      FIRQ: FIR --Declaramos   el filtro FIR.
      Port map( Xin => buffer6,
                        sel => selec,
                        coef => coefic,
         Yout => Qout);

      QAM161: QAM16 --Declaramosel decisor para 16qam.
      Port map( I => Iout,
         Q => Qout,
         salida => salidademod);

      process (clock)
      begin
      end process;
end demodarch;
```

ANEXO 8 – DIAGRAMA DE BLOQUES DEL DEMODULADOR



ANEXO 9 – COMPARATIVAS VIRTEX 4

Device	Configurable Logic Blocks (CLBs)(1)				XtremeDSP Slices(2)	Block RAM		DCMs	PMCDs	PowerPC Processor Blocks	Ethernet MACs	RocketIO Transceiver Blocks	Total I/O Banks	Max User I/O
	Array(3) Row x Col	Logic Cells	Slices	Max Distributed RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)							
XC4VLX15	64 x 24	13,824	6,144	96	32	48	864	4	0	N/A	N/A	N/A	9	320
XC4VLX25	96 x 28	24,192	10,752	168	48	72	1,296	8	4	N/A	N/A	N/A	11	448
XC4VLX40	128 x 36	41,472	18,432	288	64	96	1,728	8	4	N/A	N/A	N/A	13	640
XC4VLX60	128 x 52	59,904	26,624	416	64	160	2,880	8	4	N/A	N/A	N/A	13	640
XC4VLX80	160 x 56	80,640	35,840	560	80	200	3,600	12	8	N/A	N/A	N/A	15	768
XC4VLX100	192 x 64	110,592	49,152	768	96	240	4,320	12	8	N/A	N/A	N/A	17	960
XC4VLX160	192 x 88	152,064	67,584	1056	96	288	5,184	12	8	N/A	N/A	N/A	17	960
XC4VLX200	192 x 116	200,448	89,088	1392	96	336	6,048	12	8	N/A	N/A	N/A	17	960

Device	Configurable Logic Blocks (CLBs)[1]				XtremeDSP Slices[2]	Block RAM		DCMs	PMCDs	PowerPC Processor Blocks	Ethernet MACs	RocketIO Transceiver Blocks	Total I/O Banks	Max User I/O
	Array[3] Row x Col	Logic Cells	Slices	Max Distributed RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)							
XC4VSX25	64 x 40	23,040	10,240	160	128	128	2,304	4	0	N/A	N/A	N/A	9	320
XC4VSX35	96 x 40	34,560	15,360	240	192	192	3,456	8	4	N/A	N/A	N/A	11	448
XC4VSX55	128 x 48	55,296	24,576	384	512	320	5,760	8	4	N/A	N/A	N/A	13	640
XC4VFX12	64 x 24	12,312	5,472	86	32	36	648	4	0	1	2	N/A	9	320
XC4VFX20	64 x 36	19,224	8,544	134	32	68	1,224	4	0	1	2	8	9	320
XC4VFX40	96 x 52	41,904	18,642	291	48	144	2,592	8	4	2	4	12	11	448
XC4VFX60	128 x 52	56,880	25,280	395	128	232	4,176	12	8	2	4	16	13	576
XC4VFX100	160 x 68	94,896	42,176	659	160	376	6,768	12	8	2	4	20	15	768
XC4VFX140	192 x 84	142,128	63,168	987	192	552	9,936	20	8	2	4	24	17	896