Proyecto Fin de Carrera
Ingeniería Aeroespacial

Realización electrónica en FPGA de un convertidor Digital/Analógico con PWM como interfaz analógica

Autor: Jacobo Manuel Ruiz Bocanegra

Tutor: Francisco Colodro Ruiz

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Proyecto Fin de Carrera
Ingeniería Aeroespacial

Realización electrónica en FPGA de un convertidor Digital/Analógico con PWM como interfaz analógica

Autor:
Jacobo Manuel Ruiz Bocanegra

Tutor:
Francisco Colodro Ruiz
Profesor Titular

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Proyecto Fin de Carrera: Realización electrónica en FPGA de un convertidor Digital/Analógico con PWM como interfaz analógica

Autor: Jacobo Manuel Ruiz Bocanegra
Tutor: Francisco Colodro Ruiz

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mi familia, por haberme apoyado y motivado siempre a seguir adelante, tanto antes como tras empezar la carrera; a los compañeros de clase, amigos y profesores que me hayan ayudado, tanto con explicaciones como con consejos, y en especial a mi tutor Francisco Colodro Ruiz por poner tanto de su parte en la realización de este proyecto.

Jacobo Manuel Ruiz Bocanegra

Sevilla, 2022

Resumen

El objetivo de este proyecto es realizar e implementar en una FPGA un convertidor Digital/Analógico de ancho de banda parametrizable, basado en un modulador Sigma Delta de segundo orden de 16 niveles y en un modulador de ancho de pulsos.

Para ello, se ha partido del diseño de dicho convertidor en Simulink. Se han llevado a cabo las simulaciones necesarias en MATLAB y Simulink para analizar el comportamiento del convertidor y para darle dimensiones a las señales que se han usado para describirlo en lenguaje VHDL.

Finalizados todos los estudios previos, se han desarrollado los códigos necesarios para realizar el convertidor en VHDL empleando la herramienta ISE de Xilinx. Dicha herramienta ha permitido realizar simulaciones cuyos resultados se han analizado para comprobar que el diseño del convertidor en VHDL se había realizado de forma satisfactoria.

Una vez acabadas las simulaciones en VHDL, se ha realizado el último paso de este proyecto: implementar los códigos que conforman al convertidor en una placa FPGA. Usando un osciloscopio digital PicoScope 6, se han obtenido resultados experimentales que se han estudiado y comparado con los obtenidos en las simulaciones de los pasos anteriores para así determinar si los resultados finales han sido satisfactorios.

Abstract

The objective of this project is to make and implement in a FPGA a parameterizable bandwidth Digital/Analog converter, based on a 16-level second order Sigma Delta modulator and a pulse width modulator.

To do this, the design of said converter in Simulink has been used. The necessary simulations have been carried out in MATLAB and Simulink to analyze the behavior of the converter and to give dimensions to the signals that have been used to describe it in VHDL language.

Once all the previous studies have been completed, the necessary codes have been developed to make the converter in VHDL using the Xilinx ISE tool. This tool has allowed simulations to be carried out, the results of which have been analyzed to verify that the design of the converter in VHDL had been carried out satisfactorily.

Once the VHDL simulations have been completed, the last step of this project has been carried out: implementing the codes that make up the converter on an FPGA board. Using a PicoScope 6 digital oscilloscope, experimental results have been obtained that have been studied and compared with those obtained in the simulations of the previous steps in order to determine if the final results have been satisfactory.

Índice Abreviado

Resumen III
Abstract V
Índice Abreviado VII
Notación XIII

1 Introducción 1

2 Marco Teórico 3
2.1 Señal 3
2.2 Muestreo de una señal 3
2.3 Efecto de Aliasing 4
2.4 Teorema de Nyquist 6
2.5 Complemento a dos 6
2.6 Proceso de cuantificación 7
2.7 Paso y error de cuantificación 8
2.8 Tipos de cuantificación 9
2.9 Convertidor digital analógico 9
2.10 Transformada Z 9
2.11 SNR Y HD 9
2.12 Convertidor sobremuestreado 10
2.13 Modulador Sigma/Delta 10
2.14 Modulador de ancho de pulsos 13
2.15 FPGA 16

3 Simulación en MATLAB y SIMULINK 17
3.1 Modelos Simulink del Modulador Sigma/Delta y del Modulador de Ancho de Pulsos 17
3.2 Comparación 22

4 Simulación en VHDL 27
4.1 Bloque 1 28
4.2 Bloque 2 34
4.3 Bloque 3 41

4.4 Bloque 4 45
4.5 Bloque TOP 47
4.6 Simulaciones en ISE de Xilinx 47

5 Obtención de resultados 55
5.1 Filtro de paso bajo 55
5.2 Almacenamiento de los resultados experimentales 57
5.3 Análisis de los resultados experimentales 58
5.4 SNR, HD2 y HD3 63

6 Conclusiones y posibles mejoras 69

Apéndice A Códigos de MATLAB 71
A.1 Códigos previos a los análisis de la descripción en VHDL 71
A.2 Códigos para analizar resultados de VHDL 76
A.3 Códigos para analizar resultados de la FPGA 79

Apéndice B Códigos VHDL 81
B.1 Bloque TOP 81
B.2 Bloque 1 86
B.3 Bloque 2 98
B.4 Bloque 3 115
B.5 Bloque 4 119
B.6 Simulación del TOP 120

Apéndice C Gráficas de las frecuencias de corte 125

Apéndice D Efecto de solapamiento a la salida (Crosstalk) 129

Apéndice E Resultados de las simulaciones para los pines B4 de J1 y A6 de J2 133

Índice de Figuras 139
Índice de Tablas 143
Índice de Códigos 145
Bibliografía 147

Índice

Resumen III
Abstract V
Índice Abreviado VII
Notación XIII

1 Introducción **1**

2 Marco Teórico **3**
2.1 Señal 3
2.2 Muestreo de una señal 3
2.3 Efecto de Aliasing 4
2.4 Teorema de Nyquist 6
2.5 Complemento a dos 6
2.6 Proceso de cuantificación 7
2.7 Paso y error de cuantificación 8
2.8 Tipos de cuantificación 9
2.9 Convertidor digital analógico 9
2.10 Transformada Z 9
2.11 SNR Y HD 9
2.11.1 SNR 9
2.11.2 HD 10
2.12 Convertidor sobremuestreado 10
2.13 Modulador Sigma/Delta 10
2.14 Modulador de ancho de pulsos 13
2.14.1 Corrección del error asociado a los PWM 15
2.15 FPGA 16

3 Simulación en MATLAB y SIMULINK **17**
3.1 Modelos Simulink del Modulador Sigma/Delta y del Modulador de Ancho de Pulsos 17
3.1.1 Modelo 1: Modulador Sigma/Delta 18
3.1.2 Modelo 2: Modulador Sigma/Delta y de ancho de pulsos sin corrección 19
3.1.3 Modelo 3: Modulador Sigma/Delta y de ancho de pulsos con corrección 20

3.1.4 Modelo 4: Modulador Sigma/Delta y de ancho de pulsos con corrección y señales enteras 20
3.2 Comparación 22

4 Simulación en VHDL 27
4.1 Bloque 1 28
4.1.1 Generación de PulsoSD y PulsoPWM 29
4.1.2 Generador pulsoRZ 30
4.1.3 Contador n y Valores 31
4.2 Bloque 2 34
4.2.1 Programa sumador 34
4.2.2 Programa restador 35
4.2.3 Cuantificador Q 35
4.2.4 Cuantificador LUT 37
4.2.5 Retraso tras LUT 37
4.2.6 Retraso tras yn 40
4.3 Bloque 3 41
4.3.1 Registro de desplazamiento 41
4.3.2 Obtención de la salida 43
4.3.3 Bloques en VHDL 44
4.4 Bloque 4 45
4.5 Bloque TOP 47
4.6 Simulaciones en ISE de Xilinx 47
4.6.1 Entrada al modulador Sigma/Delta 47
4.6.2 Salida del modulador Sigma/Delta 48
4.6.3 Salida del programa 49
4.6.4 Simulación Post-Implementación 50
4.6.5 Preparación para la simulación con FPGA 51
4.6.6 Conexión de la FPGA 52

5 Obtención de resultados 55
5.1 Filtro de paso bajo 55
5.2 Almacenamiento de los resultados experimentales 57
5.3 Análisis de los resultados experimentales 58
5.3.1 Señal salNRZ para NPWM=2 58
5.3.2 Señal salRZ para NPWM=2 59
5.3.3 Señal salNRZ para NPWM=12 60
5.3.4 Señal salRZ para NPWM=12 60
5.3.5 Señal salNRZ para NPWM=20 61
5.3.6 Señal salRZ para NPWM=20 62
5.4 SNR, HD2 y HD3 63
5.4.1 Comportamiento de HD3 64
5.4.2 Análisis de HD2: forma de los pulsos 64
5.4.3 Análisis de la SNR 66

6 Conclusiones y posibles mejoras **69**

Apéndice A Códigos de MATLAB **71**
A.1 Códigos previos a los análisis de la descripción en VHDL 71
A.1.1 Analizador modelos Simulink 71
A.1.2 Generador de la entrada 75
A.2 Códigos para analizar resultados de VHDL 76
A.2.1 Análisis entrada y salida SDM 76
A.2.2 Análisis salida del PWM 77
A.3 Códigos para analizar resultados de la FPGA 79
A.3.1 Análisis salida FPGA 79

Apéndice B Códigos VHDL **81**
B.1 Bloque TOP 81
B.2 Bloque 1 86
B.3 Bloque 2 98
B.4 Bloque 3 115
B.5 Bloque 4 119
B.6 Simulación del TOP 120

Apéndice C Gráficas de las frecuencias de corte **125**

Apéndice D Efecto de solapamiento a la salida (Crosstalk) **129**

Apéndice E Resultados de las simulaciones para los pines B4 de J1 y A6 de J2 **133**

Índice de Figuras 139
Índice de Tablas 143
Índice de Códigos 145
Bibliografía 147

Notación

DAC	Convertidor Digital a Analógico
SDM	Modulador Sigma Delta
PWM	Modulador de Ancho de Pulsos
δ	Delta de Dirac
Δ	Paso de cuantificación
CA2	Complemento a dos
SNR	Relación señal-ruido
HD	Distorsión armónica
NTF	Función de transferencia del ruido
STF	Función de transferencia de la señal
LPF	Filtro paso bajo
L	Niveles del PWM
FPGA	Matriz de Puertas Programables en Campo
B	Ancho de Banda
A	Amplitud
OSR	Ratio de Sobremuestreo
LUT	Lookup Table. Tabla de correspondencia

1 Introducción

En el mundo tecnológico actual, el procesamiento digital de señales presenta una importancia vital debida al gran número de ventajas que aporta, siendo algunas de ellas la altísima velocidad que alcanza, su adaptabilidad, su durabilidad, su versatilidad y la constante disminución del peso de sus componentes.

Sin embargo, para procesar señales digitalmente se necesitan muchas herramientas, como sensores, transductores o convertidores analógico-digitales. Y es de estos últimos, de los convertidores, en los que se ha basado este trabajo.

En este proyecto se va la realizar un convertidor digital analógico (en inglés Digital-to-analogue converter, DAC) caracterizado por:

- La salida será una señal de 1 bit que trabaja a alta velocidad.
- Ancho de banda y frecuencia de señal de entrada estarán parametrizados en el código VHDL.
- Reloj maestro que controla el comportamiento del circuito
- Cuantificador interno del modulador Sigma/Delta (SDM) de 16 niveles y modulación por ancho de pulsos (PWM)

Este convertidor trabajará con números binarios enteros y codificados en complemento a dos.

El proceso para llevar a cabo el convertidor ha estado dividido en varias etapas:

- Partiendo del convertidor diseñado en Simulink, se hicieron simulaciones y análisis de los resultados que dieron dichas simulaciones en *Matlab* y *Simulink*, lo que permitió hacer una estimación a priori de qué resultados experimentales se obtendrían y tener una referencia con la que comparar dichos resultados una vez que se consiguieron. Estas simulaciones también fueron de vital importancia debido a que permitieron conocer el número de bits necesarios para cada una de las señales que se deben crear al describir el convertidor en VHDL.
- Tras hacer las simulaciones, se ha tenido que describir el convertidor en VHDL, codificando las distintas partes que conforman el convertidor en Simulink de forma que se consiguiera que cumpliera la misma función. Una vez hecho esto, se han

realizado una síntesis e implementación de los códigos en VHDL, y se han comparado los resultados que se obtenían con los que se obtuvieron en Simulink y Matlab.

- Tras comprobar que con los modos de síntesis e implementación se obtenían resultados aceptables a la salida del programa, se ha realizado una post-síntesis y post-implementación, que permite sacar resultados teniendo en cuenta las variaciones en la salida que pueden producirse por los posibles retrasos que pueden aparecer al exportar los códigos a una placa.
- Una vez comprobado que los resultados de todas las simulaciones son válidos, se ha procedido a conectar el programa con una FPGA. Dicha placa ha generado la salida, que ha sido filtrada con un filtro paso bajo y medida con un osciloscopio digital, lo que ha permitido representar en el ordenador dicha salida.
- Por último, se han almacenado estos datos y se han obtenido los valores que definen la calidad de la señal de salida en Matlab, comparándolos con los que se obtuvieron en las diferentes simulaciones que se han hecho hasta ese momento. Una vez acabados todos estos análisis, se ha dado por terminado este proyecto.

2 Marco Teórico

En este capítulo se van a explicar los conceptos teóricos que envuelven a este trabajo, de los que se ha partido y que son necesarios para comprender los propósitos, resultados y conclusiones de este proyecto.

2.1 Señal

Una señal es una perturbación física que puede medirse y que transporta información. Las señales, al expresar magnitudes físicas, suelen mostrar variaciones continuas en un tiempo continuo. A este tipo de señales se las denomina señales analógicas.

La naturaleza analógica de las señales impide que se puedan manipular mediante sistemas digitales. Es por ello que deben digitalizarse. El muestreo es el primer paso que se debe llevar a cabo cuando se va a digitalizar una señal.

2.2 Muestreo de una señal

Muestrear una señal analógica consiste en tomar muestras del valor de su amplitud en intervalos de tiempo equidistantes, lo que implica que las muestras de la señal a muestrear sean temporalmente equidistantes. De esta forma, se habrá obtenido una secuencia de valores a partir de la señal continua, tal y cómo se ve ejemplificado en la figura 2.1.

La diferencia de tiempo entre dos muestras consecutivas se denomina período de muestreo, T_s, y su inversa se denomina frecuencia de muestreo, f_s, que representa el número de muestras que se toman por segundo.

El proceso del muestreo se puede representar matemáticamente con la siguiente expresión:

$$y(t) = x(t) \cdot \sum_{n=-\infty}^{\infty} \delta(t - n \cdot T_s) \tag{2.1}$$

Por lo tanto, el proceso de muestrear se puede ver cómo convertir la señal en un sumatorio de deltas de Dirac cuya amplitud pasa a ser el valor de la señal analógica en cada punto. Dichos puntos estarán asociados a instantes de tiempo múltiplos de T_s.

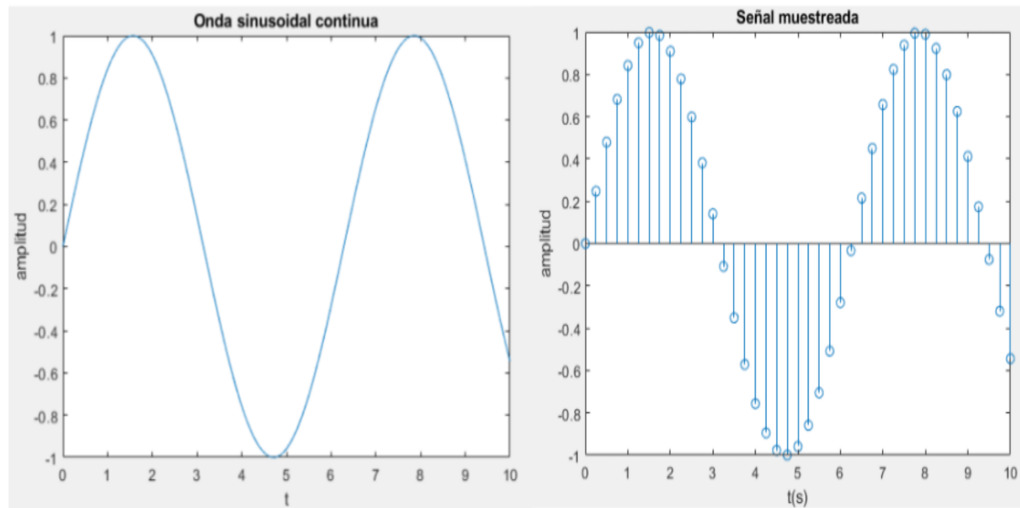

Figura 2.1 Ejemplo de señal muestreada.

Considerando ahora que el espectro de $x(t)$ en el dominio de la frecuencia es $X(f)$, la expresión anterior puede reescribirse como:

$$Y(f) = f_s \cdot \sum_{n=-\infty}^{\infty} X(f - n \cdot f_s) \tag{2.2}$$

Donde se observa que, en el dominio de la frecuencia, la señal muestreada se forma como repeticiones del espectro de la señal original, distanciadas entre ellas un valor igual al de la frecuencia de muestreo, tal y como se ve en la siguiente imagen:

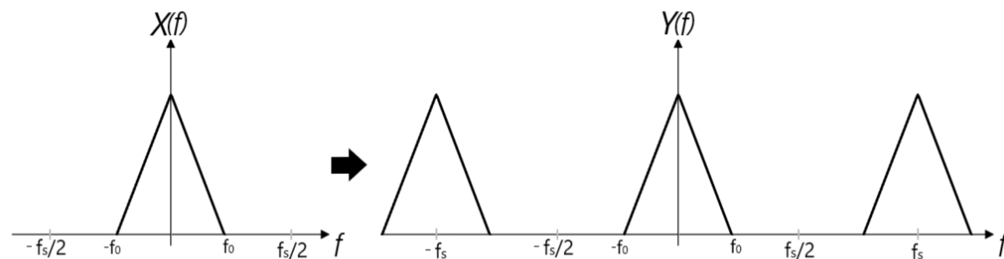

Figura 2.2 Espectro de una señal muestreada.

En este caso, mediante un filtro se puede recuperar el espectro de la señal original, como se ve en la figura 2.3. Este filtro $H(f)$ no alterará la señal para valores de frecuencia comprendidos entre $-f_0$ y f_0, eliminará aquellos asociados a las réplicas que se producen al muestrear la señal.

2.3 Efecto de Aliasing

El aliasing es un efecto que se presenta cuando la frecuencia con la que se muestrea una señal, f_s, no es la adecuada, lo que da lugar a que la señal que se obtiene al muestrear no sea lo suficientemente buena como para permitir recuperar la señal original a partir de ella.

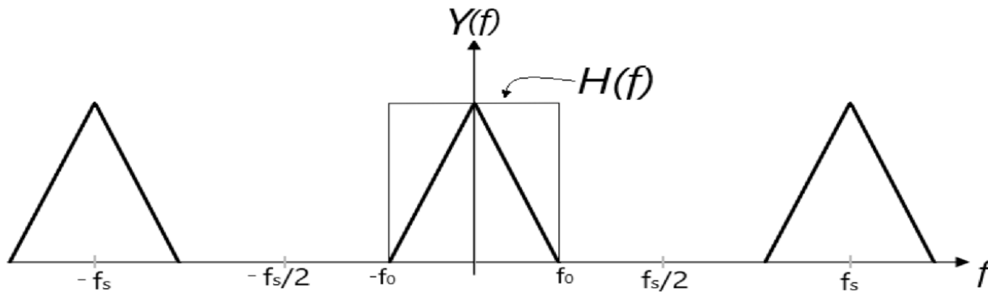

Figura 2.3 Espectro de una señal muestreada con filtro.

La razón por la cual el valor de f_s influye en la posibilidad de recuperar la señal original se puede ver representado en la siguiente imagen:

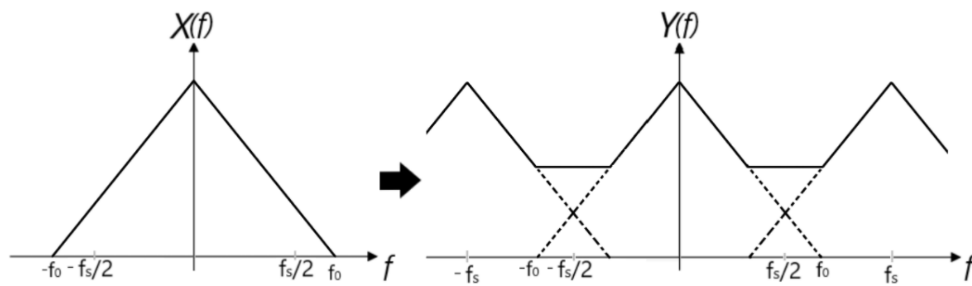

Figura 2.4 Espectro de una señal muestreada con aliasing.

Como se observa, el espectro de la señal original y de sus repeticiones ahora se superponen, debido a que el valor de f_s no es lo suficientemente grande para que las repeticiones del espectro estén lo suficientemente alejadas del espectro de la señal antes de ser muestreada. Esto da lugar a que si se usa un filtro, $H(f)$, similar al del caso anterior, se obtiene que ya no es posible recuperar la señal original a partir de la muestreada, debido a que la frecuencia de muestreo no ha sido lo suficientemente grande como para que no se produzca solapamiento.

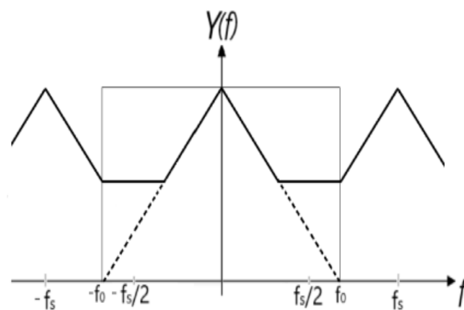

Figura 2.5 Espectro de una señal muestreada y filtrada con aliasing.

Se necesita por lo tanto un criterio que permita conocer cuál es la frecuencia mínima de muestreo, f_s, de forma que se pueda recuperar la señal una vez muestreada.

2.4 Teorema de Nyquist

El teorema de Nyquist establece el valor mínimo que debe tener la frecuencia de muestreo, f_s, para que la señal original pueda recuperarse a partir de la muestreada. Dicho teorema establece que la frecuencia de muestreo debe cumplir la siguiente relación con la frecuencia f_0:

$$\frac{f_s}{2} \geq f_0 \rightarrow f_s \geq 2f_0 = f_{Nyquist} \tag{2.3}$$

Se cumplirá, por lo tanto, que siempre que la frecuencia de muestreo sea mayor que la de Nyquist, teóricamente, se podrá reconstruir la señal original a partir de sus muestras.

Por otro lado, la relación entre la $f_{Nyquist}$ y la f_s se denomina OSR (Ratio de Sobremuestreo):

$$\frac{f_s}{f_{Nyquist}} = OSR \tag{2.4}$$

Para una frecuencia $f_{Nyquist}$ dada, cuanto mayor sea el OSR mayor será la frecuencia de muestreo f_s.

2.5 Complemento a dos

Se va a explicar a continuación en qué consiste codificar los números binarios en complemento a dos, ya que en este trabajo los valores de las señales de los códigos en VHDL van a estar codificadas de esta forma.

Esta codificación es especialmente útil cuando se trata con números negativos. Para llevarla a cabo, cuando se tenga un número negativo entero, x, codificarlo en complemento a dos será tan fácil como codificar en binario natural el valor $2^n + x$, siendo n el número de bits disponible para realizar la conversión a binario.

Una vez que se tenga el número negativo codificado en CA2, para para obtener su valor absoluto se le deberá hacer su complemento a dos, que consiste en cambiar sus ‘0’ por ‘1’ y viceversa (es decir, hacerle su complemento a uno) y tras esto sumarle una unidad al resultado.

Para el caso de n bits, el rango de números enteros codificado será $[-2^{n-1}, 2^{n-1}-1]$.

Otra de las ventajas de usar el complemento a dos es que simplifica el proceso de restar, ya que la resta del número binario ‘b’ al número binario ‘a’ se reduce a la suma de ‘a’ y el complemento a dos de ‘b’. Esta propiedad se usará a lo largo de la codificación del convertidor en VHDL, tal y como se verá en los siguientes capítulos.

En la tabla 2.1 se puede ver como se representan los números en binario para el caso de n=3:

Tabla 2.1 Codificación en CA2 para 3 bits .

x_2	x_1	x_0	$x\|_{10}$
1	0	0	-4
1	0	1	-3
1	1	0	-2
1	1	1	-1
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3

Como se observa, los números positivos, al ser pasados a binario natural, ya están automáticamente en CA2. Por otro lado, para identificar qué números son negativos y cuáles positivos en CA2, se debe prestar atención al valor de su primer dígito: si este es ’0’, el número en cuestión es positivo. Si, por otro lado, es ’1’, el número será negativo, y para conocer su valor en decimal habrá que realizarle la conversión que se ha explicado en el los apartados anteriores.

2.6 Proceso de cuantificación

Tras haber realizado el muestreo de la señal analógica, el siguiente paso es realizar la cuantificación de las muestras, que consiste en asociar a cada uno de los valores muestreados discretos uno de los niveles del cuantificador. De esta forma, al cuantificar la señal muestreada se reduce el número de posibles valores que pueden tener las muestras que la componen. El nivel del cuantificador que determinará el nuevo valor de cada una de las muestras de la señal original se asigna de forma que sea el más cercano posible al valor de la muestra original, disminuyendo así el error que puede generar el proceso de cuantificación. Por este proceso, la señal muestreada pasa a ser discreta en su amplitud, y no solo en el tiempo.

Tal y como se ve en la figura 2.6, el proceso de cuantificación está limitado a un rango de entrada. Para valores de muestras dentro de dicho rango, el nuevo valor de la muestra será necesariamente cercano a esta, o lo que es lo mismo, la diferencia entre el valor de la muestra después y antes de ser cuantificada estará limitado. Sin embargo, para entradas fuera de dicho rango, el valor de la diferencia entre la salida y la entrada no puede acotarse.

Como se ve en la imagen, el proceso de cuantificar puede venir acompañado de una codificación, si se hace que los valores de los niveles que puede tomar la salida estén codificados en binario. Debido a que en este proyecto la entrada va a estar codificada en binario directamente, el paso de la cuantificación no estará realmente acompañado de un proceso de codificación en este trabajo.

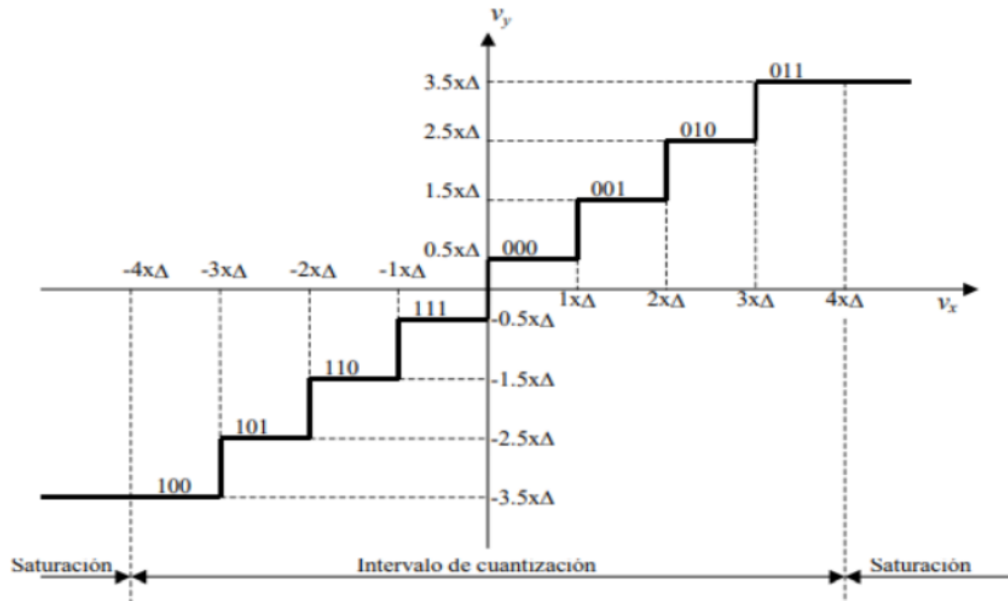

Figura 2.6 Representación del proceso de cuantificar una señal con 3 bits en CA2[1].

2.7 Paso y error de cuantificación

El proceso de cuantificar da lugar al denominado error o ruido de cuantificación, que se define como la diferencia entre el valor de las muestras antes de ser cuantificadas y tras haber pasado por el cuantificador. Este error podrá comportarse de dos formas distintas dependiendo del valor muestreado a cuantificar; si dicho valor está dentro del intervalo de cuantificación de entrada, el error estará limitado. Sin embargo, para el caso de entradas fuera de ese rango de entrada, el error de cuantificación podrá tomar cualquier valor, tal y como se representa en la figura 2.7.

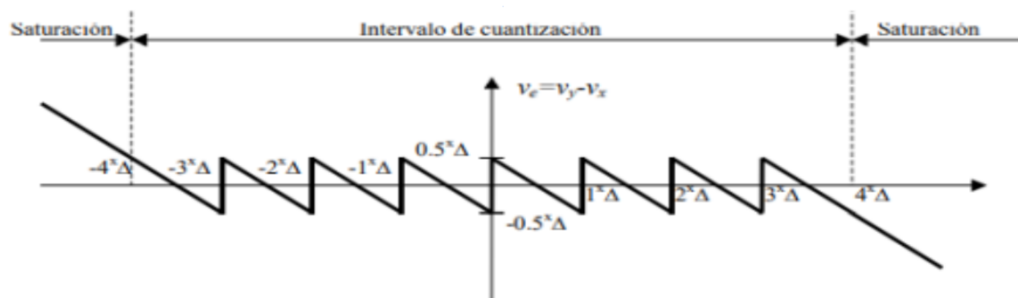

Figura 2.7 Representación de la evolución error de un cuantificador[1].

Dicho error depende del denominado paso de cuantificación, que se define como:

$$\Delta = \frac{Rango\ del\ cuantificador}{N^o\ de\ niveles\ del\ cuantificador - 1} \tag{2.5}$$

Por definición, el paso de cuantificación será la diferencia de valores entre dos niveles consecutivos del cuantificador.

2.8 Tipos de cuantificación

Dependiendo de si uno de los niveles del cuantificador es o no es '0', se pueden diferenciar dos tipos de cuantificadores, los midrise (en los que 0 no es un nivel del cuantificador) y los midthread (en los que 0 si es un nivel de cuantificador). En este trabajo, el cuantificador será de tipo midrise.

2.9 Convertidor digital analógico

Los convertidores digitales analógicos (DAC) reciben como entrada una secuencia de valores digitales y discretos y genera como salida una secuencia de valores continuos y analógicos.

2.10 Transformada Z

Es necesario explicar la transformada Z, ya que permite realizar análisis frecuenciales de señales discretas en el tiempo (como $x(n)$), siendo útil en este proyecto. La transformada Z de $x(n)$ se define como:

$$X(z) = Z\{x[n]\} = \sum_{n=0}^{\infty} x[n]z^{-n} \tag{2.6}$$

Dónde n es un entero y z es una variable compleja.

2.11 SNR Y HD

Estos valores permiten evaluar la calidad con la que se ha llevado a cabo el proceso de obtención de la señal original tras haber sido digitalizada.

2.11.1 SNR

La SNR (Signal to Noise Ratio o Relación Señal/Ruido) se trata de un parámetro que mide la relación entre la potencia de la señal deseada y la del ruido, tal y como se ve en la siguiente expresión:

$$SNR = \frac{P_{Signal}}{P_{Noise}} \tag{2.7}$$

Esta expresión suele medirse en decibelios, de forma la relación anterior finalmente queda:

$$SNR(dB) = 10\log_{10}\frac{P_{Signal}}{P_{Noise}} \tag{2.8}$$

De esta forma, obtener un valor de la SNR mayor a 0 dB implica que el nivel de potencia de la señal supera a la del ruido, y mientras mayor sea este valor, la calidad de la señal será mejor.

2.11.2 HD

El parámetro HD (Harmonic Distortion o Distorsión Armónica) mide la relación entre la cantidad de contenido asociado a los distintos armónicos de la señal respecto a la frecuencia fundamental, de forma que cada armónico (2º, 3º, 4º...) tiene asociado su propio HD. Su valor se calcula con la siguiente expresión:

$$HD_i(dB) = 20\log_{10}\frac{V_{1F}}{V_{iF}} \tag{2.9}$$

Donde V_{1F} es el valor de pico de la frecuencia fundamental y V_{iF} el del armónico i-ésimo. En este proyecto se calcularán el segundo y tercer armónico, es decir, HD2 y HD3.

2.12 Convertidor sobremuestreado

Recordando el concepto de frecuencia de Nyquist, se define un convertidor sobremuestreado como aquel cuya frecuencia de trabajo es mucho mayor a $f_{Nyquist} = 2 \cdot f_0$. El sobremuestreo presenta varias ventajas; facilita el diseño de los filtros anti-aliasing en la conversión Analógico/Digital (ADC) y de reconstrucción en la conversión Digital/Analógica, respectivamente; aumenta la resolución y disminuye el ruido.

Esta reducción del ruido se puede explicar con la expresión de la densidad espectral de potencia, que se puede ver a continuación:

$$| Q(f) | = \frac{\Delta^2}{12 \cdot f_s} \tag{2.10}$$

Se observa que cuanto mayor sea el valor de f_s menor será la densidad espectral de potencia del ruido de cuantificación. Es por ello que la potencia de ruido, que se extiende por todo el rango de frecuencias hasta $\pm\frac{f_s}{2}$, disminuye conforme mayor sea f_s, ya que mayor será las zonas en las que no habrá señal, y el ruido podrá eliminarse fácilmente con un filtro paso bajo.

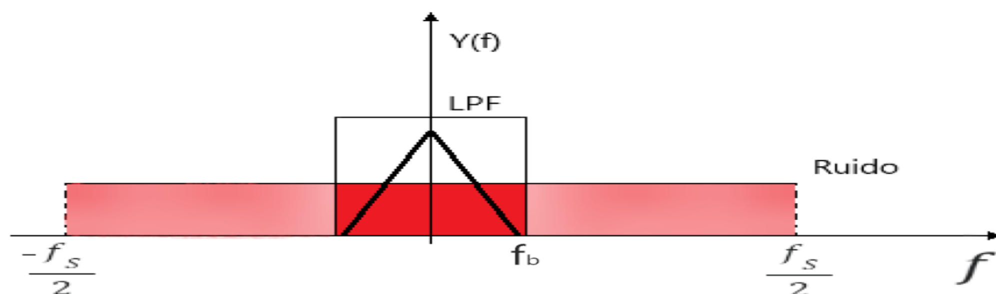

Figura 2.8 Representación del comportamiento del ruido con sobremuestreo.

2.13 Modulador Sigma/Delta

Los moduladores Sigma/Delta (SDM) son un tipo de convertidor sobremuestrado, que reduce el ruido en la banda de frecuencia en la que se encuentra la señal de interés. A

este efecto se le conoce como *Noise Shaping* o Conformación del Ruido. Además, al ser sobremuestreado, se puede relajar el filtro anti-aliasing.
En la figura 2.9 se puede ver una representación de un modulador Sigma/Delta de primer orden.

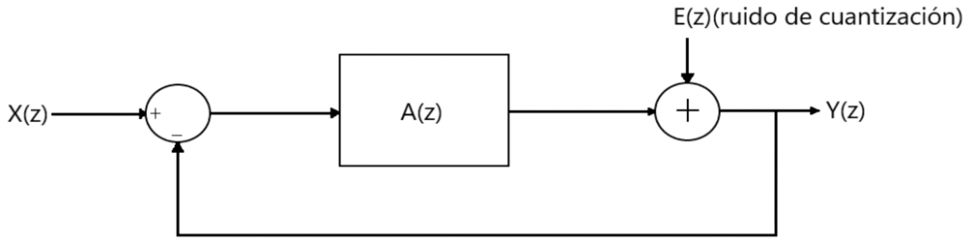

Figura 2.9 Representación de modulador Sigma/Delta de primer orden.

De dicha figura se puede obtener la siguiente expresión:

$$Y(z) = E(z) + A(z) \cdot X(z) - A(z) \cdot Y(z) \tag{2.11}$$

Reorganizándola, se obtiene la forma de las funciones de transferencia por las que pasa tanto la señal de entrada como el ruido de cuantificación:

$$Y(z) = \frac{1}{1+A(z)} \cdot E(z) + \frac{A(z)}{1+A(z)} \cdot X(z) \tag{2.12}$$

Donde:

- **NTF(z) =**$\frac{1}{1+A(z)}$ es la función de transferencia del ruido de cuantificación
- **STF(z) =**$\frac{A(z)}{1+A(z)}$ es la función de transferencia de la señal

Por lo tanto, se debe elegir $A(z)$ de forma que $STF(z) \to 1$ y $NTF(z) \to 0$ en las frecuencias donde se encuentre la señal. Esto se logra si, en dichas frecuencias, $A(z) \to \infty$, lo que teóricamente se consigue en el caso de que $A(z)$ sea un integrador, ya que es un modulador paso-bajo (y la señal se encontrará a bajas frecuencias).

En un integrador en tiempo discreto, la salida debe ser la suma de la entrada anterior con la salida anterior, tal y como se ve en la figura 2.10. Es decir, $V(n) = V(n-1) + U(n-1)$, que en el dominio de z será:

$$V(z) = z^{-1} \cdot V(z) + z^{-1} \cdot V(z) \to V(z) = \frac{z^{-1}}{1-z^{-1}} U(z) \tag{2.13}$$

De esta forma, la función de transferencia del integrador será:

$$A(z) = \frac{V(z)}{U(z)} = \frac{z^{-1}}{1-z^{-1}} = \frac{1}{z-1} \tag{2.14}$$

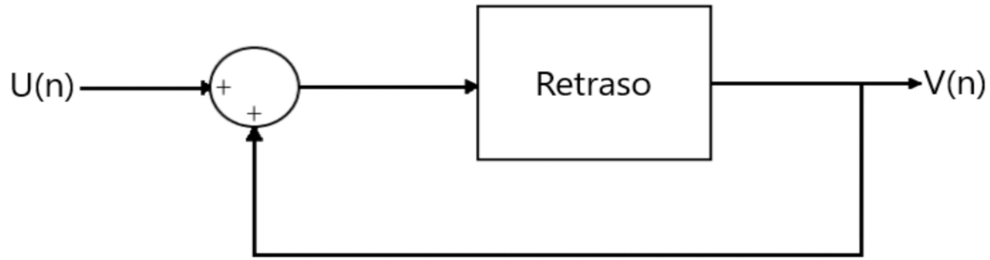

Figura 2.10 Integrador en tiempo discreto.

Una vez obtenida la función *A(z)*, se terminan de definir así las funciones *NTF(z)* y *STF(z)*:

$$NTF(z) = \frac{1}{1+\frac{1}{z-1}} = \frac{z-1}{z} = 1 - z^{-1} \tag{2.15}$$

$$STF(z) = \frac{\frac{1}{z-1}}{1+\frac{1}{z-1}} = \frac{1}{z} = z^{-1} \tag{2.16}$$

Por lo tanto, se tiene finalmente que

$$Y(z) = (1 - z^{-1}) \cdot E(z) + z^{-1} \cdot X(z) \tag{2.17}$$

Recordando ahora que $z = e^{jwT_s} = e^{j2\pi fT_s}$, se obtiene así la expresión que modifica al ruido de cuantización en función de la frecuencia:

$$NTF(jf) = 1 - e^{-j2\pi fT_s} = 2 \cdot e^{-j\pi fT_s}(\frac{e^{j\pi fT_s} - e^{-j\pi fT_s}}{2}) = 2 \cdot e^{-j\pi fT_s} \cdot \sin(\pi fT_s) \tag{2.18}$$

En la figura 2.11 se puede ver un esquema de la cantidad de ruido de cuantificación que le queda a la señal al filtrarse con un filtro de paso bajo (LPF). Se observa que el ruido, al haberse modulado con la función *NTF(f)*, es pequeño a frecuencias bajas donde se encuentra la señal, y que crece al aumentar la frecuencia. Si se compara esta imagen con la Figura 2.8,

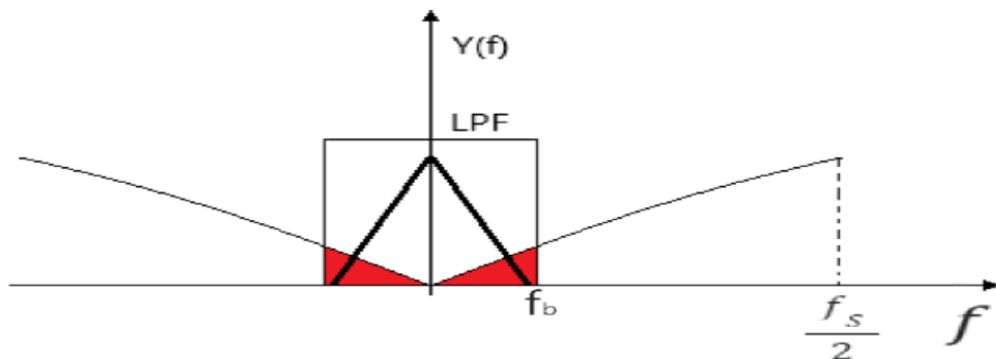

Figura 2.11 Espectro de la señal y potencia del ruido de cuantificación.

se puede observar la ventaja del *Noise Shaping*, ya que la cantidad de ruido que presentará ahora la señal es considerablemente menor al caso del ruido uniformemente distribuido por todo el rango de frecuencias hasta $\frac{f_s}{2}$.

Se debe tener en cuenta que el modulador que se ha obtenido es de primer orden. El orden vendrá dado por el número de integradores que se tengan, de forma que a mayor orden mayor resolución del SDM, ya que menor cantidad ruido quedará alojado en las bajas frecuencias y más ruido se desplazará a las altas. Se muestra a continuación una representación de cómo varía el error de cuantificación dependiendo de si el modulador es de primer o segundo orden. Éste último, el de segundo orden, es el que se implementará en este trabajo:

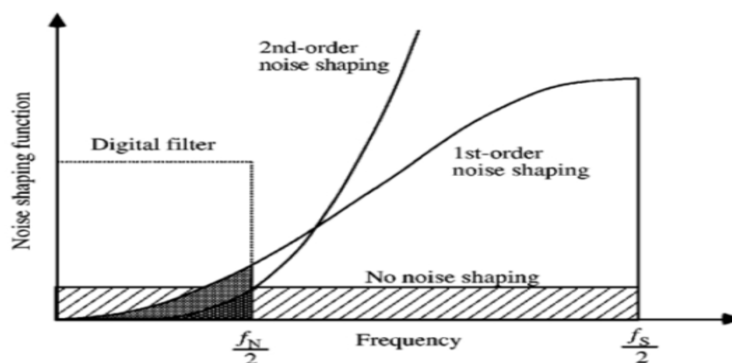

Figura 2.12 Error de cuantificación sin modulador de ruido y con SDM de orden 1 y 2[7].

Como se observa, aumentar el orden del modulador a 2 reduce considerablemente el ruido de cuantificación de la señal, lo que justifica su uso en este proyecto. La expresión de la función *NTF(F)* para el caso del SDM de segundo orden será:

$$NTF(z) = (1 - z^{-1})^2 \tag{2.19}$$

Por otro lado, se muestra la figura 2.13 como es el espectro que caracteriza a los moduladores Sigma/Delta. Como se ha visto hasta ahora, el espectro presenta ruido pequeño cerca de la señal, pero este crece considerablemente al alejarse de ella, donde no supone problema su valor gracias al uso de un filtro paso bajo.

2.14 Modulador de ancho de pulsos

La modulación de ancho de pulsos es un tipo de codificación donde los valores, analógicos o digitales, pueden codificarse como una forma de onda de dos niveles. El ciclo de trabajo (es decir, el cociente entre el tiempo que la señal se encuentra en estado activo y el período total de la señal) de dicha forma de onda es proporcional al valor que se está codificando en cada momento.

Para llevar a cabo la modulación PWM, se necesita una onda de referencia, cuyo período será T_{ond}, con la que se comparará a la señal de entrada. De esta forma, la salida del

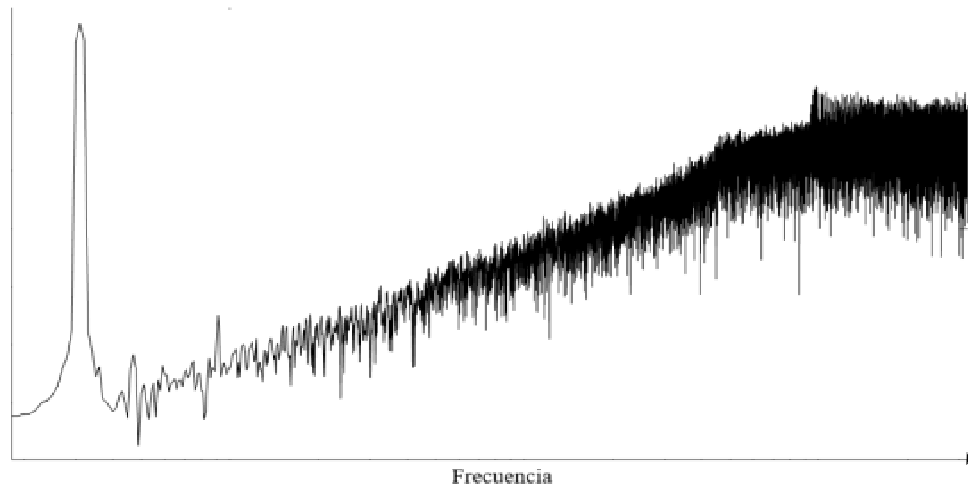

Figura 2.13 Ejemplo de espectro de un SDM [8].

modulador PWM tendrá un valor alto si la señal es mayor al valor de la referencia, y bajo en el caso contrario.

Las señales que se suelen tomar como referencia suelen ser cuadradas o triangulares. Se muestra a continuación una imagen en la que se puede ver representado el comportamiento de este modulador, para el caso en el que solo haya 4 niveles posibles de *y(n)* y en el que la señal de referencia es triangular.

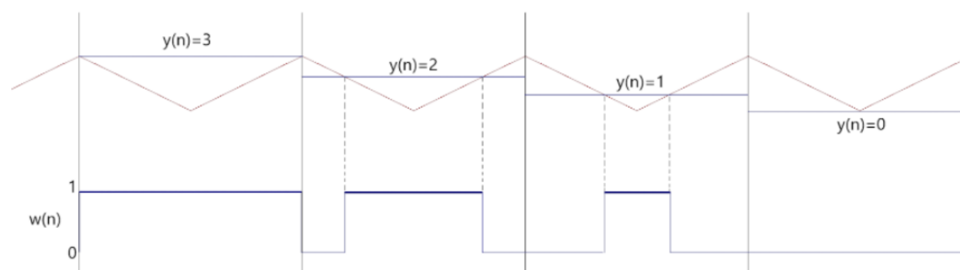

Figura 2.14 Comparación con referencia triangular ideal.

Como se observa, la señal de entrada *y(n)* se ha comparado con el valor de la referencia triangular, lo que ha dado lugar a una señal de un solo bit cuyo ciclo de trabajo es proporcional al valor de la entrada, siendo el ciclo de trabajo 0 para la entrada más pequeña y 1 para el valor de entrada más grande.

Sin embargo, en este caso la señal triangular era ideal. En el caso de realizar la PWM digitalmente, tal y como se ve en la figura 2.15, la señal triangular de referencia pasa a estar compuesta por *L-1* niveles (siendo *L* el número de posibles valores de *y(n)*) ya que estos son los necesarios para poder asignar una salida diferente a cada valor posible de *y(n)*.

Por lo tanto, la salida de *w(n)* deberá actualizarse cada vez que se compare con la señal de referencia, y se compara con la referencia 2 veces por cada nivel (la primera vez descendiendo la referencia y la segunda ascendiendo), tal y como se ve en la figura 2.16

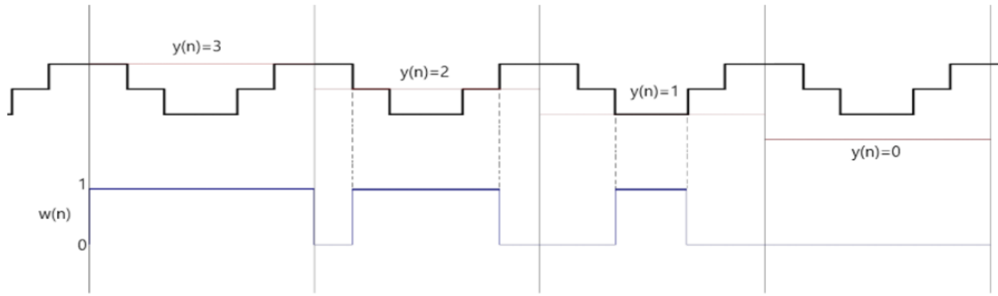

Figura 2.15 Comparación con referencia triangular digital.

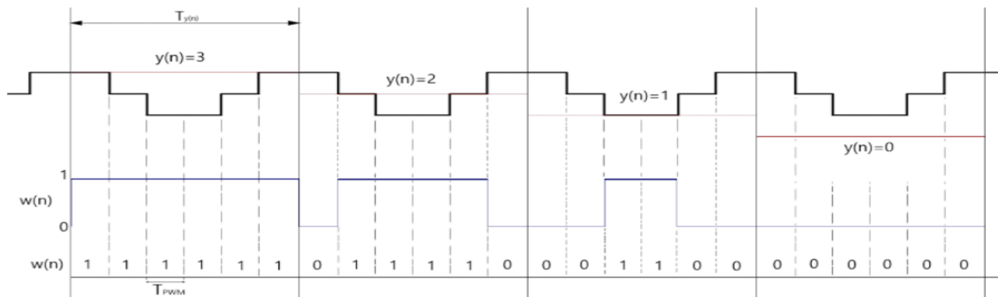

Figura 2.16 Ejemplo de obtención de la salida de un PWM.

De esta forma, se establece que la relación entre la frecuencia a la que se actualiza *y(n)* y la frecuencia a la que se actualiza *w(n)* deberá ser:

$$\frac{T_{y(n)}}{T_{PWM}} = \frac{f_{PWM}}{f_{y(n)}} = 2 \cdot (L-1) \tag{2.20}$$

Es decir, que el modulador PWM trabajará *2(L-1)* veces más rápido que el SDM del que provendrá su entrada en este trabajo, pero solo tendrá 1 bit de salida.

2.14.1 Corrección del error asociado a los PWM

Realizar la modulación PW sobre señales discretas o digitales da lugar a una disminución de la calidad de la forma de onda de la señal modulada debida a la distorsión armónica que genera. En el documento *Digital correction of errors and harmonic distortion in pulse-width modulation of digital signals* [5] se analizan los mecanismos que producen esta distorsión, lo que permite cuantificar el error y, por lo tanto, corregirlo o reducirlo. En dicho documento se propone la siguiente estructura de corrección que se ve en la figura 2.17.

Donde, para el caso de que la señal de referencia del *PWM* sea triangular y con el mismo periodo que el SDM, las expresiones de $y_2(n)$ e $y_3(n)$ son:

$$y_2(n) = 0 \tag{2.21}$$

$$y_3(n) = \frac{(1+y(n))^3}{4} \tag{2.22}$$

Por lo tanto, a partir de ahora se va a denominar G_3 al conjunto de operaciones necesarias

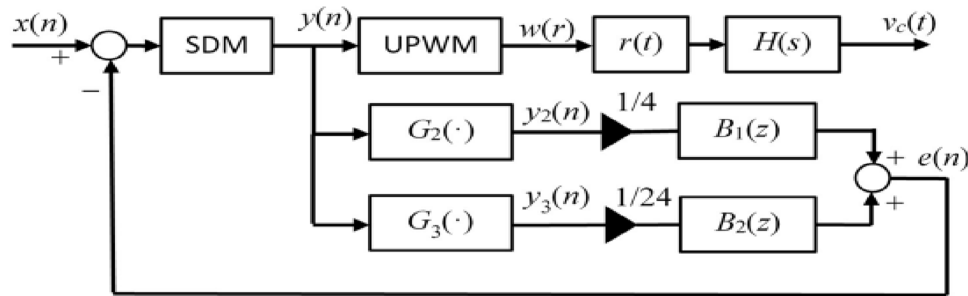

Figura 2.17 Estructura de corrección para señales de entrada muestreadas [5].

para transformar $y(n)$ en $y_3(n)$. Por simplicidad, se introducirá también en esta estructura la ganancia $\frac{1}{24}$ que se puede ver en la figura 2.17.

Por otro lado, la función de transferencia $B_2(z)$ de la figura anterior es:

$$B_2(z) = z^{-1}(1 - z^{-1})^2 \tag{2.23}$$

Esta corrección se basa en la validez de aproximar derivadas continuas en el tiempo empleando funciones de z ($B_2(z)$). Esto será posible siempre que la frecuencia de la señal de entrada sea lo suficientemente pequeña comparada con la frecuencia del SDM, lo que ocurre siempre que el *OSR* sea lo suficientemente grande. Por lo tanto, para este trabajo se ha tomado un valor de *OSR*=32.

2.15 FPGA

Una FPGA (Field Programmable Gate Arrays en inglés o Matriz de Puertas Programables en Campo en español) se trata de un conjunto de dispositivos semiconductores pequeños que se pueden programar, realizando así diferentes tareas y acciones específicas rápida y eficientemente. Es por ello que son usadas en la industria automovilística, sistemas de emulación de hardware, aplicaciones militares, de reconocimiento de voz. . .

El hecho de que sean reprogramables tras haber sido fabricados le aporta a las FPGA una flexibilidad que las hace la herramienta perfecta con la que trabajar en este proyecto. Para fijar el comportamiento de las FPGA, se debe realizar una descripción de un circuito en un lenguaje de descripción de hardware o con un diseño esquemático, aunque la primera de las dos opciones presenta ventajas frente a la segunda al ser más adecuada para grandes estructuras al permitir especificar comportamientos funcionales de alto nivel.

3 Simulación en MATLAB y SIMULINK

El propósito de este capítulo es mostrar el modelo de convertidor que conformó el punto de partida de este trabajo. Se hará un análisis del comportamiento de este modelo empleando las herramientas *Simulink* y *Matlab*, ya que estos programas permiten diseñar convertidores y modificarlos de forma mucho más rápida y sencilla que en la herramienta ISO de Xilinx.

Por ello, se va a comenzar mostrando un convertidor basado en un SDM al que se le va a añadir el PWM y los elementos correctores pertinentes, y se va a realizar el cálculo de la *SNR*, *HD2* y *HD3* de cada uno de los convertidores que se obtengan, comprobando de esta forma que el que se ha decidido implementar en VHDL presenta buenos resultados comparado con los demás.

3.1 Modelos Simulink del Modulador Sigma/Delta y del Modulador de Ancho de Pulsos

En este capítulo se va a analizar el comportamiento de 4 moduladores. El primero de ellos será el SDM de segundo orden básico. El segundo de ellos es el SDM al que se le añade el PWM pero al que no se le hacen las correcciones vistas en la figura 2.17 del capítulo anterior. El tercero de ellos es el SDM y el PWM al que sí que se le añaden estas correcciones. El cuarto es similar al tercero, pero aplicando los cambios necesarios para que toda su aritmética sea entera (es decir, que los valores de sus señales serán siempre números enteros).

Comparar los resultados del cuarto modelo con los que se obtienen de los modelos 1, 2 y 3 permite comprobar que, aunque las señales del convertidor sean enteras (tal y como ocurrirá una vez se describa el convertidor en VHDL), los valores de la *SNR*, *HD2* y *HD3* siguen siendo aceptables o incluso mejores que en los casos en los que la aritmética no es entera.

Todos estos convertidores recibirán como entrada una onda sinusoidal de una amplitud

A=$10^{\frac{-3}{20}} \cdot SatuQ$ (siendo $satuQ$=8280). Como se observa, esta amplitud está compuesta por el producto de dos valores. El primero de ellos, $10^{\frac{-3}{20}}$, es la que podría considerarse “Amplitud real” de la onda: es la que tendrá la señal de salida una vez filtrada. El segundo valor, $satuQ$=8280, se puede ver como un amplificador necesario para que la señal de entrada sinusoidal del convertidor puedan ser entera, ya que esto no sería posible en el caso de que su amplitud fuera $10^{\frac{-3}{20}}$, puesto que en ese caso la entrada del modelo 4 solo podría tomar tres valores: *0*,*-1* y *1*.

Se muestra a continuación una imagen de los 4 convertidores que se van a ver en este capítulo. Todos ellos presentan un filtro de Butterworth, que permite eliminar las frecuencias altas de la salida y recuperar así la señal sinusoidal.

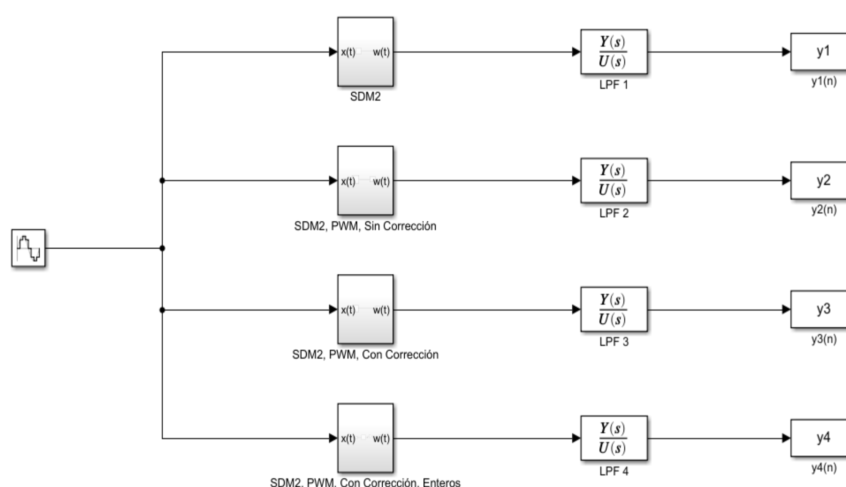

Figura 3.1 Convertidores en Simulink.

Se procede a continuación a explicar los componentes de cada uno de los moduladores por separado:

3.1.1 Modelo 1: Modulador Sigma/Delta

Este modelo se corresponde con el SDM de segundo orden obtenido en el capítulo anterior (2.19). Se recuerda que la expresión de la función de transferencia que modificaba a la diferencia entre la señal de entrada y de salida (el ruido) es $NTF(z) = (1 - z^{-1})^2 = 1 + z^{-2} - 2 \cdot z^{-1}$. De esta forma, la implementación de esta función de transferencia en $Simulink$ se hará cómo se ha indicado en la figura 3.2.

Respecto al *Cuantificador Q*, es conveniente mencionar que el valor del paso del que depende dicho cuantificador, *pasoQ*, será el mismo tanto para este como para los siguientes modelos, ya que en ninguno de ellos cambia el valor de la amplitud de entrada. El valor de *pasoQ* viene dado por la expresión 2.5, donde el rango del cuantificador será *Rango*=$2 \cdot satuQ$ (por lo tanto su valor mínimo es *-8280* y el máximo *8280*) y su número de niveles 16. La salida de este modelo, y_1, se dividirá entre el valor *satuQ* antes de ser filtrada, para que de esta forma la amplitud a la salida esté acotada al tango *[-1,1]*.

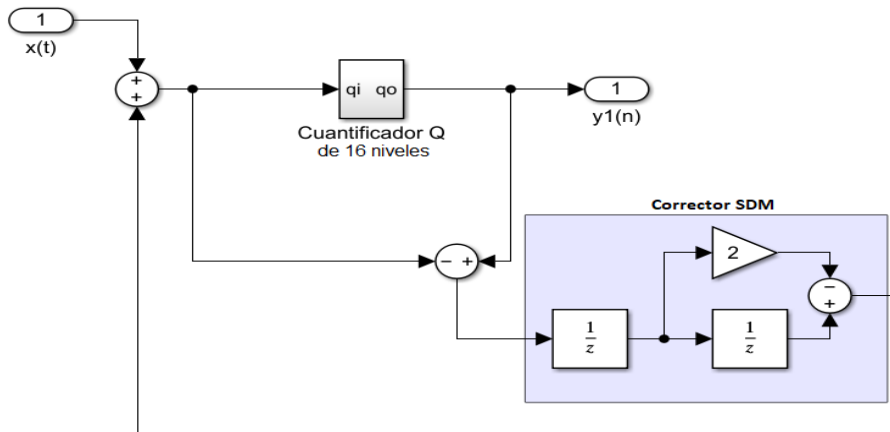

Figura 3.2 Modelo en Simulink del SDM de segundo orden.

3.1.2 Modelo 2: Modulador Sigma/Delta y de ancho de pulsos sin corrección

Esta estructura está compuesta por el SDM y el PWM. El bloque Modulador Sigma/Delta (SDM en la siguiente imagen) es igual al que se ha tenía en el diseño de la figura 3.2. Sin embargo, se debe recordar que al incluir un PWM al convertidor, es conveniente llevar a cabo una corrección digital. Este modelo no cuenta con dicha corrección, por lo que servirá para observar cómo es el comportamiento de la salida en el caso de que se omita el corrector. Como se observa, al igual que en el modelo anterior a la salida del SDM se ha

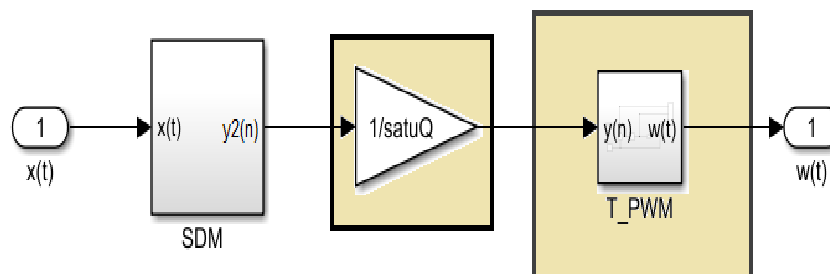

Figura 3.3 Modelo 2. Marcado: Ganancia y bloque PWM.

introducido una ganancia de $\frac{1}{satuQ}$ con el propósito de reducir la amplitud máxima de la señal de salida del SDM a [-1,1], necesario para que el valor de la salida pueda compararse con la señal triangular de referencia del bloque PWM. Los componentes de dicho bloque se muestran en la figura 3.4.

Se puede observar que se obtiene la entre la señal de entrada, *y(n)*, con la señal de referencia triangular. Dicha diferencia entra en un cuantificador, cuya salida será -1 para valores negativos o 1 para positivos. De esta forma, se obtiene la señal *w(t)*, que es la salida del convertidor.

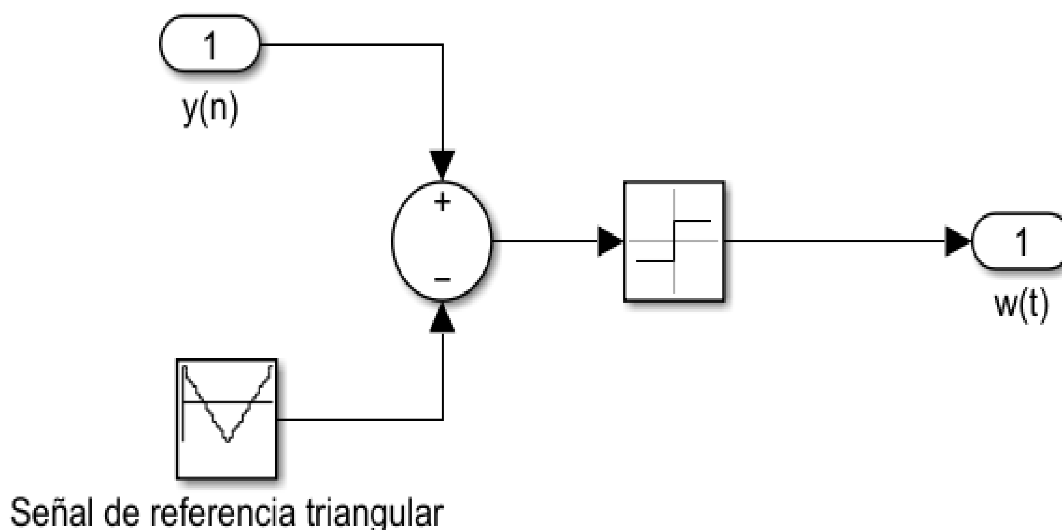

Figura 3.4 PWM en Simulink.

3.1.3 Modelo 3: Modulador Sigma/Delta y de ancho de pulsos con corrección

Al contrario que en el caso anterior, en este modelo sí se ha añadido un corrector del ruido que disminuye el error de la señal de salida, permitiendo aumentar la calidad del convertidor. El diseño de dicho corrector, que se ve remarcado en la figura 3.5, se explicó en el capítulo anterior y se ha realizado conforme a la figura 2.17.

Tal y como se vio en dicho capítulo, el corrector se podía separar en dos bloques. El primero, G_3, es el asociado a las operaciones que se deben realizar sobre la entrada del corrector (la denominada $y_3(n)$ en la figura 2.17) y cuya salida se ha denominado *a(n)* en la siguiente figura. Sin embargo, como se ve en la figura 3.5, ha sido necesario añadir dos ganancias a las operaciones que conforman este bloque.

La primera de ellas, $\frac{1}{satuQ}$, es necesaria para restringir la amplitud máxima que puede presentar $y_3(n)$ antes de aplicarle las operaciones pertinentes para realizar la corrección. La segunda de ellas, $\frac{satuQ}{96}$, consta del 96, que viene dado por la expresión del corrector, y del valor *satuQ*, utilizado para que la señal *a(n)* tenga de nuevo la amplitud que le corresponde. Por otro lado, se nombró B_2 a la estructura que recoge los retrasos que se le deben aplicar a *a(n)*.

Como se aprecia en el bloque B_2, los retrasos que lo conforman dan lugar a la expresión 4.9:

$$B_2(z) = z^{-1} - 2z^{-2} + z^{-3} = z^{-1}(1 - 2z^{-1} + z^{-2}) = z^{-1}(1 - z^{-1})^2 \tag{3.1}$$

3.1.4 Modelo 4: Modulador Sigma/Delta y de ancho de pulsos con corrección y señales enteras

La variación que presenta este modelo respecto al anterior es que se han introducido dos cuantificadores que no aparecen en las estructuras anteriores. Estos cuantificadores son de

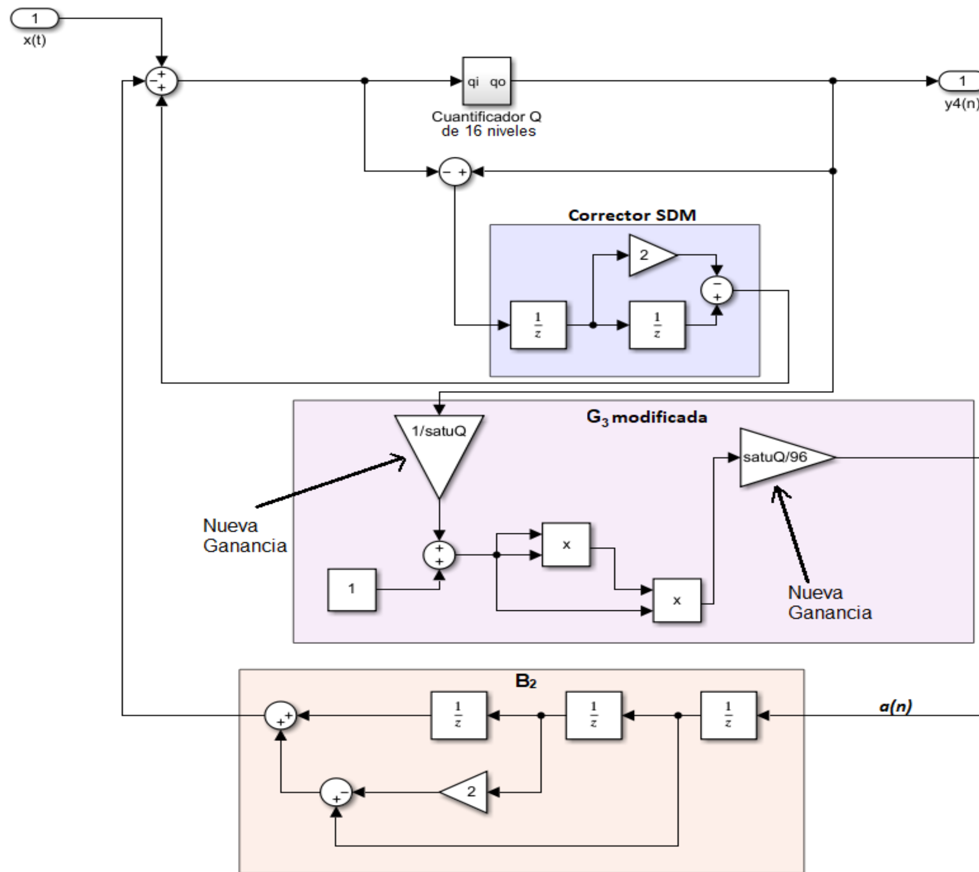

Figura 3.5 Modelo 3.

paso 1, lo que permite que las señales dentro del SDM sean enteras (ya que, como se sabe, este modelo es el que se implementará en VHDL). Uno de ellos se ha colocado a la entrada del bloque SDM, tal y como se ve en la figura 3.6.

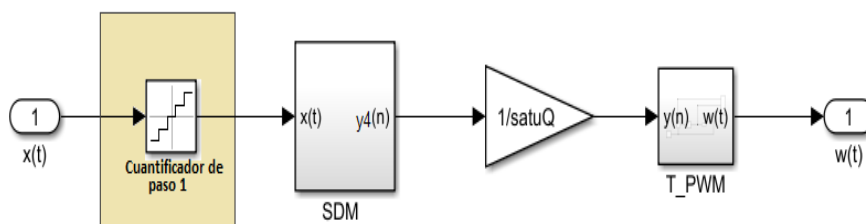

Figura 3.6 Modelo 4. Introducido: Cuantificador de paso 1.

Por otro lado, el segundo cuantificador que se ha añadido (que también tiene paso unidad) se encuentra a la salida de la denominada G_3 del corrector de ruido (tal y como se puede ver en la figura 3.7). A partir de este punto, cuando sea necesario, se denominará **Cuantificador LUT** a la denominada G_3 del corrector que se ha representado en la figura 3.7, y **Retraso tras LUT** a la estructura que se denominó B_2. Por otro lado, se denominará **Retraso tras yn** al bloque que hasta ahora era *Corrector SDM*.

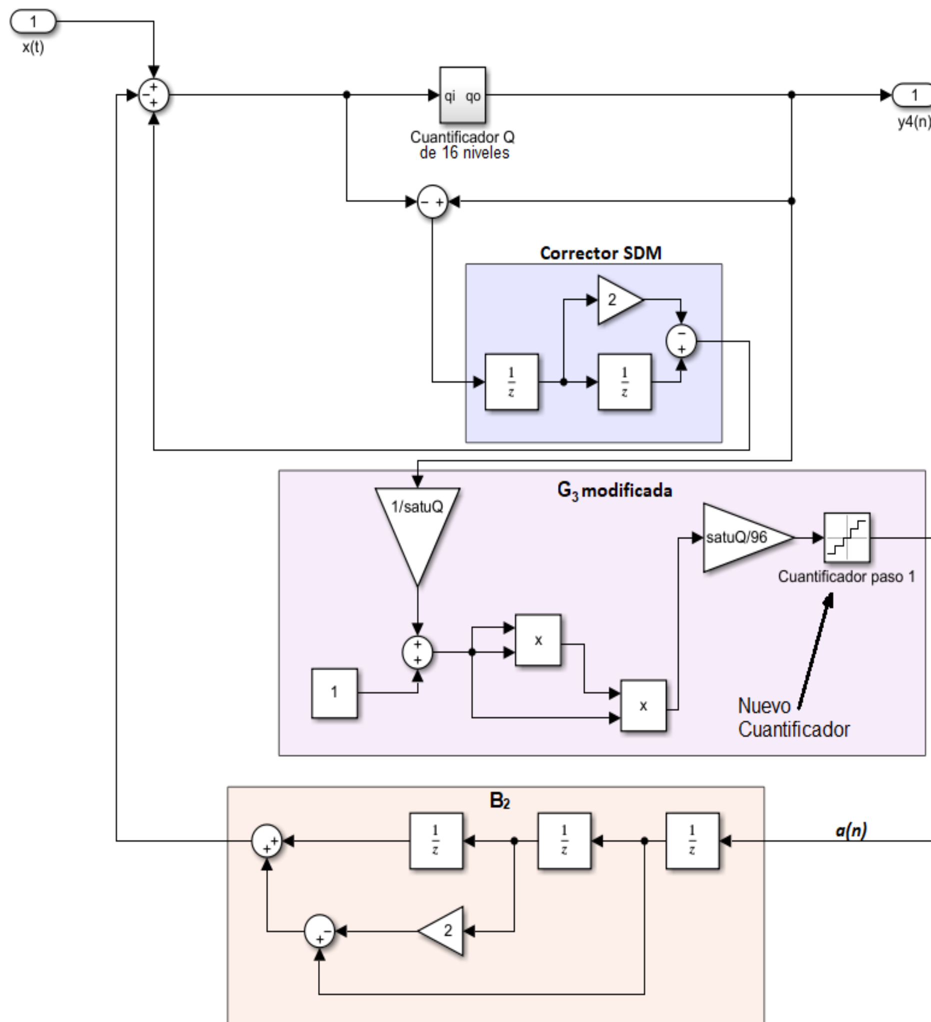

Figura 3.7 SDM del modelo 4. Introducido: Cuantificador de paso 1 del G_3.

3.2 Comparación

En las siguientes imágenes se puede ver la representación espectral que se obtiene al comprobar el comportamiento de los modelos de convertidor que se han mostrado en este capítulo. Para llevar a cabo las simulaciones de las que se han obtenido los resultados que se van a presentar a continuación, se ha fijado como frecuencia de la señal de entrada $f_b = 361.7\ Hz$ (aunque esta frecuencia no permanecerá fija en los análisis que se hagan en los siguientes capítulos).

Tabla 3.1 Comparación de los valores de la SNR de cada modelo.

Valor	Modelo 1	Modelo 2	Modelo 3	Modelo 4
SNR(dB)	84.727569	84.496260	85.085676	86.187610

Se observa que se logra mejorar la calidad de la salida al acoplarse al SDM no solo el PWM, sino que también el corrector del ruido que se ha mencionado en los apartados anteriores. Por otro lado, se comprueba que:

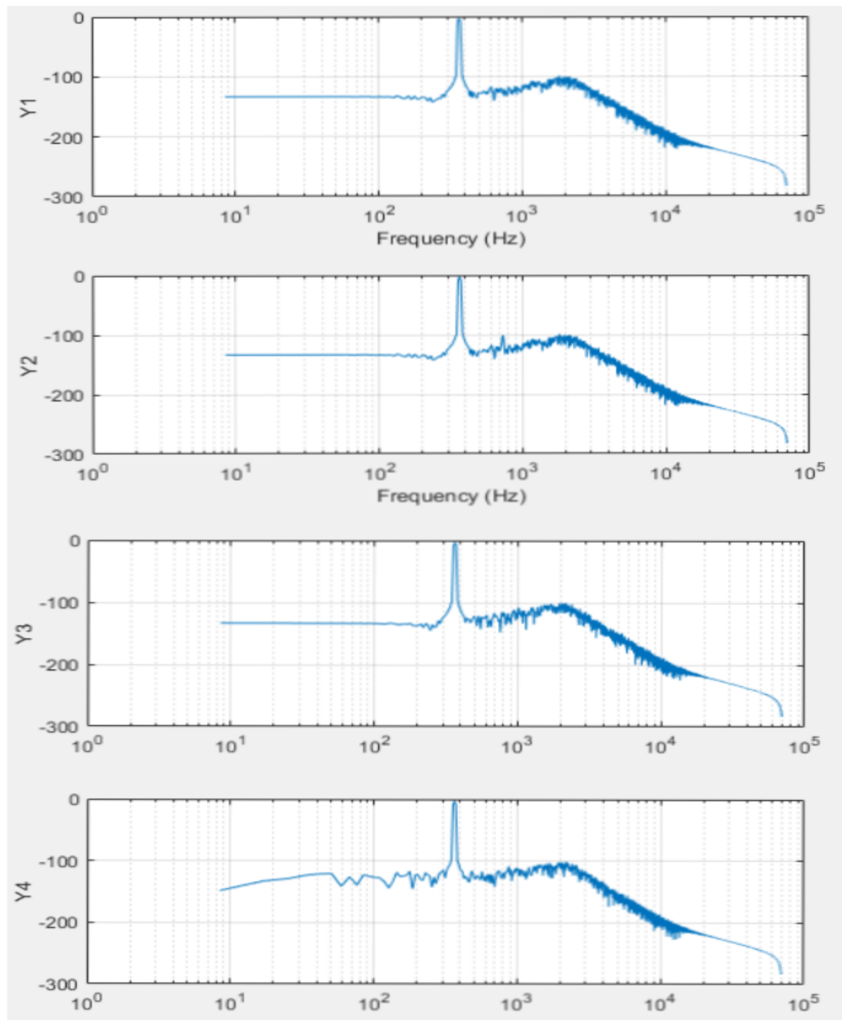

Figura 3.8 Representación espectral de las salidas.

Tabla 3.2 Comparación de los valores de la *HD2* de cada modelo.

Valor	Modelo 1	Modelo 2	Modelo 3	Modelo 4
HD2(dB)	111.476795	97.698332	111.154133	108.216532

Tabla 3.3 Comparación de los valores de la *HD3* de cada modelo.

Valor	Modelo 1	Modelo 2	Modelo 3	Modelo 4
HD3(dB)	104.055757	102.796954	104.666577	106.084888

- Los valores de la SNR de los 4 modelos son equivalentes.
- Emplear el corrector de los modelos 3 y 4 permite que disminuya el segundo armónico del modelo 2, lo cuál es apreciable en la segunda gráfica de la figura 3.8).
- Se consigue corregir el tercer armónico en los modelos 3 y 4, lo que mejora el valor del *HD3*

Esto justifica que se haya decidido incluir el PWM en el diseño del convertidor. Por lo tanto, el modelo 4 será es el que se describirá en VHDL (en el que ser recuerda que la aritmética es entera) y una vez que se haya implementado el convertidor en VHDL, se podrá comprobar que los códigos se han realizado correctamente comparando la *SNR*, *HD2*

y *HD3* las que se han obtenido al hacer simulaciones con la que se ha obtenido en este apartado.

Finalmente, se muestra en la siguiente imagen tanto la salida filtrada del convertidor seleccionado junto con la entrada al mismo (en el que la amplitud se ha dividido entre el factor *SatuQ*, por lo que solo será A=$10^{\frac{-3}{20}}$). Esta gráfica se ha obtenido mediante el uso de la herramienta *Scope* en Simulink:

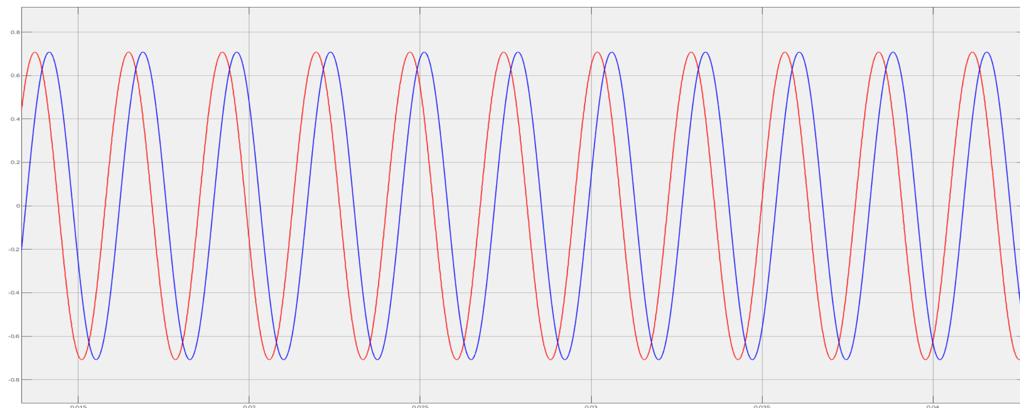

Figura 3.9 Señal de entrada (roja) y salida(azul) del convertidor 4.

Se observa que ambas señales son prácticamente idénticas, con la diferencia de que la de salida está retrasada respecto a la de entrada debido a los retrasos del SDM.

El siguiente paso tras tener definida la estructura que se iba a llevar a cabo en VHDL fue medir todas las señales que la conformaban. Esto es debido a que, para programar en el lenguaje VHDL, se debe conocer a priori el número máximo de bits necesarios para cada una de las señales que se van a crear. Para poder hacer esto, se deberá conocer primero el valor máximo al que se puede llegar en cada parte del convertidor en cualquier momento.

Para ello, se han simulado y representado los valores que se obtienen en Simulink de cada señal del modulador mediante bloques Scope, y se ha obtenido el máximo que alcanza cada una de ellas. Con estos valores se ha calculado el número de bits máximo (que es el número de bits necesario para escribir dicho valor máximo en binario), mediante la siguiente expresión:

$$n_{bitmax} = \lceil \log_2(valor\ max) \rceil + 1 \tag{3.2}$$

Donde $\lceil x \rceil$ es el mínimo número entero no inferior a x. A $\lceil \log_2(valor\ max) \rceil$ se le suma una unidad debido a que los números en binario se representarán en complemento a dos, para lo cuál se necesita un bit adicional que indique el signo. En la figura 3.10 se puede ver el número de bits máximo que se ha obtenido para cada una de las partes que conforman el modulador Sigma/Delta.

No se ha especificado el número de bits necesarios a la salida de los bloques z^{-1} porque esta operación, como se verá más adelante, no modifica el número de bits de la entrada.

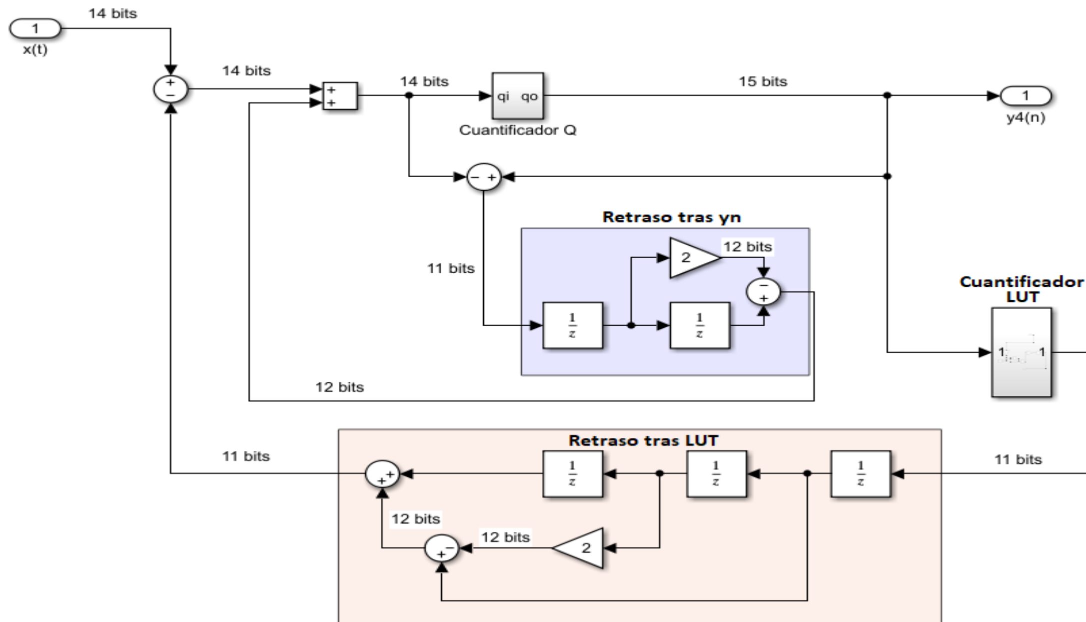

Figura 3.10 Bits máximos en cada parte del SDM.

Por otro lado, se recuerda que la salida del SDM no es la entrada del PWM, sino que antes del PWM hay una ganancia de $\frac{1}{SatuQ}$, lo que restringirá el rango de valores que podrá tomar como entrada del PWM a [-1,1].

Esos valores, como se sabe, se comparan con la señal de referencia, y dependiendo de cual de los dos es mayor, la salida será *-1* ó *1*. Como se verá en el siguiente capítulo, este comportamiento va a simplificarse ligeramente, de forma que sea más sencilla su descripción en VHDL (aunque seguirá generando la misma salida que si no se hiciera esta simplificación).

Por otro lado, se adelanta que en VHDL se generará la salida del PWM de forma distinta a como se genera en *Simulink* ya que, por simplicidad, no se guardará el valor '-1', sino '0'. Una vez que se analicen los resultados, si se hará la "conversión" de los valores, los '0'pasarán a ser '1'.

4 Simulación en VHDL

En este capítulo, partiendo de la estructura en Simulink que se vio en el apartado anterior (3.6), se van a explicar los diferentes códigos que se han desarrollado en VHDL para que describen el mismo comportamiento que hacía el modulador en Simulink. Dichos códigos se han dividido en función de los componentes del circuito en Simulink a los que están asociados.

Una vez explicado el comportamiento del modulador y los códigos en VHDL que permiten implementarlo, se van a mostrar las simulaciones que permite realizar el programa ISE de Xilinx y a analizar los resultados de dichas simulaciones.

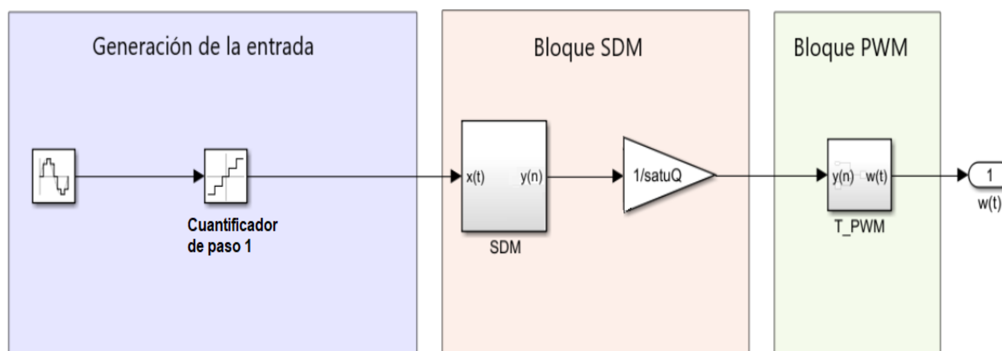

Figura 4.1 Partes del convertidor en Simulink según su función.

Se recuerda que, ya que se va a programar en VHDL, será necesario conocer a priori el valor de ciertos parámetros, puesto que así lo exige este lenguaje de programación a la hora de definir las variables que necesita. Es por ello que era de vital importancia el análisis de las señales que se hizo en el apartado anterior, y que permitió definir el número de bits necesario para definir cada una de ellas.

Dicho esto, se va comenzar separando el convertidor definido en *Simulink* en 4 partes bien diferenciadas, cumpliendo cada una de ellas una función distinta, de forma similar a como se hizo en la imagen anterior en *Simulink*. Tras esto, se van a explicar los códigos

que conforman cada una de estas partes, prestando especial atención a la relación que hay entre las señales de entrada y salida de cada uno de los bloques, puesto que es necesario comprenderlas bien para entender correctamente cómo interactúan los bloques entre ellos.

Dicho esto, los 4 bloques funcionales en los que se ha dividido el circuito son:

- Bloque 1: Generador de pulsos y datos
- Bloque 2: Modulador Sigma/Delta (SDM)
- Bloque 3: Modulador por ancho de pulsos (PWM)
- Bloque 4: Generador de las salidas

Como se observa, han resultado 4 bloques de la división, en lugar de los 3 que se obtuvieron en Simulink (4.1). Esto es debido a que, tal y como se explicará en detalle más adelante, se ha decidido transformar una de las salidas en una señal con retorno a cero.

En la siguiente imagen se puede ver una representación gráfica de los 4 bloques y como están relacionados entre ellos mediante sus señales de entrada y salida:

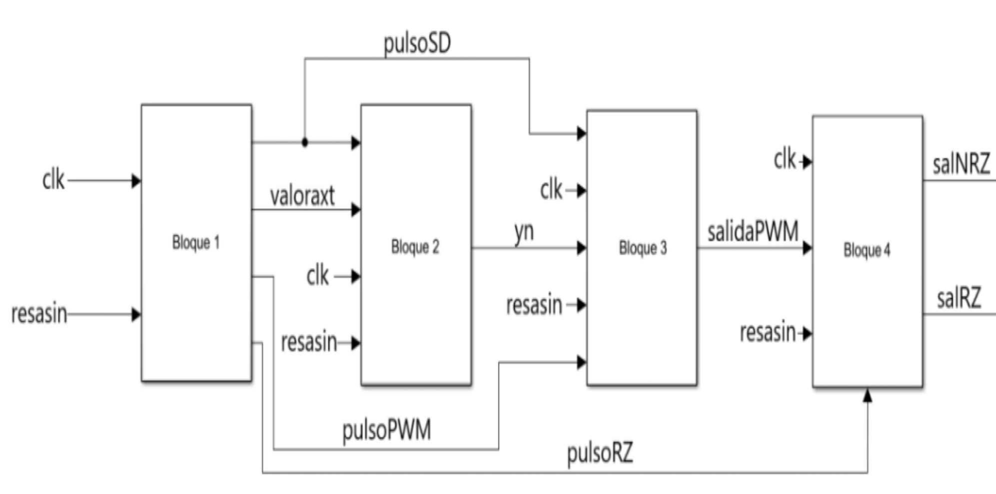

Figura 4.2 Esquema del convertidor en VHDL.

Por lo tanto, se va a comenzar explicando el bloque 1, cuyas salidas son los pulsos de los que dependen el resto de bloques (*PulsoSD* y *PulsoPWM*), la señal *PulsoRZ* y la entrada del bloque SDM, *valoraxt*.

4.1 Bloque 1

Como ya se ha dicho, este bloque genera la entrada del modulador Sigma/Delta. La entrada del modulador deberá ser sinusoidal, de amplitud $A=10^{\frac{-3}{20}} \cdot SatuQ$, frecuencia f_b y muestreado a una frecuencia f_s.

Se puede ver a continuación una representación esquemática de los programas que generan este bloque, y de las señales que entran y salen de cada uno de ellos:

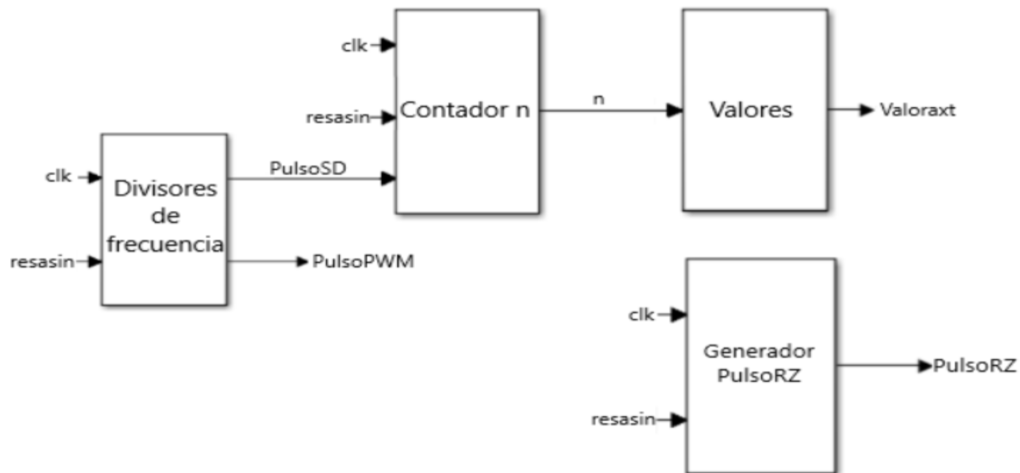

Figura 4.3 Componentes del Bloque 1 del convertidor.

Como se ve, todas las variables que se tienen están sincronizadas con el reloj maestro, clk, cuya frecuencia es f_{clk}=50 MHz (ó lo que es lo mismo, su período es es T_{clk}=$\frac{1}{f_{clk}}$=20 ns). Además de éste, el circuito también presenta una señal de reseteo asíncrona, *resasin*, que podrá tomar 2 valores, 0 ó 1. De esta forma, cuando se verifique que *resasin*= “1”, todas las señales con lógica secuencial se anularán, permitiendo reiniciar el circuito siempre que se considere necesario.

Una vez aclarado esto, se van a comenzar a explicar los programas que se vieron en la imagen anterior:

4.1.1 Generación de PulsoSD y PulsoPWM

Las señales *PulsoSD* y *PulsoPWM* permiten fijar la frecuencia de trabajo de los bloques 2 y 3. Sin embargo, estas señales deben obtenerse a partir de la señal *clk*. Los programas necesarios para generar las señales *pulso* son los que se explican en este apartado. Tal y como se verá a continuación, se trata de un único programa que presenta dos versiones, generando cada una de ellas uno de los dos pulsos.

El programa encargado de generar los pulsos, que ha sido denominado *Divisor de frecuencias*, se asemeja a un contador. Tiene como entradas el reloj del sistema (*clk*), el reset asíncrono (*resasin*), y como parámetro el valor *entrada*, que indica el valor máximo al que debe llegar el contador. El valor del parámetro *entrada* se corresponderá con $N_{SDM} = \frac{f_{clk}}{f_{SDM}}$ para el programa que genere la señal *pulsoSD* y con $N_{PWM} = \frac{f_{clk}}{f_{PWM}}$ para el que genere al *pulsoPWM*. Debe tenerse en cuenta que tanto N_{SDM} como N_{PWM} serán enteros.

Como salida tiene la señal *pulso*, que podrá valer 1 ó 0.

La señal pulso será 1 cuando el contador alcance al valor del parámetro *entrada*, y será 0 cuando aún no se haya dado esta condición. Por su parte, el contador aumentará su valor cada vez que la señal *clk* pase a valer de 0 a 1. Siempre que el contador alcance el valor máximo (es decir, el valor *entrada*), pasará a valer de nuevo 0, repitiéndose así el proceso continuamente.

La señal *entrada*, al ser el valor máximo al que llega el contador, es la que indica la relación que habrá entre la frecuencia del reloj del sistema, clk, y la frecuencia de la señal pulso. Concretamente, se cumplirá que $f_{pulso} = \frac{f_{clk}}{entrada}$, tal y como se ve ejemplificado en la siguiente imagen:

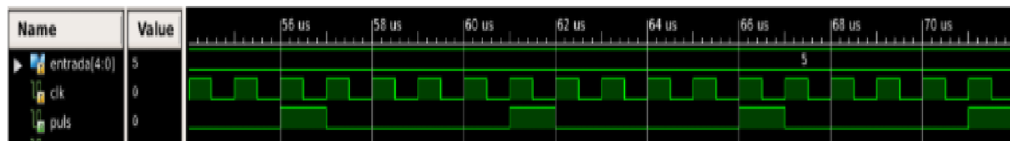

Figura 4.4 Formas de onda (reloj y salida) de un divisor de frecuencia cuando se divide por 5.

En esta imagen se observa que, para el caso en el que el valor de entrada sea 5 y el período de la señal clk es 1 us, el programa genera la señal pulso, que valdrá 1 cada 5 pulsos de clk. También es apreciable que el ancho de la señal puls es el doble del de la señal clk.

Por lo tanto, este programa permitirá generar los pulsos de los que dependerá el comportamiento de tanto el bloque 2 (en este caso el *pulsoSD*) como del bloque 3 (en este caso tanto *pulsoSD* como *pulsoPWM*).

4.1.2 Generador pulsoRZ

El comportamiento de este programa es similar al anterior. También está basado en un contador, y tiene como entradas *clk*, *resasin* y un valor máximo al que debe llegar dicho contador (denominado *entRZ*). Como salida tiene la señal pulsoRZ, que también podrá valer 1 o 0.

La diferencia con el programa anterior es que en éste la salida pulsoRZ cambiará su estado sin que el contador alcance el valor *entRZ*, sino la mitad de dicho valor. Cuando se de esa condición, la señal pulsoRZ (que hasta entonces, al contrario que en el programa anterior, se habrá mantenido a 1) pasará a ser 0. Una vez que el contador alcance, esta vez sí, el valor *entRZ*, el valor del pulsoRZ pasará a ser 1 de nuevo, de forma que dicho proceso se repetirá constantemente. En la figura 4.5 se puede ver un ejemplo de cómo se comportaría este programa, para el caso de que el valor de *entRZ* sea 8 y el período de la señal *clk* 1 us.

Se observa que, como se ha dicho, la salida será un pulso que valdrá la mitad del tiempo 1 y la otra mitad 0. Para este caso concreto, la entrada es 8, por lo que una vez que se hayan producido 8 ciclos de reloj (ó lo que es lo mismo, que el valor del contador haya aumentado 8 veces, puesto que está sincronizado con el del reloj), el comportamiento de la

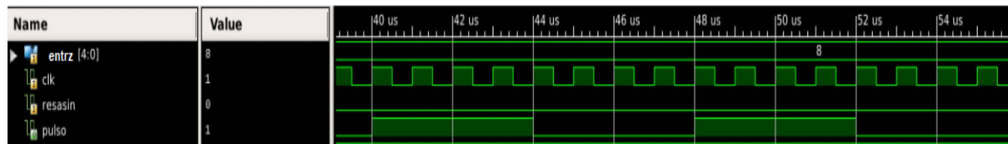

Figura 4.5 Comportamiento de la señal pulsoRZ.

señal *pulsoRZ* (pulso en la imagen) se repetirá, siendo 4 us el tiempo que está a 1 y 4 us el tiempo que está a 0.

La utilidad de la señal *pulsoRZ* será generar señales con retorno a cero (RZ), tal y como se verá en la figura 4.24, lo que se explicará en detalle al llegar al bloque 4:*Generación de las salidas*.

4.1.3 Contador n y Valores

Estos dos programas, cuya relación puede verse en la figura 4.3, son los que permiten generar la señal de entrada muestreada. Para ello, es necesario también comentar *cómo* se va a generar la señal de entrada.

El primer paso será determinar el número de muestras que se tomarán de la señal de entrada en cada uno de sus períodos. Como ya se sabe, la forma de la señal de entrada se puede ver en la figura 4.6, donde se observa la duración de un período, T. Dicho valor T

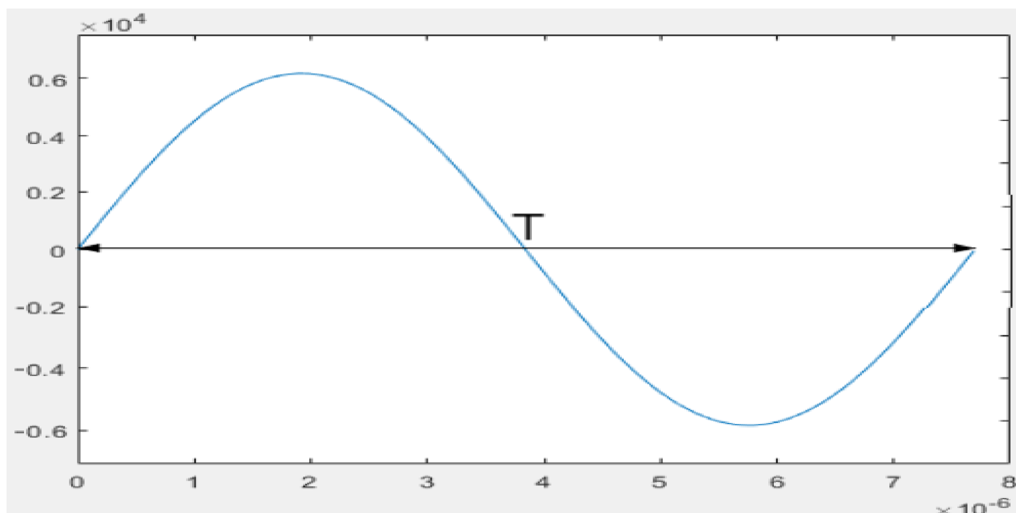

Figura 4.6 Representación de la señal de entrada.

debe cumplir una relación con el número de muestras que se tomará en cada período, N_T, y el tiempo que transcurre entre la toma de dos muestras consecutivas, t_s. De esta forma, la expresión que se verifica es:

$$\frac{N_T}{f_s} = T \tag{4.1}$$

siendo $f_s = \frac{1}{t_s}$ la frecuencia de muestreo.

Dicha frecuencia deberá coincidir con la frecuencia con la que se actualiza la entrada del SDM (ó lo que es lo mismo, la frecuencia con la que trabaja el SDM , f_{SDM}).

Ahora, teniendo en cuenta la expresión del ancho de banda:

$$B = \frac{f_{SDM}}{2 \cdot OSR} \tag{4.2}$$

Donde OSR es el ratio de sobremuestreo, que es OSR=32, por lo que se tiene que $B = \frac{f_{SDM}}{64}$. En la práctica, como mínimo el tercer armónico del seno de entrada debe caer en el ancho de banda B, lo que se refleja en la siguiente relación entre la frecuencia de muestreo (f_{SDM}) y la frecuencia del seno de entrada (f_b):

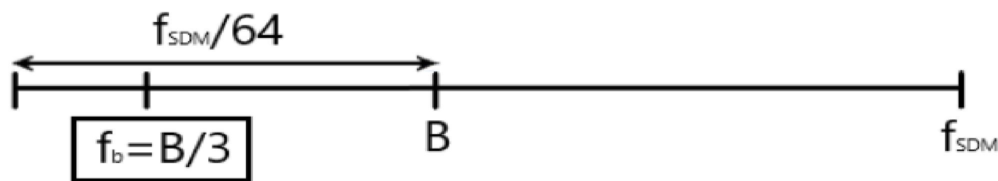

Figura 4.7 Representación de las relaciones entre frecuencias.

Por ello, $f_b = \frac{B}{3} = \frac{f_{SDM}}{3 \cdot 2 \cdot OSR} = \frac{f_{SDM}}{192}$. Sin embargo, se va a tomar una frecuencia que sea 2 veces más pequeña que la frecuencia límite obtenida, de forma que se cumpla la condición del apartado anterior ($f_b < \frac{f_{SDM}}{192}$). Por lo tanto, finalmente se obtiene que:

$$f_b = \frac{f_{SDM}}{192 \cdot 2} = \frac{f_{SDM}}{384} \rightarrow T = 384 \cdot ts \tag{4.3}$$

Esto significa que por cada período de la señal de entrada se tomarán 384 muestras. Una vez definido este valor, se necesitarán 2 programas en VHDL que lleven a cabo la función de generar la señal de entrada. El primero de ellos deberá ser un contador que vaya de 0 a 383 (es decir, que cuente 384 veces), que determinará cuál de las 384 muestras posibles es la que debe entrar al modulador Sigma/Delta en cada momento (programa **Contador n**). El otro deberá ser un programa que le asigne a la entrada del modulador el valor correcto, que vendrá definido por el índice (que se ha denominado n) que se obtiene del programa contador. Este último programa será el denominado **Valores**.

Contador n

Como su nombre indica, este programa se trata de un contador, que aparte de recibir como entrada el reloj del sistema y el reset asíncrono recibe el *pulsoSD*. De esta forma, el valor de las señales internas dependerá de cuánto valga el *pulsoSD*, además de *clk* y *resasin*. Concretamente, la señal que recoge el valor del contador aumentará en una unidad cuando el valor de *pulsoSD* sea 1 (salvo en el caso de que se haya alcanzado el máximo valor al que debe llegar el contador, ya que entonces lo que hará dicha señal será reiniciarse).

Aunque pueda parecer que no es necesaria, en este programa debe entrar también la señal *clk*, debido a que esta es la que permite que las señales que dependen del valor de *pulsoSD* comprueben si *pulsoSD* ha cambiado de valor.

La salida de este programa será el índice n, que determinará en cada momento cuánto será el valor de la entrada del modulador Sigma/Delta, tal y como se verá al explicar el siguiente programa.

Valores

Este programa consiste en una tabla de correspondencia. Una tabla de correspondencia es una estructura que se usará en varias partes de los códigos de VHDL, de forma que le asigna un valor a una salida dependiendo del valor que tenga la entrada.

De esta forma, cada valor de entrada n tiene asociado un valor de salida, que se denominará *valorxt*. Dicho valor será la entrada del SDM. Los valores de la tabla de correspondencia serán las 384 muestras que se habrán tomado del seno de entrada, de forma que cada vez que se reinicie la señal n, también lo hará la señal *valorxt*. Dichas muestras, como ya se sabe, serán números enteros en complemento a 2 expresados en el sistema binario.

Aunque no reciba como entrada las señales clk y resasin, valoraxt depende de ellas puesto que su valor viene dado por el índice n que, como ya se vio en la explicación del programa anterior, está determinado por esas señales.

Los valores de la tabla de correspondencia han sido muestreados del seno de entrada, cuya amplitud se recuerda que es A=$10^{\frac{-3}{20}} \cdot SatuQ$ y su frecuencia es f_b. El valor de f_b se puede poner en función de la frecuencia de muestreo, como se vio con anterioridad en 4.3

$$f_b = \frac{f_{SDM}}{384} = \frac{\frac{f_{clk}}{N_{SDM}}}{384} = \frac{f_{clk}}{384 \cdot N_{SDM}} \tag{4.4}$$

Por lo tanto, ya que la expresión de la señal de entrada es $x(t) = A \cdot \sin(2\pi \cdot f_b \cdot t + \frac{\pi}{2})$, los valores de la tabla de correspondencia deberían cambiar con N_{SDM}, de la que depende f_b. Sin embargo, se debe tener en cuenta que la señal x(t) se va a muestrear, y que el periodo de muestreo será f_s, de forma que x(t) cumpliría la siguiente expresión:

$$x(t) \equiv x(nt_s) \tag{4.5}$$

Donde n será siempre un entero entre 0 y 383. La expresión anterior se puede escribir de la siguiente forma:

$$x(n) = A \cdot \sin(2\pi \cdot f_b \cdot n \cdot t_s + \frac{\pi}{2}) \tag{4.6}$$

Y ya que $t_s = \frac{1}{f_{SDM}} = \frac{1}{f_b \cdot 384}$, la expresión de la señal muestreada finalmente será:

$$x(n) = A \cdot \sin(2\pi \cdot \frac{n}{384} + \frac{\pi}{2}) \tag{4.7}$$

Por lo tanto, se demuestra así que los valores de la tabla de correspondencia serán independientes de la frecuencia del seno de entrada, y que solo dependen de la amplitud, que será constante.

Estos valores se redondean al entero más cercano, y estarán codificados en complemento a

dos de binario.

4.2 Bloque 2

Se ha denominado bloque 2 de la figura 4.2 al conjunto de programas que conforman el modulador Sigma/Delta, y que por lo tanto recibe como entrada la señal *valoraxt* y emite como salida la señal *yn*, que será a su vez la entrada del tercer bloque. Los programas que conforman este bloque son retrasos, sumas, restas, duplicadores y tablas de correspondencia. Para hacer más sencilla su programación, algunos de ellos se agruparon en conjuntos según su función. Dichos conjuntos se mantendrán a la hora de explicar cómo se comporta este bloque. En la figura 4.8 se muestra de qué programas está compuesto el bloque SDM y las señales que los relacionan. Tras ello, se procederá a explicar el comportamiento de cada uno de ellos en VHDL.

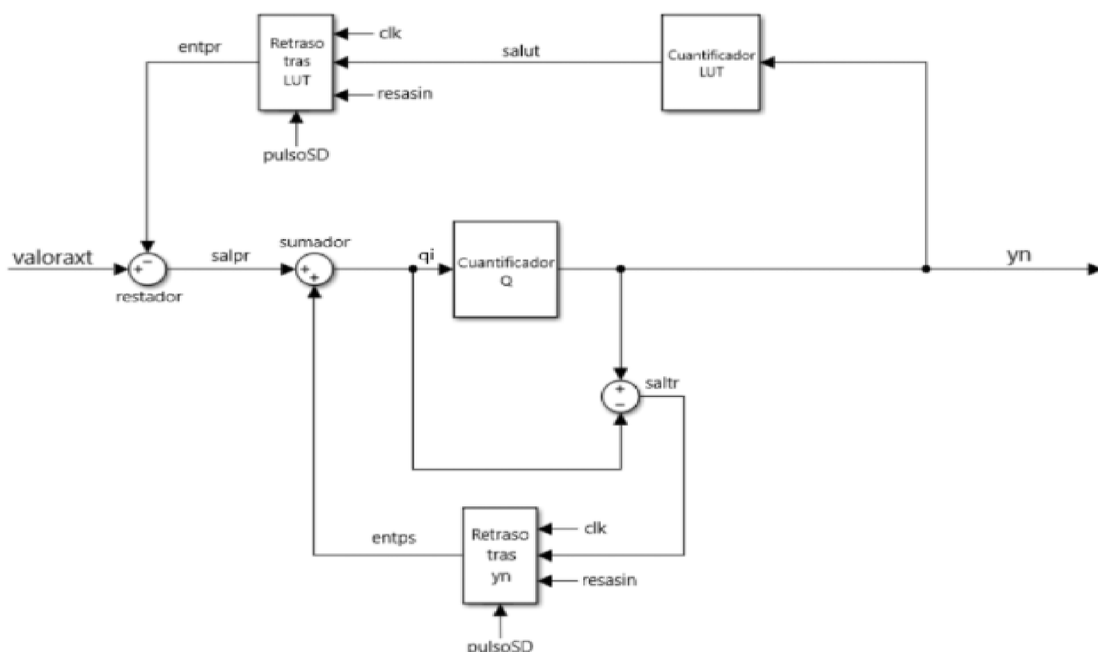

Figura 4.8 Componentes del Bloque 2 del convertidor.

4.2.1 Programa sumador

Como su nombre indica, este código permite sumar dos entradas, *a* y *b*, que se recuerda que serán dos números binarios en complemento a dos.

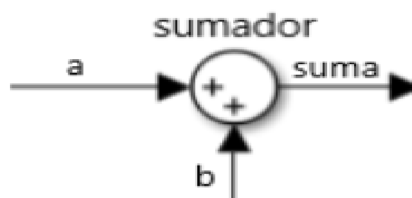

Figura 4.9 Sumador.

Se debe tener en cuenta que, cuando el signo (primer bit) de a y b sean el mismo, se puede producir desbordamiento a la salida, por lo que se deberá comprobar que en ese caso, el signo (primer bit) del resultado sea el mismo que el de a y b. En caso contrario, se habrá producido desbordamiento, ya que el resultado debería ser del mismo signo que los sumandos (positivo si ambos son positivos, negativo si ambos son negativos).

Tabla 4.1 Criterio para identificar el desbordamiento en el caso de la suma .

Signo de la entrada **a**	Signo de la entrada **b**	Signo esperado de la salida	Signo de la salida si desbordamiento
+	+	+('0')	-('1')
-	-	-('1')	+('0')

Si se detecta desbordamiento, el valor a la salida será el más grande posible con los bits disponibles cuando el signo de los sumandos sea positivo, y el valor más pequeño posible cuando el signo de los sumandos sea negativo, para así lograr minimizar el error.

4.2.2 Programa restador

Este programa es similar al anterior, con la diferencia de que a la primera de las entradas (a) se le resta la segunda (b). Ya que, como se sabe, se trabaja en complemento a dos, la resta se reduce a sumar al primer número el complemento a dos del segundo.

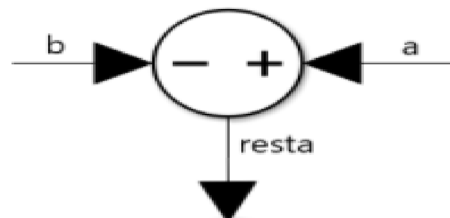

Figura 4.10 Restador.

Al igual que en el caso del sumador, también se puede producir desbordamiento; aunque esta vez si las entradas a y b tienen signos distintos. Por ello, se deberá comprobar si, en ese caso, el signo de la salida (su primer bit) es el mismo que el de la entrada a.

Tabla 4.2 Criterio para identificar el desbordamiento en el caso de la resta.

Signo de la entrada **a**	Signo de la entrada **b**	Signo esperado de la salida	Signo de la salida si desbordamiento
+	-	+('0')	-('1')
-	+	-('1')	+('0')

Si se detecta desbordamiento, el proceso que se realizará para minimizar el error será similar al del programa anterior. El valor a la salida será el más grande posible con los bits disponibles cuando el signo de la entrada a sea positivo, y el valor más pequeño posible cuando el signo de la entrada a sea negativo.

4.2.3 Cuantificador Q

Este bloque realiza la primera cuantificación del SDM, de forma que genera la salida de este modulador, y_n, a partir de q_i. Sin embargo, no solo cuantifica q_i, sino que hace las operaciones que se ven en la figura 4.11 (en la que se recuerda que $pasoQ$=1104) con

el propósito de que el cuantificador sea de tipo *midrise*. Aunque se podrían codificar las

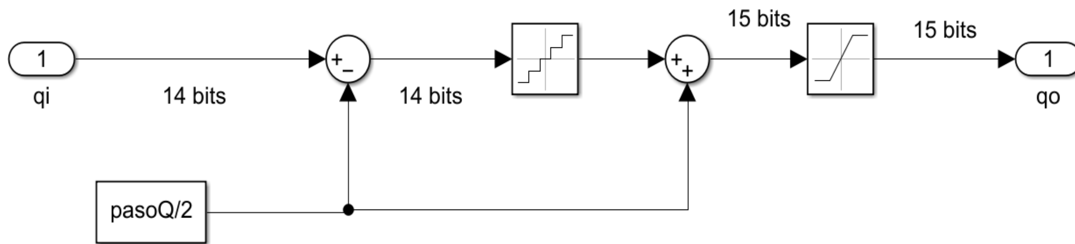

Figura 4.11 Componentes del bloque Cuantificador Q en Simulink.

operaciones que conforman este bloque en VHDL, se ha considerado más sencillo calcular cuanto valdrá la salida para cada intervalo de valores posible de la entrada, lo cuál es factible ya que en todo este bloque las operaciones son de lógica combinacional (es decir, que la salida en cada momento solo dependerá de cuanto valga la entrada en ese momento, y no de cuánto valía esta en instantes anteriores).

De esta forma, como en este bloque se cuantificará la entrada, se puede calcular los intervalos que definirán cada una de sus salidas, y el valor que tendrá según cada uno de estos intervalos. En esta imagen se muestran los resultados de estos cálculos, se ha indicado en la columna de la izquierda en qué rangos de valores se puede encontrar la entrada (q_i) y en la columna de la derecha el valor de la salida de este bloque, q_0, dependiendo de en que rango de valores se encuentre q_i :

qi	q_0
qi ≤ -7729	-8280
-7728 ≤ qi ≤ -6625	-7176
-6624 ≤ qi ≤ -5521	-6072
-5520 ≤ qi ≤ -4417	-4968
-4416 ≤ qi ≤ -3313	-3864
-3312 ≤ qi ≤ -2209	-2760
-2208 ≤ qi ≤ -1105	-1656
-1104 ≤ qi ≤ -1	-552
0 ≤ qi ≤ 1103	552
1104 ≤ qi ≤ 2207	1656
2208 ≤ qi ≤ 3311	2760
3312 ≤ qi ≤ 4415	3864
4416 ≤ qi ≤ 5519	4968
5520 ≤ qi ≤ 6623	6072
6624 ≤ qi ≤ 7727	7176
7728 ≤ qi	8280

Figura 4.12 Salida del Cuantificador Q según el valor de qi .

El valor de q_0 se corresponde con el de y_n que se ha mencionado con anterioridad.

4.2.4 Cuantificador LUT

Como se sabe, el cuantificador LUT tomaba la salida del cuantificador Q, que en la imagen se ha denominado *enlut*, y tras aplicarle las operaciones que se ven en la figura 4.13, le hace una segunda cuantificación de paso 1 para que su salida sea entera. Sin embargo, ya

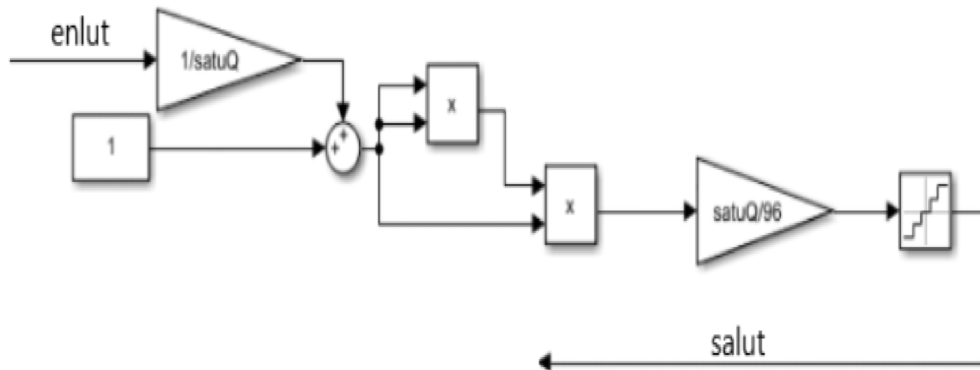

Figura 4.13 Cuantificador LUT en Simulink.

que el valor de la salida del cuantificador LUT solo depende del valor de la entrada (como en el caso del Cuantificador Q), se ha simplificado este programa transformándolo en una tabla de correspondencia, con la diferencia de que mientras en el caso anterior la entrada podía ser cualquier valor, ahora la entrada está limitada a los 16 posibles valores de salida que tiene el cuantificador Q. Por lo tanto, se han realizado a mano el valor que debe tener la salida (en la siguiente tabla *salut*) de este cuantificador para cada posible valor de entrada, tal y como se puede ver en la tabla **??**.

Tabla 4.3 Posibles entradas y salidas del Cuantificador LUT .

enlut	-8280	-7176	-6072	-4968	-3864	-2760	-1656	-552
salut	0	0	2	6	13	26	44	70

enlut	552	1656	2760	3864	4968	6072	7176	8280
salut	105	149	204	272	353	449	561	690

Como en otras ocasiones, se recuerda que en los códigos estos valores están codificados en complemento a dos, y que se han mostrado en decimal para facilitar la comprensión del programa.

4.2.5 Retraso tras LUT

En la figura 4.14 se puede ver en Simulink las operaciones que el bloque Retraso tras LUT aplica a la salida del bloque Cuantificador LUT. Se va a proceder a explicar las operaciones que conforman este bloque para más adelante mostrar como se ha implementado en VHDL.

Operación z^{-1}(retraso)

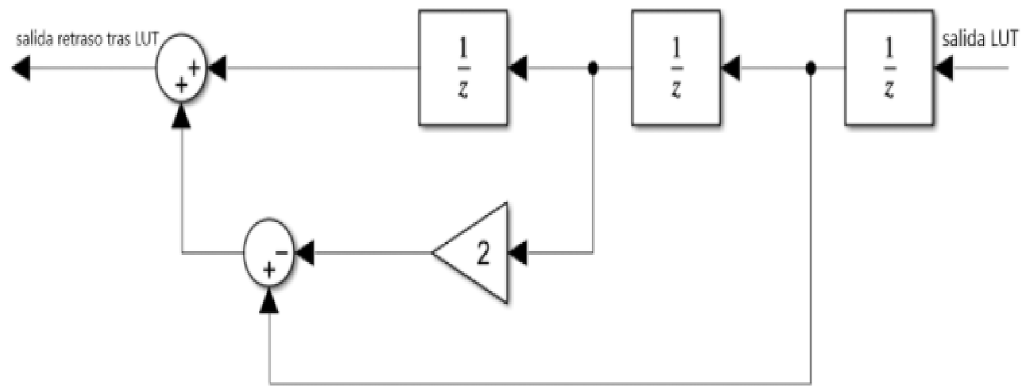

Figura 4.14 Bloque *Retraso tras LUT* en Simulink.

Al trabajar con la transformada z, dividir entre z es lo mismo que retrasar la señal un intervalo de muestreo, lo que significa que su salida será el valor de su entrada un período de muestreo anterior. Es por ello que esta operación se describirá en VHDL como un biestable tipo D (delay). Dicho biestable se puede representar de la siguiente forma:

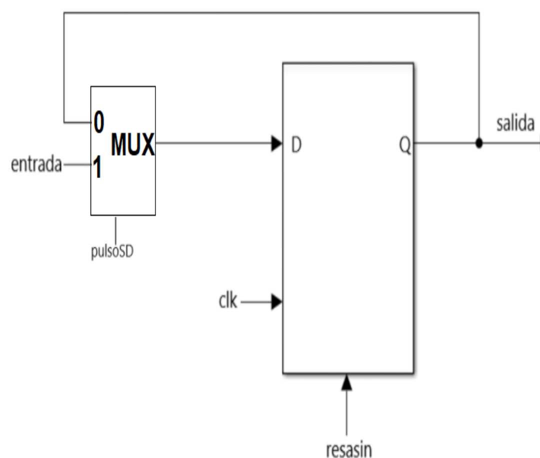

Figura 4.15 Esquema de un biestable.

El comportamiento del biestable se puede resumir en los siguientes pasos:

- La entrada llega al biestable, y si la señal *pulsoSD* está a '1', la señal interna *D* pasa a tener el valor de la entrada. Cuando se produzca un flanco de subida de la señal *clk*, la señal *Q* pasará a tomar el valor de *D*. La salida será el valor de *Q*.

- Si la señal *pulsoSD* está a '0' cuando la entrada llega al biestable, el valor que pasa a ser la señal *D* es el de la salida, Q. De igual forma, el valor de *Q* (y por lo tanto de la salida) pasará a ser *D* cuando se produzca un flanco de subida de la señal *clk*. Por lo tanto, el valor de la salida se mantendrá igual hasta que la señal *pulsoSD* no pase a ser 1.

- En el caso de que la señal de reseteo *resasin* esté activada (valga ‘1’) la salida y las señales intermedias serán nulas.

El comportamiento del bloque biestable se puede ver en la figura 4.16. Como se observa,

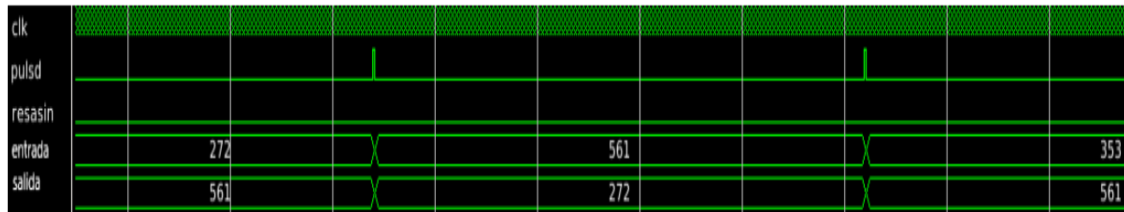

Figura 4.16 Ejemplo del comportamiento del biestable.

la salida es el valor anterior que tenía la entrada, hasta que se produce un flanco de subida de la señal *pulsoSD*, momento en el que se actualizan tanto la entrada como la salida: la salida pasa a tener el ahora antiguo valor de la entrada, y la entrada el próximo valor que se le asignará a la salida una vez se produzca el nuevo flanco de subida del *pulsoSD*.

Ganancia

La operación ganancia permite multiplicar la entrada por un valor, siendo dicho valor 2 para este programa, tal y como se ve en el esquema de la figura 4.14. Ya que estará codificada en binario, duplicar el valor de la entrada se reducirá a añadir un 0 a la derecha de la última cifra que conforme su cadena de bits, sea cual sea su signo. El resto de operaciones que conforman este bloque son sumas y restas. Su explicación se omite al haberse realizado ya en apartados anteriores. Una vez explicado el comportamiento de este bloque, se muestra a continuación un esquema de como se ha descrito en VHDL, mostrando las señales que relacionan cada una de sus partes:

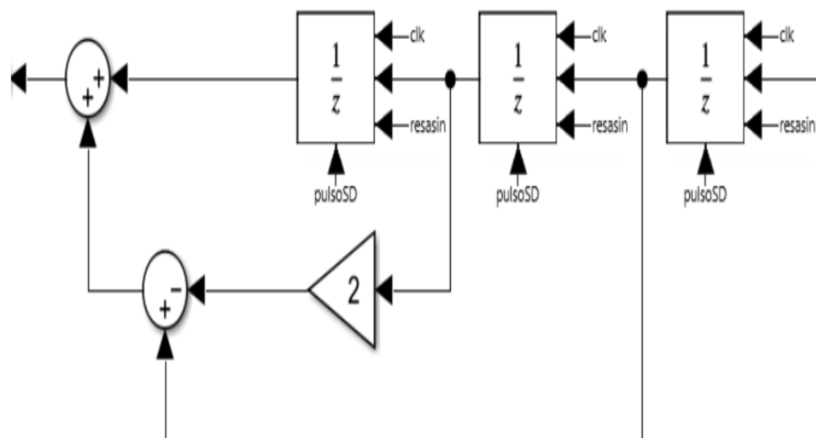

Figura 4.17 Entradas/salidas de los componentes del bloque retraso tras LUT en VHDL.

4.2.6 Retraso tras yn

Los elementos que forman este bloque pueden verse en la siguiente imagen: Como se

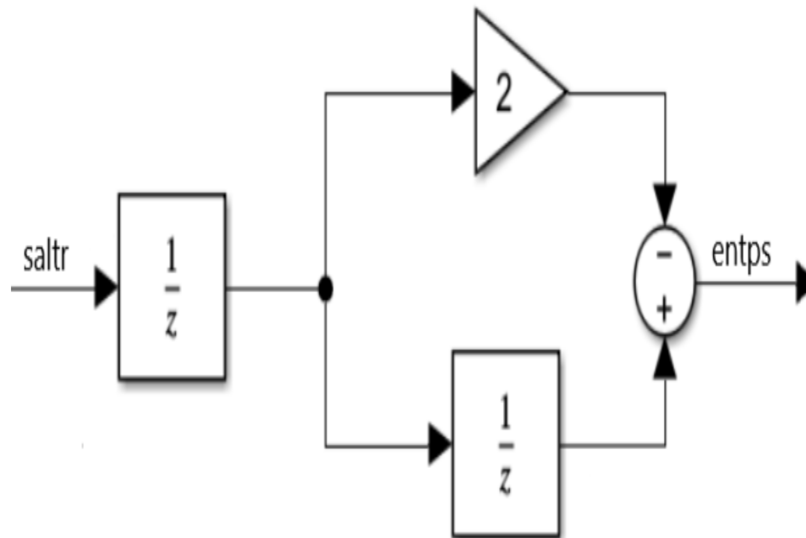

Figura 4.18 Bloque Retraso tras yn en Simulink.

observa, ya se ha explicado cómo se implementan en VHDL los elementos que conforman este sub bloque, por lo que se procede a mostrar una representación esquemática de dichos elementos y cuáles son las señales que los relacionan:

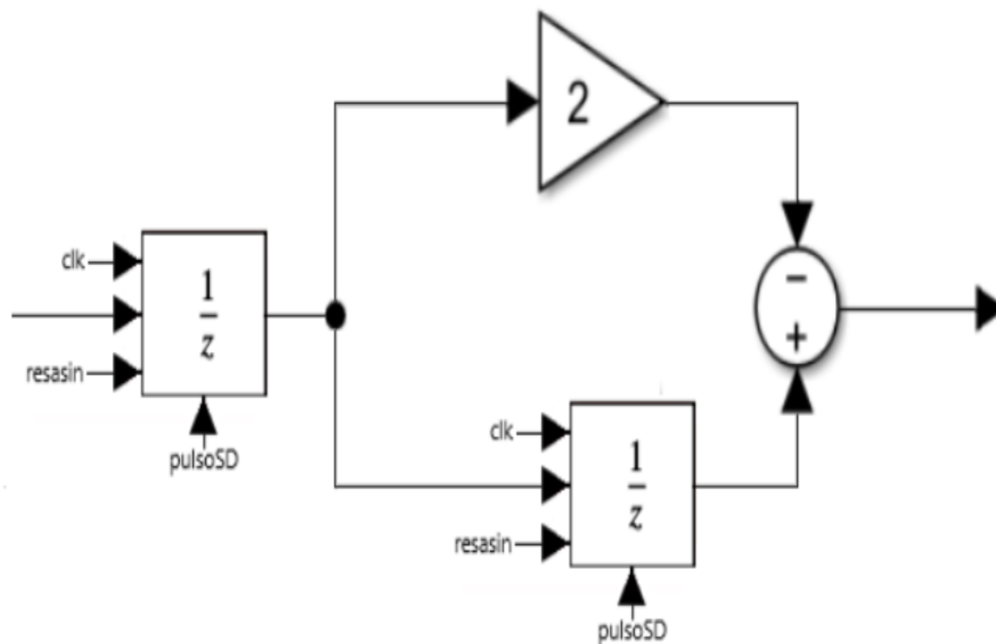

Figura 4.19 Componentes del bloque *Retraso tras yn* en VHDL.

4.3 Bloque 3

Una vez explicado el bloque del SDM, se va a continuar explicando el bloque PWM y los códigos que describen su comportamiento. Dicho comportamiento es más sencillo que en el caso del bloque anterior, puesto que mientras en el bloque SDM estaba compuesto por diferentes operaciones (sumas, restas, varias LUT, varios retrasos…), el bloque PWM solo está compuesto por una LUT, un biestable y un registro de desplazamiento.

El bloque PWM recibirá como entrada la salida del SDM que, como se recuerda, puede ser uno entre 16 valores posibles. El valor de dicha entrada, tal y como se vio en los capítulos anteriores, se debería comparar con una señal triangular, lo que resultaría en una señal de un solo bit que podría valer 0 ó 1. Sin embargo, para simplificar el comportamiento de este bloque, lo que se ha hecho es almacenar en la variable de la que se obtendrá la salida del PWM la cadena de bits que resultaría de comparar el valor de la entrada con la señal de referencia triangular, tal y como se indica en la siguiente imagen:

y_n	variable P
-8280	000000000000000000000000000000
-7176	000000000000001100000000000000
-6072	000000000000011110000000000000
-4968	000000000000111111000000000000
-3864	000000000001111111100000000000
-2760	000000000011111111110000000000
-1656	000000000111111111111000000000
-552	000000001111111111111100000000
552	000000011111111111111110000000
1656	000000111111111111111111000000
2760	000001111111111111111111100000
3864	000011111111111111111111110000
4968	000111111111111111111111111000
6072	001111111111111111111111111100
7176	011111111111111111111111111110
8280	111111111111111111111111111111

Figura 4.20 Variabe *P* según la entrada al bloque PWM.

Como se ve, la variable de la que se obtiene la salida del bloque PWM (a partir de ahora, P) tiene 30 bits, lo que se corresponde con 1 bit por cada nivel diferente del triángulo de referencia. Ahora, se cargará esta variable en un *registro de desplazamiento*.

4.3.1 Registro de desplazamiento

Para explicar el comportamiento de un registro de desplazamiento, se va a hacer un ejemplo con el caso en el que el número de niveles que se pueden recibir como entrada sea 4 (en lugar de 16). Se considerará que dichos niveles son, por simplicidad, 0, 1, 2 y 3; y que el valor de *P* se actualiza con el pulso del bloque SDM (*pulsoSD*). Por lo tanto, se recogen en la siguiente tabla los valores que se almacenarían en la variable *P* dependiendo del nivel de entrada en este ejemplo :

Tabla 4.4 Ejemplo comportamiento señal P .

Nivel de entrada	Variable *P* para dicho nivel
0	000000
1	001100
2	011110
3	111111

Como se ha dicho, se va a cargar la señal *P* en un registro de desplazamiento, se va a definir la señal *D*, que será la variable que recoge la salida de dicho registro de desplazamiento. Dicha variable comparte similitudes con *P*, tendrá su mismo número de bits (en este ejemplo, 6, y en el caso real 30) y al igual que *P* se actualizará con el *pulsoSD*. Sin embargo, mientras que la señal *P* se mantiene igual entre dos *pulsoSD* consecutivos, los bits de la variable *D* se irán desplazando a la izquierda cada vez que se de un flanco de subida del *pulsoPWM*, tal y como se ejemplifica en la imagen que se ve a continuación: Como se

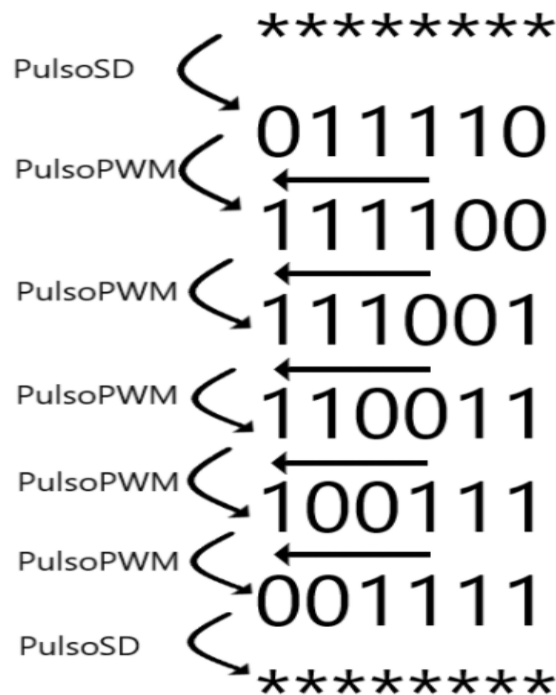

Figura 4.21 Funcionamiento del registro de desplazaamiento.

ve, el *pulsoSD* hace que el valor de *D* se actualice, tomando el valor que *P* tenga en ese momento (pasa de ser ******* a 011110 en el ejemplo), y con cada *pulsoPWM* los bits se van desplazando una posición a la izquierda, mientras que el primero pasa a estar en última posición. Al final, el siguiente *pulsoSD* hará que el valor de la señal *D* vuelva a actualizarse, tomando un nuevo valor que le vendrá dado por *P* (que también se habrá actualizado con *pulsoSD*).

4.3.2 Obtención de la salida

Una vez explicado qué es un registro de desplazamiento, se va a mostrar cual es su importancia al obtener la salida del PWM. Para ello, se puede volver a emplear la imagen anterior:

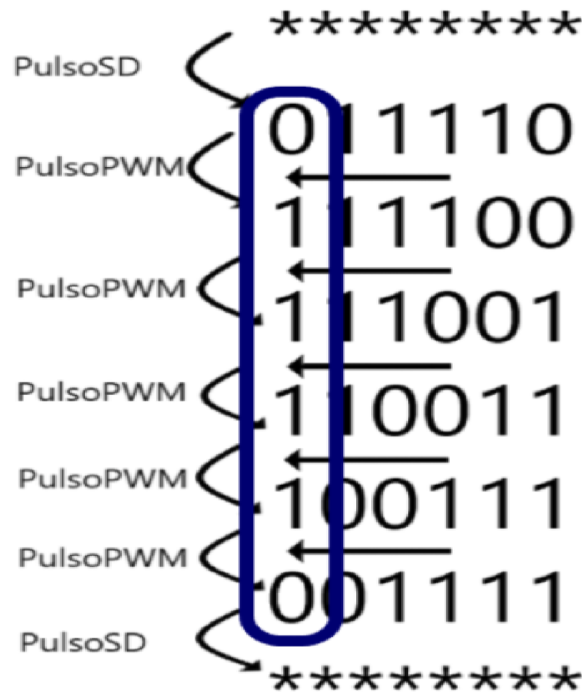

Figura 4.22 Obtención de salida del registro de desplazaamiento.

Se observa que, gracias al registro de desplazamiento, la unión del primer bit de cada valor que toma *D* forma la misma cadena de bits de *D* justo después de actualizarse con *pulsoSD* (es decir, el valor que tenía P). De esta forma, si la salida del PWM es el valor de este bit (el primero de la cadena de bits de *D*), se puede obtener el valor que toma *P* en cada momento, que a su vez viene dado por el valor de la salida del SDM, y todo esto necesitando únicamente un bit. Se debe tener en cuenta que, por lo tanto, la salida del PWM deberá ser (en este ejemplo) 6 veces más rápida que la del SDM para que por cada valor de entrada del SDM se produzcan 6 salidas (en este ejemplo) del PWM.

Para el caso real, se recuerda que el número de bits que conforman la variable *P*, y por lo tanto la variable *D*, es 30. Teniendo en cuenta todo lo dicho en el apartado anterior, se establece así que la relación entre la frecuencia del PWM y del SDM (ó lo que es lo mismo, entre el *pulsoPWM* y el *pusloSD*) debe ser:

$$\frac{N_{SD}}{N_{PWM}} = \frac{f_{PWM}}{f_{SD}} = 30 \tag{4.8}$$

Es decir, que entre 2 *pulsoSD* consecutivos habrá 30 *pulsoPWM*. Esto implica que, una vez que se establezca N_{SD} o N_{PWM} (la relación entre la frecuencia del reloj maestro y la frecuencia de alguno de los dos moduladores), ambos valores estarán definidos por la

relación que se acaba de obtener, $N_{SD} = 30 \cdot N_{PWM}$.

4.3.3 Bloques en VHDL

Una vez aclarado el comportamiento de este bloque, se va a analizar y explicar su estructura y los programas que lo conforman: **Tabla de correspondencia:**

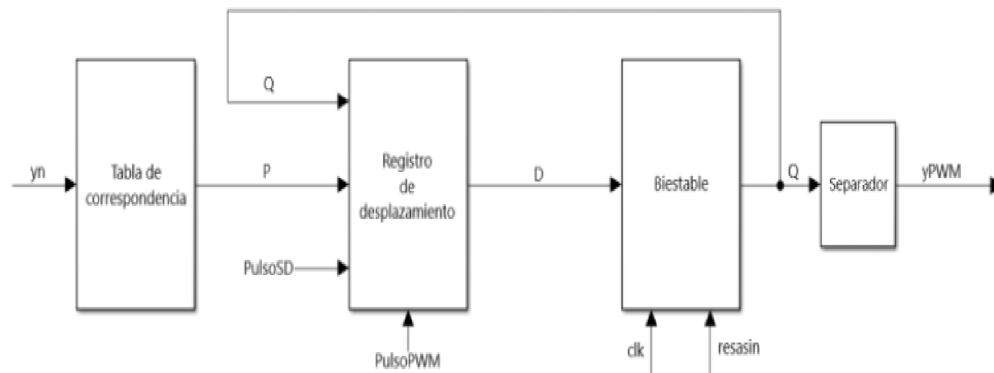

Figura 4.23 Componentes del bloque 3.

Como ya se ha dicho, la tabla de correspondencia sustituirá el proceso de comparación de la señal de entrada con la señal triangular (y por lo tanto la operación de la ganancia de $\frac{1}{SatuQ}$ que se tenía a la salida del SDM antes de llegar al PWM), lo que hace más sencillo describir el comportamiento del PWM en VHDL.

Registro de desplazamiento y biestable:

Estos dos bloques, en conjunto, tienen un comportamiento similar al del bloque z^{-1}. Cuando actúa de forma normal, la señal P llega (lo que coincide con el momento en el que se activa el *pulsoSD*), y le da su valor a D, que tras sufrir un retraso al pasar por el bloque biestable se convierte en Q.

La señal Q tendrá dos funciones; por un lado, su primer bit será la salida del bloque PWM, y por otro, volverá al bloque Registro de desplazamiento, donde cada vez que se active el *pulsoPWM* su valor se almacenará en la señal D con sus bits desplazados a la izquierda, pasando así su primer bit a última posición. De esta forma, la señal D, en la que estará almacenada la señal Q con sus bits desplazados, pasará de nuevo por el bloque biestable y se transformará en la nueva Q, cuyo primer bit (que será el segundo de la señal P original) pasará a ser la nueva salida, repitiéndose así el proceso constantemente, hasta que se vuelva a activar la señal *PulsoSD*.

En el momento en el que *pulsoSD* se vuelve a activar, la P tendrá un nuevo valor, que pasa a D, repitiéndose así el proceso.

Separador:

Este programa simplemente toma el primer bit de la señal *Q*, que será la salida del bloque PWM, y le asigna el nombre *yPWM* (o *salidaPWM*, como se llama en el siguiente bloque).

4.4 Bloque 4

El bloque 4 (que se puede ver en la figura 4.2) toma la salida del bloque 3, *salidaPWM*, y la transforma, originando 2 salidas distintas. La diferencia entre ellas será que una de las dos salidas será una señal con retorno a cero.

Una señal con retorno a cero es aquella que pasa a valer '0' en algún instante de tiempo dentro del período de la señal, tras haber tenido el valor '1' de la señal de forma continuada.

Su utilidad reside en que, cuando se producen cambios del valor de la señal (de '0' a '1') , aparecen ciertas perturbaciones antes de que la señal se estabilice completamente en el valor '1'. Esto puede ser un problema cuando se dan dos o más pulsos consecutivos con valor '1', ya que la forma de onda no será la misma debido a estas perturbaciones. Dicho efecto puede verse representado en la figura 4.24.

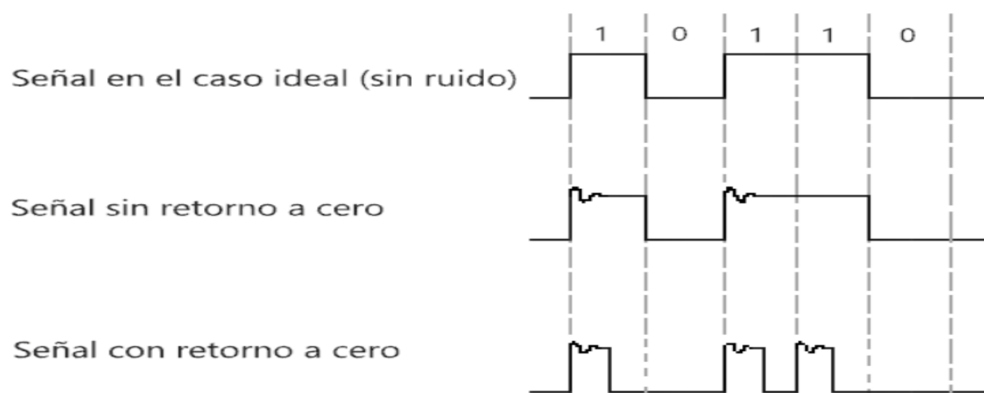

Figura 4.24 Esquema de señales con y sin retorno a cero.

Como se observa en esta imagen, en el caso de la señal sin retorno a cero, se puede producir una pérdida de información ocasionada por la diferencia que hay entre la forma del tercer pulso con valor '1' y la forma de los dos primeros pulsos con valor '1'. Esto es debido a que, en el caso del tercer pulso con valor '1', la señal ya se ha estabilizado en dicho valor.

Mediante el retorno a cero, se consigue que todos los pulsos con valor '1' tengan aproximadamente la misma forma (tal y como se representa en el caso de la señal con retorno a cero) gracias a que se fuerza que haya más transiciones de '0' a '1', con lo que se logrará disminuir el ruido de cuantificación del modulador.

Una vez explicado a esto, se muestra una representación esquemática del bloque 4 en la figura 4.25. Se observa en la imagen anterior que se hace pasar la señal salidaPWM por una puerta lógica AND, junto con la señal *PulsoRZ*. Se recuerda que la señal *PulsoRZ* era

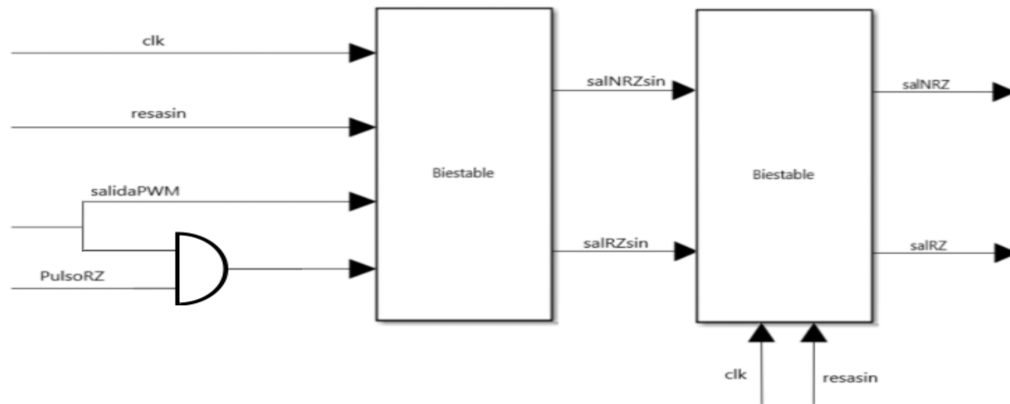

Figura 4.25 Bloque 4 en VHDL.

creada en el bloque 1, pero que no había sido usada hasta ahora. Por sus características, permite realizar la función de transformar una señal en una señal con retorno a cero, tal y como se ve en la figura 4.26.

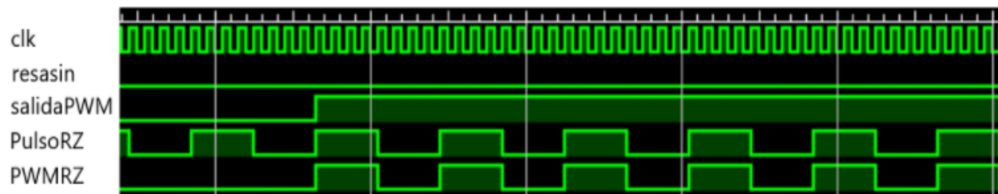

Figura 4.26 Obtención de salida con retorno a cero.

Por lo tanto, la frecuencia de la señal *PulsoRZ* deberá ser la misma con la que se actualiza la señal *salidaPWM*, ya que la utilidad de *PulsoRZ* es aplicarle el retorno a cero a cada uno de los valores de salida del PWM. Esto demuestra que la frecuencia de trabajo de la señal *PulsoRZ*, f_{RZ}, debe ser igual a la que genera resultados el PWM, es decir, la frecuencia f_{PWM}. Por ello,el valor *entRZ*, que se recuerda que era el parámetro del que dependía el comportamiento del programa que generaba la señal *pulsoRZ*, deberá ser:

$$entRZ = N_{PWM} \tag{4.9}$$

La señal resultante, *PWMRZ*, será igual a *salidaPWM* solo que ahora es una señal con retorno a cero. Esta señal entra en el primer bloque *Biestable* junto con la señal *salidaPWM* sin modificar. La primera de ellas, tras pasar por el biestable (lo que genera un retraso) pasará a denominarse *salRZsin*, mientras que la *salidaPWM* tras pasar por el biestable será *salNRZsin*. Una vez que han salido, ambas señales vuelven a entrar en un segundo biestable, lo que les provoca un segundo retraso, formando así las salidas finales del programa, *salRZ* y *salNRZ*. Estos retrasos no modifican la forma de la salida, solo la retrasan, por lo que no hay problema en aplicarlos. Su función es tratar de facilitar y aumentar la calidad de la toma de datos experimental que se hará en el siguiente capítulo.

Como aclaración, se muestran en la siguiente tabla las frecuencias de trabajo que se han visto a lo largo de la explicación del modelo, y como están relacionadas entre ellas:

Tabla 4.5 Frecuencias de trabajo.

Frecuencia	f_{clk}	f_{SDM}	f_{PWM}	f_{RZ}
Valor(Hz)	$50 \cdot 10^6$	$\frac{f_{clk}}{30 \cdot N_{PWM}}$	$\frac{f_{clk}}{N_{PWM}}$	$\frac{f_{clk}}{N_{PWM}}$

4.5 Bloque TOP

Se define al bloque TOP como a la unión de los bloques que se han visto en los apartados anteriores, de forma que se puede ver como el programa que relaciona a los bloques ya vistos y en el que se describen las señales de entradas y salidas que los unen. Es en el bloque TOP donde se define el valor de los parámetros de los que depende el comportamiento del modulador (el valor de N_{PWM}, N_{SD}...) y de los que depende el tamaño de las señales que se usan en el programa (que se obtuvieron como se explicó en el capítulo anterior).

El bloque TOP es el que se sintetizará para sacar resultados de la FPGA, de forma que es interesante realizar simulaciones de dicho bloque, ya que permite poder comprobar que el modulador cumple correctamente su función de reconstruir la señal de entrada. De este bloque no solo se comprobará la salida, sino que también la entrada y salida del modulador Sigma/Delta, ya que es interesante mostrar como cambia la señal de entrada a lo largo del modulador.

4.6 Simulaciones en ISE de Xilinx

Para poder simular el bloque TOP, se debe crear un código adicional denominado banco de pruebas o test bench, en el que se indica como deben comportarse las entradas (clk y resasin) del programa para realizar una simulación correcta. A este proceso se le denomina generación de estímulos del TOP.

En este código se deben especificar cuáles serán las entradas y salidas, y las rutas en las que se encuentran los ficheros en los que se almacenarán los resultados de las simulaciones.

Por lo tanto, se muestran a continuación los resultados de las simulaciones del convertidor, fijando la frecuencia del reloj maestro a f_{clk}=50 MHz y la entrada resasin a 0 tras 20 ns a 1 (para hacer que todas las señales valgan 0 al inicio).

4.6.1 Entrada al modulador Sigma/Delta

Es interesante comprobar que el comportamiento del bloque de entrada (bloque 1) sea el correcto y que por lo tanto la señal sinusoidal muestreada se genere correctamente. Por ello, se ha simulado el funcionamiento del programa y se han tomado muestras de como varía la señal de entrada al bloque 2 de la figura 4.2. Tras ellos, se han tomado estos datos y se han representado en MATLAB la variación temporal de esta señal y su espectro, tal y como se ve en las siguientes imágenes:

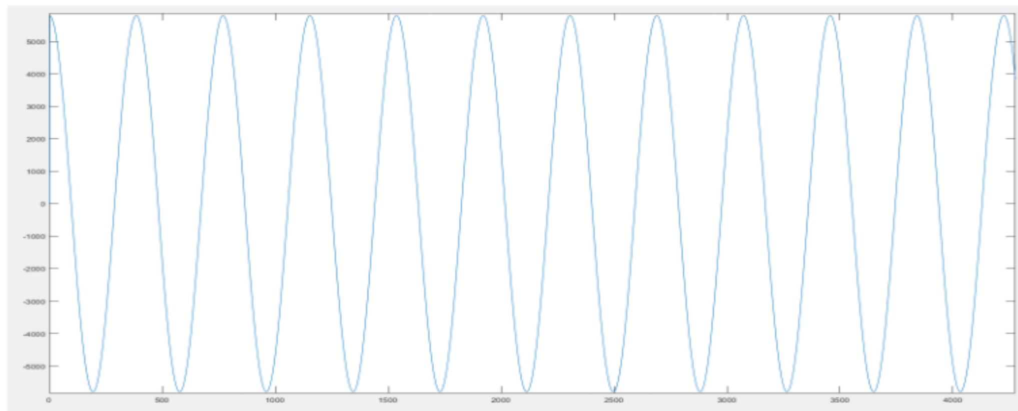

Figura 4.27 Evolución temporal entrada al SDM.

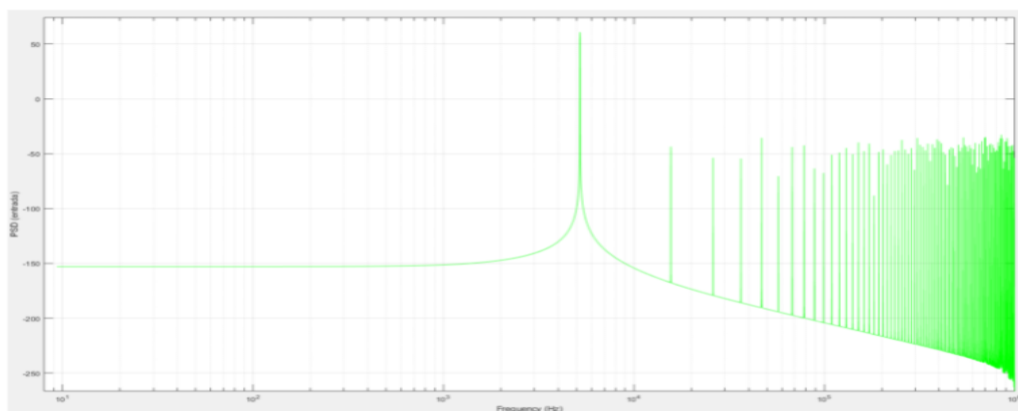

Figura 4.28 Espectro de la entrada al SDM.

Observando ambas imágenes, se comprueba que tanto la forma temporal como el espectro de la señal de entrada son las asociadas a una senoide muestreada, lo que indica que se forma correctamente.

Se ha calculado además el valor de la SNR y HD2 de la entrada, y se ha obtenido:

Tabla 4.6 SNR y HD2 a la entrada del SDM .

SNR(dB)	HD2(dB)
103.9044	104.3732

4.6.2 Salida del modulador Sigma/Delta

Al igual que se hizo para la entrada del modulador Sigma/Delta, se procede a continuación a analizar como evoluciona la salida del mismo modulador. Como ya se ha explicado, la salida del modulador Sigma/Delta está cuantificada, lo que significa que el valor de la salida está restringido a uno de los 16 niveles del cuantificador de dicho modulador.

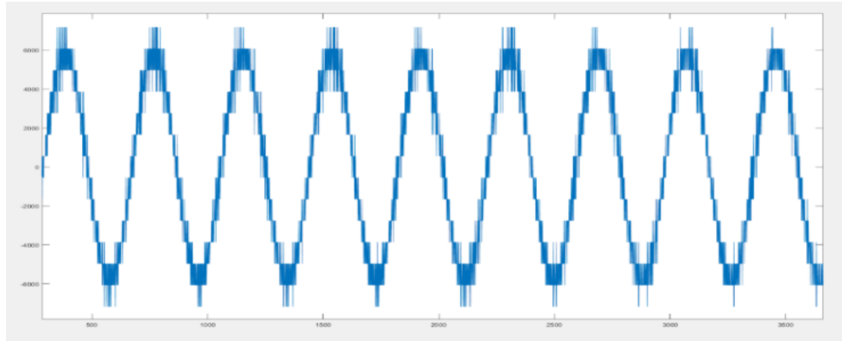

Figura 4.29 Evolución temporal salida del SDM.

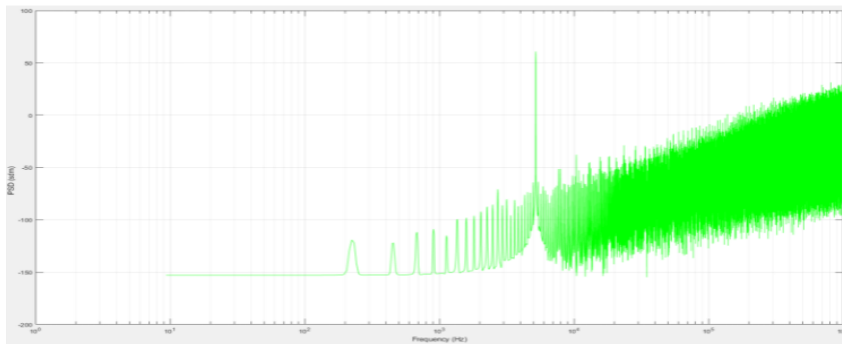

Figura 4.30 Señal espectral de la salida del SDM en VHDL.

Como se observa, el espectro de esta señal muestra similitudes con el que ya se vio que caracterizaba la salida de los SDM (ruido que aumenta al aumentar la frecuencia y más bajo alrededor de la frecuencia fundamental). Al igual que en el caso anterior, se ha calculado el valor de la SNR y la HD2, y se ha obtenido:

Tabla 4.7 *SNR* y *HD2* de la salida del SDM .

SNR(dB)	*HD2(dB)*
83.9806	98.4997

Por otro lado, se ha calculado la SNR asociada a la salida del SDM mediante los modelos de Simulink. Se ha obtenido un valor de SNR_{Ref}=85.772 dB, por lo que se concluye que el valor que se ha obtenido en este apartado es aceptable y señal de que los códigos se han realizado de forma adecuada.

4.6.3 Salida del programa

Al igual que en los casos anteriores, se muestra la forma de la señal de salida y su espectro en las figuras 4.31 y 4.32.

Se observa que el comportamiento del espectro es similar al que se obtuvo en el apartado *Simulación en MATLAB y Simulink*, lo que es señal de que el convertidor funciona correctamente. Se muestran a continuación los valores de *SNR*,*HD2* y *HD3* calculados para este caso:

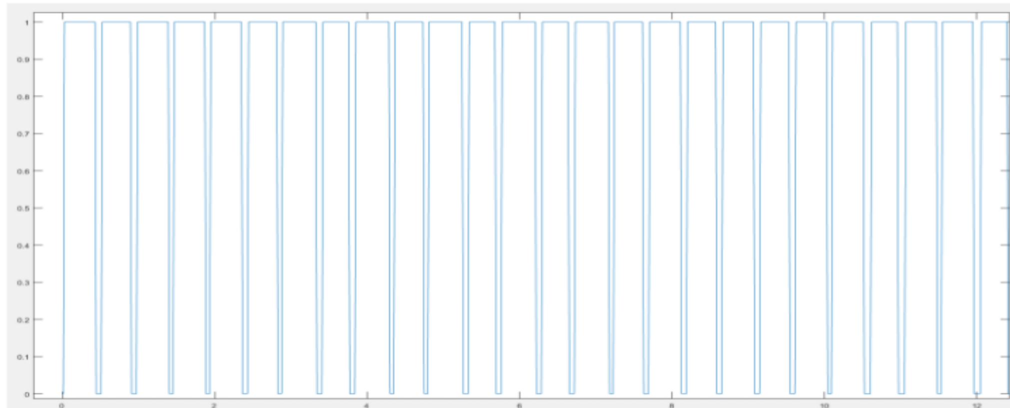

Figura 4.31 Salida del convertidor en el dominio del tiempo.

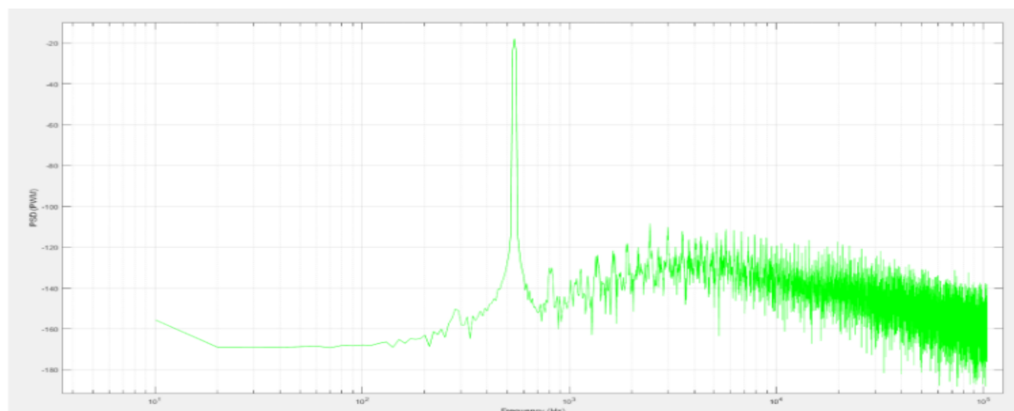

Figura 4.32 Espectro salida del convertidor.

Tabla 4.8 *SNR*, *HD2* y *HD3* de la salida del *PWM*.

SNR(*dB*)	*HD2*(*dB*)	*HD3*(*dB*)
85.1579	112.9475	100.8443

Se recuerda que, en el capítulo anterior, se obtuvo una SNR de referencia de SNR_{Ref}= 86.1876 dB (en la tabla 4.5), $HD2_{Ref}$=108.216532 dB (en la tabla 3.2) y $HD3_{Ref}$=106.084888 dB (en la tabla 3.3). Dado a que los valores obtenidos en esta simulación son similares al que se obtuvieron en la simulación realizada en *Simulink*, se ve justificado asumir que se ha podido llevar a cabo la descripción completa y correcta del convertidor en VHDL.

4.6.4 Simulación Post-Implementación

Los resultados de las simulaciones que se han mostrado en las páginas anteriores han sido hechas en modo Behavioral. Este modo permite simular y comprobar que los códigos se han realizado de forma correcta, pero se debe hacer también una simulación en modo Post-Route, ya que dicho modo emula teniendo en cuenta retrasos y otras perturbaciones que tienen los circuitos reales y que no son tenidos en cuenta en el modo *Behavioral*. Para poder hacer este tipo de simulaciones, se debe indicar dentro del programa que modelo de FPGA se va a utilizar, que en este caso es FPGA XC3S500E-4FG320.

Por ello, se han obtenido por último los resultados de las simulaciones de la salida final, para el modo Post-Route. Para este caso, se han calculado los siguientes resultados:

Tabla 4.9 *SNR*, *HD2* y *HD3* de la salida del *PWM* Post-Implementación.

SNR(*dB*)	*HD2*(*dB*)	*HD3*(*dB*)
84.8100	110.3546	100.3154

Como se observa, el orden de los valores de la *SNR*, *HD2* y la *HD3* son del orden de los que se obtuvieron en el caso de Behavioral, aunque se observe cierta caída producida por los retrasos que han aparecido durante la realización de las simulaciones.

4.6.5 Preparación para la simulación con FPGA

Una vez comprobados los resultados que da el convertidor, se debe conectar el programa hecho en Xilinx con la FPGA . El primer paso para realizar este proceso es indicarle al programa que pines de la placa con la que se va a trabajar serán utilizados. Dichos pines se deberán conectar a las cuatro señales principales del convertidor: clk, resasin, salRZ y salNRZ. El documento en el que se establecen los pines se denomina *Archivo de constraints.*

Los pines que se han escogido y sus características son las siguientes:

clk	Input		C9	☑	0	LVCMOS33*	3.300				NONE
resasin	Input		D18	☑	1	LVTTL*	3.300				PULLDOWN*
SALNRZ	Output		B4	☑	0	LVTTL*	3.300		12	SLOW	NONE
SALRZ	Output		A6	☑	0	LVTTL*	3.300		12	SLOW	NONE

Figura 4.33 Pines asociados a las señales *clk*, *resasin*, *salRZ* y *salNRZ.*

Tanto la posición de los pines como el resto de características que se ven en la tabla anterior deben elegirse por el usuario, y han sido seleccionados siguiendo las recomendaciones que se pueden encontrar en el documento *Spartan-3E FPGA Starter Kit Board User Guide*.[6]

A la hora de seleccionar los pines se debe tener en cuenta que los vinculados a las salidas deberían estar lo más alejados entre sí que se pueda, para así tratar de evitar, en la medida de lo posible, que haya solapamiento entre ellas (es decir, que los cambios de una de las salidas afecten a la otra y viceversa). En la figura 4.34 se muestran los pines de salida que se eligieron en un primer momento, cuya posición se corresponde con la que se veía en la tabla anterior. El pin asociado a la salida salNRZ pertenece al conector J1, mientras que el pin de la salida salRZ al conector J2. No existe diferencia entre pertenecer al conector J1 o J2, ya que sus pines presentan las mismas características.

También se han indicado los puertos correspondientes a los cables de conexión de la placa con el ordenador y con la alimentación eléctrica. En la parte inferior de la imagen se puede observar la posición del interruptor asociado a la señal de entrada *resasin*, que como

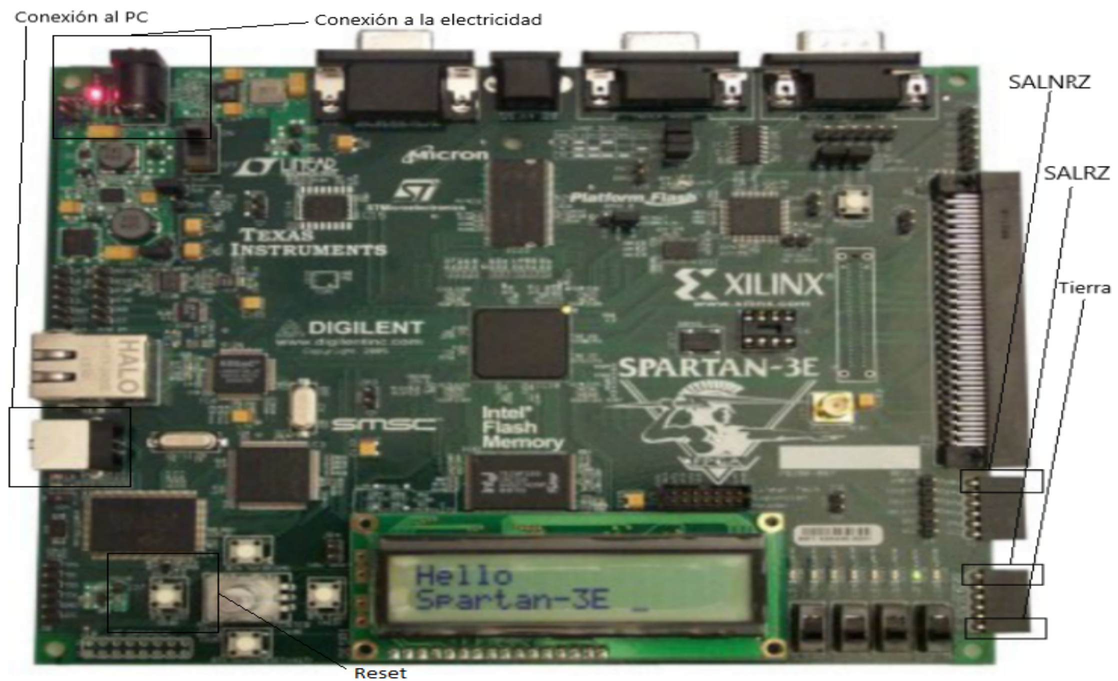

Figura 4.34 FPGA, pines de salida y puertos de conexión.

se sabe resetea (pasa a 0) todas las señales del programa y permite reiniciarlo en cualquier momento al pulsarlo.

Aunque, como se ha dicho, se trató de evitar que hubiera acoplamiento entre las señales de salida alejando los pines de los que se obtienen las salidas, los resultados experimentales, tal y como se muestra en el *Apéndice D Efecto de solapamiento de salida (Crosstalk)*, mostraron que existía cierta dependencia entre el comportamiento de las señales salRZ y salNRZ. Esto dió lugar a que el programa que se ha explicado hasta ahora tuviera que duplicarse, de forma que una de las copias solo genera la salida salRZ y la otro la salida salNRZ. Esto, como tal, no altera el comportamiento de los componentes de los programas que se han explicado hasta ahora, por lo que todas las explicaciones que se han hecho hasta este momento siguen siendo válidas.

4.6.6 Conexión de la FPGA

Una vez que se han realizado todas las simulaciones necesarias, se deben realiza los pasos para programar la FPGA.

Una vez definido el Archivo de constraints, se debe crear el archivo Bitstream. Dicho archivo contiene la información que necesita la FPGA para generar la misma salida que el convertidor que se ha programado. Tras generarlo, se debe abrir la pestaña Tools, donde aparece la opción IMPACT.

Al abrirse la ventana IMPACT, si la FPGA está encendida y se ha conectado el ordenador mediante el cable adecuado, debe ser reconocida por éste. Una vez que se haya obtenido el mensaje de que la programación de la placa ha sido un éxito, en los pines de salida de

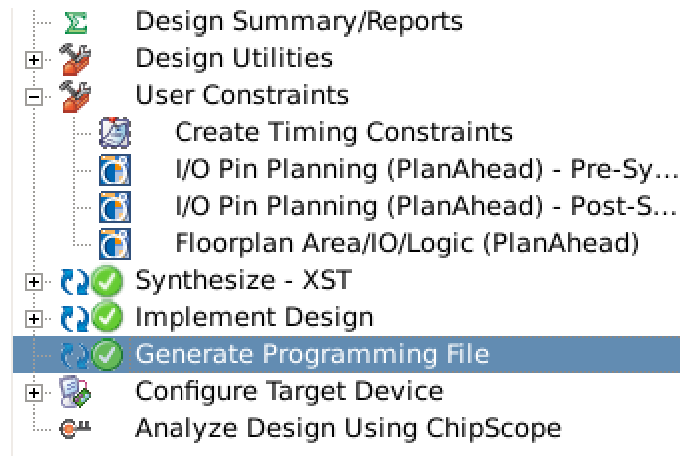

Figura 4.35 Indicación de generación correcta del archivo Bitstream.

la FPGA indicados en el archivo de las Constraints se podrá comenzar a medir la salida del convertidor que se ha diseñado con las herramientas correspondientes. Este proceso se describirá más detalladamente en el siguiente capítulo.

5 Obtención de resultados

Una vez explicado cómo se han descrito en VHDL las diferentes partes del modulador y como se conecta a la FPGA, se procede a continuación a explicar cómo se obtienen los resultados experimentales de esta y a mostrar los análisis de dichos resultados.

Para llevar a cabo este análisis, se han utilizado osciloscopios PicoScope (de 12 y 16 bits de resolución), el software PicoScope6 y los programas de *Matlab* que se pueden ver en el *Apéndice A*. Las características de los osciloscopios se pueden ver en la siguiente tabla:

Tabla 5.1 Características de los osciloscopios.

Resolución	12 bits	16 bits
Ancho de banda máximo	100 MHz	5 MHz
Tasa de muestreo	250 MS/s	10 MS/s
Memoria	32 MS	16 MS

Ya que, como se observa, el osciloscopio de 12 bits presenta mayor tasa de muestreo, servirá para tomar las muestras de los pulsos de salida antes de ser filtrados.

Además, se ha diseñado y realizado un filtro paso bajo que permite la medición de resultados a la salida del modulador, eliminando el ruido que aparece a altas frecuencias (que como se sabe es especialmente alto debido a las técnicas de conformación de ruido que se emplean en el SDM).

5.1 Filtro de paso bajo

Se debe tener en cuenta que, como ya se dijo, el valor de la frecuencia de muestreo (al igual que el del ancho de banda y la frecuencia de la señal original) no son fijos, sino que dependen del parámetro N_{PWM}. Es por esto que los valores de los elementos que conforman el filtro también deberán depender de dicho valor, ya que la frecuencia de corte que debe tener asociado dependerá de N_{PWM}.

Teniendo esto en cuenta, el filtro (que se ha diseñado con la herramienta *Micro-Cap 12*) se puede ver a continuación:

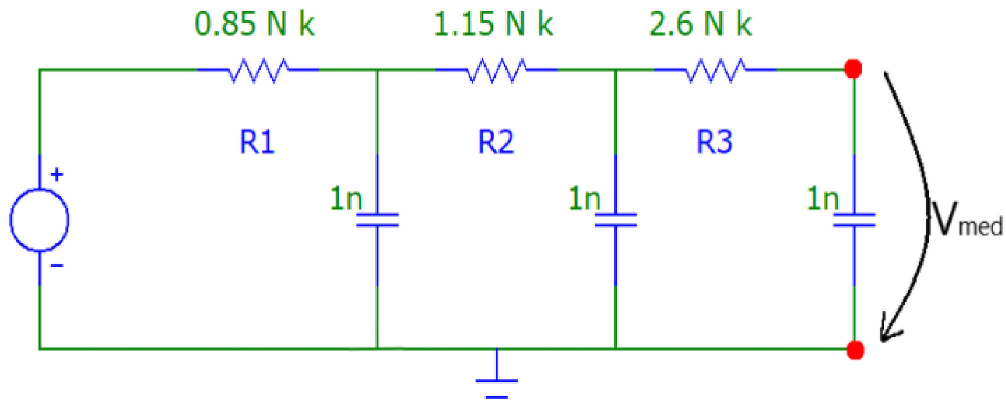

Figura 5.1 Filtro paso bajo en Micro-Cap.

Donde, como se había dicho ya, el valor de las resistencias depende de N_{PWM} (N en la imagen anterior). Los valores se han calculado considerando que la capacidad de los condensadores se va a mantener fija e igual a $1nF$. Las N_{PWM} que se han decidido tomar son:

Tabla 5.2 Resistencias teóricas para cada valor de N_{PWM} .

NPWM	2	6	12	16	20	24	40	80
R1	1.7K	5.1K	10.2K	13.6K	17K	20.4K	34K	68K
R2	2.3K	6.9K	13.8K	18.4K	23K	27.6K	46K	92K
R3	5.2K	15.6K	31.2K	41.6K	52K	62.4K	104K	208K

Sin embargo, los valores de resistencia se han aproximado a valores de resistencias comerciales:

Tabla 5.3 Resistencias disponibles para cada valor de N_{PWM} .

NPWM	2	6	12	16	20	24	40	80
R1	1.74K	5.1K	10K	13.3K	16.9K	20.5K	34.8K	68.1K
R2	2.26K	6.8K	13.3K	18.7K	22K	27.4K	46.4K	91K
R3	5.1K	15.4K	33K	41.6K	52.3K	62K	100K	220K

La frecuencia de corte -3 dB de estos filtros debe coincidir con el ancho de banda de la señal para cada uno de los posibles valores de N_{PWM}. Se recuerda que la expresión del ancho de banda es:

$$B = \frac{f_{SDM}}{OSR \cdot 2} = \frac{f_{clk}}{OSR \cdot 2 \cdot 30 \cdot N_{PWM}} \tag{5.1}$$

Los valores de frecuencia de corte f_c calculada con la expresión anterior serán:

Tabla 5.4 Frecuencias teóricas necesarias para cada valor de N_{PWM} .

NPWM	2	6	12	16	20	24	40	80
f_c(KHz)	13.02	4.34	2.17	1.63	1.30	1.08	0.65	0.32

Estos valores son los teóricos que se deberían poder alcanzar si se tuvieran las resistencias exactas. Sin embargo, como ya se ha dicho, se han tenido que tomar valores aproximados de las resistencias. Aun así, el valor de las frecuencias de corte que se obtienen con las resistencias aproximadas, tal y como se puede apreciar en el *Apéndice C Gráficas de las frecuencias de corte*, son lo suficientemente cercanas a las teóricas, lo que permite que los filtros puedan emplearse.

En la siguiente imagen se puede ver uno de los filtros; el cable rojo es el asociado a la salida del convertidor, mientras que el cable negro es el de retorno a tierra. Por otro lado, la zona marcada es en la que se toman los datos: se deberá conectar a ella la punta de sonda del cable de pruebas que va al osciloscopio. El otro conector de este cable, el cocodrilo, se conectará a otro cable de tierra.

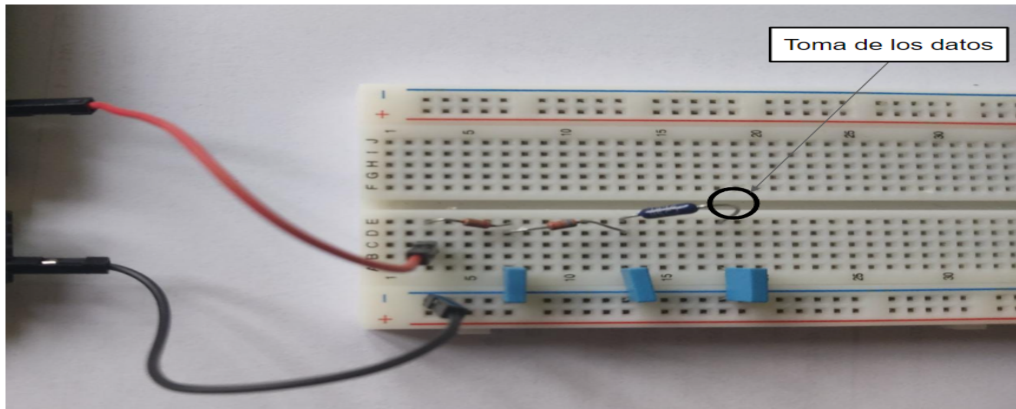

Figura 5.2 Filtro paso bajo en placa de pruebas.

5.2 Almacenamiento de los resultados experimentales

Una vez explicado el filtro paso bajo que se va a utilizar, se va a comenzar a mostrar los resultados experimentales que se obtuvieron. Se deberá tener en cuenta que:

- Los osciloscopios PicoScope tienen un filtro paso bajo interno que elimina las altas frecuencias junto con el que se ha diseñado en el apartado anterior. La frecuencia de corte asociada a dicho filtro interno es ajustable, así que para cada valor de N_{PWM} se ha elegido la frecuencia de corte que se vio en la tabla anterior.

- Antes de tomar los datos definitivos, se hicieron varias pruebas con los equipos de medidas para encontrar la mejor configuración con la que medir los datos. Los valores que se obtuvieron se encuentran recogidos en el *Apéndice E.Resultados de las simulaciones para pines B4 de J1 y A6 de J2.*

Los resultados que se pueden ver en dicho apéndice indican que la mejor forma de medir los datos es con el osciloscopio de 16 bits, en alterna, con la sonda con atenuación X1 y tomando 20 períodos.

5.3 Análisis de los resultados experimentales

Tras explicar cómo se ha configurado la herramienta PicoScope para guardar los datos, se muestra a continuación una representación gráfica de los resultados obtenidos, para cada valor de N_{PWM} y para los casos de salRZ y salNRZ, hecha en MATLAB.

5.3.1 Señal salNRZ para NPWM=2

Antes de mostrar las gráficas de los resultados, se recogen en la siguiente tabla los valores de las frecuencias de relevancia para el caso de $N_{PWM} = 2$. Para calcularlas, se ha recurrido a las expresiones 4.4, 4.8 y 5.1:

Tabla 5.5 Frecuencias para $N_{PWM} = 2$.

N_{PWM}	f_b (Hz)	B (Hz)	f_{SDM} (Hz)	f_{PWM} (Hz)
2	2170,14	13020,8	833,3 $\cdot 10^3$	25 $\cdot 10^6$

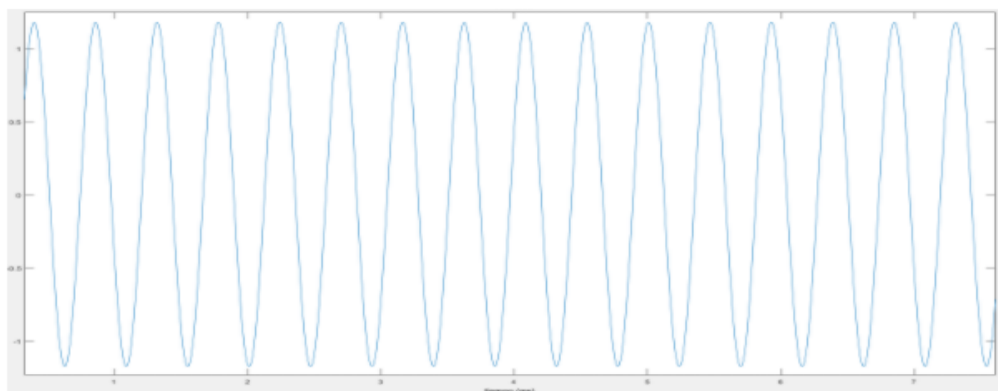

Figura 5.3 Señal salNRZ para N_{PWM}=2 en el dominio del tiempo.

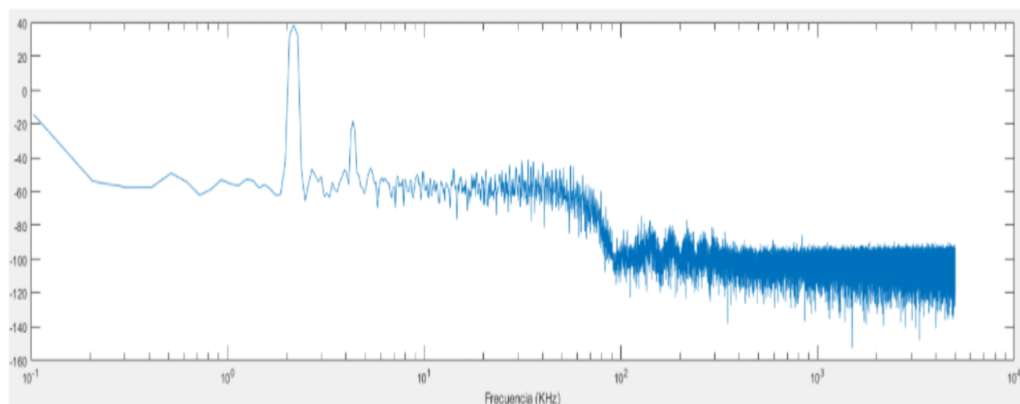

Figura 5.4 Señal espectral salNRZ para N_{PWM}=2.

5.3.2 Señal salRZ para NPWM=2

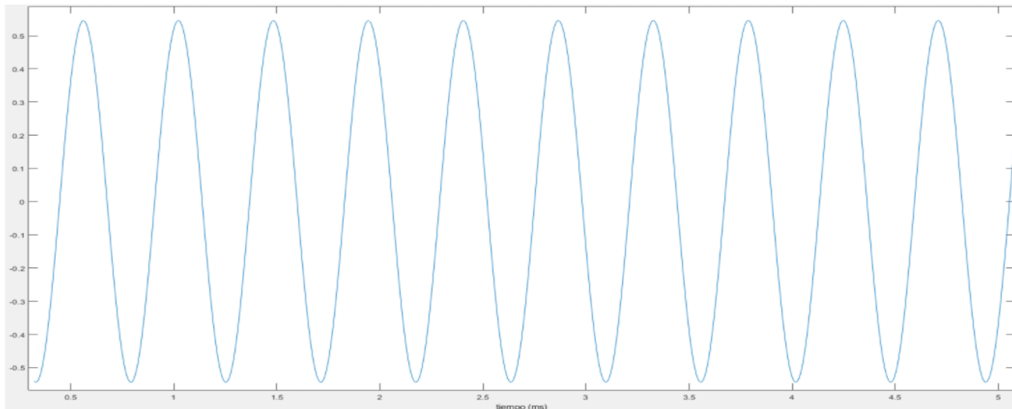

Figura 5.5 Señal salRZ para N_{PWM}=2 en el dominio del tiempo .

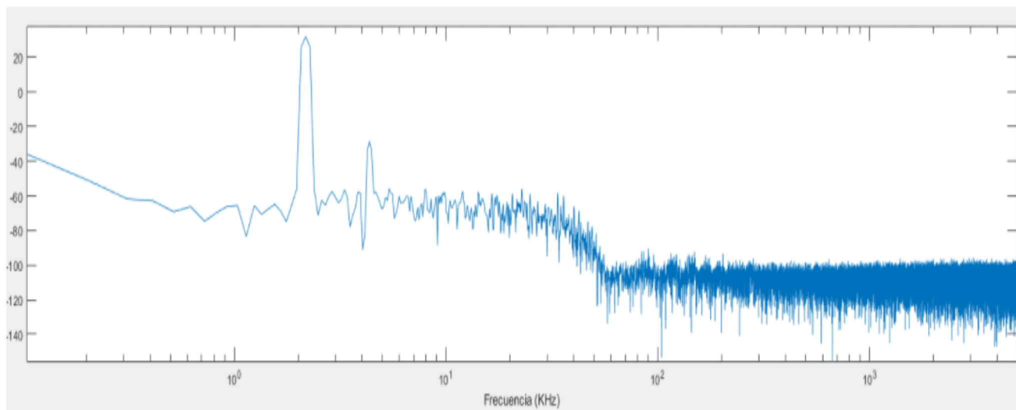

Figura 5.6 Señal espectral salRZ para N_{PWM}=2.

Si se compara en el dominio del tiempo la amplitud de las señales salNRZ y salRZ, se puede observar que la de la primera es el doble que la de la segunda. Esto es debido a que, como se sabe, los pulsos que generan la señal salRZ presentan el retorno a cero, lo que hace que se mantengan en '1' aproximadamente la mitad del tiempo que lo hacen los pulsos de la señal salNRZ, lo que explica que el valor de la amplitud de la salRZ sea aproximadamente la mitad. Por otro lado, se han mostrado en la tabla 5.6 los valores de la SNR, HD2 y HD3 para cada una de las salidas con N_{PWM}=2.

Tabla 5.6 Resultados para $N_{PWM} = 2$.

$N_{PWM} = 2$	SNR(dB)	HD2(dB)	HD3(dB)
salNRZ	76.700	61.302	86.868
salRZ	77.167	60.221	85.735

Como ya se ha indicado en la tabla 5.5, para el caso de $N_{PWM} = 2$ la frecuencia de la onda es $f_b = 2{,}170$ kHz. Se puede observar que, en las gráficas que muestran la representación espectral de las salidas, la frecuencia fundamental se da aproximadamente a dicha

frecuencia, y el segundo armónico aparece aproximadamente a $2 \cdot f_b$, lo que se corresponde con lo esperado.

5.3.3 Señal salNRZ para NPWM=12

Al igual que para el caso anterior, en la siguiente tabla se pueden ver el valor de las frecuencias de las que depende el comportamiento del convertidor para el caso de $N_{PWM} = 12$:

Tabla 5.7 Frecuencias para $N_{PWM} = 12$.

N_{PWM}	f_b (Hz)	B (Hz)	f_{SDM} (Hz)	f_{PWM} (Hz)
12	361,7	2170,14	$139 \cdot 10^3$	$4{,}17 \cdot 10^6$

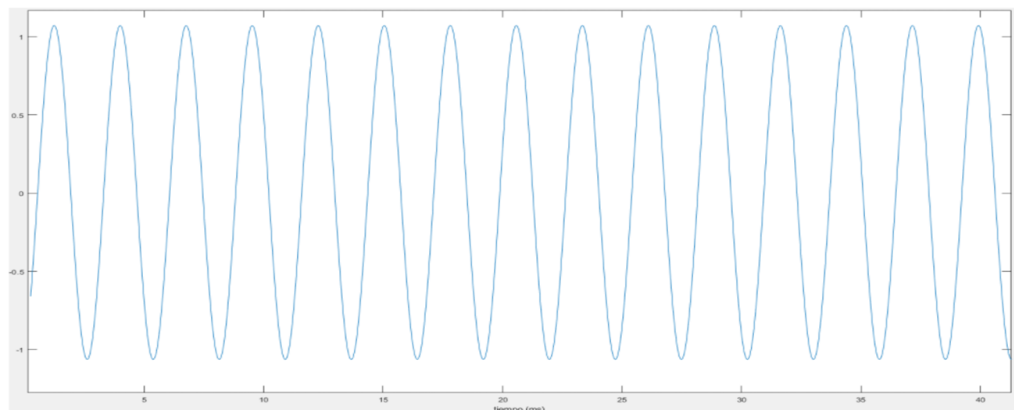

Figura 5.7 Señal salNRZ para N_{PWM}=12 en el dominio del tiempo.

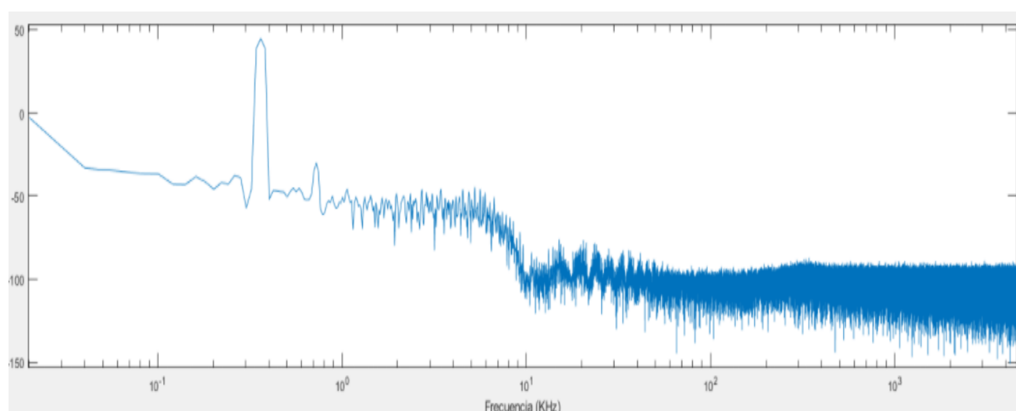

Figura 5.8 Señal espectral salNRZ para N_{PWM}=12.

5.3.4 Señal salRZ para NPWM=12

Se muestran a continuación los resultados de la SNR, HD2 y HD3 que se obtienen al analizar las salidas del convertidor cuando N_{PWM}=12:

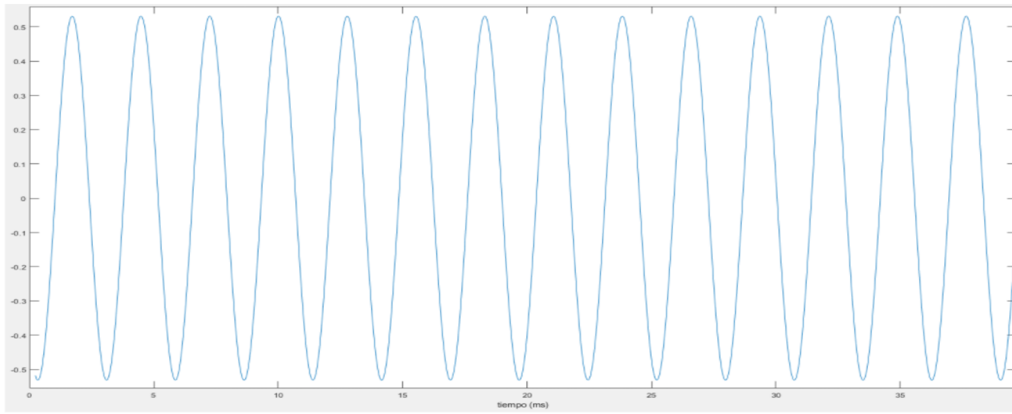

Figura 5.9 Señal salRZ para N_{PWM}=12 en el dominio del tiempo.

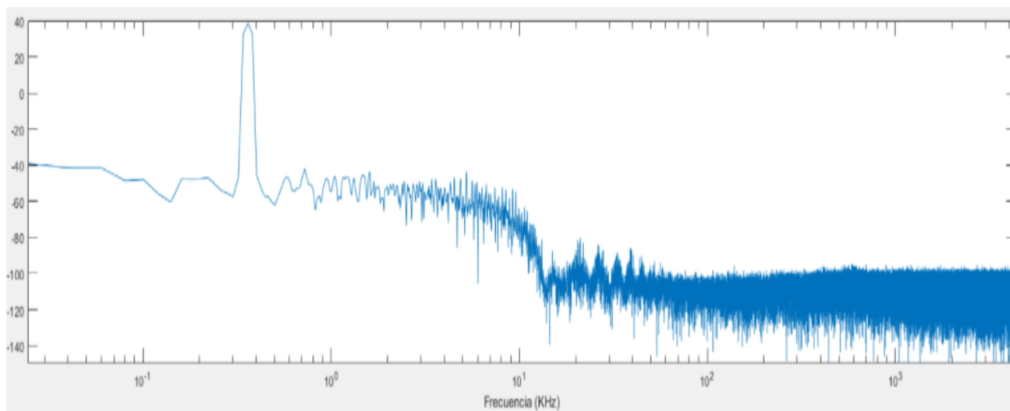

Figura 5.10 Señal espectral salRZ para N_{PWM}=12.

Tabla 5.8 Resultados para $N_{PWM} = 12$.

$N_{PWM} = 12$	SNR(dB)	HD(dB)2	HD3(dB)
salNRZ	74.804	74.429	87.390
salRZ	71.931	78.486	79.900

Para este caso se puede observar, en las gráficas que muestran el espectro de las salidas, que el valor de la frecuencia fundamental se corresponde con el que se mostró en la tabla 5.7, $f_b = 361$ Hz. Además, se puede comprobar que el segundo armónico coincide con el que puede ser calculado analíticamente: $2 \cdot f_c = 722$ Hz.

5.3.5 Señal salNRZ para NPWM=20

Por último, en la siguiente tabla se pueden ver el valor de las frecuencias que definen el comportamiento del convertidor para el caso de $N_{PWM} = 20$:

Tabla 5.9 Frecuencias para $N_{PWM} = 20$.

N_{PWM}	f_b (Hz)	B (Hz)	f_{SDM} (Hz)	f_{PWM} (Hz)
20	217	1203,08	$83{,}3 \cdot 10^3$	$2{,}5 \cdot 10^6$

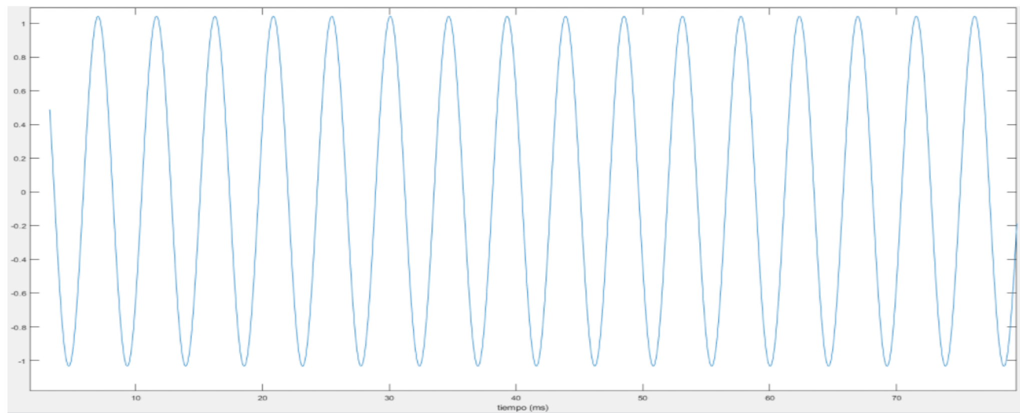

Figura 5.11 Señal salNRZ para N_{PWM}=20 en el dominio del tiempo.

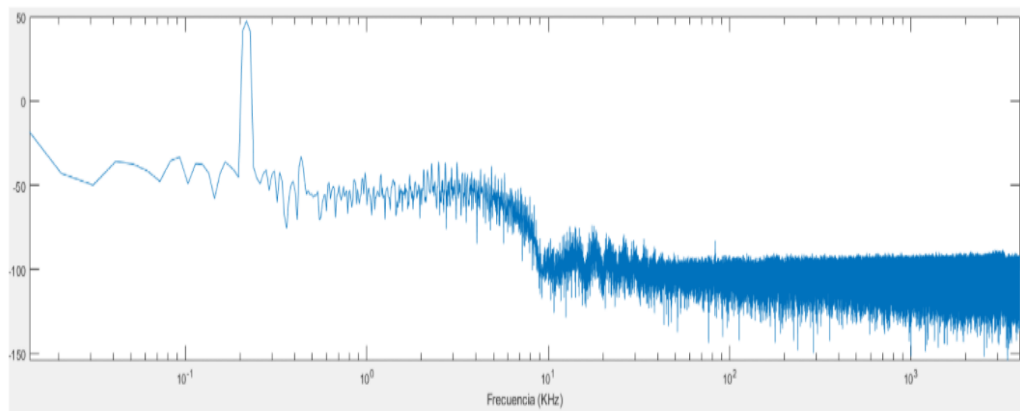

Figura 5.12 Señal espectral salNRZ para N_{PWM}=20.

5.3.6 Señal salRZ para NPWM=20

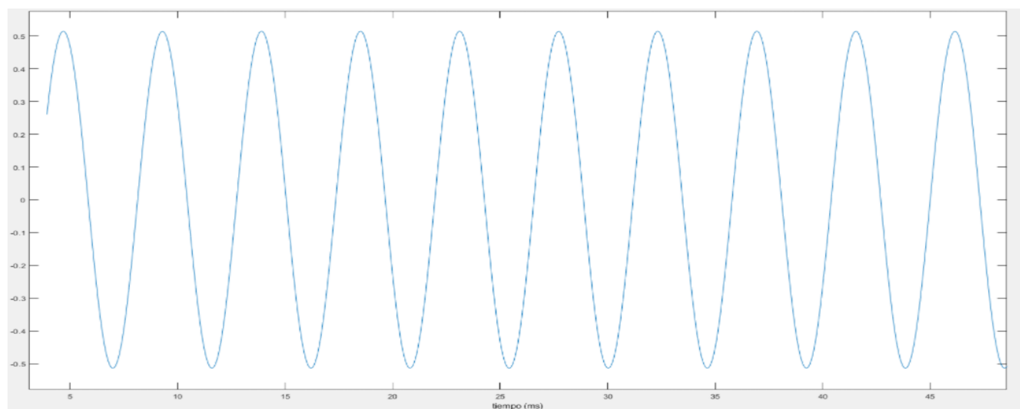

Figura 5.13 Señal salRZ para N_{PWM}=20 en el dominio del tiempo.

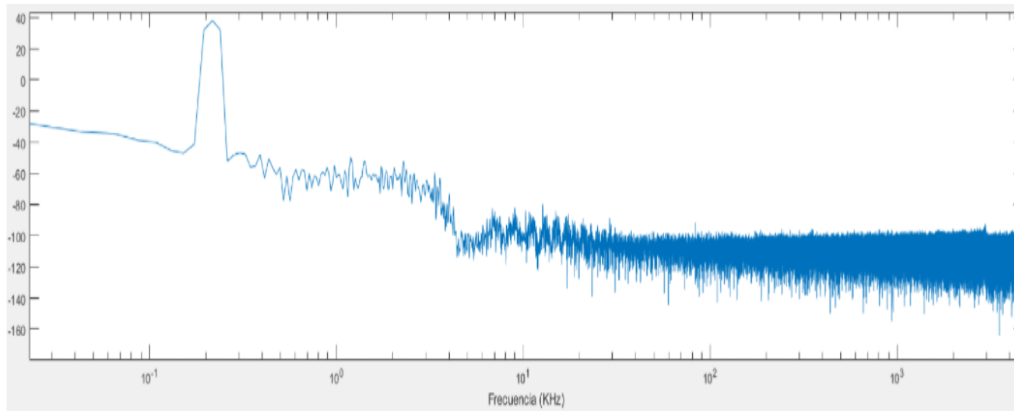

Figura 5.14 Señal espectral salRZ para N_{PWM}=20.

En la siguiente tabla se pueden ver los valores de la SNR, HD2 y HD3 que se han calculado para las salidas cuando N_{PWM}=20:

Tabla 5.10 Resultados para $N_{PWM} = 20$.

$N_{PWM} = 20$	SNR(dB)	HD2(dB)	HD3(dB)
salNRZ	75.265	79.830	91.759
salRZ	74.249	81.125	90.864

Por otro lado, para N_{PWM}=20, la frecuencia fundamental debe estar alrededor de f_b =217 Hz (como se vio en la la tabla 5.9). Si se observan las gráficas anteriores, se puede ver que en este caso la frecuencia fundamental también se corresponde con la calculada analíticamente. Por otra parte, el segundo armónico se debe encontrar en torno a los 450 Hz, lo que coincide con el doble de la frecuencia fundamental. En este caso, se observa que es más complicado identificar el segundo armónico que en casos anteriores, y solo es posible para la gráfica de la salNRZ. Esto es debido a que, como se verá a continuación, la tendencia general de la HD2 es aumentar conforme menor sea la frecuencia de entrada, y además de eso, también tiende a ser más alta para la salRZ que para la salNRZ, lo que explica porque no es visible en la gráfica del espectro de la salRZ.

5.4 SNR, HD2 y HD3

En las siguientes gráficas se puede ver una estimación de como varían los valores máximos de *SNR*, *HD2* y *HD3* según el ancho de banda de la señal (para el caso de la gráfica de SNR) y según la frecuencia de la onda (para el caso de la gráfica de HD2 y HD3). Para realizar estas gráficas, se han necesitado valores de N_{PWM} adicionales a los que se habían tomado hasta ese momento. Dichos valores son los que se indicaron en la tabla del comienzo de este capítulo. Para todos ellos, se realizaron los mismos cálculos que se han explicado para los casos de N_{PWM}=2 ,12 y 20. Los datos que se han calculado para generar estas gráficas se pueden encontrar en el Apéndice D: *Resultados de las simulaciones para los pines B4 de J1 y A6 de J2*.

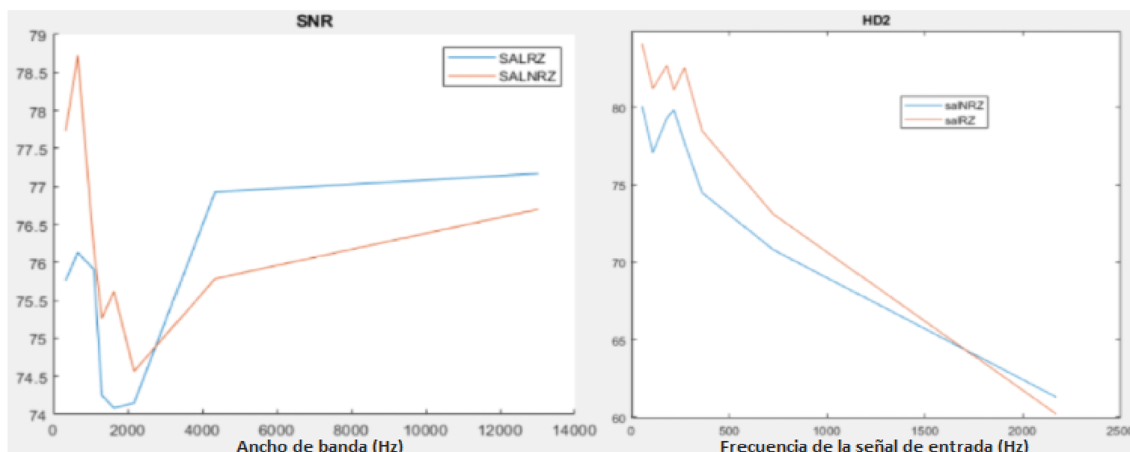

Figura 5.15 Evolución de la SNR y HD2 con la frecuencia.

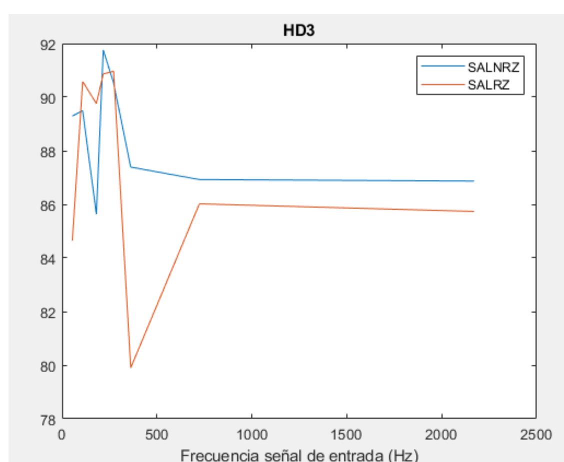

Figura 5.16 Evolución de la HD3 con la frecuencia.

5.4.1 Comportamiento de HD3

Los valores que presenta la *HD3* son más altos en general que los de la *HD2* y la *SNR*, aunque se ha producido una caída de hasta 20 *dB* respecto a la *HD3* obtenida en las simulaciones realizadas en VHDL.

El hecho de que la *HD3* sea mayor a la *HD2* era esperable comprobando las señales espectrales del apartado anterior, ya que el segundo armónico podía apreciarse fácilmente, cosa que no ocurría con el tercero al estar por debajo del fondo de ruido.

Los valores que se han obtenido, al ser mayores que 80 *dB*, indican que la potencia asociada tercer armónico para todos los casos considerados es considerablemente menor a la de la frecuencia principal.

5.4.2 Análisis de HD2: forma de los pulsos

La gráfica del comportamiento de la HD2 frente a la frecuencia de la onda indica que, en general, la señal salRZ presenta una mayor HD2 que la señal salNRZ. Esto es debido a que, como se vio al explicar la justificación de las señales con retorno a cero, el error de salRZ

es menor que el de la señal salNRZ. En las siguientes imágenes se puede apreciar cómo se comportan los pulsos de salida dependiendo si la señal es con y sin retorno a cero. Como

Figura 5.17 Pulsos con retorno a cero.

se aprecia en la gráfica anterior, el error de la señal salRZ se compensa porque siempre es aproximadamente el mismo, por lo que el valor medio de la señal siempre será igual, tal y como se puede ver remarcado en la figura 5.18.

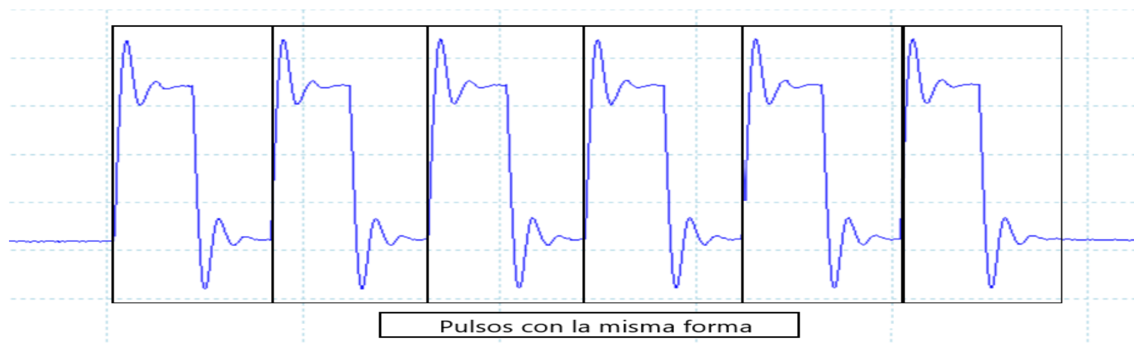

Figura 5.18 Pulsos de la señal salRZ.

Por otro lado, en el caso de las señales con retorno a cero, lo que se obtiene es:

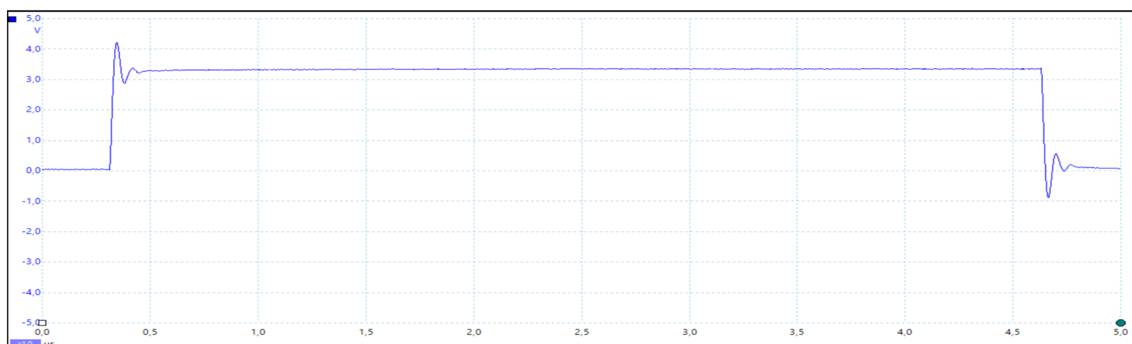

Figura 5.19 Pulsos sin retorno a cero.

Como se ha visto, en este caso el valor medio no es siempre igual, por lo que el error no puede compensarse (tal y como si que podía hacer en el caso de la salRZ). En la siguiente imagen se pueden ver marcados los pulsos que conforman la señal salNRZ, y como el

primero, por presentar el pico de subida, es diferente a los demás:

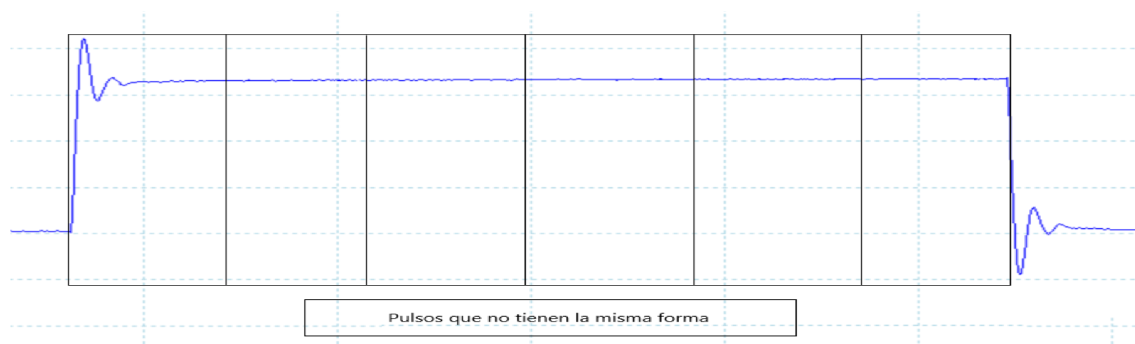

Figura 5.20 Pulsos de la señal salNRZ.

5.4.3 Análisis de la SNR

El comportamiento de la gráfica de la SNR de la figura 5.16 (concretamente la caída que aparece alrededor de los 2000 Hz de ancho de banda de la señal) indica que la forma de obtener los datos no es adecuada. En el caso de que la placa estuviera generando los datos de forma correcta, el valor de la SNR debería aumentar conforme disminuye la frecuencia del seno de entrada, ya que la placa estaría trabajando con menos frecuencias. Además, si se comparan las SNR experimentales con las obtenidas en las simulaciones Post-Implementación que se realizaron en el apartado anterior, se observa que se ha producido una caída considerable (de hasta 10 dB en los casos más extremos) de este valor. Por otra parte, si se recuerda la forma que presentaban los pulsos que se obtienen al comprobar la salida sin filtrar:

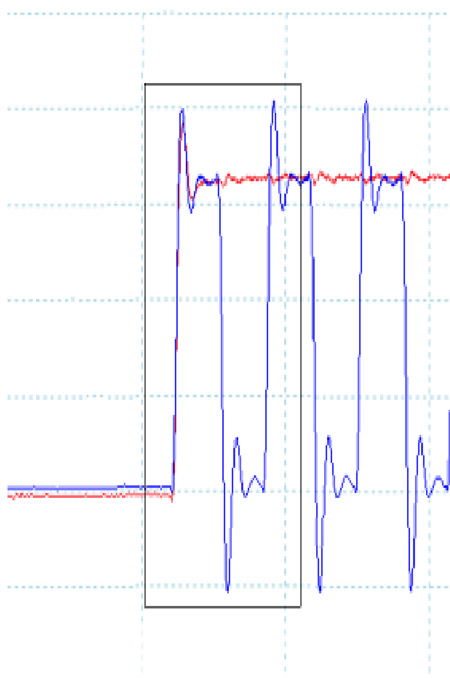

Figura 5.21 Picos de los pulsos.

Los pulsos de la señal, cuando se generaban en las simulaciones en VHDL, tenían forma perfectamente cuadrada. Sin embargo, al generarse en la FPGA, presentan unos picos, tanto en las subidas como en las bajadas, de gran tamaño, lo que explica que el comportamiento de la SNR no sea el adecuado. Este efecto podría deberse a varios factores, como que los equipos no permiten medidas de mayor rango dinámico (por lo que se deberían cuidar mejor las conexiones de la placa de pruebas), que no se haya usado una placa impresa o el hecho de que los pads de salida de la FPGA no fueron diseñados para ser DAC.

Sin embargo, en un primer momento se comprobó si el motivo de este comportamiento defectuoso podía estar relacionado con las características de los pines de salida, por lo que se probó a cambiarlos por los que se ven en la figura 5.22, ya que debían ser más adecuados debido a que, según *Spartan-3E FPGA Starter Kit Board User Guide*[6], alcanzan mayor velocidad de trabajo a la hora de generar la salida:

Figura 5.22 Pines del conector J3.

La forma de estos pines es diferente a la que tenían los que pertenecen a los conectores J1 y J2, por lo que la manera de obtener datos de ellos también lo es. Mientras que la conexión de la placa con el filtro antes se realizaba introduciendo el extremo del cable conector macho en el pin de salida, en este caso se necesita acoplar una pieza intermedia entre el cable conector y el pin de la FPGA, tal y como se aprecia en la figura 5.23.

Se ha tenido que definir un nuevo *Archivo de constraints*, en el que como ya se sabe se fijan las características de los pines asociados a las señales de entrada y a la de salida. Como se observa en la siguiente imagen, las de *clk* y *resasin* no se han modificado; solo se han cambiado las características de la salida *salNRZ*.

Además, se debe crear otro archivo para el caso del programa que produce la señal *salRZ* (se recuerda que el programa que se realizó originalmente tuvo que duplicarse para evitar el solapamiento que aparecía entre las salidas debido a que se generaban simultáneamente).

Una vez generados ambos, se ha procedido a realizar las mediciones y análisis de la

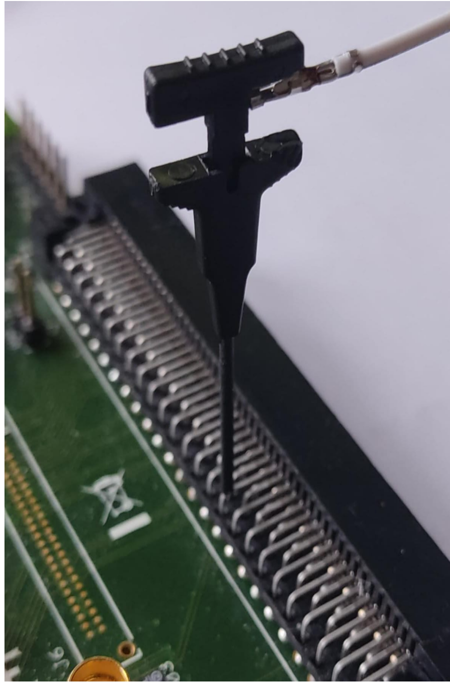

Figura 5.23 Pieza que permite conexión con J3.

clk	Input		C9	☑	0	LVCMOS33*	3.300				NONE
resasin	Input		D18	☑	1	LVTTL*	3.300				PULLDOWN*
SALNRZ	Output		A4	☑	0	LVCMOS33*	3.300		8*	FAST*	NONE

Figura 5.24 Pines asociados a *clk*, *resasin* y salNRZ.

misma forma que se hizo con los resultados que se han obtenido ahora. Sin embargo, los nuevos valores de la *SNR*, *HD2* y *HD3* han resultado ser menores que los obtenidos anteriormente, por lo que se ha concluido que lo que limita la calidad de la salida no es la velocidad a la que trabajan los pines, sino a la que se comporta la propia FPGA. Por lo tanto, se considera que una posible solución a este problema sería recurrir a una FPGA más rápida o utilizar un DAC de un bit a la salida, lo cual sale de los límites de este proyecto.

6 Conclusiones y posibles mejoras

Este capítulo recoge las conclusiones que se pueden sacar de este proyecto y de los distintos pasos que lo han conformado, además de posibles mejoras que se le podrían aplicar en el futuro.

Como se sabe, a lo largo de este trabajo se han usado varios programas que han permitido realizar diferentes simulaciones, progresivamente más *complejas* pasando del *Matlab* y del *Simulink* al lenguaje VHDL. Este proceso se corresponde con la metodología de diseño top-down, que consiste en aproximar funcional y estructuralmente un sistema a su realización final disponiendo de herramientas que permiten comprobar la bondad y validad de los resultados experimentales que finalmente se obtienen.

Aun así, tal y como se ha visto ejemplificado en este proyecto, las simulaciones intermedias al fin y al cabo no pueden tener en cuenta todos los factores que envuelven a la toma de datos experimental.

Gracias a que se parametrizaron los códigos, se han podido realizar simulaciones y medidas a multitud de señales de entrada distintas, aunque esto también ha sido posible gracias a la adaptabilidad y capacidad de reprogramación que presentan las FPGA.

Aunque no se han logrado obtener los valores de SNR, HD2 y HD3 de las simulaciones, estos siempre han sido relativamente altos, lo que se ha visto reflejado en el capítulo anterior en que las salidas, en el dominio del tiempo, siempre presentaban la forma sinusoidal de la entrada, lo que se ve reflejado en que, si bien la SNR no ha alcanzado los aproximadamente 85 dB de las últimas simulaciones, también es cierto que para ningún valor de N_{PWM} ha sido más pequeña de 74 dB. Además, se ha logrardo que el tercer armónico no se observara al estar por debajo del fondo de ruido. Estos resultados experimentales concuerdan con los que se obtuvieron en un TFG anterior [8], en el que se usó una FPGA más rápida para implementar un SDM de sólo un bit de salida y sin interfaz PWM.

Al contrario que en este trabajo, en ese proyecto se obtuvieron mejoras más significativas en el HD2 cuando se usaban pulsos RZ (con retorno a cero) en comparación con los NRZ. Este hecho puede estar justificado porque la señal PWM lleva en cierto modo un retorno a zero (Fig. 5.19 y Fig. 5.20), y por lo tanto no cabría esperar tanta mejora del HD2 al usar la señal con RZ.

Respecto a las posibles mejores que se podrían realizar al proyecto, la primera de ellas sería emplear una placa impresa, puesto que el problema de que la SNR y HD2 no sean lo suficientemente altas debe residir en la forma con la que se generan los pulsos ya que, como se ha visto en el trabajo, todas las simulaciones previas a este paso presentaban buenos resultados. Para solucionar el problema de que los pulsos no se están generando de la forma adecuada, se podría recurrir a un DAC de 1 bit activado por la salidas salRZ y salNRZ.

Más allá de la calidad de los resultados, otro aspecto del trabajo que podría mejorarse es el grado de parametrización de los códigos: aunque es el suficientemente alto como para que se pueda cambiar la frecuencia del seno de entrada y de trabajo con solo modificar el valor de la N_{PWM}, se podrían haber parametrizado más señales, o concretamente sus tamaños (es decir, el número de bits máximo que necesitan). La mayoría de las señales intermedias tiene un tamaño fijo que no depende de la frecuencia del seno de entrada, pero si de su amplitud. Estos valores deberían cambiarse, uno a uno, en el caso de que la amplitud máxima del seno de entrada fuera muy diferente a la que se ha tomado en este trabajo (ya que, en este caso, se deberían cambiar el paso que define al cuantificador del SDM en *Simulink*), por lo que sería conveniente realizar esta mejora ya que permitiría simular y analizar el comportamiento de este convertidor al trabajar con entradas senoidales de diferente amplitud.

También podría realizarse una optimización del Hardware sustituyendo las estructuras del convertidor que se han descrito mediante una *Look-Up Table* (en la red de corrección del PWM y a la hora de comparar la salida del *SDM* con la señal de referencia triangular) por las operaciones de las que se componían estas estructuras originalmente, evitando así el uso de *Look-Up Table* y tratando así de usar menos recursos Hardware.

Apéndice A
Códigos de MATLAB

En este apartado se presentan los códigos de Matlab que han sido necesarios para llevar a cabo este proyecto. Éstos se han separado según su funcionalidad.

A.1 Códigos previos a los análisis de la descripción en VHDL

A.1.1 Analizador modelos Simulink

Este código fue facilitado junto con los modelos en *Simulink* de los diferentes convertidores. Este código permite calcular, entre otros valores importantes, la SNR ; y define los parámetros de los que dependerán las simulaciones de los convertidores, como la amplitud de la señal de entrada, el período de las señales triangulares de referencia o el valor de los saturadores que aparecían en ellos.

Código A.1 generadorentrada.

```
%% NOTA: EL NUMERO DE NIVELES (NL) TIENE QUE SER PAR PARA QUE EL
%% CUANTIFICADOR SEA MID-RISE

clear all;

%%PARAMETROS
Amp=10^(-3/20); %%Amplitud inicial
Nptos=1024*16*1; %%Número de puntos de la transformada
Ndemas=10;
Neje=Nptos+Ndemas; %Número de muestras de ejecución

NL=16; % Numero de niveles del cuantificador del SDM. Tiene que ser
    par
F=NL-1;
paso=2/(NL-1);
paso96=paso*96;
Umbrales_1T=[ (F-1)/2:-1:-(F-1)/2 -(F-1)/2:1:(F-1)/2 ]*paso;
```

```
Umbrales_2T=[-(F-1)/2:1:(F-1)/2 (F-1)/2:-1:-(F-1)/2 ]*paso;
Umbrales=  [-(F-1)/2:1:(F-1)/2  (F-1)/2:-1:-(F-1)/2 ]*paso;
Lm=2;



FD=32;
OSR=FD;
fp=361.7;  %Frecuencia de señal de entrada
Bw=6*fp;  %Relación ancho de banda- frecuencai señal de entrada
fs=FD*(2*Bw); %% Muestreo de la señal de salida para PSD
ts=1/fs;

fsi=fs;     %% Muestreo de la señal de entrada
tsi=1/fsi;    %% Perioro triangulo_2T

fos=2*F*fsi;  %% Generacion digital PWM
tos=1/fos;




Nfs=round(Nptos*fp/fs); Nbw=round(Nptos*Bw/fs);
fp=Nfs*fs/Nptos;


%% Amplitud distorsion del PWM (k>1)
Ak=@(A,k) (pi*k*fp*ts)^(k-1)*A^k/2^(k-1)/factorial(k);


%% Filtro Butterworth

[Nbutter, Dbutter]=butter(9,2*pi*Bw,'s');
ventana=window(@hann,Nptos);

%% AJUSTAMOS EL STEP DEL CUANTIFICADOR PARA QUE SU SALIDA (pasoQ/2)
     SEA
%  UN NUMERO ENTERO Y PARA QUE LA SALIDA DE LA GANANCIA (satu_Nb
    /96)
%  TAMBIEN LO SEA. CONDICION: (pasoQ/2) multiplo de 4x3
Nb=14; %% Numeros de bits de señal a convertir
paso_Nb=2^(Nb-1)*paso; % Paso ideal
pasoQ=24*round(paso_Nb/24);
satuQ=pasoQ/2 + (NL/2 - 1)*pasoQ;
satu_ideal=2^(Nb-1);
satu_Nb=96*round ( 0.5 + 2^(Nb-1) / 96);
```

```
fprintf(1,'Frecuencias (Hz): fp=%f Bw=%f fsi=%f fs=%f fos=%f\n',fp,
   Bw,fsi,fs,fos);
fprintf(1,'Fact: diezm.: %d \nÍndices de frecuencias: Nfs=%d Nbw=%d
    \n',FD,Nfs,Nbw);
fprintf(1,'paso_ideal=%f pasoQ=%f satu_ideal=%f satu_Nb=%f satuQ=%f
   \n',paso_Nb,pasoQ,satu_ideal,satu_Nb,satuQ);





   sim('sdminteger.slx');
   %%Calculo transformada
   t1=y1(Ndemas:Neje-1);
%    clear y1;
   t2=y2(Ndemas:Neje-1);
%    clear y2;
   t3=y3(Ndemas:Neje-1);
   %clear y3;
   t4=y4(Ndemas:Neje-1);
%    clear y4;
   t5=y5(Ndemas:Neje-1);

%    Lt=length(t1);
%    fprintf(1,'Longitud de FFT: %d ',Lt);

   [tt1,freq]=pwelch(t1,ventana,Nptos-1,Nptos,fs);
   [tt2,freq]=pwelch(t2,ventana,Nptos-1,Nptos,fs);
   [tt3,freq]=pwelch(t3,ventana,Nptos-1,Nptos,fs);
   [tt4,freq]=pwelch(t4,ventana,Nptos-1,Nptos,fs);
   [tt5,freq]=pwelch(t5,ventana,Nptos-1,Nptos,fs);

   % Escalamos para que el pico sea 20*log10(Amp)
   f_esc=(2*fs*sum(tt2(Nfs-2:Nfs+4))/tt2(Nfs+1)/Nptos)^(+1);
   tt1=f_esc*tt1;
   tt2=f_esc*tt2;
   tt3=f_esc*tt3;
   tt4=f_esc*tt4;
   tt5=f_esc*tt5;

   %Cálculo de la relación señal/ruido
   senal1=0; senal2=0; senal3=0; senal4=0; senal5=0;
    senal1=sum(tt1(Nfs-2:Nfs+4)); %senal1+tt1(i); tt1(i)=0;
    senal2=sum(tt2(Nfs-2:Nfs+4));
    senal3=sum(tt3(Nfs-2:Nfs+4)); %senal1+tt1(i); tt1(i)=0;
    senal4=sum(tt4(Nfs-2:Nfs+4));
    senal5=sum(tt5(Nfs-2:Nfs+4));
```

```
    snr1=10*log10(senal1/(sum(tt1(5:Nbw))-senal1));
    snr2=10*log10(senal2/(sum(tt2(5:Nbw))-senal2));
    snr3=10*log10(senal3/(sum(tt3(5:Nbw))-senal3));
    snr4=10*log10(senal4/(sum(tt4(5:Nbw))-senal4));
    snr5=10*log10(senal5/(sum(tt5(5:Nbw))-senal5));

    INB=10*log10( paso^2 * pi^(2*Lm)/12/(2*Lm+1)/OSR^(2*Lm+1) );
    SNRt= 10*log10(Amp^2/2) - INB;
    SFDR2=20*log10( Amp / Ak(Amp,2) );
    SFDR3=20*log10( Amp / Ak(Amp,3) );




fprintf(1,'%f %f %f %f %f SNRt=%f SFDR: (2)=%f (3)=%f\n',snr1,snr2,
    snr3,snr4,snr5,SNRt,SFDR2,SFDR3);

%% REPRESENTACION GRAFICA
figure


subplot(2,1,1), semilogx(freq,10*log10(tt1));
grid on
ylabel('Y1');
xlabel('Frequency (Hz)')

subplot(2,1,2), semilogx(freq,10*log10(tt2));
grid on;
ylabel('Y2');
xlabel('Frequency (Hz)')

figure
subplot(2,1,1), semilogx(freq,10*log10(tt3));
grid on;
ylabel('Y3');
subplot(2,1,2), semilogx(freq,10*log10(tt4));
grid on;
ylabel('Y4');

figure
subplot(2,1,1), semilogx(freq,10*log10(tt5));
grid on;
ylabel('Y5');
subplot(2,1,2), semilogx(freq,10*log10(tt5));
grid on;
```

```
ylabel('Y5');

%% REALIZAMOS TEST SI LAS VARIABLES INTERNAS DEL SDM SON NUMEROS
   ENTEROS
NO_I=1;
for ind=1:1:length(y_I),
    frac=abs( y_I(ind) - round(y_I(ind)) );
    if(frac > 1e-6)
        NO_I=0;
        break;
    end
end

if NO_I==0,
    fprintf(1,'HAY REALES NO ENTEROS\n');
else
    fprintf(1,'HAY SOLO ENTEROS\n');
end
```

A.1.2 Generador de la entrada

Este código ha permitido generar en VHDL los valores de entrada al modulador Sigma/Delta, codificados en CA2. El resultado de ejecutar este programa es el conjunto de líneas de código que se podrán ver en el apartado **Código B.5**: valores.

Código A.2 generadorentrada.

```
fclk=10^8;% Frecuencia reloj master
Nbits= 14;   % Numero de bits de la señal de entrada
N=1;%Factor por el que se divide la frecuencia del SDM
fsdm=fclk/N; % Frecuencia del SDM
tsdm=1/fsdm;
OSR=32;     % Oversampling ratio
Narm=6;     % Numero de armónicos que caben en la banda de señal
fb=fsdm/(Narm*2*OSR);
Nmuestras=2*OSR*Narm; % Numero de muestras de un periodo de señal
   (1/fb)
Nb_address=floor( log2(Nmuestras) ) + 1; %% Numero de bits de la
   direccion de la look-up table
t=[0:1:Nmuestras-1].*tsdm;
A= 2^(Nbits-1) ;%Amplitud de la señal de entrada
x=round( A*sin(pi*2*fb*(t) + pi/2 ) );
plot(t,x)
for j=1:Nmuestras

    if x(j)<0
        y=2^Nbits+x(j);
```

```
    else
        y=x(j);
    end

    if j==1 %primera fila de la tabla
        fprintf(1,'\nsalida<="%s" when n="%s" else\n',dec2bin(y,14),
            dec2bin(j-1,Nb_address));
    elseif j==Nmuestras %ultima fila de la tabla
        fprintf('"%s";\n',dec2bin(y,14));
    else %filas intermedias
    fprintf('"%s" when n="%s" else\n',dec2bin(y,14),dec2bin(j-1,
        Nb_address));
    end
end
```

A.2 Códigos para analizar resultados de VHDL

A.2.1 Análisis entrada y salida SDM

Este código permite calcular tanto la SNR como la HD2 de los datos de entrada y salida del modulador Sigma/Delta. También permite representar estos datos tanto en el dominio del tiempo como de la frecuencia que se han podido ver en este trabajo.

Se debe recordar que tanto la entrada como la salida del SDM estarán codificadas en CA2, por lo que estos valores, al pasarlos a decimal, se deberá restar la cantidad de 2^n, siendo n el número de bits, a todos los valores que en decimal sean iguales o superen la cantidad denominada *umbralsd* en el código ($2^{(n-1)}$) para que así los números negativos pasen a tener su valor correcto en decimal:

Código A.3 Análisis entrada y salida SDM.

```
clear all
OSR=32;
Nbsd=14;
umbralsd=2^(Nbsd-1);
Nptsd=384;


%% Datos SDM
load("vsimreal4.txt")
x=vsimreal4(:,1);%entrada del modulador sigma delta
y=vsimreal4(:,2);%salida del modulador sigma delta
Nptos=length(x);
```

```
for k=1:Nptos
    if x(k)>=umbralsd,
        x(k)=x(k)-2^Nbsd;
    end
end
for k=1:Nptos
    if y(k)>=2*umbralsd,
        y(k)=y(k)-2^(Nbsd+1);
    end
end

Nptos=Nptsd*floor(Nptos/Nptsd);
Nbw=round(Nptos/2/OSR);

plot(y);
ventana=window(@hann,Nptos);
N=2;
fclk=50*10^6;
fs=fclk/(30*N)
ts=1/fs;
 [PSX,freq]=pwelch(x,ventana,Nptos-1,Nptos,fs);
 [PSY,freq]=pwelch(y,ventana,Nptos-1,Nptos,fs);
 [PSYmax,Nfs]=max(PSY);

 %% Potencias de ruido y señal
 Psenal=sum(PSY(Nfs-5:Nfs+5));
  harm2=sum(PSY(2*Nfs-5:2*Nfs+5));
 Pruido=sum(PSY(3:Nbw))- Psenal;

 figure
 semilogx(freq,10*log10(PSY),'g');
grid on;
ylabel('PSD (entrada y sdm)');
xlabel('Frequency (Hz)');

fprintf(1,'Nptos=%d Nbw=%d Nfs=%d\n',Nptos,Nbw,Nfs);
fprintf(1,'SNR = %f dB\n',10*log10(Psenal/Pruido));
HD2=10*log10(Psenal/harm2);
```

A.2.2 Análisis salida del PWM

Al igual que en el caso anterior, el siguiente código permite calcular los valores de la SNR y HD2 de la salida, en este caso del PWM. Ya que esta es de un solo bit, no se necesita hacer la operación del apartado anterior de convertir los números negativos en CA2 al pasarlos a decinaml, pero si se deberá tener en cuenta que la salida del modulador PWM deberá modificarse ligeramente antes de analizarse: las salidas con valor '1'tendrán el mismo valor,

mientras que las salidas que valgan ‘0’deberán valer ahora -1 tal y como opcurría en la salida del diseño de este modulador en Simulink. Para ello, bastará con duplicar el valor de todos los datos (los que valgan ‘0’tendrán el mismo valor mientras que los que valgan ‘1’pasarán a valer ‘2’) y restarle una unidad a todos ellos. De esta forma, se consigue que la salida sea de la misma forma que la que se obtenía de Simulink (el valor más alto ‘1’y el más bajo ‘-1’).

Código A.4 Análisis salida PWM.

```
load("simtrasvhdl.txt")
N=2;
fclk=50*10^6;
fs=fclk/(30*N)
ts=1/fs;

tpwm=ts/30; %% Ranura temporal del PWM
NT_PWM=length(simtrasvhdl); %% Numero total de puntos PWM
tiempo=tpwm*[0:1:NT_PWM-1]';
x=simtrasvhdl(:,2);
x_PWM=[tiempo 2*x-1]; %Entrada de simulink

[Nbutter, Dbutter]=butter(3,2*pi*fs/(OSR*2),'s');

sim('PWM2LP'); %% Filtra la señal PWM en SIMULINK

load('sal_pwm.mat');
yf=salida(2,:);

Np_yf=length(yf);
Np_yf=Nptsd*floor(Np_yf/Nptsd);
Nbw_yf=round(Np_yf/2/OSR);

ventana=window(@hann,Np_yf);
[PS_yf,freq]=pwelch(yf,ventana,Np_yf-1,Np_yf,fs);
[PSYmax,Nfs_yf]=max(PS_yf);

% Potencias de ruido y señal
 P_senal_yf=sum(PS_yf(Nfs_yf-2:Nfs_yf+2));
 P_ruido_yf=sum(PS_yf(3:Nbw_yf))-P_senal_yf;
fprintf(1,'SNR PWM= %f dB\n',10*log10(P_senal_yf/P_ruido_yf));
harm2=sum(PS_yf(2*Nfs_yf-3:2*Nfs_yf+3));
HD2=10*log10(P_senal_yf/harm2)

figure
semilogx(freq,10*log10(PS_yf),'g');
grid on;
```

```
ylabel('PSD (PWM)');
xlabel('Frequency (Hz)');
```

A.3 Códigos para analizar resultados de la FPGA

A.3.1 Análisis salida FPGA

Este código permite, a partir de los valores de la salida de la FPGA, calcular cuanto vale la SNR, HD2 y HD3 de los resultados experimentales. Como se observa, muchos de los valores dependerán del parámetro N_{PWM}, por lo que será necesario especificarlo para que los cálculos sean correctos. Además, este código representa también la señal de salida tanto en el dominio del tiempo como de la frecuencia.

Código A.5 Análisis salida FPGA.

```
clear all

load('prb.txt');
y=prb(:,2); %datos salida FPGA
t=prb(:,1); %tiempo salida FPGA


plot(y);
xlabel('Numero de muestra');
title('Antes de recortar');

%% PARAMETROS
Nsample=length(y);
OSR=32;
Narm=6;
Ndiv=30;
Npwm=; %Valor de NPWM
segundos=0;
if segundos==1
   ts=(t(2)-t(1))
else
   ts=(t(2)-t(1))*1e-3; % Periodo de muestreo
end
fs=1/ts;
fp=50e6/Ndiv/Npwm/2/OSR/Narm; %% Frecuencia de senal
Tp=1/fp;

Np1Tp=fs/fp;          %% Numero de puntos en un periodo de senal
Nptos=round(Np1Tp*floor(Nsample/Np1Tp));
```

```
figure
plot(t(Nsample-Nptos+1:Nsample),y(Nsample-Nptos+1:Nsample));
xlabel('tiempo (ms)');
title('Recortada');

fprintf(1,'Nsample=%d Nptos=%d Np1Tp=%f fs=%f (MHz) fp=%f (KHz)\n',
   Nsample,Nptos,Np1Tp,fs*1e-6,fp*1e-3);
%%CALCULAR SNR

%% Calulo de la PSD y de la SNR
ventana=window(@hann,Nptos);
psd=pwelch(y(Nsample-Nptos+1:Nsample),ventana,Nptos-1,Nptos);

[Py_max,ind_fpp]=max(psd);
fpp=ind_fpp*fs/Nptos;
fprintf(1,'Frecuencia señal; medida: % f y teorica: %f (KHz)\n',fpp
   *1e-3,fp*1e-3);

Lpsd=length(psd);

figure
freq=[0:1:Lpsd-1]*fs/2/Lpsd;
subplot(2,1,1), semilogx(10*log10(psd));
xlabel('Indice de frecuencia');
subplot(2,1,2), semilogx(1e-3*freq,10*log10(psd));
xlabel('Frecuencia (KHz)');

%% PARAMETROS A DAR VALOR
Nfs=round(Nptos*fp/fs);
Nbw=6*Nfs;
senal=sum(psd(Nfs-5:Nfs+5));
harm2=sum(psd(2*Nfs-5:2*Nfs+5));
harm3=sum(psd(3*Nfs-5:3*Nfs+5));
ruido=sum(psd(4:Nbw))-senal;
%%%%%%

snr=10*log10(senal/ruido);
HD2=10*log10(senal/harm2);
HD3=10*log10(senal/harm3);

fprintf(1,'SNR (dB): %f HD2: %f (dB) HD3: %f (dB)\n',snr,HD2,HD3);
```

Apéndice B
Códigos VHDL

En este apéndice se pueden observar los códigos VHDL que se han usado a lo largo de este proyecto para que describan el comportamiento del convertidor. Se han separado según el bloque del modulador al que corresponden, para hacer más más sencilla su compresión:

B.1 Bloque TOP

Este código es el que recoge todos los demás que se verán en los siguientes apartados, además de definir las señales que los relacionan. Se observa que, ya que es el programa al que se le harán las simulaciones, sus entradas son el reloj del sistema y el reset (*clk* y *resasin*) y sus salidas las mismas que las que tiene la FPGA: *salRZ*y *salNRZ*. En este programa, además, se especifica el valor de los parámetros de los que dependen el resto de señales y constantes del programa, y se especifica el valor de la N_{PWM}.

Código B.1 top.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
entity top is
generic (Nbxt: integer:=14;
        Nbparan: integer:=9;
      Ncont:integer:=8;
      NcontPWM:integer:=4);
    Port (clk : in STD_LOGIC;
       resasin:in STD_LOGIC;
       SALNRZ: out STD_LOGIC;
       SALRZ: out STD_LOGIC
       );
end top;

architecture Behavioral of top is
  COMPONENT contn
```

```
  generic (Nn: integer );
  PORT(
    entrada : IN std_logic_vector(Nn-1 downto 0);
    resasin : IN std_logic;
    pulso : IN std_logic;
      clk : IN std_logic;
    n : OUT std_logic_vector(Nn-1 downto 0)
    );
  END COMPONENT;



COMPONENT valores
generic (Nbxt: integer );
  PORT(
    n : IN std_logic_vector(8 downto 0);
    salida : OUT std_logic_vector(Nbxt-1 downto 0)
    );
  END COMPONENT;
  COMPONENT divisorfrec
  generic (N: integer );
  PORT(
    entrada : IN std_logic_vector(N-1 downto 0);
    clk : IN std_logic;
    resasin : IN std_logic;
    pulso : OUT std_logic
    );
  END COMPONENT;
COMPONENT sdm
   generic (Nb: integer);
  PORT(
    xt : IN std_logic_vector(Nb-1 downto 0);
    clk : IN std_logic;
    resasin : IN std_logic;
    pulsosd : IN std_logic;
    yn : OUT std_logic_vector(Nb downto 0)
    );
  END COMPONENT;
COMPONENT PWM
generic (Nb: integer);
  PORT(
    ySMN : IN std_logic_vector(Nb downto 0);
    clk : IN std_logic;
    pulsosd : IN std_logic;
    resasin : IN std_logic;
    pulsoPWM : IN std_logic;
    salidaPWM : OUT std_logic
```

```
    );
  END COMPONENT;

  COMPONENT salidaD
  PORT(
    yPWM : IN std_logic;
    pulsoPWM : IN std_logic;
    clk : IN std_logic;
    resasin : IN std_logic;
    salRZ : OUT std_logic;--;
    salNRZ : OUT std_logic
    );



  END COMPONENT;


  COMPONENT genpulRZ
  generic (N: integer );
  PORT(
    entrada : IN std_logic_vector(N-1 downto 0);
    resasin : IN std_logic;
    pulso : out std_logic;
      clk : IN std_logic
    );
  END COMPONENT;



  signal pulsosd,pulsoPWM,salidaPWM,salRZsin,salNRZsin,Q,D,pulsoRZ,
      A,B : STD_LOGIC:='0';

--  constant contsd : STD_LOGIC_VECTOR (Ncont-1 downto 0)
    :="111100";---para Npwm=2
--  constant contpwm : STD_LOGIC_VECTOR (NcontPWM-1 downto 0)
    :="10";---para Npwm=2
--  constant contsd : STD_LOGIC_VECTOR (Ncont-1 downto 0)
    :="10110100";---para Npwm=6
--  constant contpwm : STD_LOGIC_VECTOR (NcontPWM-1 downto 0)
    :="110";---para Npwm=6
  constant contsd : STD_LOGIC_VECTOR (Ncont-1 downto 0):="11110000"
      ;---para Npwm=8
  constant contpwm : STD_LOGIC_VECTOR (NcontPWM-1 downto 0):="1000"
      ;---para Npwm=8
--  constant contsd : STD_LOGIC_VECTOR (Ncont-1 downto 0)
    :="111100000";---para Npwm=16
```

```
-- constant contpwm : STD_LOGIC_VECTOR (NcontPWM-1 downto 0)
   :="10000";---para Npwm=16
-- constant contsd : STD_LOGIC_VECTOR (Ncont-1 downto 0)
   :="1001011000";---para Npwm=20
-- constant contpwm : STD_LOGIC_VECTOR (NcontPWM-1 downto 0)
   :="10100";---para Npwm=20
-- constant contsd : STD_LOGIC_VECTOR (Ncont-1 downto 0)
   :="1011010000";---para Npwm=24
-- constant contpwm : STD_LOGIC_VECTOR (NcontPWM-1 downto 0)
   :="11000";---para Npwm=24
-- constant contsd : STD_LOGIC_VECTOR (Ncont-1 downto 0)
   :="10010110000";---para Npwm=40
-- constant contpwm : STD_LOGIC_VECTOR (NcontPWM-1 downto 0)
   :="101000" ;---para Npwm=40
-- constant contsd : STD_LOGIC_VECTOR (Ncont-1 downto 0)
   :="100101100000";---para Npwm=80
-- constant contpwm : STD_LOGIC_VECTOR (NcontPWM-1 downto 0)
   :="1010000" ;---para Npwm=80
-- constant contsd : STD_LOGIC_VECTOR (Ncont-1 downto 0)
   :="1001011000000";---para Npwm=160
-- constant contpwm : STD_LOGIC_VECTOR (NcontPWM-1 downto 0)
   :="10100000" ;---para Npwm=160


  constant maxn: STD_LOGIC_VECTOR (Nbparan-1 downto 0):="101111111"
     ;
  signal n: STD_LOGIC_VECTOR (Nbparan-1 downto 0):="000000101";
  signal valoraxt : STD_LOGIC_VECTOR (Nbxt-1 downto 0):=(others
     =>'0');
  signal yn : STD_LOGIC_VECTOR (Nbxt downto 0);
begin

--------------------------------Bloque 1---------------------




Inst_divisorfrec: divisorfrec generic map (N=>Ncont) PORT MAP(
    entrada =>contsd ,
    clk =>clk ,
    resasin =>resasin ,
    pulso => pulsosd
  );

DFdelPWM: divisorfrec generic map (N=>NcontPWM) PORT MAP(
    entrada =>contpwm ,
    clk =>clk ,
    resasin => resasin ,
```

```
    pulso =>pulsoPWM
  );

Inst_contn: contn generic map (Nn=>Nbparan) PORT MAP(
    entrada =>maxn ,
    resasin =>resasin ,
    clk=>clk,
    n =>n ,
    pulso =>pulsosd
  );


Inst_genpulRZ: genpulRZ generic map (N=>NcontPWM) PORT MAP(
    entrada =>contpwm,
    resasin =>resasin ,
    clk=>clk,
    pulso =>pulsoRZ
  );


Inst_valores: valores generic map (Nbxt=>Nbxt) PORT MAP(
    n =>n ,
    salida =>valoraxt
  );



--------------------------------Bloque 2---------------------


Inst_sdm: sdm generic map (Nb=>Nbxt) PORT MAP(
    xt =>valoraxt ,
    clk =>clk ,
    resasin =>resasin ,
    pulsosd =>pulsosd ,
    yn =>yn
  );

--------------------------------Bloque 3---------------------

Inst_PWM: PWM generic map (Nb=>Nbxt) PORT MAP(
    ySMN =>yn ,
    clk =>clk ,
    resasin =>resasin ,
    pulsosd =>pulsoSD ,
    pulsoPWM =>pulsoPWM ,
    salidaPWM => salidaPWM
```

```
  );


-------------------------------Bloque 4--------------------
Inst_salidaD: salidaD PORT MAP(
    yPWM =>salidaPWM ,
    pulsoPWM =>pulsoRZ,
    salNRZ =>salNRZsin ,
    salRZ =>salRZsin ,
    clk =>clk ,
    resasin => resasin
    );




B<= salNRZsin;
D<= salRZsin;
process(resasin,clk)
  begin
    if (resasin='1') then
    Q <= '0';
    A <= '0';
    elsif (clk'event and clk='1') then
    Q <= D;
    A <= B;
    end if;
  end process;
  salRZ<=Q ;
  salNRZ<=A;



end Behavioral;
```

B.2 Bloque 1

Se recuerda que los programas que forman este bloque son el divisor de frecuencias, que genera tanto el *pulsoPWM* como el *pulsoSD* , el generador del *pulsoRZ*, el contador de n y el programa *valores* con el que se le asigna el valor correspondiente a la entrada al bloque 2.

Código B.2 divisorfrec.

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

entity divisorfrec is
generic( N : integer:=5);
    Port ( entrada : in STD_LOGIC_vector(N-1 downto 0);
           clk : in STD_LOGIC;
           resasin : in STD_LOGIC;
           pulso : out STD_LOGIC);
end divisorfrec;

architecture Behavioral of divisorfrec is
 constant uno: STD_LOGIC_vector(N-1 downto 0) := (0=>'1',others
    =>'0');
 signal cont,comp : STD_LOGIC_vector(N-1 downto 0) := (others=>'0')
    ;
begin

process (clk,resasin)
begin

   if resasin='1' then
      cont<=(0=>'0',others=>'0');
   elsif (clk'event and clk='1') then
      cont<=comp;
    end if;

end process;

pulso <= '1' when entrada-1=cont else
          '0';

     comp <= (0=>'0',others=>'0') when entrada-1=cont else
            cont+uno;

end Behavioral;
```

Código B.3 contn.

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

```
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;


-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity contn is
generic( Nn : integer:=5);
    Port ( entrada : in STD_LOGIC_vector(Nn-1 downto 0);
           resasin : in STD_LOGIC;
           n : out STD_LOGIC_vector(Nn-1 downto 0);
           pulso,clk : in STD_LOGIC);
end contn;

architecture Behavioral of contn is
 constant uno: STD_LOGIC_vector(Nn-1 downto 0) := (0=>'1',others
    =>'0');
 signal cont,comp : STD_LOGIC_vector(Nn-1 downto 0) := (others
    =>'0');
begin

process (clk,resasin)
begin

   if resasin='1' then
      cont<=(0=>'0',others=>'0');
   elsif ( clk'event and clk='1' ) then
      cont<=comp;
    end if;

end process;

n<=cont ;
     comp <= (0=>'0',others=>'0') when entrada=cont and pulso='1'
        else
           cont+uno when pulso='1' else
         cont;


end Behavioral;
```

Código B.4 genpulRZ.

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

entity genpulRZ is
generic( N : integer:=5);
    Port ( entrada : in STD_LOGIC_vector(N-1 downto 0);
           clk : in STD_LOGIC;
           resasin : in STD_LOGIC;
           pulso : out STD_LOGIC);
end genpulRZ;

architecture Behavioral of genpulRZ is
 constant uno: STD_LOGIC_vector(N-1 downto 0) := (0=>'1',others
    =>'0');
 signal mitentr: STD_LOGIC_vector(N-2 downto 0) ;
 signal cont,comp : STD_LOGIC_vector(N-1 downto 0) := (others=>'0')
    ;
begin
mitentr<= entrada(N-1 downto 1);
process (clk,resasin)
begin

   if resasin='1' then
      cont<=(0=>'0',others=>'0');
   elsif (clk'event and clk='1') then
      cont<=comp;
    end if;

end process;

pulso <= '1' when mitentr>cont else
          '0';
     comp <= (0=>'0',others=>'0') when entrada-1=cont else
            cont+uno;

end Behavioral;
```

Código B.5 valores.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity valores is
generic (Nbxt: integer:=14 );
    Port ( n : in STD_LOGIC_VECTOR (8 downto 0);
           salida : out STD_LOGIC_VECTOR (Nbxt-1 downto 0));
end valores;

architecture Behavioral of valores is

begin

salida<= "01011010100111" when n="000000000" else
      "01011010100111" when n="000000001" else
      "01011010100100" when n="000000010" else
      "01011010100001" when n="000000011" else
      "01011010011011" when n="000000100" else
      "01011010010100" when n="000000101" else
      "01011010001100" when n="000000110" else
      "01011010000001" when n="000000111" else
      "01011001110110" when n="000001000" else
      "01011001101001" when n="000001001" else
      "01011001011010" when n="000001010" else
      "01011001001010" when n="000001011" else
      "01011000111000" when n="000001100" else
      "01011000100101" when n="000001101" else
      "01011000010000" when n="000001110" else
      "01010111111010" when n="000001111" else
      "01010111100010" when n="000010000" else
      "01010111001001" when n="000010001" else
      "01010110101110" when n="000010010" else
      "01010110010001" when n="000010011" else
      "01010101110100" when n="000010100" else
      "01010101010100" when n="000010101" else
      "01010100110100" when n="000010110" else
```

```
        "01010100010010" when n="000010111" else
        "01010011101110" when n="000011000" else
        "01010011001001" when n="000011001" else
        "01010010100011" when n="000011010" else
        "01010001111011" when n="000011011" else
        "01010001010001" when n="000011100" else
        "01010000100111" when n="000011101" else
        "01001111111011" when n="000011110" else
        "01001111001101" when n="000011111" else
        "01001110011111" when n="000100000" else
        "01001101101110" when n="000100001" else
        "01001100111101" when n="000100010" else
        "01001100001010" when n="000100011" else
        "01001011010110" when n="000100100" else
        "01001010100001" when n="000100101" else
        "01001001101010" when n="000100110" else
        "01001000110010" when n="000100111" else
        "01000111111001" when n="000101000" else
        "01000110111111" when n="000101001" else
        "01000110000011" when n="000101010" else
        "01000101000110" when n="000101011" else
        "01000100001000" when n="000101100" else
        "01000011001001" when n="000101101" else
        "01000010001001" when n="000101110" else
        "01000001000111" when n="000101111" else
        "01000000000101" when n="000110000" else
        "00111111000001" when n="000110001" else
        "00111101111100" when n="000110010" else
        "00111100110111" when n="000110011" else
        "00111011110000" when n="000110100" else
        "00111010101000" when n="000110101" else
        "00111001011111" when n="000110110" else
        "00111000010101" when n="000110111" else
        "00110111001011" when n="000111000" else
        "00110101111111" when n="000111001" else
        "00110100110010" when n="000111010" else
        "00110011100100" when n="000111011" else
        "00110010010110" when n="000111100" else
        "00110001000111" when n="000111101" else
        "00101111110111" when n="000111110" else
        "00101110100110" when n="000111111" else
        "00101101010100" when n="001000000" else
        "00101100000001" when n="001000001" else
        "00101010101110" when n="001000010" else
        "00101001011010" when n="001000011" else
        "00101000000101" when n="001000100" else
        "00100110110000" when n="001000101" else
```

```
        "00100101011001" when n="001000110" else
        "00100100000011" when n="001000111" else
        "00100010101011" when n="001001000" else
        "00100001010011" when n="001001001" else
        "00011111111011" when n="001001010" else
        "00011110100010" when n="001001011" else
        "00011101001000" when n="001001100" else
        "00011011101110" when n="001001101" else
        "00011010010100" when n="001001110" else
        "00011000111000" when n="001001111" else
        "00010111011101" when n="001010000" else
        "00010110000001" when n="001010001" else
        "00010100100101" when n="001010010" else
        "00010011001000" when n="001010011" else
        "00010001101011" when n="001010100" else
        "00010000001110" when n="001010101" else
        "00001110110001" when n="001010110" else
        "00001101010011" when n="001010111" else
        "00001011110101" when n="001011000" else
        "00001010010111" when n="001011001" else
        "00001000111000" when n="001011010" else
        "00000111011010" when n="001011011" else
        "00000101111011" when n="001011100" else
        "00000100011101" when n="001011101" else
        "00000010111110" when n="001011110" else
        "00000001011111" when n="001011111" else
        "00000000000000" when n="001100000" else
        "11111110100001" when n="001100001" else
        "11111101000010" when n="001100010" else
        "11111011100011" when n="001100011" else
        "11111010000101" when n="001100100" else
        "11111000100110" when n="001100101" else
        "11110111001000" when n="001100110" else
        "11110101101001" when n="001100111" else
        "11110100001011" when n="001101000" else
        "11110010101101" when n="001101001" else
        "11110001001111" when n="001101010" else
        "11101111110010" when n="001101011" else
        "11101110010101" when n="001101100" else
        "11101100111000" when n="001101101" else
        "11101011011011" when n="001101110" else
        "11101001111111" when n="001101111" else
        "11101000100011" when n="001110000" else
        "11100111001000" when n="001110001" else
        "11100101101100" when n="001110010" else
        "11100100010010" when n="001110011" else
        "11100010111000" when n="001110100" else
```

```
        "1110000101111O" when n="001110101" else
```

```
        "10101110101111" when n="010100100" else
        "10101110000101" when n="010100101" else
        "10101101011101" when n="010100110" else
        "10101100110111" when n="010100111" else
        "10101100010010" when n="010101000" else
        "10101011101110" when n="010101001" else
        "10101011001100" when n="010101010" else
        "10101010101100" when n="010101011" else
        "10101010001100" when n="010101100" else
        "10101001101111" when n="010101101" else
        "10101001010010" when n="010101110" else
        "10101000110111" when n="010101111" else
        "10101000011110" when n="010110000" else
        "10101000000110" when n="010110001" else
        "10100111110000" when n="010110010" else
        "10100111011011" when n="010110011" else
        "10100111001000" when n="010110100" else
        "10100110110110" when n="010110101" else
        "10100110100110" when n="010110110" else
        "10100110010111" when n="010110111" else
        "10100110001010" when n="010111000" else
        "10100101111111" when n="010111001" else
        "10100101110100" when n="010111010" else
        "10100101101100" when n="010111011" else
        "10100101100101" when n="010111100" else
        "10100101011111" when n="010111101" else
        "10100101011100" when n="010111110" else
        "10100101011001" when n="010111111" else
        "10100101011001" when n="011000000" else
        "10100101011001" when n="011000001" else
        "10100101011100" when n="011000010" else
        "10100101011111" when n="011000011" else
        "10100101100101" when n="011000100" else
        "10100101101100" when n="011000101" else
        "10100101110100" when n="011000110" else
        "10100101111111" when n="011000111" else
        "10100110001010" when n="011001000" else
        "10100110010111" when n="011001001" else
        "10100110100110" when n="011001010" else
        "10100110110110" when n="011001011" else
        "10100111001000" when n="011001100" else
        "10100111011011" when n="011001101" else
        "10100111110000" when n="011001110" else
        "10101000000110" when n="011001111" else
        "10101000011110" when n="011010000" else
        "10101000110111" when n="011010001" else
        "10101001010010" when n="011010010" else
```

```
        "10101001101111" when n="011010011" else
        "10101010001100" when n="011010100" else
        "10101010101100" when n="011010101" else
        "10101011001100" when n="011010110" else
        "10101011101110" when n="011010111" else
        "10101100010010" when n="011011000" else
        "10101100110111" when n="011011001" else
        "10101101011101" when n="011011010" else
        "10101110000101" when n="011011011" else
        "10101110101111" when n="011011100" else
        "10101111011001" when n="011011101" else
        "10110000000101" when n="011011110" else
        "10110000110011" when n="011011111" else
        "10110001100001" when n="011100000" else
        "10110010010010" when n="011100001" else
        "10110011000011" when n="011100010" else
        "10110011110110" when n="011100011" else
        "10110100101010" when n="011100100" else
        "10110101011111" when n="011100101" else
        "10110110010110" when n="011100110" else
        "10110111001110" when n="011100111" else
        "10111000000111" when n="011101000" else
        "10111001000001" when n="011101001" else
        "10111001111101" when n="011101010" else
        "10111010111010" when n="011101011" else
        "10111011111000" when n="011101100" else
        "10111100110111" when n="011101101" else
        "10111101110111" when n="011101110" else
        "10111110111001" when n="011101111" else
        "10111111111011" when n="011110000" else
        "11000000111111" when n="011110001" else
        "11000010000100" when n="011110010" else
        "11000011001001" when n="011110011" else
        "11000100010000" when n="011110100" else
        "11000101011000" when n="011110101" else
        "11000110100001" when n="011110110" else
        "11000111101011" when n="011110111" else
        "11001000110101" when n="011111000" else
        "11001010000001" when n="011111001" else
        "11001011001110" when n="011111010" else
        "11001100011100" when n="011111011" else
        "11001101101010" when n="011111100" else
        "11001110111001" when n="011111101" else
        "11010000001001" when n="011111110" else
        "11010001011010" when n="011111111" else
        "11010010101100" when n="100000000" else
        "11010011111111" when n="100000001" else
```

```
        "11010101010010" when n="100000010" else
        "11010110100110" when n="100000011" else
        "11010111111011" when n="100000100" else
        "11011001010000" when n="100000101" else
        "11011010100111" when n="100000110" else
        "11011011111101" when n="100000111" else
        "11011101010101" when n="100001000" else
        "11011110101101" when n="100001001" else
        "11100000000101" when n="100001010" else
        "11100001011110" when n="100001011" else
        "11100010111000" when n="100001100" else
        "11100100010010" when n="100001101" else
        "11100101101100" when n="100001110" else
        "11100111001000" when n="100001111" else
        "11101000100011" when n="100010000" else
        "11101001111111" when n="100010001" else
        "11101011011011" when n="100010010" else
        "11101100111000" when n="100010011" else
        "11101110010101" when n="100010100" else
        "11101111110010" when n="100010101" else
        "11110001001111" when n="100010110" else
        "11110010101101" when n="100010111" else
        "11110100001011" when n="100011000" else
        "11110101101001" when n="100011001" else
        "11110111001000" when n="100011010" else
        "11111000100110" when n="100011011" else
        "11111010000101" when n="100011100" else
        "11111011100011" when n="100011101" else
        "11111101000010" when n="100011110" else
        "11111110100001" when n="100011111" else
        "00000000000000" when n="100100000" else
        "00000001011111" when n="100100001" else
        "00000010111110" when n="100100010" else
        "00000100011101" when n="100100011" else
        "00000101111011" when n="100100100" else
        "00000111011010" when n="100100101" else
        "00001000111000" when n="100100110" else
        "00001010010111" when n="100100111" else
        "00001011110101" when n="100101000" else
        "00001101010011" when n="100101001" else
        "00001110110001" when n="100101010" else
        "00010000001110" when n="100101011" else
        "00010001101011" when n="100101100" else
        "00010011001000" when n="100101101" else
        "00010100100101" when n="100101110" else
        "00010110000001" when n="100101111" else
        "00010111011101" when n="100110000" else
```

```
    "00011000111000" when n="100110001" else
    "00011010010100" when n="100110010" else
    "00011011101110" when n="100110011" else
    "00011101001000" when n="100110100" else
    "00011110100010" when n="100110101" else
    "00011111111011" when n="100110110" else
    "00100001010011" when n="100110111" else
    "00100010101011" when n="100111000" else
    "00100100000011" when n="100111001" else
    "00100101011001" when n="100111010" else
    "00100110110000" when n="100111011" else
    "00101000000101" when n="100111100" else
    "00101001011010" when n="100111101" else
    "00101010101110" when n="100111110" else
    "00101100000001" when n="100111111" else
    "00101101010100" when n="101000000" else
    "00101110100110" when n="101000001" else
    "00101111110111" when n="101000010" else
    "00110001000111" when n="101000011" else
    "00110010010110" when n="101000100" else
    "00110011100100" when n="101000101" else
    "00110100110010" when n="101000110" else
    "00110101111111" when n="101000111" else
    "00110111001011" when n="101001000" else
    "00111000010101" when n="101001001" else
    "00111001011111" when n="101001010" else
    "00111010101000" when n="101001011" else
    "00111011110000" when n="101001100" else
    "00111100110111" when n="101001101" else
    "00111101111100" when n="101001110" else
    "00111111000001" when n="101001111" else
    "01000000000101" when n="101010000" else
    "01000001000111" when n="101010001" else
    "01000010001001" when n="101010010" else
    "01000011001001" when n="101010011" else
    "01000100001000" when n="101010100" else
    "01000101000110" when n="101010101" else
    "01000110000011" when n="101010110" else
    "01000110111111" when n="101010111" else
    "01000111111001" when n="101011000" else
    "01001000110010" when n="101011001" else
    "01001001101010" when n="101011010" else
    "01001010100001" when n="101011011" else
    "01001011010110" when n="101011100" else
    "01001100001010" when n="101011101" else
    "01001100111101" when n="101011110" else
    "01001101101110" when n="101011111" else
```

```
      "01001110011111" when n="101100000" else
      "01001111001101" when n="101100001" else
      "01001111111011" when n="101100010" else
      "01010000100111" when n="101100011" else
      "01010001010001" when n="101100100" else
      "01010001111011" when n="101100101" else
      "01010010100011" when n="101100110" else
      "01010011001001" when n="101100111" else
      "01010011101110" when n="101101000" else
      "01010100010010" when n="101101001" else
      "01010100110100" when n="101101010" else
      "01010101010100" when n="101101011" else
      "01010101110100" when n="101101100" else
      "01010110010001" when n="101101101" else
      "01010110101110" when n="101101110" else
      "01010111001001" when n="101101111" else
      "01010111100010" when n="101110000" else
      "01010111111010" when n="101110001" else
      "01011000010000" when n="101110010" else
      "01011000100101" when n="101110011" else
      "01011000111000" when n="101110100" else
      "01011001001010" when n="101110101" else
      "01011001011010" when n="101110110" else
      "01011001101001" when n="101110111" else
      "01011001110110" when n="101111000" else
      "01011010000001" when n="101111001" else
      "01011010001100" when n="101111010" else
      "01011010010100" when n="101111011" else
      "01011010011011" when n="101111100" else
      "01011010100001" when n="101111101" else
      "01011010100100" when n="101111110" else
      "01011010100111";
end Behavioral;
```

B.3 Bloque 2

Este bloque es el que conforma el modulador Sigma/Delta, de forma que toma la salida del bloque 1 y le aplica las operaciones que se han explicado a lo largo de este proyecto. La mayoría de las relaciones entre las señales de las distintas partes del modulador Sigma/Delta se establecen en el código *SDM*, aunque algunas de ellas se establecen en los subprogramas *rtlut* y *tryn*.

Código B.6 sdm.

```

library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;



entity sdm is
generic (Nb: integer := 14);
    Port ( xt : in STD_LOGIC_VECTOR (Nb-1 downto 0);
        clk : IN std_logic:='0';
        resasin : IN std_logic;
        pulsosd :IN  STD_LOGIC;
           yn : out STD_LOGIC_VECTOR (Nb downto 0));
end sdm;

architecture Behavioral of sdm is

COMPONENT restadorA2
  generic (Nb: integer );
  PORT(
    entr_p : IN std_logic_vector(Nb-1 downto 0);
    entr_n : IN std_logic_vector(Nb-1 downto 0);
    sal : OUT std_logic_vector(Nb-1 downto 0);
    cco : OUT std_logic
    );
  END COMPONENT;
COMPONENT sumar
   generic (Nb: integer);
    PORT(
         a : IN std_logic_vector(Nb-1 downto 0);
         b : IN std_logic_vector(Nb-1 downto 0);
         suma : OUT std_logic_vector(Nb-1 downto 0);
         ci : IN std_logic;
         co : OUT std_logic
        );
    END COMPONENT;

COMPONENT cuantificadorpasoq
  generic (Nb: integer);
  PORT(
    qi : IN std_logic_vector(Nb-1 downto 0);
    qo : OUT std_logic_vector(Nb downto 0)
    );
  END COMPONENT;


COMPONENT lut
```

```
  generic (Nb: integer);
  PORT(
    enlut : IN std_logic_vector(Nb downto 0);
    salut : OUT std_logic_vector(Nb-4 downto 0)
    );
  END COMPONENT;


  COMPONENT rtlut
  generic (Nb: integer);
  PORT(
    salut : IN std_logic_vector(Nb-1 downto 0);
    clk : IN std_logic;
    resasin : IN std_logic;
    pulsoSD : IN std_logic;
    senret : OUT std_logic_vector(Nb-1 downto 0)
    );
  END COMPONENT;

  COMPONENT rtyn
  generic (Nb: integer);
  PORT(
    saltr : IN std_logic_vector(Nb-4 downto 0);
    clk : IN std_logic;
    resasin : IN std_logic;
    pulsoSD : IN std_logic;
    entps : OUT std_logic_vector(Nb-3 downto 0)
    );
  END COMPONENT;



signal entpr : STD_LOGIC_VECTOR (Nb-4 downto 0):=(others=>'0');
signal entpralar,salpr,entpsalar,auxqi,qi,auxsalpr :
   STD_LOGIC_VECTOR (Nb-1 downto 0):=(others=>'0');
signal ssalpr,sqi,ssaltralar: std_logic;
signal entps : STD_LOGIC_VECTOR (Nb-3 downto 0):=(others=>'0');
signal aux,qialar,saltralar : STD_LOGIC_VECTOR (Nb downto 0):=(
   others=>'0');
signal salut,saltr : STD_LOGIC_VECTOR (Nb-4 downto 0);

begin

  entpralar <= "000"&entpr when entpr(Nb-4)='0' else
         "111"&entpr;
```

```
pr:  restadorA2 generic map (Nb=>Nb) PORT MAP(
    entr_p =>xt ,
    entr_n =>entpralar ,
    sal => auxsalpr ,
    cco => ssalpr
  );

salpr<= ssalpr&auxsalpr(Nb-2 downto 0) when ((xt(Nb-1)=entpralar(Nb
   -1)) or (xt(Nb-1)='0' and entpralar(Nb-1)='1' and ssalpr='0')
   or (xt(Nb-1)='1' and entpralar(Nb-1)='0' and ssalpr='1'))else
      ("01111111111111") when xt(Nb-1)='0' else
      ("10000000000000") ;


entpsalar <= "00"&entps when (entps(Nb-3)='0') else
         "11"&entps;

psu: sumar generic map (Nb=>Nb) PORT MAP(
    a => salpr ,
    b => entpsalar ,
    suma => auxqi ,
    ci => '0' ,
    co => sqi
  );

qi<=  sqi&auxqi(Nb-2 downto 0) when (salpr(Nb-1)=not(entpsalar(Nb
   -1))) or ((salpr(Nb-1)=entpsalar(Nb-1) and sqi=salpr(Nb-1)) and
    (sqi='1' or sqi='0')) else
      ("01111111111111") when salpr(Nb-1)='0' else
      ("10000000000000");

cuantificadorq: cuantificadorpasoq generic map (Nb=>Nb) PORT MAP(
    qi => qi ,
    qo => aux
  );
yn<=aux;

qialar <= "0"&qi when qi(Nb-1)='0' else
       "1"&qi;


tr:  restadorA2 generic map (Nb=>(Nb+1)) PORT MAP(
    entr_p =>aux ,
    entr_n =>qialar ,
    sal => saltralar ,
    cco => ssaltralar
  );
```

```
saltr<= ssaltralar&saltralar(Nb-5 downto 0) when ((aux(Nb)=qialar(
    Nb)) or (aux(Nb)='0' and qialar(Nb)='1' and ssaltralar='0') or
    (aux(Nb)='1' and qialar(Nb)='0' and ssaltralar='1')) else
      ("01111111111") when aux(Nb)='0' else
      ("10000000000") ;









cuantificadorlut: lut generic map (Nb=>Nb) PORT MAP(
    enlut =>aux ,
    salut =>salut
  );


bloquertlut: rtlut generic map (Nb=>(Nb-3)) PORT MAP(
    salut =>salut ,
    clk =>clk ,
    pulsoSD=>pulsoSD,
    resasin =>resasin ,
    senret => entpr
  );


bloquertyn: rtyn generic map (Nb=>Nb) PORT MAP(
    saltr =>saltr ,
    clk =>clk ,
    pulsoSD=>pulsoSD,
    resasin =>resasin ,
    entps => entps
  );
```

```
end Behavioral;
```

Código B.7 restadorA2.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
entity restadorA2 is
  generic (Nb: integer := 5);
    Port ( entr_p : in STD_LOGIC_VECTOR (Nb-1 downto 0);
           entr_n : in STD_LOGIC_VECTOR (Nb-1 downto 0);
           sal : out STD_LOGIC_VECTOR (Nb-1 downto 0);
           cco : out STD_LOGIC);
end restadorA2;
architecture Behavioral of restadorA2 is

COMPONENT sumar
   generic (Nb: integer :=5); -- tamano por defecto
    PORT(
         a : IN std_logic_vector(Nb-1 downto 0);
         b : IN std_logic_vector(Nb-1 downto 0);
         suma : OUT std_logic_vector(Nb-1 downto 0);
         ci : IN std_logic;
         co : OUT std_logic
        );
    END COMPONENT;
signal a , b, suma: std_logic_vector(Nb-1 downto 0) := (others =>
    '0');
   signal ci, co : std_logic := '0';
begin
  uut: sumar generic map (Nb=>Nb) PORT MAP (
          a => a,
          b =>b,
          suma => suma,
          ci => ci,
          co => co );
  a <= entr_p;
  b <= not(entr_n);
  ci <= '1';
  sal <= suma;
  cco <= co;
end Behavioral;
```

Código B.8 sumar.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity sumar is
generic (Nb: integer := 5);
    Port ( a : in STD_LOGIC_VECTOR (Nb-1 downto 0);
           b : in STD_LOGIC_VECTOR (Nb-1 downto 0);
           suma : out STD_LOGIC_VECTOR(Nb-1 downto 0);
           ci : in STD_LOGIC:='0';
           co : out STD_LOGIC);
end sumar;
architecture arq_sumador of sumar is
signal ax,bx,sumax: std_logic_vector(Nb downto 0);
constant cinx: std_logic_vector(Nb downto 1) := (others =>'0');

begin



ax <= '0'&a when a(Nb-1)='0' else
      '1'&a ;
bx <= '0'&b when b(Nb-1)='0' else
      '1'&b ;





sumax <= ax + bx + (cinx&ci) ;

suma <= sumax(Nb-1 downto 0) ;
co <= sumax(Nb) ;


end architecture;
```

Código B.9 cuantificadorpasoq.

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

entity cuantificadorpasoq is
```

```
generic (Nb: integer := 14);
    Port ( qi : in STD_LOGIC_VECTOR (Nb-1 downto 0);
           qo : out STD_LOGIC_VECTOR (Nb downto 0));
end cuantificadorpasoq;

architecture Behavioral of cuantificadorpasoq is
constant nivel1 : std_logic_vector(Nb downto 0) := ("
   101111110101000");
constant nivel2 : std_logic_vector(Nb downto 0) := ("
   110001111111000");
constant nivel3 : std_logic_vector(Nb downto 0) := ("
   110100001001000");
constant nivel4 : std_logic_vector(Nb downto 0) := ("
   110110010011000");
constant nivel5 : std_logic_vector(Nb downto 0) := ("
   111000011101000");
constant nivel6 : std_logic_vector(Nb downto 0) := ("
   111010100111000");
constant nivel7 : std_logic_vector(Nb downto 0) := ("
   111100110001000");
constant nivel8 : std_logic_vector(Nb downto 0) := ("
   111110111011000");
constant nivel9 : std_logic_vector(Nb downto 0) := ("
   000001000101000");
constant nivel10: std_logic_vector(Nb downto 0) := ("
   000011001111000");
constant nivel11: std_logic_vector(Nb downto 0) := ("
   000101011001000");
constant nivel12: std_logic_vector(Nb downto 0) := ("
   000111100011000");
constant nivel13: std_logic_vector(Nb downto 0) := ("
   001001101101000");
constant nivel14: std_logic_vector(Nb downto 0) := ("
   001011110111000");
constant nivel15: std_logic_vector(Nb downto 0) := ("
   001110000001000");
constant nivel16: std_logic_vector(Nb downto 0) := ("
   010000001011000");


constant limsup1 : std_logic_vector(Nb-2 downto 0) := ("
   1111000110001");
constant liminf2 : std_logic_vector(Nb-2 downto 0) := ("
   1111000110000");
constant limsup2 : std_logic_vector(Nb-2 downto 0) := ("
   1100111100001");
```

```
constant liminf3 : std_logic_vector(Nb-2 downto 0) := ("
   1100111100000");
constant limsup3 : std_logic_vector(Nb-2 downto 0) := ("
   1010110010001");
constant liminf4 : std_logic_vector(Nb-2 downto 0) := ("
   1010110010000");
constant limsup4 : std_logic_vector(Nb-2 downto 0) := ("
   1000101000001");
constant liminf5 : std_logic_vector(Nb-2 downto 0) := ("
   1000101000000");
constant limsup5 : std_logic_vector(Nb-2 downto 0) := ("
   0110011110001");
constant liminf6 : std_logic_vector(Nb-2 downto 0) := ("
   0110011110000");
constant limsup6 : std_logic_vector(Nb-2 downto 0) := ("
   0100010100001");
constant liminf7 : std_logic_vector(Nb-2 downto 0) := ("
   0100010100000");
constant limsup7 : std_logic_vector(Nb-2 downto 0) := ("
   0010001010001");
constant liminf8 : std_logic_vector(Nb-2 downto 0) := ("
   0010001010000");
constant limsup8 : std_logic_vector(Nb-2 downto 0) := ("
   0000000000001");
constant liminf9 : std_logic_vector(Nb-2 downto 0) := ("
   0000000000000");
constant limsup9 : std_logic_vector(Nb-2 downto 0) := ("
   0010001001111");
constant liminf10 : std_logic_vector(Nb-2 downto 0) := ("
   0010001010000");
constant limsup10 : std_logic_vector(Nb-2 downto 0) := ("
   0100010011111");
constant liminf11 : std_logic_vector(Nb-2 downto 0) := ("
   0100010100000");
constant limsup11 : std_logic_vector(Nb-2 downto 0) := ("
   0110011101111");
constant liminf12 : std_logic_vector(Nb-2 downto 0) := ("
   0110011110000");
constant limsup12 : std_logic_vector(Nb-2 downto 0) := ("
   1000100111111");
constant liminf13 : std_logic_vector(Nb-2 downto 0) := ("
   1000101000000");
constant limsup13 : std_logic_vector(Nb-2 downto 0) := ("
   1010110001111");
constant liminf14 : std_logic_vector(Nb-2 downto 0) := ("
   1010110010000");
```

```
constant limsup14 : std_logic_vector(Nb-2 downto 0) := ("
   1100111011111");
constant liminf15 : std_logic_vector(Nb-2 downto 0) := ("
   1100111100000");
constant limsup15 : std_logic_vector(Nb-2 downto 0) := ("
   1111000101111");
constant uno : std_logic_vector(Nb-2 downto 0) := (0=>'1',others
   =>'0');
signal valoracomparar:STD_LOGIC_VECTOR (Nb-2 downto 0);
begin

valoracomparar<= qi(Nb-2 downto 0) when qi(Nb-1)='0' else
          "1111111111111" when qi="10000000000000" else
              not(qi(Nb-2 downto 0))+uno;



  qo <= nivel1 when qi(Nb-1)='1' and limsup1<=valoracomparar else
       nivel2 when qi(Nb-1)='1' and liminf2>=valoracomparar and
          limsup2<=valoracomparar else
       nivel3 when qi(Nb-1)='1' and liminf3>=valoracomparar and
          limsup3<=valoracomparar else
     nivel4 when qi(Nb-1)='1' and liminf4>=valoracomparar and
        limsup4<=valoracomparar else
     nivel5 when qi(Nb-1)='1' and liminf5>=valoracomparar and
        limsup5<=valoracomparar else
     nivel6 when qi(Nb-1)='1' and liminf6>=valoracomparar and
        limsup6<=valoracomparar else
       nivel7 when qi(Nb-1)='1' and liminf7>=valoracomparar and
          limsup7<=valoracomparar else
       nivel8 when qi(Nb-1)='1' and liminf8>=valoracomparar and
          limsup8<=valoracomparar else
     nivel9 when qi(Nb-1)='0' and liminf9<=valoracomparar and
        limsup9>=valoracomparar else
     nivel10 when qi(Nb-1)='0' and liminf10<=valoracomparar and
        limsup10>=valoracomparar else
     nivel11 when qi(Nb-1)='0' and liminf11<=valoracomparar and
        limsup11>=valoracomparar else
       nivel12 when qi(Nb-1)='0' and liminf12<=valoracomparar and
          limsup12>=valoracomparar else
       nivel13 when qi(Nb-1)='0' and liminf13<=valoracomparar and
          limsup13>=valoracomparar else
     nivel14 when qi(Nb-1)='0' and liminf14<=valoracomparar and
        limsup14>=valoracomparar else
     nivel15 when qi(Nb-1)='0' and liminf15<=valoracomparar and
        limsup15>=valoracomparar else
     nivel16;
```

```
end Behavioral;
```

Código B.10 lut.

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity lut is
generic (Nb: integer := 14);
    Port ( enlut : in STD_LOGIC_VECTOR (Nb downto 0);
           salut : out STD_LOGIC_VECTOR (Nb-4 downto 0));
end lut;

architecture Behavioral of lut is
constant nivel1 : std_logic_vector(Nb-4 downto 0) := ("00000000000"
   );
constant nivel2 : std_logic_vector(Nb-4 downto 0) := ("00000000010"
   );
constant nivel3 : std_logic_vector(Nb-4 downto 0) := ("00000000110"
   );
constant nivel4 : std_logic_vector(Nb-4 downto 0) := ("00000001101"
   );
constant nivel5 : std_logic_vector(Nb-4 downto 0) := ("00000011010"
   );
constant nivel6 : std_logic_vector(Nb-4 downto 0) := ("00000101100"
   );
constant nivel7 : std_logic_vector(Nb-4 downto 0) := ("00001000110"
   );
constant nivel8 : std_logic_vector(Nb-4 downto 0) := ("00001101001"
   );
constant nivel9 : std_logic_vector(Nb-4 downto 0) := ("00010010101"
   );
constant nivel10: std_logic_vector(Nb-4 downto 0) := ("00011001100"
   );
```

```
constant nivel11: std_logic_vector(Nb-4 downto 0) := ("00100010000"
   );
constant nivel12: std_logic_vector(Nb-4 downto 0) := ("00101100001"
   );
constant nivel13: std_logic_vector(Nb-4 downto 0) := ("00111000001"
   );
constant nivel14: std_logic_vector(Nb-4 downto 0) := ("01000110001"
   );
constant nivel15: std_logic_vector(Nb-4 downto 0) := ("01010110010"
   );
constant entr0 : std_logic_vector(Nb downto 0) := ("101111110101000
   ");
constant entr1 : std_logic_vector(Nb downto 0) := ("110001111111000
   ");
constant entr2 : std_logic_vector(Nb downto 0) := ("110100001001000
   ");
constant entr3 : std_logic_vector(Nb downto 0) := ("110110010011000
   ");
constant entr4 : std_logic_vector(Nb downto 0) := ("111000011101000
   ");
constant entr5 : std_logic_vector(Nb downto 0) := ("111010100111000
   ");
constant entr6 : std_logic_vector(Nb downto 0) := ("111100110001000
   ");
constant entr7 : std_logic_vector(Nb downto 0) := ("111110111011000
   ");
constant entr8 : std_logic_vector(Nb downto 0) := ("000001000101000
   ");
constant entr9 : std_logic_vector(Nb downto 0) := ("000011001111000
   ");
constant entr10 : std_logic_vector(Nb downto 0) := ("
   000101011001000");
constant entr11 : std_logic_vector(Nb downto 0) := ("
   000111100011000");
constant entr12 : std_logic_vector(Nb downto 0) := ("
   001001101101000");
constant entr13 : std_logic_vector(Nb downto 0) := ("
   001011110111000");
constant entr14 : std_logic_vector(Nb downto 0) := ("
   001110000001000");

begin

  salut <= nivel1 when enlut=entr0 or enlut=entr1 else
           nivel2 when enlut=entr2 else
           nivel3 when enlut=entr3 else
        nivel4 when enlut=entr4 else
```

```
        nivel5 when enlut=entr5 else
        nivel6 when enlut=entr6 else
           nivel7 when enlut=entr7 else
           nivel8 when enlut=entr8 else
        nivel9 when enlut=entr9 else
        nivel10 when enlut=entr10 else
        nivel11 when enlut=entr11 else
           nivel12 when enlut=entr12 else
           nivel13 when enlut=entr13 else
        nivel14 when enlut=entr14 else
        nivel15;


end Behavioral;
```

Código B.11 rtlut.

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity rtlut is
generic (Nb: integer := 11);
    Port ( salut : in STD_LOGIC_VECTOR (Nb-1 downto 0);
        clk : IN std_logic;
        resasin,pulsoSD: IN std_logic;
           senret : out STD_LOGIC_VECTOR (Nb-1 downto 0));
end rtlut;

architecture Behavioral of rtlut is



  COMPONENT zmenos1
  generic (Nb: integer);
  PORT(
    entrada : IN std_logic_vector(Nb-1 downto 0);
```

```
    clk : IN std_logic;
    pulsoSD : IN std_logic;
    resasin : IN std_logic;
    salida : OUT std_logic_vector(Nb-1 downto 0)
    );
  END COMPONENT;

  COMPONENT restadorA2
  generic (Nb: integer );
  PORT(
    entr_p : IN std_logic_vector(Nb-1 downto 0);
    entr_n : IN std_logic_vector(Nb-1 downto 0);
    sal : OUT std_logic_vector(Nb-1 downto 0);
    cco : OUT std_logic
    );
  END COMPONENT;

  COMPONENT sumar
   generic (Nb: integer); -- tamano por defecto
    PORT(
         a : IN std_logic_vector(Nb-1 downto 0);
         b : IN std_logic_vector(Nb-1 downto 0);
         suma : OUT std_logic_vector(Nb-1 downto 0);
         ci : IN std_logic;
         co : OUT std_logic
        );
    END COMPONENT;


signal psr,ssr,tsr : STD_LOGIC_VECTOR (Nb-1 downto 0);
signal ssrdoble,psralar,tsralar,auxsalcralar,salcralar,senretalar :
    STD_LOGIC_VECTOR (Nb downto 0);
signal ssalcralar,ssenretalar : STD_LOGIC;
begin


  pzm1: zmenos1 generic map (Nb=>Nb) PORT MAP(
    entrada => salut ,
    salida => psr,
    clk => clk ,
    pulsoSD=>pulsoSD,
    resasin => resasin
  );


  szm1: zmenos1 generic map (Nb=>Nb) PORT MAP(
    entrada => psr ,
```

```
    salida => ssr,
    clk => clk ,
    pulsoSD=>pulsoSD,
    resasin => resasin
  );

  tzm1: zmenos1 generic map (Nb=>Nb) PORT MAP(
    entrada => ssr ,
    salida => tsr,
    clk => clk ,
    pulsoSD=>pulsoSD,
    resasin => resasin
  );
psralar<='0'&psr;
ssrdoble<=ssr&'0';
tsralar<='0'&tsr;
cr:restadorA2 generic map (Nb=>(Nb+1)) PORT MAP(
    entr_p =>psralar ,
    entr_n =>ssrdoble ,
    sal => auxsalcralar ,
    cco => ssalcralar
  );

salcralar<= ssalcralar&auxsalcralar(Nb-1 downto 0) when (psralar(Nb
   )=ssrdoble(Nb)) or (psralar(Nb)='0' and ssrdoble(Nb)='1' and
   ssalcralar='0') or (psralar(Nb)='1' and ssrdoble(Nb)='0' and
   ssalcralar='1') else
     ("011111111111") when psralar(Nb)='0' else
     ("100000000000");


tsu: sumar generic map (Nb=>(Nb+1)) PORT MAP(
    a => salcralar ,
    b => tsralar ,
    suma => senretalar ,
    ci => '0' ,
    co => ssenretalar
  );


senret<= ssenretalar&senretalar(Nb-2 downto 0) when (((salcralar(Nb
   )=not(tsralar(Nb))) or (salcralar(Nb)=tsralar(Nb) and salcralar
   (Nb)=ssenretalar)) and (ssenretalar='0' or ssenretalar='1'))
   else---evitamos que la salida sea sea XXXXXXXXXX imponiendo que
     ssenretalar no sea X
```

```
      ("01111111111") when salcralar(Nb)='0' else ---se comprueba
         para evitar resultados extraños al principio de la
         simulacion
      ("10000000000");


end Behavioral;
```

Código B.12 zmenos1.

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


entity zmenos1 is
    generic (Nb: integer := 2);
    Port ( entrada : in STD_LOGIC_VECTOR (Nb-1 downto 0);
           salida : out STD_LOGIC_VECTOR (Nb-1 downto 0);
           clk,resasin,pulsoSD : in STD_LOGIC);
end zmenos1;
architecture Behavioral of zmenos1 is
signal D,Q :STD_LOGIC_VECTOR (Nb-1 downto 0);
begin






  process(resasin,clk)
  begin
    if (resasin='1') then Q <= (0 => '0',others => '0');
    elsif (clk'event and clk='1') then Q <= D;
    end if;
  end process;
D<= entrada when pulsoSD='1' else
   Q;
  salida<=Q ;

end Behavioral;
```

Código B.13 rtyn.

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

```
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity rtyn is
   generic (Nb: integer := 14);
    Port ( saltr : in STD_LOGIC_VECTOR (Nb-4 downto 0);
        clk : IN std_logic;
        resasin : IN std_logic;
        pulsoSD : IN std_logic;
           entps : out STD_LOGIC_VECTOR (Nb-3 downto 0));
end rtyn;

architecture Behavioral of rtyn is


  COMPONENT zmenos1
  generic (Nb: integer);
  PORT(
    entrada : IN std_logic_vector(Nb-1 downto 0);
    clk : IN std_logic;
    resasin : IN std_logic;
    pulsoSD : IN std_logic;
    salida : OUT std_logic_vector(Nb-1 downto 0)
    );
  END COMPONENT;

  COMPONENT restadorA2
  generic (Nb: integer );
  PORT(
    entr_p : IN std_logic_vector(Nb-1 downto 0);
    entr_n : IN std_logic_vector(Nb-1 downto 0);
    sal : OUT std_logic_vector(Nb-1 downto 0);
    cco : OUT std_logic
    );
  END COMPONENT;
signal csr,qsr: STD_LOGIC_VECTOR (Nb-4 downto 0);
signal dobledecsr,qsralar,salqr,auxsalqr:STD_LOGIC_VECTOR (Nb-3
   downto 0);
signal ssalqr:STD_LOGIC;
begin
```

```
czm1: zmenos1 generic map (Nb=>(Nb-3)) PORT MAP(
    entrada => saltr ,
    salida => csr ,
    clk =>clk ,
    pulsoSD=>pulsoSD,
    resasin =>resasin);
dobledecsr <= csr&'0';--duplicar a un negativo se hace igual que a
   un positivo

qzm1: zmenos1 generic map (Nb=>(Nb-3)) PORT MAP(
    entrada => csr ,
    salida => qsr ,
    clk =>clk ,
    pulsoSD=>pulsoSD,
    resasin =>resasin );
qsralar<= "0"&qsr when qsr(Nb-4)='0' else ----salida del quinto z-1
         "1"&qsr;
qr:restadorA2 generic map (Nb=>(Nb-2)) PORT MAP(
    entr_p =>qsralar ,
    entr_n =>dobledecsr ,
    sal => auxsalqr ,
    cco => ssalqr);
salqr<= ("011111111111") when qsralar(Nb-3)='0' and dobledecsr(Nb
   -3)='1' and ssalqr='1' else
      ("100000000000") when qsralar(Nb-3)='1' and dobledecsr(Nb-3)
         ='0' and ssalqr='0' else
       ssalqr&auxsalqr(Nb-4 downto 0);
entps<=salqr ;
end Behavioral;
```

B.4 Bloque 3

Este bloque, como ya se explicó, está formado por dos códigos: el primero de ellos es el que ha denominado PWM, que toma la salida del bloque 2, la transforma en una señal de la forma típica de este tipo de modulador en el segundo programa (*definirP*) y genera la salida de este modulador al final del primer código, separando el primer bit del resto de la salida del programa *definirP*.

Código B.14 PWM.

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
```

```
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity PWM is
generic (Nb: integer := 14);
    Port ( ySMN : in STD_LOGIC_VECTOR (Nb downto 0);
           clk : in STD_LOGIC;
        resasin : in STD_LOGIC;
        pulsosd : in STD_LOGIC;
        pulsoPWM : in STD_LOGIC;
           salidaPWM : out STD_LOGIC);
end PWM;

architecture Behavioral of PWM is
COMPONENT definirP
generic (Nb: integer := 14);
  PORT(
    ySDM : IN std_logic_vector(Nb downto 0);
    P : OUT std_logic_vector(29 downto 0)
    );
  END COMPONENT;

signal P,Q,D : std_logic_vector(29 downto 0);

begin
Inst_definirP: definirP generic map (Nb=>Nb) PORT MAP(
    ySDM =>ySMN ,
    P => P
  );

  process(clk,resasin)
  begin
    if resasin='1' then Q<=(others => '0');
    elsif (clk='1' and clk'event) then Q<=D;
    end if;
  end process;


D<=P when pulsoSD='1' and pulsoPWM='1' else
   Q(28 downto 0)&Q(29) when pulsoPWM='1' else
   Q;
```

```
salidaPWM<= Q(29) ;

end Behavioral;
```

Código B.15 definirP.

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity definirP is
generic (Nb: integer := 14);
    Port ( ySDM : in STD_LOGIC_VECTOR (Nb downto 0);
           P : out STD_LOGIC_VECTOR (29 downto 0));
end definirP;

architecture Behavioral of definirP is
constant salida1 : STD_LOGIC_VECTOR (29 downto 0) := "
   000000000000000000000000000000";
constant salida2 : STD_LOGIC_VECTOR (29 downto 0) := "
   000000000000001100000000000000";
constant salida3 : STD_LOGIC_VECTOR (29 downto 0) := "
   000000000000011100000000000000";
constant salida4 : STD_LOGIC_VECTOR (29 downto 0) := "
   000000000000111110000000000000";
constant salida5 : STD_LOGIC_VECTOR (29 downto 0) := "
   000000000001111111000000000000";
constant salida6 : STD_LOGIC_VECTOR (29 downto 0) := "
   000000000011111111100000000000";
constant salida7 : STD_LOGIC_VECTOR (29 downto 0) := "
   000000000111111111110000000000";
constant salida8 : STD_LOGIC_VECTOR (29 downto 0) := "
   000000001111111111111000000000";
constant salida9 : STD_LOGIC_VECTOR (29 downto 0) := "
   000000011111111111111100000000";
```

```
constant salida10: STD_LOGIC_VECTOR (29 downto 0) := "
   000000111111111111111111000000";
constant salida11: STD_LOGIC_VECTOR (29 downto 0) := "
   000001111111111111111111100000";
constant salida12: STD_LOGIC_VECTOR (29 downto 0) := "
   000011111111111111111111110000";
constant salida13: STD_LOGIC_VECTOR (29 downto 0) := "
   000111111111111111111111111000";
constant salida14: STD_LOGIC_VECTOR (29 downto 0) := "
   001111111111111111111111111100";
constant salida15: STD_LOGIC_VECTOR (29 downto 0) := "
   011111111111111111111111111110";
constant salida16: STD_LOGIC_VECTOR (29 downto 0) := "
   111111111111111111111111111111";
constant nivel1 : std_logic_vector(Nb downto 0) := ("
   101111110101000");
constant nivel2 : std_logic_vector(Nb downto 0) := ("
   110001111111000");
constant nivel3 : std_logic_vector(Nb downto 0) := ("
   110100001001000");
constant nivel4 : std_logic_vector(Nb downto 0) := ("
   110110010011000");
constant nivel5 : std_logic_vector(Nb downto 0) := ("
   111000011101000");
constant nivel6 : std_logic_vector(Nb downto 0) := ("
   111010100111000");
constant nivel7 : std_logic_vector(Nb downto 0) := ("
   111100110001000");
constant nivel8 : std_logic_vector(Nb downto 0) := ("
   111110111011000");
constant nivel9 : std_logic_vector(Nb downto 0) := ("
   000001000101000");
constant nivel10: std_logic_vector(Nb downto 0) := ("
   000011001111000");
constant nivel11: std_logic_vector(Nb downto 0) := ("
   000101011001000");
constant nivel12: std_logic_vector(Nb downto 0) := ("
   000111100011000");
constant nivel13: std_logic_vector(Nb downto 0) := ("
   001001101101000");
constant nivel14: std_logic_vector(Nb downto 0) := ("
   001011110111000");
constant nivel15: std_logic_vector(Nb downto 0) := ("
   001110000001000");


begin
```

```
  P<= salida1 when ySDM=nivel1 else
      salida2 when ySDM=nivel2 else
    salida3 when ySDM=nivel3 else
    salida4 when ySDM=nivel4 else
    salida5 when ySDM=nivel5 else
    salida6 when ySDM=nivel6 else
    salida7 when ySDM=nivel7 else
    salida8 when ySDM=nivel8 else
    salida9 when ySDM=nivel9 else
    salida10 when ySDM=nivel10 else
    salida11 when ySDM=nivel11 else
    salida12 when ySDM=nivel12 else
    salida13 when ySDM=nivel13 else
    salida14 when ySDM=nivel14 else
    salida15 when ySDM=nivel15 else
      salida16;



end Behavioral;
```

B.5 Bloque 4

Este bloque, como ya se sabe, prepara las salidas para que una de ellas pase a ser una señal con retorno a cero. Además, aparte de este programa el bloque 4 consta de un retraso que como se ve en el bloque TOP se aplica a la salida del siguiente código:

Código B.16 salidaD.

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity salidaD is
    Port ( yPWM : in STD_LOGIC;
            pulsopwm: in STD_LOGIC;
  salRZ: out STD_LOGIC;
```

```
   salNRZ: out STD_LOGIC;
     clk,resasin : in STD_LOGIC);
end salidaD;

architecture Behavioral of salidaD is

signal yRZ: std_logic;
signal Q2: std_logic;
signal Q1: std_logic;


begin

yRZ<= yPWM and pulsoPWM;


salNRZ <=Q1;
salRZ  <=Q2;

  process(resasin,clk)
  begin
    if (resasin='1') then
    Q1 <= '0';
      Q2 <= '0';
    elsif (clk'event and clk='1') then
    Q1 <= yPWM;
      Q2 <= yRZ;
    end if;
  end process;



end Behavioral;
```

B.6 Simulación del TOP

Este bloque permite simular y obtener los resultados que se han simulado y analizado a lo largo de este proyecto. Este código ha debido modificarse varias veces, ya que dependerá de si se quieren obtener los resultados del SDM o del PWM, y del valor de N_{PWM} (ya que su valor influirá en la frecuencia con la que el programa guarda los datos en el archivo de texto de los resultados, y en el tiempo que debe esperar para comenzar a guardarlos).

Código B.17 Simulación.

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;
use STD.textio.all;
use ieee.std_logic_textio.all;
use ieee.std_logic_unsigned.all;
--use WORK.ti_ol_sdm_types.all;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY simulacion IS
END simulacion;

ARCHITECTURE behavior OF simulacion IS

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


    COMPONENT top
    PORT(
         clk : IN std_logic;
         resasin : IN std_logic;
         SALNRZ : OUT std_logic;
         SALRZ : OUT std_logic
        );
    END COMPONENT;
     file file_RESULTS : text;

   --Inputs
   signal clk : std_logic := '0';
   signal resasin : std_logic := '1';

  --Outputs
   signal SALNRZ : std_logic;
   signal SALRZ : std_logic;

signal j:integer range 0 to 1000;
signal cont:integer range 0 to 30;



   -- Clock period definitions
   constant clk_period : time := 20 ns;
  ---------------------------------------------------------------
  signal entrada:integer range 0 to (2**13-1);
 signal salida:integer range 0 to (2**14-1);
```

```
--
   -----------------------------------------------------------------------

BEGIN

  -- Instantiate the Unit Under Test (UUT)
   uut: top PORT MAP (
          clk => clk,
       resasin=>resasin,
       SALNRZ=>SALNRZ,
          SALRZ => SALRZ);

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

-----------------------------
  process
    variable v_OLINE   : line;
    variable v_SPACE : character;
    begin
        file_open(file_RESULTS, "/home/ise/Desktop/
           ARCHIVOS_DEFINITIVOS_SALRYSALNR/SALRZ/file_RESSD.txt",
           write_mode);
 wait for 20 ns;
resasin<='0';
--wait for 1190 ns;
wait for 4800 ns;
for cont in 1 to 2**30 loop

           wait for 160 ns;


           write(v_OLINE, SALRZ);
           write(v_OLINE, string'(" "));
           write(v_OLINE, SALNRZ);
           writeline(file_RESULTS, v_OLINE);



end loop;
```

```
        file_close(file_RESULTS);

        wait;
    end process;

END;
```

Apéndice C

Gráficas de las frecuencias de corte

Las siguientes gráficas muestran el resultado de hacer simulaciones en AC de los filtros que se han diseñado. Permiten comprobar si, para las resistencias que ha proporcionado el laboratorio de la Escuela, el valor de la frecuencia de corte de cada filtro es lo suficientemente cercano al que se calculó teóricamente para justificar que puedan usarse.

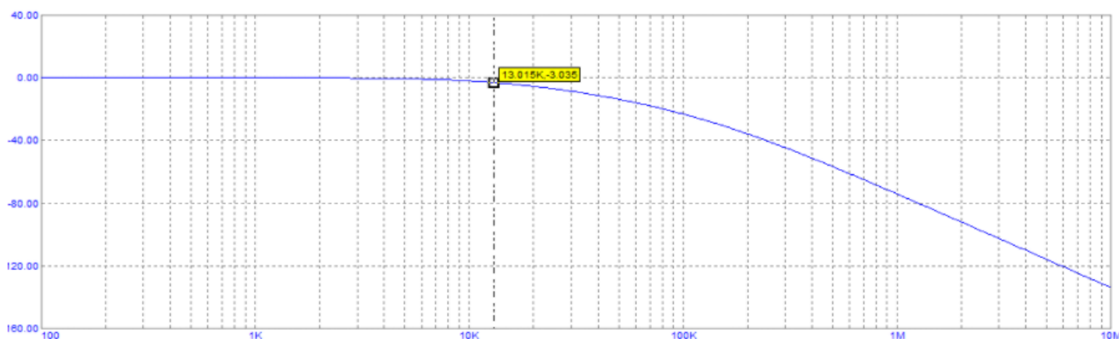

Figura C.1 Ganancia del filtro paso bajo con $N_{PWM} = 2$.

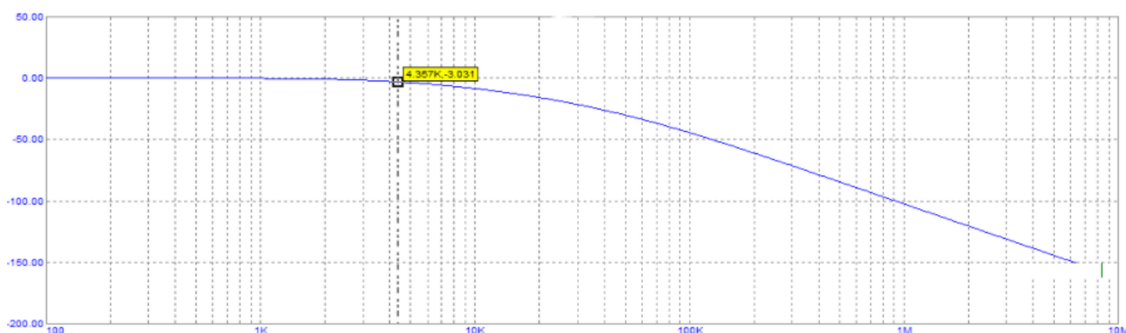

Figura C.2 Ganancia del filtro paso bajo con $N_{PWM} = 6$.

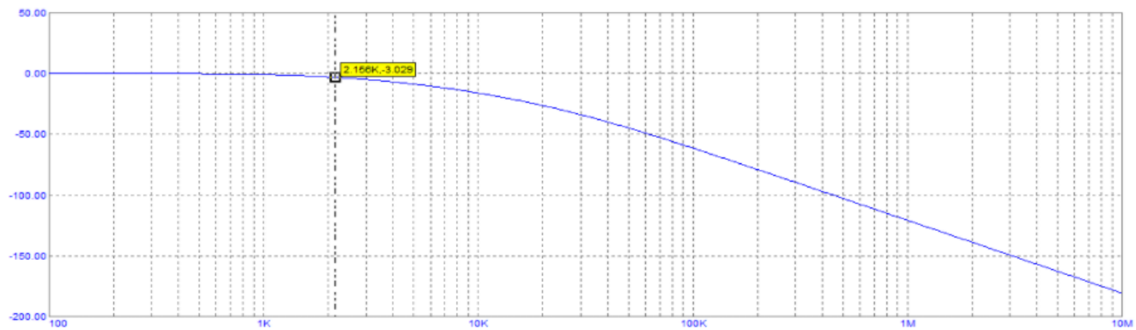

Figura C.3 Ganancia del filtro paso bajo con $N_{PWM} = 12$.

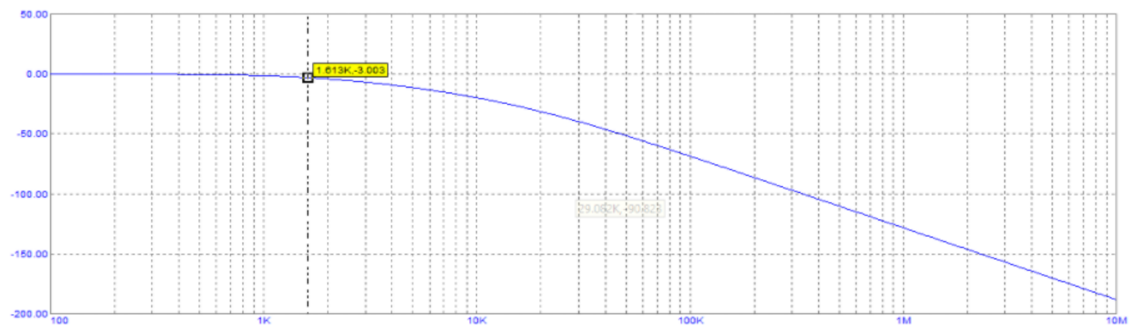

Figura C.4 Ganancia del filtro paso bajo con $N_{PWM} = 16$.

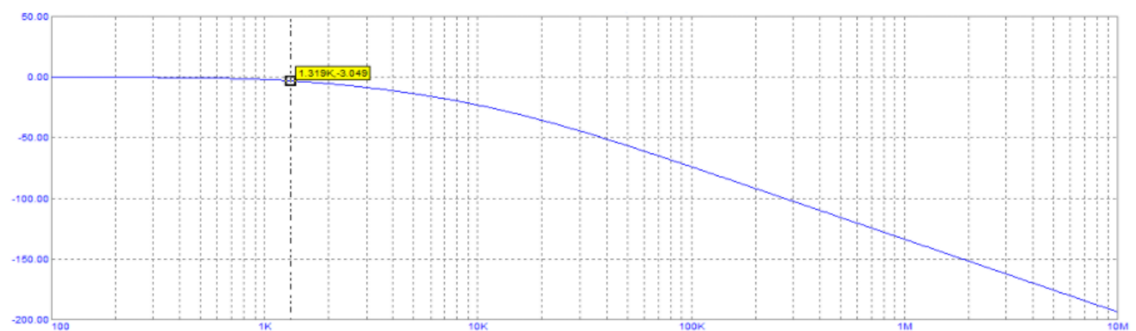

Figura C.5 Ganancia del filtro paso bajo con $N_{PWM} = 20$.

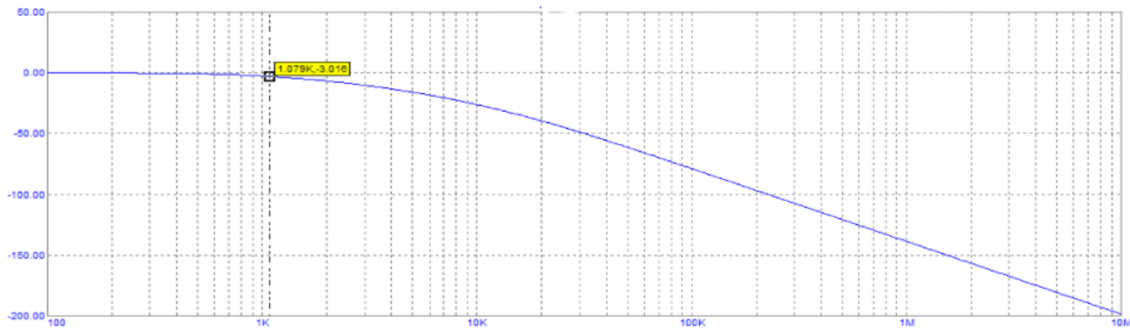

Figura C.6 Ganancia del filtro paso bajo con $N_{PWM} = 24$.

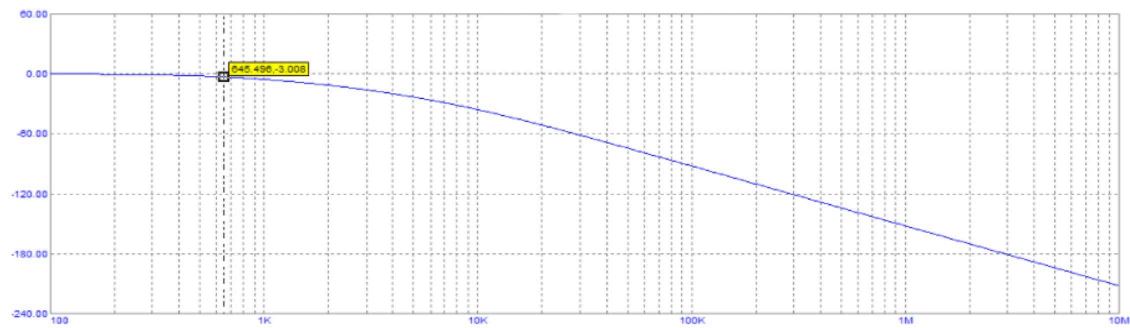

Figura C.7 Ganancia del filtro paso bajo con $N_{PWM} = 40$.

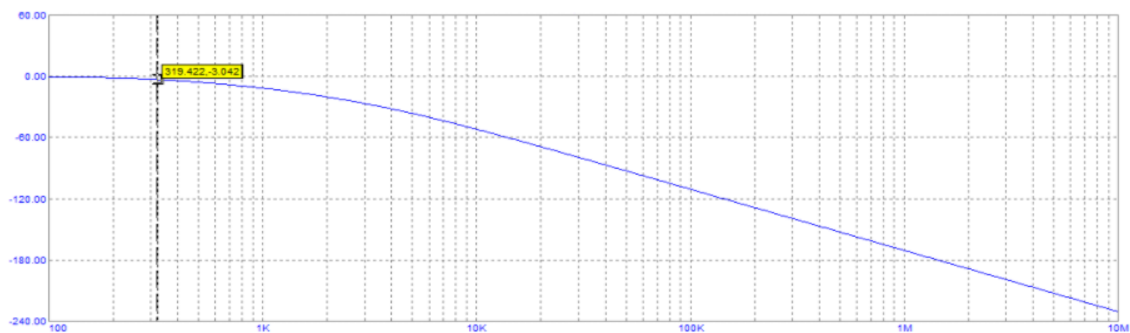

Figura C.8 Ganancia del filtro paso bajo con $N_{PWM} = 80$.

En la siguiente tabla se han recogido los valores de la frecuencia de corte que se pueden ver en las gráficas anteriores, junto con los que se obtuvieron con la expresión ya vista, $f_c = \frac{f_{clk}}{OSR \cdot 2 \cdot 30 \cdot N_{PWM}}$, para comprobar que los valores son lo suficientemente cercanos y que por lo tanto, la frecuencia de corte asociada a los filtros cumplirá adecuadamente su función:

Tabla C.1 Comparación del valor de las frecuencias de los filtros .

Valor de N_{PWM}	Valores óptimos de f_c (kHz)	Valores obtenidos de f_c por Micro-Cap 12 (kHz)
2	13.02	13.015
6	4.34	4.357
12	2.17	2.156
16	1.63	1.613
20	1.30	1.319
24	1.08	1.079
40	0.65	0.646
80	0.32	0.319

Apéndice D

Efecto de solapamiento a la salida (Crosstalk)

Como ya se ha explicado, el solapamiento a la salida es un efecto que se puede presentar cuando la placa FPGA genera más de una señal de salida a la vez. Los cambios de una de ellas afectan a la otra, de forma que se pueden observar pequeñas perturbaciones en una señal que coinciden con los cambios que presenta otra de las señales.

En un primer momento, la descripción en VHDL del convertidor se realizó de forma que generara sus dos salidas, *salrRZ* y *salNRZ*, a la vez. De esta forma, si se miden las salidas sin filtrar (y con el osciloscopio de 12 bits, ya que permite alcanzar mayores velocidades de muetsreo), se podrán ver los pulsos que las conforman, tal y como se aprecia en la siguiente imagen:

Figura D.1 Salida conjunta de salRZ y salNRZ.

Como se observa, la señal roja debe corresponderse con la señal en la que no hay retorno a cero, mientras que la azul será la *salRZ.* Aunque a simple vista no es apreciable, los cambios que produce el retorno a cero en la *salRZ* modifican ligeramente el comportamiento de la *salNRZ*, tal y como se puede ver a continuación:

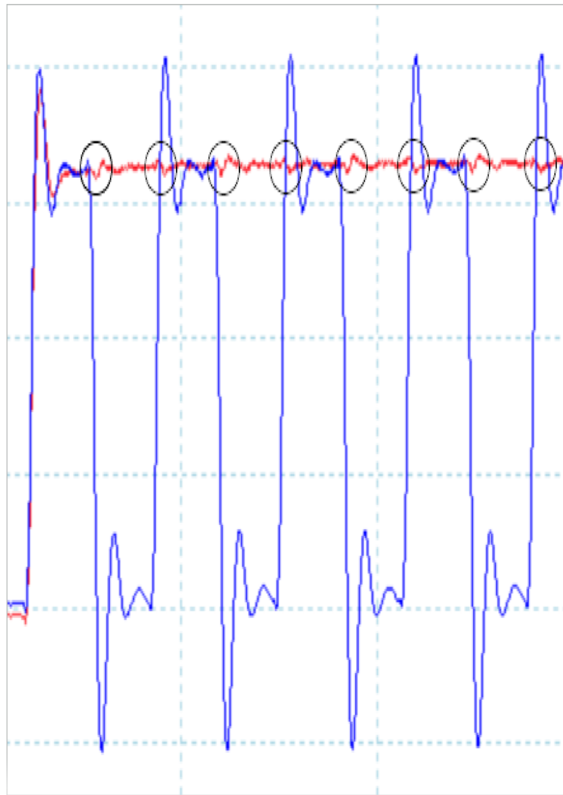

Figura D.2 Efecto del retorno a cero de *salRZ* en *salNRZ*.

Para hacer más evidente este efecto, se muestra a continuación solo el comportamiento de *salNRZ*, aunque es generada a la vez que *salNRZ*, por lo que sus perturbaciones siguen apareciendo:

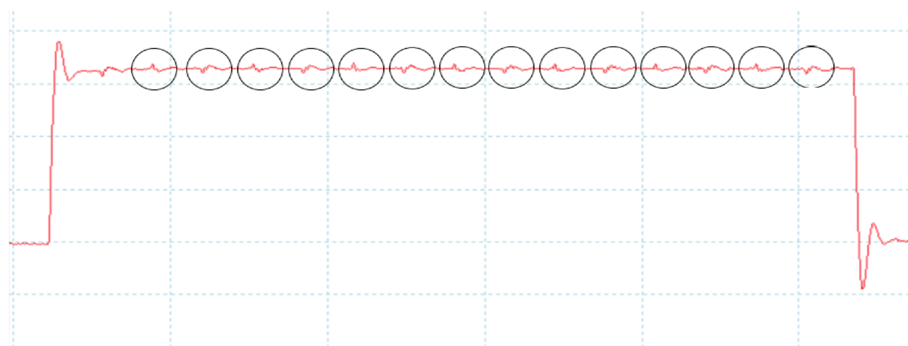

Figura D.3 Efecto de la señal *salRZ* en *salNRZ*.

Como se ve, estos pequeños saltos se producen de forma periódica, con el período aproximado de la señal *salRZ*. Para acabar de justificar que las señales se generen por separado, se muestra en la siguiente imagen el comportamiento de *salrNRZ* (ahora en azul) cuando es generada sin *salRZ*. Se podrá observar que las perturbaciones periódicas han desaparecido, puesto que, como se había adelantado, su efecto estaba producido por la generación simultánea de las dos señales, *salRZ* y *salNRZ*:

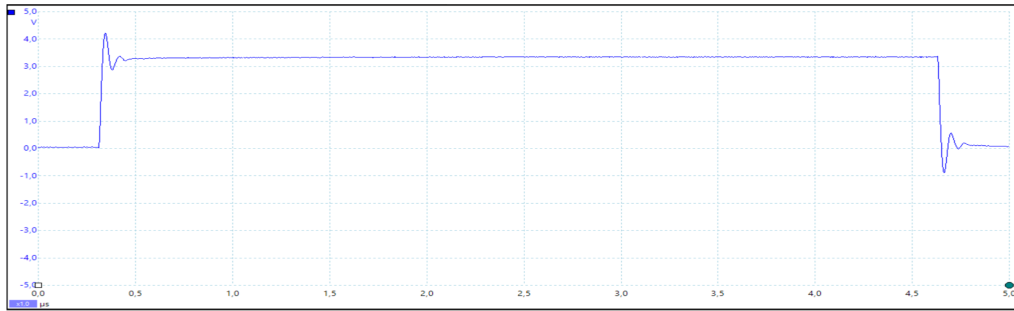

Figura D.4 Obtención de salNRZ sin solapamiento.

De esta forma, se han podido eliminar estas perturbaciones separando el programa que se hizo en un primer momento en 2 , uno que generará la *salRZ* y otro que generará la *salNRZ*. Así, se ha logrado aumentar la calidad de los pulsos (y por lo tanto de las señales) de salida.

Apéndice E

Resultados de las simulaciones para los pines B4 de J1 y A6 de J2

En las siguientes tablas, se pueden observar los valores de **SNR**, **HD2** y **HD3** que se han obtenido al analizar en *MATLAB* los resultados experimentales que ha generado la FPGA con los pines B4 y A6. Como se ha explicado a en el trabajo, para determinar con que configuración se tienen los mejores resultados, se han obtenido todas las salidas posibles para los valores de $N_{PWM} = 2,\ 12\ y\ 20$. Se ha considerado que, al repetirse para los tres valores de N_{PWM} la configuración con la que se obtenían los mejores resultados, estaba justificado considerar que esa configuración sería también la óptima para tomar los datos del resto de valores de N_{PWM} que se han decidido analizar en el trabajo.

Por ello, en las siguientes tablas se ha marcado la configuración con la que se conseguían mejores resultados (en decir, con la que se obtenía el mayor valor de **SNR** posible y mantuviera unos valores de **HD2** y **HD3** similares o superiores al del resto de configuraciones).

Tabla E.1 SNR para N_{PWM}=2 y SALRZ.

SALRZ /SNR	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	39.979	39.742	22.746	22.858	39.868	39.794	22.784	22.775
16 bits	**77.167**	74.001	72.280	73.018	69.589	64.626	72.993	70.605

Tabla E.2 HD2 para N_{PWM}=2 y SALRZ.

SALRZ /HD2	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	54.939	55.076	53.904	55.218	57.800	57.743	57.392	58.672
16 bits	**60.221**	60.300	55.617	55.730	55.635	56.098	55.859	55.767

Tabla E.3 HD3 para N_{PWM}=2 y SALRZ.

SALRZ /HD3	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	74.356	73.925	70.944	71.007	83.047	82.312	70.191	70.178
16 bits	**85.735**	88.181	82.009	85.591	78.228	82.464	88.244	87.026

Tabla E.4 SNR para N_{PWM}=2 y SALNRZ.

SALNRZ /SNR	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	73.265	70.154	66.965	64.637	60.008	54.166	56.963	55.164
16 bits	**76.700**	74.279	75.065	69.938	74.851	73.326	70.265	70.198

Tabla E.5 HD2 para N_{PWM}=2 y SALNRZ.

SALNRZ /HD2	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	55.837	56.058	56.268	55.698	58.703	57.277	56.562	57.642
16 bits	**61.302**	61.545	56.781	56.510	61.406	56.889	56.637	56.963

Tabla E.6 HD3 para N_{PWM}=2 y SALNRZ.

SALNRZ /HD3	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	73.905	73.326	73.914	72.112	71.469	71.594	70.813	69.837
16 bits	**86.868**	89.497	84.665	81.018	81.765	88.457	82.194	86.317

Tabla E.7 SNR para N_{PWM}=12 y SALRZ.

SALRZ /SNR	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	61.235	60.753	51.283	47.366	61.616	60.515	51.764	47.620
16 bits	**71.931**	69.338	67.935	66.603	70.916	69.002	67.094	68.607

Tabla E.8 HD2 para N_{PWM}=12 y SALRZ.

SALRZ /HD2	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	68.817	70.903	50.322	64.644	66.683	73.471	50.331	58.280
16 bits	**78.486**	78.894	75.303	79.340	75.375	78.873	76.027	75.224

Tabla E.9 HD3 para N_{PWM}=12 y SALRZ.

SALRZ /HD3	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	68.911	77.470	58.983	70.587	75.405	78.955	58.093	67.573
16 bits	**79.900**	81.827	79.045	77.766	81.437	84.140	79.852	78.951

Tabla E.10 SNR para N_{PWM}=12 y SALNRZ.

SALNRZ /SNR	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	69.761	69.890	70.759	67.083	57.147	57.851	54.452	57.755
16 bits	**74.804**	70.229	69.898	69.813	70.465	70.464	70.726	69.192

Tabla E.11 HD2 para N_{PWM}=12 y SALNRZ .

SALNRZ /HD2	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	70.022	73.543	70.181	68.813	66.611	69.061	67.531	71.742
16 bits	**74.429**	76.365	73.778	76.164	75.009	76.354	74.987	74.506

Tabla E.12 HD3 para N_{PWM}=12 y SALNRZ.

SALNRZ /HD3	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	73.927	73.142	73.156	73.055	68.127	70.087	67.812	71.246
16 bits	**87.390**	85.543	81.595	79.387	81.700	85.870	77.779	84.860

Tabla E.13 SNR para N_{PWM}=20 y SALRZ.

SALRZ /SNR	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	66.453	65.862	48.360	46.718	63.166	65.634	46.972	55.772
16 bits	**74.249**	69.075	60.300	60.551	64.975	65.044	65.207	61.790

Tabla E.14 HD2 para N_{PWM}=20 y SALRZ.

SALRZ /HD2	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	66.753	70.464	61.047	71.602	66.993	81.391	71.802	56.804
16 bits	**81.125**	86.583	79.824	81.375	83.825	81.618	82.088	84.400

Tabla E.15 HD3 para N_{PWM}=20 y SALRZ.

SALRZ /HD3	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	65.918	63.877	53.073	75.438	64.515	64.022	77.004	47.861
16 bits	**90.864**	91.490	88.210	89.228	89.716	91.317	88.258	87.786

Tabla E.16 SNR para N_{PWM}=20 y SALNRZ.

SALNRZ /SNR	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	72.019	70.193	69.542	68.213	57.864	54.868	58.921	55.351
16 bits	**75.265**	73.247	69.965	71.793	71.751	73.582	73.343	73.974

Tabla E.17 HD2 para N_{PWM}=20 y SALNRZ.

SALNRZ /HD2	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	73.049	73.561	68.503	68.213	69.854	73.492	71.473	74.939
16 bits	**79.830**	78.939	80.284	79.958	79.830	78.939	79.377	79.717

Tabla E.18 HD3 para N_{PWM}=20 y SALNRZ.

SALNRZ /HD3	**CA/X1 /20 per**	CA/X1 /40 per	CA/X10 /20 per	CA/X10 /40 per	CC/X1 /20 per	CC/X1 /40 per	CC/X10 /20 per	CC/X10 /40 per
12 bits	78.158	76.834	72.650	73.570	72.473	73.730	71.264	73.556
16 bits	**91.759**	93.900	90.324	94.929	92.674	94.307	88.980	92.931

Se observa que la configuración más favorable para obtener los resultados es tomarlos con el osciloscopio de 16 bits, en corriente alterna, usando una sonda con atenuación X1 y haciendo los cálculos a 20 periodos.

Por lo tanto, se muestran a continuación los valores calculados para el resto de N_{PWM} con esta configuración, tanto para *SALNRZ* como para *SALRZ*.

Tabla E.19 Valores para N_{PWM}=6 .

NPWM=6	SALNRZ	SALRZ
SNR	75.784	76.927
HD2	70.814	73.122
HD3	86.925	86.019

Tabla E.20 Valores para N_{PWM}=16 .

NPWM=16	SALNRZ	SALRZ
SNR	75.617	74.083
HD2	77.404	82.620
HD3	90.534	90.966

Tabla E.21 Valores para N_{PWM}=24 .

NPWM=24	SALNRZ	SALRZ
SNR	76.181	75.907
HD2	79.310	82.702
HD3	85.627	89.758

Tabla E.22 Valores para N_{PWM}=40 .

NPWM=40	SALNRZ	SALRZ
SNR	78.724	76.129
HD2	77.065	81.207
HD3	89.497	90.583

Tabla E.23 Valores para N_{PWM}=80 .

NPWM=80	SALNRZ	SALRZ
SNR	77.733	75.713
HD2	80.070	82.543
HD3	89.294	84.645

Índice de Figuras

2.1	Ejemplo de señal muestreada	4
2.2	Espectro de una señal muestreada	4
2.3	Espectro de una señal muestreada con filtro	5
2.4	Espectro de una señal muestreada con aliasing	5
2.5	Espectro de una señal muestreada y filtrada con aliasing	5
2.6	Representación del proceso de cuantificar una señal con 3 bits en CA2[1]	8
2.7	Representación de la evolución error de un cuantificador[1]	8
2.8	Representación del comportamiento del ruido con sobremuestreo	10
2.9	Representación de modulador Sigma/Delta de primer orden	11
2.10	Integrador en tiempo discreto	12
2.11	Espectro de la señal y potencia del ruido de cuantificación	12
2.12	Error de cuantificación sin modulador de ruido y con SDM de orden 1 y 2[7]	13
2.13	Ejemplo de espectro de un SDM [8]	14
2.14	Comparación con referencia triangular ideal	14
2.15	Comparación con referencia triangular digital	15
2.16	Ejemplo de obtención de la salida de un PWM	15
2.17	Estructura de corrección para señales de entrada muestreadas [5]	16
3.1	Convertidores en Simulink	18
3.2	Modelo en Simulink del SDM de segundo orden	19
3.3	Modelo 2. Marcado: Ganancia y bloque PWM	19
3.4	PWM en Simulink	20
3.5	Modelo 3	21
3.6	Modelo 4. Introducido: Cuantificador de paso 1	21
3.7	SDM del modelo 4. Introducido: Cuantificador de paso 1 del G_3	22
3.8	Representación espectral de las salidas	23
3.9	Señal de entrada (roja) y salida(azul) del convertidor 4	24
3.10	Bits máximos en cada parte del SDM	25
4.1	Partes del convertidor en Simulink según su función	27
4.2	Esquema del convertidor en VHDL	28
4.3	Componentes del Bloque 1 del convertidor	29
4.4	Formas de onda (reloj y salida) de un divisor de frecuencia cuando se divide por 5	30
4.5	Comportamiento de la señal pulsoRZ	31

4.6 Representación de la señal de entrada 31
4.7 Representación de las relaciones entre frecuencias 32
4.8 Componentes del Bloque 2 del convertidor 34
4.9 Sumador 34
4.10 Restador 35
4.11 Componentes del bloque Cuantificador Q en Simulink 36
4.12 Salida del Cuantificador Q según el valor de qi 36
4.13 Cuantificador LUT en Simulink 37
4.14 Bloque *Retraso tras LUT* en Simulink 38
4.15 Esquema de un biestable 38
4.16 Ejemplo del comportamiento del biestable 39
4.17 Entradas/salidas de los componentes del bloque retraso tras LUT en VHDL 39
4.18 Bloque Retraso tras yn en Simulink 40
4.19 Componentes del bloque *Retraso tras yn* en VHDL 40
4.20 Variabe *P* según la entrada al bloque PWM 41
4.21 Funcionamiento del registro de desplazaamiento 42
4.22 Obtención de salida del registro de desplazaamiento 43
4.23 Componentes del bloque 3 44
4.24 Esquema de señales con y sin retorno a cero 45
4.25 Bloque 4 en VHDL 46
4.26 Obtención de salida con retorno a cero 46
4.27 Evolución temporal entrada al SDM 48
4.28 Espectro de la entrada al SDM 48
4.29 Evolución temporal salida del SDM 49
4.30 Señal espectral de la salida del SDM en VHDL 49
4.31 Salida del convertidor en el dominio del tiempo 50
4.32 Espectro salida del convertidor 50
4.33 Pines asociados a las señales *clk*, *resasin*, *salRZ* y *salNRZ* 51
4.34 FPGA, pines de salida y puertos de conexión 52
4.35 Indicación de generación correcta del archivo Bitstream 53

5.1 Filtro paso bajo en Micro-Cap 56
5.2 Filtro paso bajo en placa de pruebas 57
5.3 Señal salNRZ para N_{PWM}=2 en el dominio del tiempo 58
5.4 Señal espectral salNRZ para N_{PWM}=2 58
5.5 Señal salRZ para N_{PWM}=2 en el dominio del tiempo 59
5.6 Señal espectral salRZ para N_{PWM}=2 59
5.7 Señal salNRZ para N_{PWM}=12 en el dominio del tiempo 60
5.8 Señal espectral salNRZ para N_{PWM}=12 60
5.9 Señal salRZ para N_{PWM}=12 en el dominio del tiempo 61
5.10 Señal espectral salRZ para N_{PWM}=12 61
5.11 Señal salNRZ para N_{PWM}=20 en el dominio del tiempo 62
5.12 Señal espectral salNRZ para N_{PWM}=20 62
5.13 Señal salRZ para N_{PWM}=20 en el dominio del tiempo 62
5.14 Señal espectral salRZ para N_{PWM}=20 63
5.15 Evolución de la SNR y HD2 con la frecuencia 64
5.16 Evolución de la HD3 con la frecuencia 64

5.17 Pulsos con retorno a cero 65
5.18 Pulsos de la señal salRZ 65
5.19 Pulsos sin retorno a cero 65
5.20 Pulsos de la señal salNRZ 66
5.21 Picos de los pulsos 66
5.22 Pines del conector J3 67
5.23 Pieza que permite conexión con J3 68
5.24 Pines asociados a *clk*, *resasin* y salNRZ 68

C.1 Ganancia del filtro paso bajo con $N_{PWM} = 2$ 125
C.2 Ganancia del filtro paso bajo con $N_{PWM} = 6$ 125
C.3 Ganancia del filtro paso bajo con $N_{PWM} = 12$ 126
C.4 Ganancia del filtro paso bajo con $N_{PWM} = 16$ 126
C.5 Ganancia del filtro paso bajo con $N_{PWM} = 20$ 126
C.6 Ganancia del filtro paso bajo con $N_{PWM} = 24$ 127
C.7 Ganancia del filtro paso bajo con $N_{PWM} = 40$ 127
C.8 Ganancia del filtro paso bajo con $N_{PWM} = 80$ 127

D.1 Salida conjunta de salRZ y salNRZ 129
D.2 Efecto del retorno a cero de *salRZ* en *salNRZ* 130
D.3 Efecto de la señal *salRZ* en *salNRZ* 130
D.4 Obtención de salNRZ sin solapamiento 131

Índice de Tablas

2.1 Codificación en CA2 para 3 bits 7

3.1 Comparación de los valores de la SNR de cada modelo 22
3.2 Comparación de los valores de la *HD2* de cada modelo 23
3.3 Comparación de los valores de la *HD3* de cada modelo 23

4.1 Criterio para identificar el desbordamiento en el caso de la suma 35
4.2 Criterio para identificar el desbordamiento en el caso de la resta 35
4.3 Posibles entradas y salidas del Cuantificador LUT 37
4.4 Ejemplo comportamiento señal P 42
4.5 Frecuencias de trabajo 47
4.6 SNR y HD2 a la entrada del SDM 48
4.7 *SNR* y *HD2* de la salida del SDM 49
4.8 *SNR*, *HD2* y *HD3* de la salida del *PWM* 50
4.9 *SNR*, *HD2* y *HD3* de la salida del *PWM* Post-Implementación 51

5.1 Características de los osciloscopios 55
5.2 Resistencias teóricas para cada valor de N_{PWM} 56
5.3 Resistencias disponibles para cada valor de N_{PWM} 56
5.4 Frecuencias teóricas necesarias para cada valor de N_{PWM} 57
5.5 Frecuencias para $N_{PWM} = 2$ 58
5.6 Resultados para $N_{PWM} = 2$ 59
5.7 Frecuencias para $N_{PWM} = 12$ 60
5.8 Resultados para $N_{PWM} = 12$ 61
5.9 Frecuencias para $N_{PWM} = 20$ 61
5.10 Resultados para $N_{PWM} = 20$ 63

C.1 Comparación del valor de las frecuencias de los filtros 128

E.1 SNR para N_{PWM}=2 y SALRZ 133
E.2 HD2 para N_{PWM}=2 y SALRZ 133
E.3 HD3 para N_{PWM}=2 y SALRZ 134
E.4 SNR para N_{PWM}=2 y SALNRZ 134
E.5 HD2 para N_{PWM}=2 y SALNRZ 134
E.6 HD3 para N_{PWM}=2 y SALNRZ 134

E.7 SNR para N_{PWM}=12 y SALRZ 134
E.8 HD2 para N_{PWM}=12 y SALRZ 134
E.9 HD3 para N_{PWM}=12 y SALRZ 134
E.10 SNR para N_{PWM}=12 y SALNRZ 135
E.11 HD2 para N_{PWM}=12 y SALNRZ 135
E.12 HD3 para N_{PWM}=12 y SALNRZ 135
E.13 SNR para N_{PWM}=20 y SALRZ 135
E.14 HD2 para N_{PWM}=20 y SALRZ 135
E.15 HD3 para N_{PWM}=20 y SALRZ 135
E.16 SNR para N_{PWM}=20 y SALNRZ 135
E.17 HD2 para N_{PWM}=20 y SALNRZ 136
E.18 HD3 para N_{PWM}=20 y SALNRZ 136
E.19 Valores para N_{PWM}=6 136
E.20 Valores para N_{PWM}=16 136
E.21 Valores para N_{PWM}=24 136
E.22 Valores para N_{PWM}=40 137
E.23 Valores para N_{PWM}=80 137

Índice de Códigos

A.1 generadorentrada 71
A.2 generadorentrada 75
A.3 Análisis entrada y salida SDM 76
A.4 Análisis salida PWM 78
A.5 Análisis salida FPGA 79

B.1 top 81
B.2 divisorfrec 86
B.3 contn 87
B.4 genpulRZ 89
B.5 valores 89
B.6 sdm 98
B.7 restadorA2 103
B.8 sumar 103
B.9 cuantificadorpasoq 104
B.10 lut 108
B.11 rtlut 110
B.12 zmenos1 113
B.13 rtyn 113
B.14 PWM 115
B.15 definirP 117
B.16 salidaD 119
B.17 Simulación 120

Bibliografía

[1] Francisco Colodro Ruiz, *Fundamentos de la conversión A/D y D/A*, Universidad de Sevilla.

[2] Behzad Razavi, *The z-Transform for Analog Designers*, *IEEE SOLID-STATE CIRCUITS MAGAZINE*, 2020.

[3] Ian Galton, *Delta–Sigma Data Conversion in Wireless Transceivers*, *IEEE TRANSACTIONS ON MICROWAVE THEORY AND TECHNIQUES,VOL. 50*, 2002.

[4] Marcos Sánchez-Élez, *Introducción a la programación en VHDL*, Facultad de Informática Universidad Complutense de Madrid.

[5] Francisco Colodro Ruiz, *Digital correction of errors and harmonic distortion in pulse-width modulation of digital signals*, Universidad de Sevilla.

[6] *MicroBlaze Development Kit Spartan-3E 1600E Edition User Guide*, v1.1, Diciembre 2007.

[7] Barry L. Shoop y Pankaj K. Das, *Mismatch-tolerant distributed photonic analog-to-digital conversion using spatial oversampling and spectral noise shaping*, *Optical Engineering 41(7)*, Julio 2002.

[8] Otero Naranjo, R. (2021). Realización de un convertidor Digital/Analógico con 12 bits de resolución y 20kHz de ancho de banda basado en un modulador Sigma/Delta de segundo orden y un bit de salida. (Trabajo Fin de Grado Inédito). Universidad de Sevilla, Sevilla.