

UNIVERSIDAD POLITÉCNICA DE MADRID
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE
TELECOMUNICACIÓN



TRABAJO FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN SOBRE FPGA DE UN PEDAL DE EFECTOS DIGITAL

GRADO EN INGENIERÍA DE TECNOLOGÍAS
Y SERVICIOS DE TELECOMUNICACIÓN

JAVIER OTERO MARTÍNEZ
Madrid, Junio 2019

DISEÑO E IMPLEMENTACIÓN SOBRE FPGA DE UN PEDAL DE EFECTOS DIGITAL

Autor: Javier Otero Martínez

Tutor: Pablo Ituero Herrero
Departamento de Ingeniería Electrónica

Me gustaría agradecer a los que me han apoyado todo este tiempo: mis padres, mi familia, Aida e Impostor mientras trabajaba en el proyecto. También me gustaría agradecer especialmente a los profesores que me han ayudado desinteresadamente cuando he acudido a ellos con alguna pregunta: Alfredo Sanz, Jose Parera, Fernando Gonzalez y por supuesto a mi tutor, Pablo, por toda su paciencia estos meses.

Resumen

El presente proyecto se enmarca en el ámbito de la producción musical en tiempo real. En concreto, plantea el diseño, desarrollo e implementación de un pedal de efectos digital, cuyo efecto principal será el de octavador. Estos dispositivos permiten variar la señal entrante en tiempo real obteniendo a la salida diferentes efectos y modulaciones.

Un octavador es un dispositivo que entrega a la salida un señal que es idéntica (idealmente) a la señal de entrada salvo que su frecuencia está dividida por 2, añadiendo un carácter mucho más sólido y profundo al sonido resultante de la mezcla de ambos.

El proyecto plantea el proyecto desde cero, es decir, desde la idea original hasta la implementación de un prototipo. Todas las decisiones y criterios de diseño se llevan a cabo pensando en su utilización profesional, refiriéndome tanto a latencia como calidad del sonido, formato, latencia y el resto de parámetros involucrados.

En consecuencia, el resultado final ha sido una evaluación de diferentes algoritmos en base a sus diferentes propiedades y características para posteriormente realizar una propuesta de implementación en VHDL utilizando la herramienta Vivado. Estos conocimientos se han plasmado en un prototipo que me ha ayudado a afianzar estos conceptos y ha mejorado mi comprensión tanto del tratamiento de señales de audio como de la implementación de un algoritmo en FPGA.

Abstract

This project belongs to the real-time music production context. Concretely, it addresses design, development and implementation of a digital effects pedal, which main feature will be an octavizer. Such devices allow us to modify the input signal in real time obtaining a variety of different effects and modulations as the outcome.

An octavizer provides a signal which is identical (ideally) to the introduced one except that its frequencies are divided by two, adding a much more solid and deep nature to the resulting mix.

This work is carried out from zero, indeed, from the original idea to the implementation of the prototype. Every single design choice and criteria has been made thinking in its professional use, refereing to either latency, audio quality, layout and all other parameters involved.

Consequently, the outcome was an evaluation of different algorithms based on its properties and characteristics to finally make an VHDL implementation proposal using Vivado tool. This acknowledgements have been put into practise via designing a prototype that helped me to ground such concepts and improved my comprehension of both audio signal proccessing and algorithm implementation on FPGA.

Palabras clave

Pedal de efectos	Tiempo real
FPGA	Algoritmo
Instrumento musical	Vivado
Vocoder	Fourier
Matlab	Ellis
VHDL	Oppenheim
Hardware	Fase
Nexys A7	Latencia
Pmod i2-s2	Controlador
Procesado de señal	Ir añadiendo
Procesado de audio	

Key words

Pedal de efectos	Tiempo real
FPGA	Algoritmo
Instrumento musical	Vivado
Vocoder	Fourier
Matlab	Ellis
VHDL	Oppenheim
Hardware	Fase
Nexys A7	Latencia
Pmod i2-s2	Controlador
Procesado de señal	Ir añadiendo
Procesado de audio	

Índice general

1. Introducción y generalidades	1
1.1. Objetivos	2
1.2. Metodología	2
1.3. Resultados	3
1.4. Consecución de los objetivos propuestos	4
2. Entrada: etapa analógica	5
2.1. Captación de sonido: selección de micrófono	5
2.2. Etapa de preamp o preamplificación	8
2.3. Alimentación	10
3. El algoritmo	11
3.1. Fundamentos del sonido y teoría musical	11
3.1.1. Timbre, armónicos y serie armónica	12
3.2. Aproximaciones a la octavación	13
3.2.1. Prescindiendo de Fourier	13
3.2.2. Retorno a la transformada	17
3.3. Vocoder	17
3.3.1. Vocoder en la música	18
3.4. Transformación a frecuencia: STFT	19
3.4.1. Transformada de Fourier: FFT e iFFT	19
3.4.2. Solapamiento y enventanado	20
3.5. Operaciones sobre la fase	22
3.5.1. Cálculo del módulo	23
3.5.2. Recalculando la fase	23
4. Implementación	25
4.1. Gestión entrada-salida	26
4.1.1. Pmod i2-s2	27

4.1.2. Consideraciones de la implementación del Pmod	28
4.2. Controlador de datos	30
4.2.1. Implementación del solapamiento	31
4.2.2. Bancos de memorias volátiles	32
4.2.3. Core FFT e iFFT	33
4.3. Controlador global	36
4.3.1. Displays	37
5. Pruebas, depuración y medidas	38
5.1. Circuito analógico de entrada	38
5.1.1. Saturación y dependencia con la tensión de entrada	38
5.1.2. Dependencia de la ganancia con la frecuencia	39
5.1.3. Otros parámetros	39
5.2. Algoritmo de <i>Vocoder de fase</i>	39
5.3. Sobre la implementación en FPGA	41
6. Conclusiones y trabajo futuro	42
Apéndices	
A. Código del algoritmo en MATLAB	44
B. Código prueba del algoritmo en MATLAB	47
C. Script de generación de valores para las ROM	49

Capítulo 1

Introducción y generalidades

Un pedal de efectos es un dispositivo que se conecta entre un instrumento (normalmente electrófono) y su amplificador encargándose de modificar la señal de entrada y sus características fundamentales como pueden ser timbre, tono y volumen. En este proyecto se pretende estudiar las posibilidades de diseño e implementación de uno de estos pedales, enfocándolo a un uso con instrumentos de viento, en concreto el saxofón puesto que es el instrumento que yo toco. Generalmente, no es habitual el uso de pedales de efectos en instrumentos de viento, ya que en general se busca mantener el sonido lo más fiel posible al producido por el instrumento. No obstante, en multitud de producciones se pueden apreciar efectos añadidos en postproducción ya sea digital o analógicamente. Algunos ejemplos son el *reverb* o el *chorus*. Sin embargo, aquí se pretende implementar un efecto de octavador, el cual se describirá posteriormente.

He decidido el combinar este efecto con la implementación en formato de pedal. Este formato se ha hecho muy popular desde su aparición, debido a que los intérpretes pueden activarlo con el pie pudiendo mantener las manos en el instrumento. Aunque los intérpretes de instrumentos de viento no están acostumbrados al uso de pedales, se mantiene la idea del pedal por analogía con otros instrumentos.

Este proyecto abarca todo el proceso desde la idea inicial, diseño y montaje del prototipo final, por tanto, las especificaciones de funcionamiento que se han utilizado pretenden facilitar un uso profesional del prototipo, de forma que sea compatible con los estándares establecidos en el contexto musical e ingenieril al mismo tiempo.

El flujo de datos será el siguiente. En primer lugar se utiliza un transductor para adquirir la señal, en este caso, un micrófono convencional. Una vez que el estímulo externo es transformado en pulsos eléctricos, atravesará un etapa de entrada analógica que preamplifica la señal y la adecúa a la entrada de la FPGA.

Para el prototipo se ha utilizado la placa proporcionada por el departamento: Nexys A7. Esta placa monta una FPGA *Xilinx XC7A100T-1CSG324C* junto con varios switches, botones, leds y displays de 7 segmentos, que harán más fácil el manejo de la misma. Esta placa tiene un micrófono integrado, pero es de tan baja calidad que se opta por diseñar la etapa de entrada, analógica completamente, y conectarla con uno de los puertos del *Pmod i2s2*, también de Digilent y proporcionado por el departamento. Este módulo contiene ADC, DAC y los conectores de mini-jack estándar en formato de audio, que servirán para gestionar la señal de entrada y la de salida.

Se implementará un algoritmo que se encargará de llevar a cabo la octavación de la señal de entrada y de proporcionarla en la salida. Este algoritmo utiliza una aproximación de *Phase Vocoder* muy común en el tratamiento de señales de audio realizando una transformación al dominio de la frecuencia mediante FFT. Durante todo el proceso se priorizará el criterio de la *baja latencia*, dado que si no, resulta imposible operar en tiempo real. El algoritmo se describirá en profundidad en su capítulo correspondiente.

Como se puede deducir el tema elegido se ha acordado directamente con el tutor sin que estuviera previamente propuesto por él o por el departamento.

1.1. Objetivos

Para la realización de este proyecto de forma satisfactoria se ha establecido la consecución de una serie de objetivos que son los siguientes:

- Diseño de un sistema completo, a partir de una problemática definida previamente mediante el estudio del instrumento y las señales.
- Implementación sobre la FPGA proporcionada.
- Construcción de un prototipo para estudiar la problemática desde la práctica.
- Afianzar, debido a todo esto, conocimientos adquiridos durante el Grado en diversos ámbitos como el procesamiento de señal y en concreto en la especialidad de Sistemas Electrónicos como la programación hardware, el montaje de un circuito analógico y la correcta comunicación entre todos los módulos.

1.2. Metodología

- En primer lugar se seleccionará el efecto que se quieren implementar, dando prioridad al octavador.
- Como segundo paso, se estudiará la literatura existente y se probarán distintas soluciones algorítmicas empleando MATLAB.
- Estudio y elección de la interfaces de entrada y salida. Selección de micrófono, ADC y DAC.
- Desarrollo y verificación en VHDL empleando la herramienta VIVADO de Xilinx.
- Montaje de un prototipo empleando la placa Nexys A7 de Digilent, el micrófono seleccionado anteriormente y el resto de dispositivos que fueran necesarios.
- Test y depuración.

1.3. Resultados

Transcurrido el tiempo de desarrollo del proyecto, que ha sido de un año natural, se han alcanzado varias metas aunque no se ha podido ver terminada una versión funcional del prototipo.

- Tras el periodo de investigación, se ha seleccionado un algoritmo y se ha implementado en Matlab. Este código es plenamente funcional y se encuentra en el apéndice A.
- Se ha montado un circuito analógico que funciona como etapa de entrada, amplificando la señal musical y adecuándola para su correcta interpretación por parte de la FPGA.
- Aunque no se tiene una versión completa de la implementación VHDL del algoritmo elegido, si que se han diseñado varias partes para poder estimar el funcionamiento final del sistema completo así como su arquitectura.

Dado que el proceso de realización del trabajo ha sido de un año natural, se han empleado muchas horas en cada una de las diferentes partes del mismo. Para comprender la magnitud del presente proyecto, se adjunta un desglose aproximado de las horas de trabajo en siguiente tabla:

Algoritmo	150h
Investigación de los diferentes algoritmos	50h
Comparativa y evaluación de los mismos	70h
Pruebas en Matlab	30h
Círculo analógico	20h
Fase de documentación y montaje del circuito elegido	15h
Pruebas y medidas	5h
Implementación	320h
Caracterización y puesta en marcha de la entrada y salida de datos	70h
Implementación del algoritmo de Matlab en VHDL usando Vivado	100h
Depuración y pruebas	150h
Otros	50h
Medidas y estimaciones sobre el prototipo final	10h
Realización de la memoria	40h
TOTAL	540h

Esta aproximación se ha hecho en base a la fecha de los *commit* de Github realizados asumiendo una media de 12 horas de trabajo a la semana.

1.4. Consecución de los objetivos propuestos

Una vez concluido el tiempo de trabajo se pueden hacer las siguientes afirmaciones en referencia al grado de consecución de los objetivos previamente propuestos, aunque se detallen en el apartado de conclusiones:

- Se ha diseñado el sistema en su totalidad, atendiendo a criterios técnicos debidamente justificados en este documento.
- Aunque no se ha llegado a implementar el sistema en su totalidad, sí que se ha trabajado en este aspecto en profundidad, permitiendo una estimación del funcionamiento de los módulos restantes que sí que se han tenido en cuenta en el diseño.
- El prototipo evidentemente no es plenamente funcional pero sí que ha permitido la prueba de los diferentes aspectos que se han ido implementando.
- Debido al trabajo de investigación realizado, se han adquirido nuevas ideas relacionadas con estos estos ámbitos que han sido puestas en práctica de forma crítica y se han asentado los conocimientos relacionados adquiridos durante el grado.

Capítulo 2

Entrada: etapa analógica

El pedal de efectos tiene una estructura definida de la siguiente manera: la señal captada atravesará una etapa a la entrada que se encarga de amplificarla y adecuarla para introducirla en la FPGA, donde se procesará. Por último, el resultado será reconvertido en analógico y amplificado para permitir su reproducción.

2.1. Captación de sonido: selección de micrófono

Como normalmente se utilizan pedales de efectos en instrumentos electrófonos, la señal de salida del instrumento ya viaja por un cable de camino a la amplificador, el pedal actúa por tanto como un intermediario entre ambos. Sin embargo, en instrumentos de viento, es necesario utilizar un transductor que sea capaz de convertir la señal acústica consistente en ondas de presión en una serie de impulsos eléctricos que puedan ser procesados posteriormente.

La solución más sencilla consistiría en utilizar el micrófono que viene integrado con la placa Nexys A7: modelo *ADMP421* de Analog Devices [9]. No obstante, la utilización de este micrófono plantea los siguientes dos problemas.

En primer lugar, es inmediato pensar que incluso en el caso de un prototipo, si se plantea usar como pedal, no resulta nada recomendable colocar el micrófono encargado de recoger todo el sonido en el suelo. Además de estar lejos de la fuente sonora, capturaría el sonido resultante de la manipulación de los controles suponiendo una bajada en la calidad que proporcionase el dispositivo. Por tanto, es mejor utilizar un micrófono no integrado en la propia placa.

En segundo lugar, pero no menos importante, conviene tener en cuenta que la respuesta de sensibilidad del micrófono integrado es omnidireccional (ver figura 2.1), es decir, captará todos los sonidos sin importar la dirección de dónde vengan. Este tipo de transductores se usan principalmente en radio y televisión, donde puede haber varias personas hablando en el mismo micrófono o para la grabación de orquestas o agrupaciones en localizaciones cerradas determinadas. Estos micrófonos son capaces de captar tanto el sonido proveniente de la fuente como los ecos y reflexiones característicos del espacio, dando una sensación de amplitud al oyente como la que produciría su escucha en esa misma localización. Pero mientras que en estos casos se trata de un efecto deseado, resulta poco agradable captar

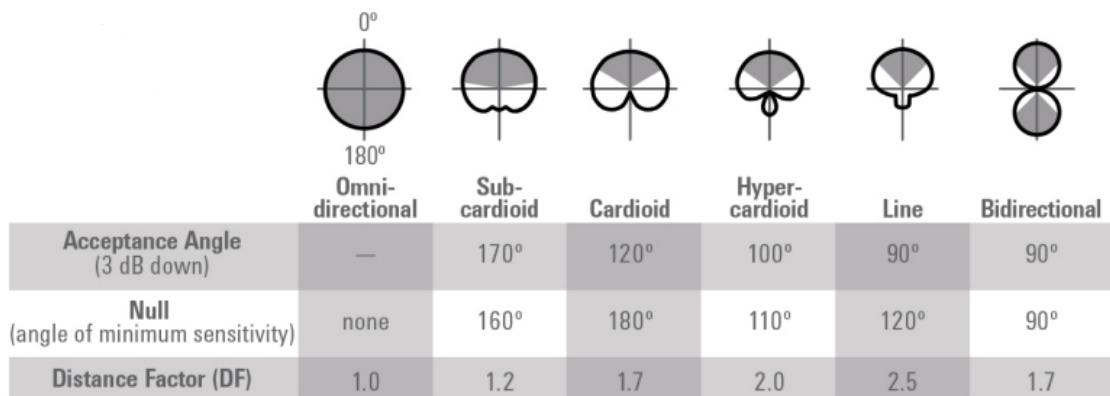


Figura 2.1: Tipos de micrófonos según su diagrama polar [2]

estas reflexiones en un ambiente no preparado para ello y encima cercano al suelo, siendo más conveniente utilizar micrófonos de tipo cardioide.

Estos micrófonos los más utilizados con instrumentos de viento, ya que el sonido suele provenir de un punto concreto. Es preciso matizar que en el caso del saxofón, contra la creencia popular, el sonido no sale siempre por la campana del instrumento. El sonido sale, por el contrario, por la llave abierta más próxima a la embocadura, que suele estar en el centro del cuerpo del instrumento. En consecuencia, no resulta conveniente acercar el micrófono mucho a la campana descuidando otras llaves. La figura 2.2 muestra un saxofón alto aunque hay varios instrumentos de la misma familia.

Además de tener en cuenta el diagrama polar para la elección de micrófono, resulta imprescindible conocer los posibles tipos en cuanto a fabricación y funcionamiento. En este



Figura 2.2: Saxofón alto

sentido tenemos los dos tipos más comunes: los micrófonos de condensador y los dinámicos los cuales se describirán brevemente teniendo en cuenta el criterio de *Shure*, uno de los fabricantes de audio profesional más reconocidos del mercado [ojo].

Los micrófonos de condensador reciben su nombre del condensador que poseen en el interior de su cápsula. El diafragma es la pieza encargada de vibrar cuando las ondas de presión del sonido lo atraviesan. Esta pieza se une con una de las placas del condensador. El resultado es que con cada movimiento del diafragma varía la distancia entre las placas y en consecuencia, se modifica la capacidad del conjunto de manera inversamente proporcional. Estos cambios modifican una señal eléctrica en la que quedan registradas las ondas de sonido recibidas. Estos micrófonos poseen una buena sensibilidad pero necesitan de alimentación *phantom* de entre 24 y 48 V, que se realiza desde la mesa de mezclas por el mismo cable de señal.

Los micrófonos dinámicos por el contrario, no requieren de alimentación, poseen buena robustez y son más baratos que los anteriores. Estos funcionan gracias a una bobina unida al diafragma que se mueve conforme a las ondas de presión recibidas del sonido dentro de un campo electromagnético. Por la acción de la inducción electromagnética se genera una corriente que atravesará la bobina de manera proporcional al estímulo entrante. La principal desventaja de estos micrófonos es que su respuesta no es del todo lineal con la frecuencia, produciendo mayor o menor ganancia en función del rango del espectro en el que se encuentre el sonido de entrada. Para compensar este efecto se suele usar ecualización posterior o diferentes diafragmas para cada rango del espectro de forma que se pueda reconstruir la señal original a base de sumas de los diferentes tramos.



Figura 2.3: Micrófono utilizado junto con el cable XLR

Teniendo en cuenta su uso en el diseño, he escogido un micrófono dinámico por la robustez que presenta que además evitara tener que preocuparse de la alimentación. Adicionalmente, estos micrófonos funcionan de manera muy similar entre sí, lo que resultará muy conveniente para mantener la flexibilidad del proyecto. El micrófono elegido finalmente será un *T.Bone MB60* [ojo] cortesía del Club Musical Delta, mostrado en la figura 2.3.

Este se colocará para capturar los sonidos en un pie en la ubicación más adecuada para el instrumento y se conectarán a la etapa siguiente ubicada en el suelo mediante un cable XLR, del que se hablará más adelante.

En cualquier caso, es necesario añadir una etapa posterior de *pre-amplificación* o *preamp* que adecúe la señal eléctrica del micrófono a otra que pueda interpretar la FPGA.

2.2. Etapa de preamp o preamplificación

La función principal que va a realizar la etapa de preamplificación será la de transformar la señal balanceada proveniente del micrófono en una sin balancear. Se realizará de manera puramente analógica aprovechando que el esquema está muy desarrollado en la literatura sobre el tema. En este caso he implementado el modelo propuesto por P.Allison en [1], el cual fue sugerido por el profesor Alfredo Sanz Hervás al que agradezco su referencia.

Típicamente, los modelos de micrófonos comerciales para aplicaciones de música, utilizan conexiones balanceadas para proporcionar su señal a la salida. El formato de estos cables de 3 hilos recibe el nombre de *XLR* pero también se pueden utilizar conectores de tipo *Jack de 6.35 mm*¹ adecuados a este tipo de señales, ilustrados en la figura 2.4.



Figura 2.4: Conector tipo XLR hembra y jack no balanceado macho de 6.35 mm

El fundamento del par balanceado consiste en enviar la señal por dos conductores entrelazados con la polaridad invertida entre sí, envueltos de un tercer conductor que actúa de barrera frente a interferencias electromagnéticas externas. De esta forma, si se produce una interferencia, afectará a ambos conductores de igual manera. Posteriormente, se introduce la señal en destino en un *Amplificador de diferencias*² que se encargará de amplificar únicamente la diferencia entre ambas señales rechazando la interferencia sufrida. Esta idea muy extendida en los diseños analógicos, permite enviar la señal por largos cables de manera robusta a cambio de introducir un tercer conductor.

Así pues, el diseño de pre-amplificación contará un amplificador de diferencia realizado con un amplificador operacional junto con una pequeña etapa de amplificación mediante

¹Este tipo de conector pero sin balancear, es el utilizado en guitarras y bajos eléctricos. Para los auriculares se utiliza jack de 3.5 mm que recibe el nombre de “minijack”.

²Traducción literal del inglés *Difference Amplifier*, la traducción al castellano puede inducir a error

transistores BJT, tal y como se puede ver en el esquema del mismo de la figura 2.5. El potenciómetro permite controlar la ganancia del sistema, es decir, el nivel de señal a la salida del mismo.

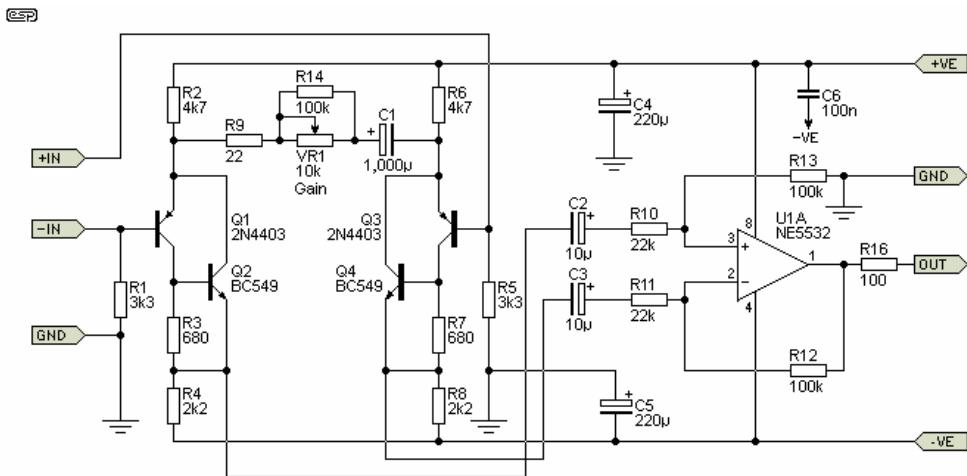


Figura 2.5: Esquema del circuito implementado

La elección de los transistores y el amplificador operacional pueden condicionar en gran medida el resultado obtenido: es importante que todos los componentes sean robustos frente al ruido. Además, la distorsión que introduzcan puede resultar más o menos agradable al oído. En consecuencia se ha seleccionado un *TL071* [ojo] por sus características frente al ruido y su reducido precio. Los transistores han sido los mismos que proponía el diseño de P.Allison.

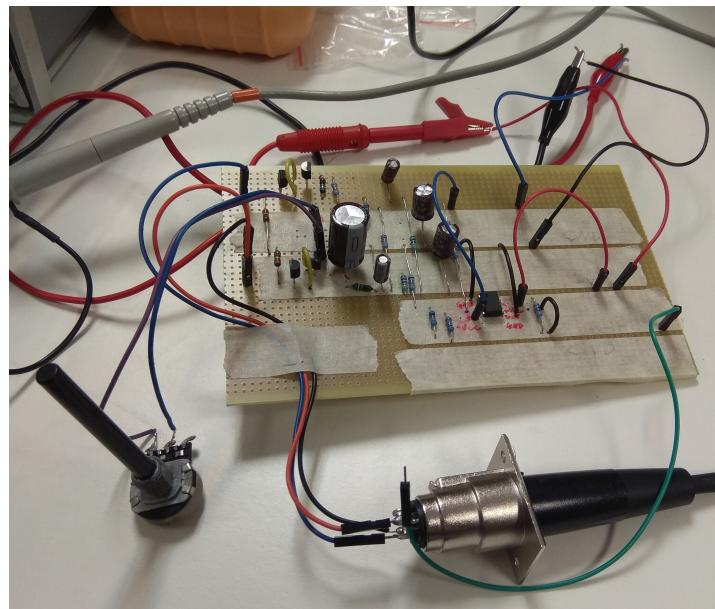


Figura 2.6: Circuito implementado

2.3. Alimentación

Para el correcto funcionamiento del circuito es necesario alimentar el operacional y proporcionar la polarización adecuada a los transistores. Aunque la bibliografía consultada es ambigua con este tema, finalmente se ha optado por utilizar una alimentación simétrica de $\pm 15V$, que será proporcionada directamente de una fuente de alimentación del laboratorio (figura 2.7). La figura 2.6 muestra el circuito una vez montado aunque la caracterización del mismo se describe posteriormente en el punto 5.1.

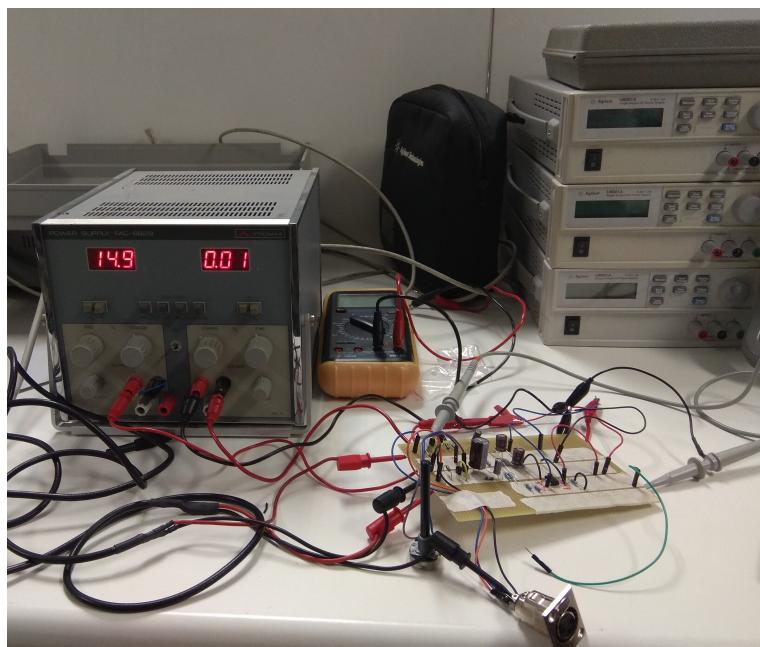


Figura 2.7: Fuente de alimentación simétrica empleada

Capítulo 3

El algoritmo

Para comenzar considero necesario hacer un pequeña introducción a algunos conceptos que tendrán importancia durante el transcurso del presente documento en relación con la física del sonido y la teoría musical, para facilitar la comprensión del documento por un público no especializado en estos ámbitos. El capítulo tercero de [3] contiene gran información al respecto, alguna de la cual se detalla a continuación.

3.1. Fundamentos del sonido y teoría musical

En primer lugar cabe destacar que como cualquier fenómeno físico, un sonido lleva asociados una serie de parámetros matemáticos que lo definen, pero tradicionalmente, estos términos han recibido otro nombre en el ámbito musical, con el que se enseñan en escuelas y conservatorios. Es importante, por tanto, tener clara la relación entre la nomenclatura musical y la equivalencia física y matemática.

- **Tono:** hace referencia a la altura de la nota en cuanto a grave o aguda. En parámetro que la define es la frecuencia, que se mide en Hercios. Aunque el oído humano puede llegar a percibir señales de hasta 20kHz, las notas fundamentales rara vez sobrepasan los 2kHz.
- **Volumen:** equivale a la intensidad que posee la señal tanto eléctricamente como cuando se propaga por el aire como una onda de presión. Cuando se mide en un circuito eléctrico se utilizan unidades de tensión mientras que como onda se suelen utilizar los dB dado su carácter logarítmico, sin embargo, posee unidades de intensidad acústica: W/m^2 .
- **Timbre:** es característico de cada fuente de sonido y es lo que nos permite diferenciar un instrumento de otro, o voces humanas entre si, aún produciendo la misma nota.

La lista de superior muestra los tres parámetros fundamentales de un sonido musical, sin embargo, no queda claro como funciona el mecanismo matemático que rige el funcionamiento del más importante de ellos: el timbre.

3.1.1. Timbre, armónicos y serie armónica

Cualquier estudiante de ingeniería está familiarizado con la generación de armónicos en el contexto de la física y las vibraciones. Sin embargo, puede que no tantos se hayan preguntado como se oyen estos modos o si se pueden percibir. La respuesta es sí, y además con mucha claridad [19]. En lo que se refiere a la música, tanto los instrumentos musicales como la voz humana son estructuras complejas que producen sonidos a muchas frecuencias diferentes al mismo tiempo. Nuestro cerebro, cuando oye un sonido se encarga de interpretar la señal recibida para producir un sonido identificable.

Es inevitable establecer una analogía entre este funcionamiento y el teorema de Fourier, que dice que podemos descomponer cualquier señal como suma de una o varias componentes sinusoidales. Esta poderosa afirmación explica como funciona la generación de cualquier sonido musical.

Así pues, cuando se produce una nota con un instrumento, se generan varias vibraciones a distintas frecuencias, en consecuencia, es la relación entre estas componentes lo que modifica el timbre de la nota generada y nos permite identificarla [8]. La trampa está en que las frecuencias que se producen no son aleatorias, son una serie de notas que reciben el nombre de *armónicos*. El conjunto de estos armónicos se llama *serie armónica*.

Así pues, podemos concluir con que cada nota está a su vez formada por varios armónicos presentándose en diferente proporción entre ellos. El armónico que define el sonido es el primero de la serie y recibe el nombre de fundamental, el resto se obtienen (idealmente [18]) multiplicando esta frecuencia por números enteros. En la imagen 3.1 vemos como un violín y una viola generan una mezcla de armónicos diferente cuando interpretan la mis-

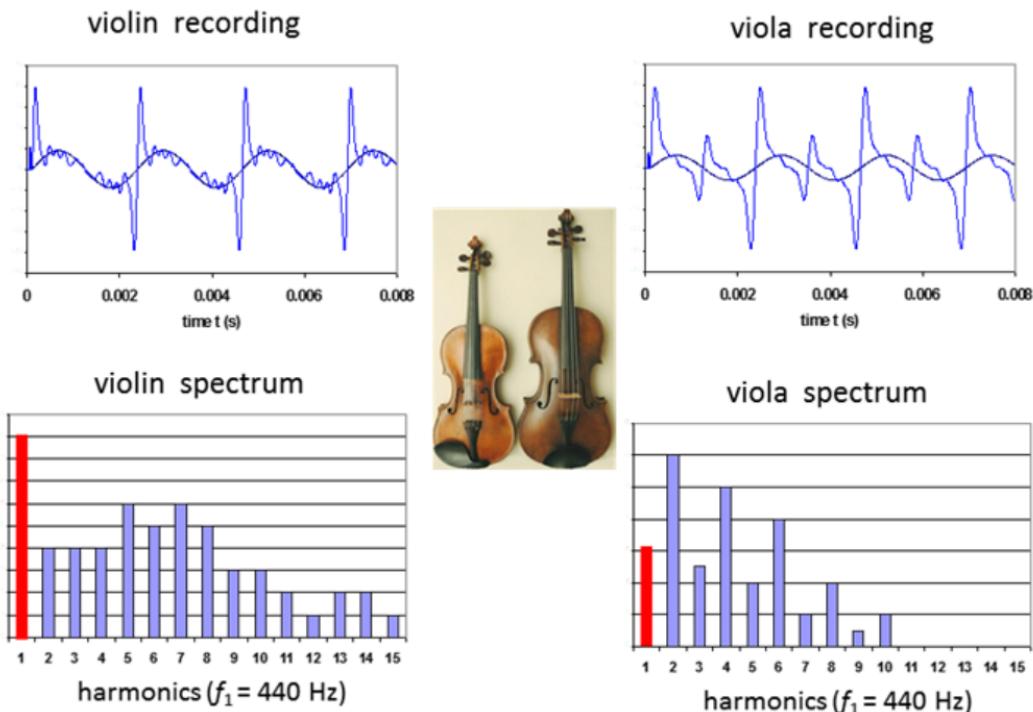


Figura 3.1: Comparativa de una nota en dos instrumentos. En rojo la fundamental.

ma nota. Las operaciones que vamos realizar en la señal de entrada tienen como objetivo modificar el tono de una nota introducida mientras mantenemos inalterados su volumen y su timbre. En la práctica veremos como tanto el algoritmo como la FPGA introducen desviaciones y no idealidades que nos permiten elaborar y clasificar diferentes métodos y operativas.

3.2. Aproximaciones a la octavación

Como ya hemos visto, el propósito de un octavador es proporcionar una señal una octava inferior a la señal introducida. En consecuencia el algoritmo debe dividir la frecuencia de cada muestra entrante por dos. Lo primero que salta a la vista es la manifiesta relación con la transformada de Fourier para operar en el dominio de la frecuencia, sin embargo, el coste computacional y temporal de implementar esta operación matemática es elevado. Es por ello que se estudian diferentes posibilidades que pudiesen simplificar la arquitectura.

El primer pensamiento es que se trata de algo sencillo: basta con eliminar una de cada dos muestras en el espectro, es decir, un diezmado en frecuencia, para obtener a la salida una señal con las frecuencias divididas por dos [15]. Sin embargo las propiedades de la transformación hacen imposible realizar esta operación: como las frecuencias de entrada son numerosas y variables no basta este diezmado, ya que el desajuste en la fase produce un desplazamiento circular en la señal de salida (ecuación 3.1).

$$Si \quad \mathcal{F}(\{x_n\})_k = X_k \quad Entonces \quad \mathcal{F}(\{x_n e^{\frac{2\pi i}{N} nm}\})_k = X_{k-m} \quad (3.1)$$

La consecuencia es que si la señal de entrada no es una única nota invariante, la salida resulta irreconocible, por lo que hay que descartar inmediatamente este proceder. [GRAFICA DE LA ENTRADA/SALIDA EN ESTE CASO MATLAB] Otra consideración que no hay que dejar de lado es la pretensión de operar en tiempo real. Esto supone que se debe tener en cuenta un mecanismo que permita trocear la señal en conjuntos finitos para poder aplicar la transformación de Fourier. Este proceso, junto con la propia implementación de la transformada va a complicar en gran medida la arquitectura, es por ello que se decide en primer lugar evaluar los algoritmos que no precisan de esta operativa.

3.2.1. Prescindiendo de Fourier

De cara a obtener un pedal de efectos para un instrumento más grave, se podría haber pensado en optar por un octavador ascendente. Esto, aunque parece que plantea los mismos problemas, resulta mucho más sencillo de implementar precisamente por ser 2 el factor de multiplicación. La fácil solución consiste en elevar cada muestra al cuadrado y realizar un filtrado pertinente que aíslle las componentes que nos interesan, de forma que $x_{out} = x_{in}^2$, según vemos en el gráfico [INSERTAR GRAFIO MATLAB]. Como se puede apreciar, se produce un aumento llamativo de los armónicos produciendo una ligera variación en el timbre que podría resultar o no conveniente. Este tipo de efectos recibe el nombre de *enhancer* pudiendo modificar el timbre de forma significativa. Esta operativa descrita me fue sugerida por José Parera, al que agradezco el tiempo que me ha dedicado. Si añadiésemos a este otros efectos como tremolo, reverb o delay, podríamos realizar una

aproximación muy válida a un pedal comercial, sin embargo, he preferido mantenerme fiel a la intención original de realizar la octavación descendente.

No obstante, merece la pena probar que ocurre si realizamos la operación opuesta, $x_{out} = \sqrt{x_{in}}$, de forma que se octavara la señal de forma descendente. El resultado es menos halagüeño de lo que pudiera parecer, en primer lugar está el inconveniente de tener que lidiar con las muestras de valor negativo¹, lo que resulta una molestia de cara al flujo de datos. Además, debido a que la entrada está limitada en banda, al reducir de forma cuadrática el valor de los armónicos más agudos, se produce una modificación en el timbre que provoca un sonido *robótico* o *artificial* por lo que se descarta también esta idea.

La última de estas operativas *sencillas* consiste en utilizar las propiedades de la multiplicación por coseno para modular la señal a la altura deseada. Aunque la idea parece sencilla, resulta muy complicado de llevar a la práctica por que habría que implementar un algoritmo que detectara los picos de frecuencia y los modulara utilizando un coseno de valor $f_{pico}/2$. La detección de picos en frecuencia ya supone otra vez la vuelta al dominio de Fourier sin contar con que la gestión de la anchura de esos picos se hace muy compleja. No obstante, el algoritmo de baja latencia que se describe posteriormente en la sección 3.2.1.2 propone algo similar.

3.2.1.1. Algoritmo NFC-TSM

Siguiendo el consejo de José Parera, recurrió a *dafx.com*, que se trata de una página donde se publican anualmente un gran número de estudios relacionados con los efectos digitales de audio y de donde he obtenido la mitad de las referencias bibliográficas. De la investigación en esta página descubrí un algoritmo que realizaba la octavación descendente sin llevar a cabo la transformada de Fourier, descrito en [12].

Este documento propone un ingenioso algoritmo al que llaman Modificación de Escala Temporal por Correlación Normalizada y Filtrada o por sus siglas en inglés *NFC-TSM*. El esquema de funcionamiento es el siguiente; primeramente se lleva a cabo un remuestreo con la tasa deseada $f_{s,original}/f_{s,replay}$ (para realizar la octavación debería ser 2:1). En segundo lugar tiene lugar el proceso de la modificación de la escala temporal que vuelve a variar la escala para obtener una salida de igual tamaño que la entrada.

En las propias palabras de los autores, la idea consiste en descartar y repetir algunos segmentos de la señal para comprimir o expandir la longitud del audio resultante. Utilizan para ello un sistema de buffer circular con dos punteros que se mueven a diferente velocidad añadiendo algunas variaciones para evitar que ambos colisionen. En consecuencia, los saltos que realiza el puntero más rápido tienen longitud variable y pueden ser en cualquier dirección, puesto que la distancia entre este y el otro puntero no es fija. Además se calcula el mejor punto para realizar el salto mediante AMDF (ecuación 3.2) para evitar una discontinuidad demasiado abrupta que resulte perceptible al oído.

$$D(m) = \sum_{k=0}^{L-1} |x(k+m) - x(k)| \quad (3.2)$$

¹Como se verá más adelante, el método de entrada en la FPGA devuelve las muestras normalizadas en el rango (-1,1)

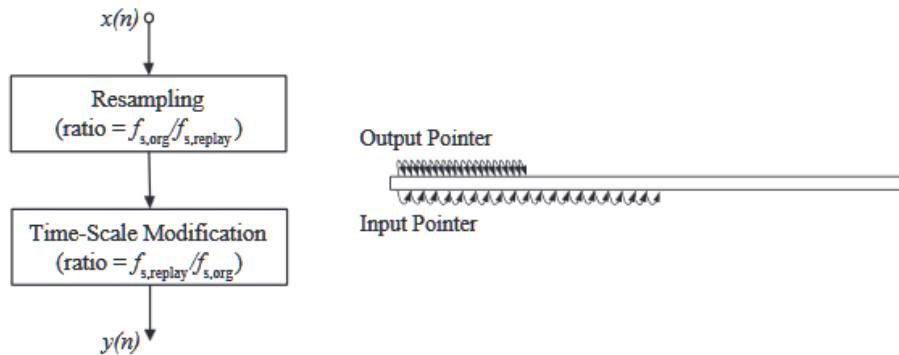


Figura 3.2: Esquema del funcionamiento NFC-TSM en octavación descendiente

La función AMDF se utiliza para ajustar el salto del puntero de manera que no se rompa la periodicidad de la señal, la cual podría no mantenerse si el punto de salto fuera aleatorio. Para ello, se compara una ventana de correlación correspondiente a donde se ubica el puntero con un área de búsqueda determinada. El punto de salto se ubica en el mínimo de la función AMDF. La longitud de las ventanas así como su ubicación son parámetros ajustables del algoritmo, tal y como aclaran los autores.

En resumen, este método combina varias operativas para lograr un algoritmo versátil que pueda ajustar al factor de octavación deseado con una calidad relativamente buena en todos. Sin embargo, para llevar a cabo la ejecución de este algoritmo es necesaria mucha capacidad de cálculo en cada instante ya que la AMDF requiere de bastantes operaciones², haciéndolo poco deseable para la FPGA. La siguiente solución presentada propone un mismo concepto de cálculo sencillo repetido en muchas ocasiones, lo cual resulta mucho más deseable a la hora de introducirlo en la placa.

3.2.1.2. Algoritmo de baja latencia

Para comprender la base de esta operativa, descrita en profundidad en [13], hay que diferenciar dos maneras de abordar la problemática del cambio de afinación a grandes rasgos ya que la nomenclatura induce a error:

- **Desplazamiento de tono:** o *pitch shifting*, se basa en que cada frecuencia se **multiplica** por una constante. Este es el caso de los algoritmos descritos antes que pretendían dividir todas las frecuencias entre 2.
- **Desplazamiento de frecuencia:** o *frequency shifting*, en este caso a cada frecuencia se le **suma** (o resta) una constante definida. Estas técnicas no se han aplicado anteriormente en algoritmos de cambios de afinación porque se rompen las relaciones armónicas entre una fundamental y sus componentes. Sin embargo, esta va a ser la solución que proponen los autores para construir el algoritmo de baja latencia.

De forma equivalente a algunos métodos anteriores, si a cada frecuencia le restamos su frecuencia mitad, $f_{out} = f_{in} - f_{in}/2$, obtenemos también la división por 2. El algoritmo se

²Estas se detallan en el artículo [12] para varios casos distintos

construye sobre esta idea, la cual, para que esta idea funcione, debe *fijar* las frecuencias entrantes de alguna manera, ya que si no, no podríamos saber a priori qué constante hay que utilizar en cada momento. La solución es utilizar un banco de filtros IIR lo suficientemente estrechos como para que se distingan correctamente dos notas sucesivas. De esta forma, se realiza la resta inmediatamente después de haber hecho el filtrado, como se muestra en 3.3. Conociendo la afinación del saxofón, se pueden conocer de antemano las frecuencias de entrada, por lo que habrá que centrar cada filtro con una de las notas del registro. El resto del espectro correspondiente a los armónicos se cubre con filtros equiespaciados siempre en escala logarítmica.

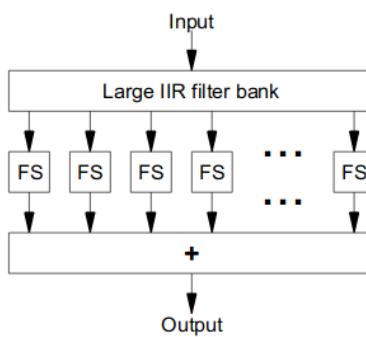


Figura 3.3: Esquema del algoritmo de baja latencia. FS equivale a cada resta de frecuencia

Tal y como proponen los autores es necesario que el ancho de banda de cada filtro vaya aumentando con la frecuencia pero estableciendo un mínimo de 50Hz para las frecuencias más bajas. En esto radica uno de los problemas de este algoritmo y es que, produce inevitablemente una ligera desafinación que se puede acentuar si se pierde la relación entera con los armónicos superiores. Esta desafinación es más pronunciada en las frecuencias inferiores debido al ancho de banda mínimo establecido ya que cada filtro coge una o dos notas.

Para realizar cada etapa de restado, son necesarios dos osciladores a 90° , una transformada de Hilbert y otros pocos componentes más, como ilustra la figura 3.4. Adicionalmente, se puede sustituir la etapa de la transformada de Hilbert por filtros IIR, reduciendo aún más el coste computacional total. No obstante, es cierto que el elevado número de módulos, aunque sencillos, tienen un coste de área grande en la FPGA aunque no resulta crítico.

La razón para no implementar este algoritmo de baja latencia ha sido puramente personal, ya que he priorizado eliminar el desafinamiento a reducir la latencia. En la fuente bibliográfica [13] hay muestras de audio de buena calidad comparando diferentes métodos, pero ninguno de ellos estaba implementado en FPGA. Es evidente que no se puede comparar de igual manera ya que las pérdidas que se producen debido al tratamiento de la señal muestra a muestra en la FPGA no se producen en un procesador. Dado que la implementación no va a ser óptima en ninguno de los casos, me ha resultado preferible implementar un algoritmo más sencillo que reduzca el número de puntos críticos en el mismo.

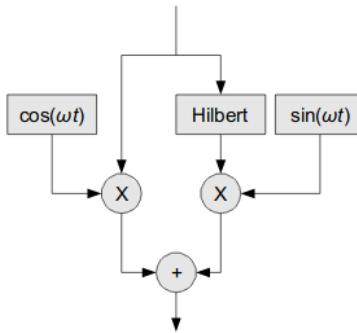


Figura 3.4: Esquema de cada etapa de restado en el dominio del tiempo

3.2.2. Retorno a la transformada

Así pues, se va a detallar la arquitectura elegida la cual está basada en un *Vocoder de fase* que se presenta en los epígrafes sucesivos [11].

3.3. Vocoder

Durante todo el siglo XX se han ido desarrollando diferentes técnicas de tratamiento y codificación para la voz, conforme iba la tecnología en aumento. La consecuencia de ello es la aparición de diversos algoritmos que permiten este tipo de operaciones con una carga computacional relativamente baja.

Un *Vocoder*³ es generalmente cualquier aparato que analiza y/o sintetiza la voz humana para lograr algún objetivo concreto, como compresión de datos, multiplexación o encriptación en la mayoría de los casos. [4]

El Vocoder de canal⁴, desarrollado por los famosos *Bell Labs* en 1928, utilizaba varios filtros multibanda seguidos por detectores de envolvente cuyas señales de control se transmitían al decodificador del receptor. Estas señales de control son mucho más lentas que la señal original a transmitir, por lo que se puede reducir el ancho de banda permitiendo a un mismo medio de transmisión soportar un mayor número de canales, ya sea por radio o cable. Finalmente, el decodificador amplifica estas señales de control y las introduce en los filtros correspondientes a cada banda para poder sintetizar de nuevo la señal original. Además de las ventajas sobre el ancho de banda, también se ayuda a proteger la señal para que no se pueda interceptar. Encriptando las señales de control y modificando los parámetros de los filtros, se puede hacer muy difícil su correcta reinterpretación si no se sincronizan el codificador y el decodificador. Esto popularizó su uso durante la Segunda Guerra Mundial en el bando aliado, patentándose diversos diseños.

El concepto se ha mantenido constante durante todo el siglo hasta nuestros días, donde podemos ver implementaciones modernas de la misma idea, por lo que se ha desarrollado una estandarización. La voz humana posee un rango de frecuencias de entre 200 y 3400 Hz típicamente, por lo que se optó por una frecuencia de muestreo de 8 kHz. Es común que se

³Del inglés *voice* (voz) junto a *encoder* (codificador)

⁴Del inglés *channel vocoder*

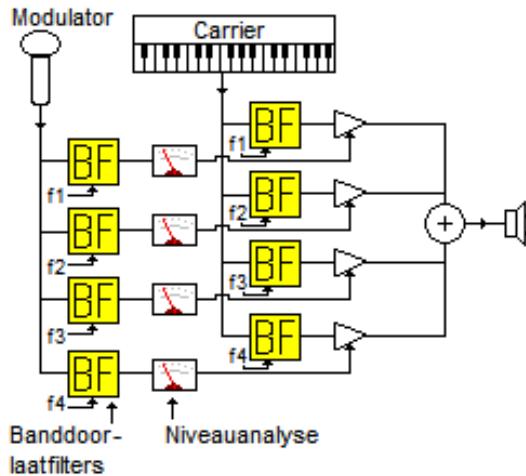


Figura 3.5: Esquema del funcionamiento de un vocoder musical[TRADUCIR]

utilice una codificación con 16 bit por muestra por analogía con el estandar CD, pero con utilizar al menos 12 la mayoría de los receptores será capaz reproducir la señal con una fidelidad razonable. Citando un ejemplo, los codificadores según la norma ITU G.729, que son utilizados en telefonía comercial, tienen una buenísima calidad con una tasa binaria de 8 kbps. Actualmente también se utilizan para desarrollar tecnologías relacionadas con la lingüística, la física y la neurociencia.

3.3.1. Vocoder en la música

Paralelamente a su utilización en comunicaciones, el vocoder se comenzó a popularizar durante la década de los 70 como método de síntesis, ya que esta estaba muy de moda en la época. Cabe mencionar, que durante esta década, surge un gran interés en los músicos por experimentar con diferentes timbres y sonidos en instrumentos conocidos o experimentales. Para aplicaciones musicales, se utiliza una frecuencia portadora proveniente de un instrumento en lugar de extraer la frecuencia fundamental del sonido que se está grabando. El resultado es una deformación del sonido capturado que, por estar afinado en una nota adecuada, produce un resultado agradable al oído. Fue el primer fabricante de sintetizadores y pionero de la música electrónica, Robert Moog, el que desarrolló un prototipo llamado *Farad* en 1968, pero no fue hasta 1970 cuando unieron el funcionamiento de esta máquina con el sintetizador modular *Moog* que había lanzado previamente al mercado. Quedaba ya conformada la esencia de utilizar la señal proveniente de un micrófono como moduladora y la proveniente de sintetizador como portadora para modularla. Algunos ejemplos tempranos de músicos reconocidos que utilizaron estos dispositivos fueron Phil Collins, Mike Oldfield, Stevie Wonder, Herbie Hancock o Michael Jackson.

Estos vocoder proporcionaban sonidos a los que el público estaba poco acostumbrado pero realmente no mantenían una fidelidad tímbrica respecto el sonido que captaban. Por ello se empezaron a utilizar los vocoder de fase los cuales permiten llevar a cabo expansión o compresión en el tiempo y *Pitch Shifting*, en otras palabras, modificar la altura musical del sonido o afinación sin cambiar la forma de onda que proporciona el timbre característico.

El método para hacerlo es el siguiente. En primer lugar se lleva a cabo una transformada mediante STFT (Short Time Fourier Transform) para posteriormente modificar la afinación mediante sub y sobremuestro. Este proceso hace que el audio resultante no resulte reconocible, por lo que es necesario ajustar el valor de la fase de cada muestra para mantener la coherencia entre ellas, de ahí el nombre de vocoder de fase. Una vez calculadas las muestras, se transforman de vuelta al dominio del tiempo, donde se rellena con ceros para obtener la misma duración que la señal entrante. A continuación se explican en detalle estas etapas, basadas en el trabajo de D.Ellis descrito en [10].

3.4. Transformación a frecuencia: STFT

Una STFT se usa para determinar el módulo y fase de muestras próximas de una señal mientras cambia con el tiempo, haciéndola muy adecuada para aplicaciones en tiempo real. Para ello, se divide la señal en segmentos más cortos de la misma longitud y se calcula la transformada de Fourier de cada uno de ellos por separado. El método para calcular la transformada es indiferente pero para obtener una latencia baja conviene decantarse por el algoritmo de la Transformada Rápida de Fourier o FFT.

3.4.1. Transformada de Fourier: FFT e iFFT

Para realizar la transformación al dominio de la frecuencia, la opción más adecuada es sin duda el algoritmo de la Transformada Rápida de Fourier o FFT. Este algoritmo calcula la Transformada de Fourier en Tiempo Discreto o DFT descomponiendo la señal original de longitud N en fragmentos de tamaño $N/2$ como muestra la figura 3.6 y posteriormente multiplicarlo por los términos W_n calculados previamente [15]. Nótese que en los bloques de $N/2$ se puede volver a aplicar el mismo principio de forma recursiva. Esto consigue reducir el tiempo de cálculo porque la transformada propiamente dicha se calcula para una longitud mucho menor. En la ecuación 3.3 correspondiente la DFT genérica podemos ver como la complejidad depende cuadráticamente de la longitud de la entrada $O(n^2)$ mientras que la FFT lo realiza únicamente con $O(n * \log(n))$.

$$X(k) = \sum_{n=0}^{N-1} x_n e^{-2\pi kni/N} \quad \text{Donde } k = 0, 1, \dots, N-1 \quad (3.3)$$

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{-2\pi kni/N} \quad \text{Donde } n = 0, 1, \dots, N-1 \quad (3.4)$$

El caso de la transformada inversa es totalmente análogo, el algoritmo de la FFT se puede aplicar de la misma forma para realizar iDFT de forma más rápida, lo que se conoce como iFFT. La ecuación 3.4 muestra la expresión genérica de la iDFT para un señal de N muestras. Cada uno de los parámetros que se utilizan para realizar las transformaciones se encuentra explicado posteriormente en la sección 4.2.3.

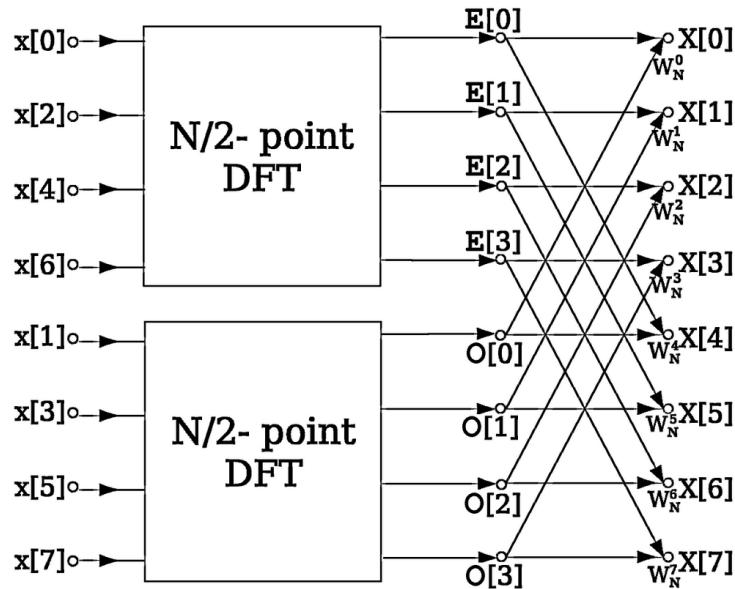


Figura 3.6: Esquema del algoritmo para la realización de la FFT

3.4.2. Solapamiento y enventanado

Dividir la señal entrante en sucesivas tramas es un proceso sencillo, únicamente se almacenan las muestras en una memoria para introducirlas posteriormente en el módulo que realiza la FFT. El problema entonces reside en la propia naturaleza de la FFT, ya que esta funciona perfectamente para señales periódicas, pero al trocear la señal en pequeñas tramas, no se garantiza que estas tramas contengan un número entero de períodos. Esto se agudiza especialmente cuando la señal es variante con el tiempo y el número de elementos por trama es independiente de la frecuencia de la señal de entrada.

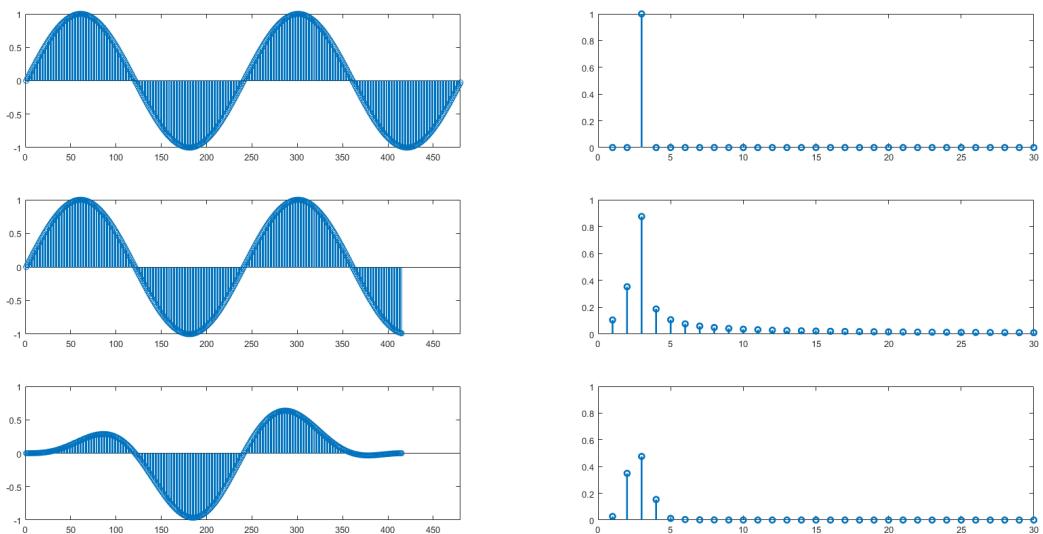


Figura 3.7: Señal sinoidal y su FFT con períodos enteros, recortada y enventanada

En la figura 3.7 se ilustra el problema del recortado arbitrario descrito y su mejora al aplicarle enventanado: al suavizar el salto de frecuencias, este resulta menos molesta al oído. Frente al caso ideal, el primero, el corte introduce componentes en otras frecuencias que se traducirán como un ruido molesto al final de cada trama, conocido en inglés como *clipping*, al escuchar la transformada inversa. Podemos comprobar como al aplicar una ventana a la señal de entrada, este se hace menor y afecta a menos muestras. Para este ejemplo se ha utilizado una ventana de Hann como la que se utilizará en el prototipo.

Junto al enventanado, se suele aplicar cierto *solapamiento*, es decir, en lugar de empezar a construir una trama a continuación de la anterior, la empezamos a llenar antes de que se haya acabado de llenar la anterior, repitiendo una misma muestra en una o varias trama sucesivas. De esta forma, las tramas están enlazadas entre ellas evitando una discontinuidad abrupta.

Generalmente se cuantifica este proceso mediante un *factor de solapamiento*, fs , expresado en tanto por ciento. Si una trama t de longitud $n = 100$ muestras tiene un solapamiento de 15 %, las primeras $n * 15\% = 15$ muestras de t son idénticas a las 15 últimas de la trama anterior, $t-1$, y así sucesivamente.

Lógicamente, cuando aumentamos el factor de solapamiento más muestras procesamos y en consecuencia, menos eficiente es el algoritmo, puesto que se procesa información redundante. Este desperdicio de recursos en el procesado supone un compromiso doble con las prestaciones. Por un lado, cuanto más disminuya esta eficiencia, más aumentará la latencia, ya que habrá que esperar al cálculo de la siguiente trama para poder finalizar la construcción de la trama presente. Por otro lado, resulta mucho más complejo de cara a la temporización en su implementación.

Como conclusión, debemos elegir un factor de solapamiento $fs > 50$ para que resulte práctico, ya que si no el efecto es demasiado sutil como para que merezca la pena dedicar esfuerzo a su implementación en la arquitectura. Tras un modelado en Matlab, he implementado finalmente un valor de $fs = 75\%$ tal y como recomienda Ellis [10] en su versión del vocoder de fase.

Como ya se ha visto antes, el solapamiento se utilizará junto con un enventanado de

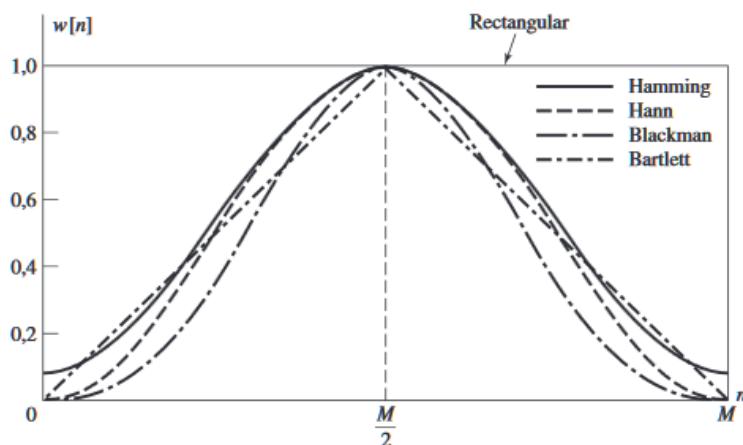


Figura 3.8: Comparativa de las ventanas más utilizadas

Hann cuya expresión se recoge en 3.5. Esta ecuación resulta sencilla de implementar y su uso está muy extendido para aplicaciones de audio en tiempo real frente a algunas de sus alternativas vistas en la figura 3.8. El propio Ellis utiliza en su algoritmo [10] una ventana de estas características.

$$H(n) = 0,5 * \left(1 - \cos \left(\frac{2\pi n}{N-1} \right) \right) \quad (3.5)$$

Estos mismos conceptos se utilizarán de forma completamente análoga en la etapa de la transformada inversa, tras introducir las muestras en el módulo iSTFT. La única excepción es que para mantener la amplitud de las muestras en la salida igual que las de la entrada, hay que aplicar un factor de escala de $2/3$. En la práctica, lo que se hará será aplicar este escalado a los valores previos quedando una ventana de Hann reducida por el factor mencionado, permitiéndonos ahorrar una multiplicación y acelerar el flujo de datos.

3.5. Operaciones sobre la fase

A grandes rasgos este algoritmo opera dividiendo cada frecuencia entrante entre dos⁵ y posteriormente modifica la fase para evitar el desplazamiento circular mencionado en 3.2.1. El procesado se realiza por *tramas consecutivas pares*: dos tramas se agrupan para formar una pareja de la que únicamente se va a poder calcular el valor de una sola trama de salida. De esta forma, la primera trama se agrupa con la segunda, la tercera con la cuarta y así sucesivamente.

⁵El proceso recibe en [15] el nombre de *diezmado*, en este caso por un valor 2.

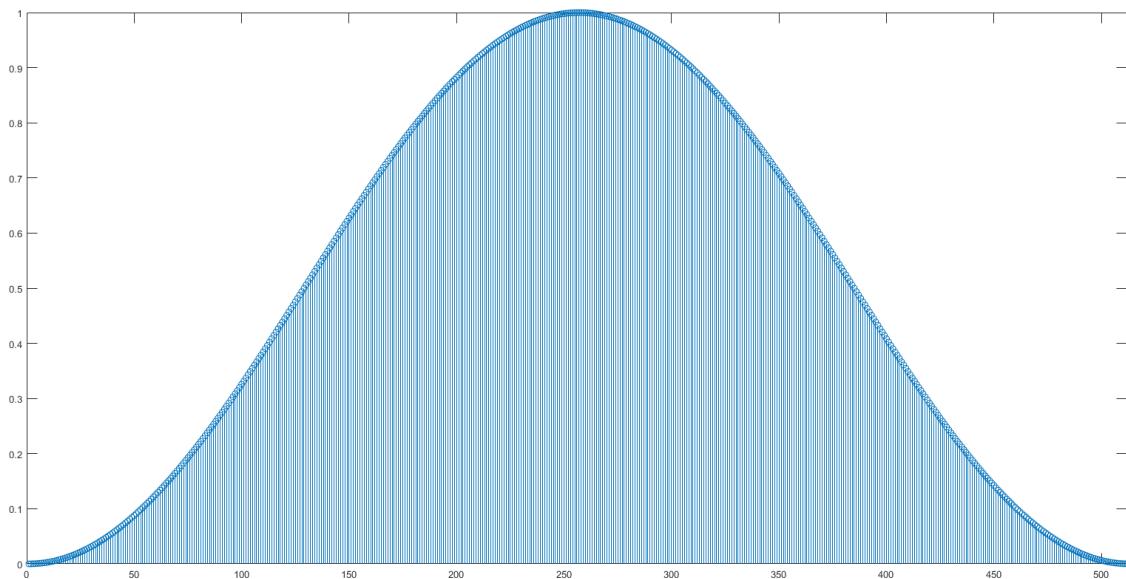


Figura 3.9: Ventana de Hann para $N = 512$ en Matlab

3.5.1. Cálculo del módulo

El primer paso consiste en obtener el módulo de las muestras. Tras repasar el algoritmo, se puede ver que solo es necesario calcular este valor para una de cada dos muestras, ya que la otra es la que será diezmada y por tanto, no merece la pena malgastar recursos en esta operación.

El cálculo del módulo a partir de la forma binomial es sencillo: $mod = \sqrt{re^2 + im^2}$ donde re es la parte real e im la imaginaria. Sin embargo, la implementación en FPGA de una raíz cuadrada no resulta inmediata. Una opción es utilizar un módulo de cálculo que procese los datos y devuelva la función raíz mientras que otra es utilizar *lookuptables* para consultar el resultado de unas entradas previamente definidas. Este último método no es práctico ya que el tamaño de la memoria ROM que almacenaría esos datos tendría que ser demasiado grande, además Vivado proporciona algunos módulos IP que realizan estas operaciones matemáticas, entre otras, a cambio de algunos ciclos de procesado.

En este caso, resulta más práctico realizar una aproximación que permita reducir el tamaño en área de la implementación así como la latencia de la misma, tal y como señala el doctor P.Chu en [5]. Si aproximamos por tanto según 3.6, donde $x = \max\{|a|, |b|\}$, $y = \min\{|a|, |b|\}$, logramos simplificar en gran medida esta operación utilizando solo operaciones sencillas.

$$\sqrt{a^2 + b^2} \approx \max\{(x - 0, 125x) + 0, 5y\}, x \quad (3.6)$$

3.5.2. Recalculando la fase

Una vez tenemos el módulo calculado, podemos calcular la nueva fase de cada muestra teniendo en cuenta que necesitamos la información de una trama t y de la siguiente, $t + 1$. El método para calcular la fase partiendo de la forma binomial es $\arctan(im/re)$ por lo que en este caso, si compensa utilizar un core IP de Vivado que realice esta operación, ya que ésta y otras operaciones trigonométricas serán necesarias más adelante.

Una vez calculada la fase de cada muestra entrante n_t , procedemos a obtener la fase de salida que se aplicará a cada muestra de salida n'_t . Para lo que calculamos una variable, dp , que se irá acumulando con n según 3.7.

$$dp = fase(n_{t+1}) - fase(n_t) - dphi \quad (3.7)$$

En esta fórmula se puede observar la existencia de una constante $dphi$ que habrá de valer $dphi = (\frac{n\pi}{2})^6$ que se calcula aparte y se introduce en una ROM para utilizarla a lo largo del procesado (ver figura 3.10). Aunque pueda parecer que el valor resultante puede llegar a ser muy elevado debido al carácter de esta constante, el siguiente paso será reducirlo al intervalo $(-\pi, \pi)$, eliminando el problema.

Esta constante dp va a servir para actualizar el valor de la fase **de la muestra siguiente** $n + 1$ operando según 3.8, de forma que la fase de cada muestra depende de la anterior dentro de la misma trama. El valor calculado, se guardará en un registro destinado

⁶Esta n también se refiere al número de muestra dentro de una trama t . Hay que recordar que $0 \leq n \leq \text{puntos de la transformada}$, 511 en este caso

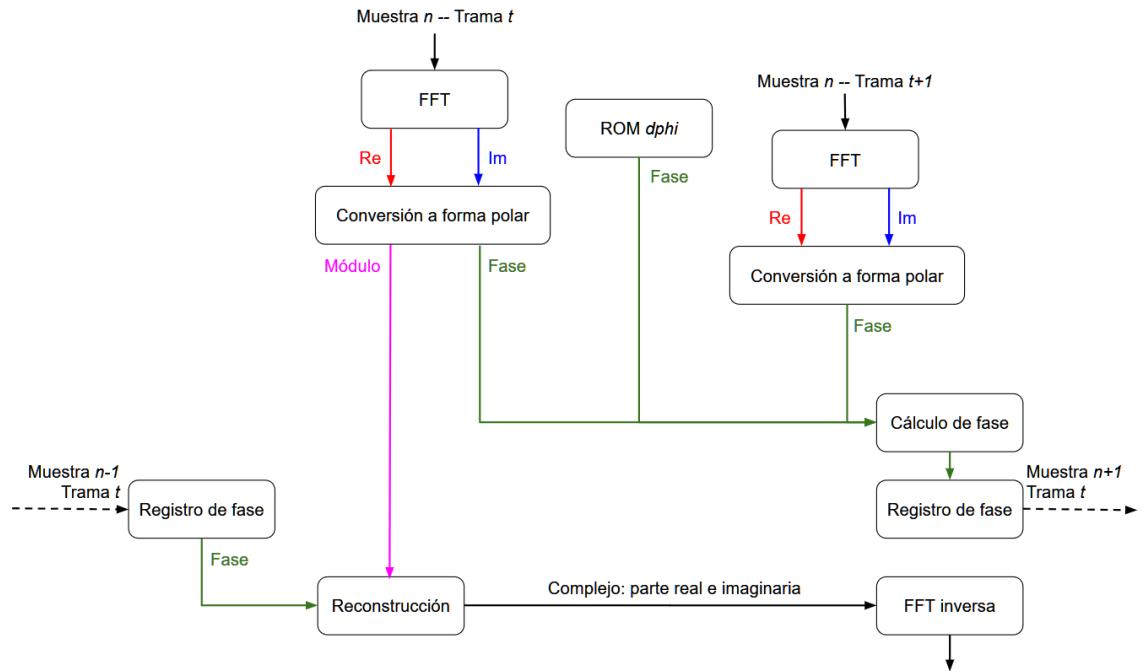


Figura 3.10: Diagrama de bloques del algoritmo empleado

a tal fin.

$$fase(n'_t + 1) = fase(n_t) + dp + dphi \quad \text{con } fase(n_t) = 0 \text{ si } n = 0 \quad (3.8)$$

Tras este proceso, se puede preparar la muestra para iniciar la transformada inversa y posteriormente reenventanar la señal de salida cuando se procesen el resto de las muestras necesarias, como se ilustra en 3.10.

Para concluir, se puede afirmar que el algoritmo que se va a implementar es una adaptación del vocoder de fase propuesto por D.Ellis [10], de forma que este orientado a la octavación y sea lo más adecuado posible para introducirlo en la FPGA. El código de Matlab empleado para probar este algoritmo se divide en dos ficheros que se adjuntan a este documento en el apéndice A.

Capítulo 4

Implementación

Una vez definido el algoritmo, se va a explicar la implementación que se ha llevado a cabo teniendo en cuenta las ventajas y limitaciones de la arquitectura. En este caso, orientar el diseño a una FPGA va a condicionar en gran medida la toma de decisiones en este aspecto.

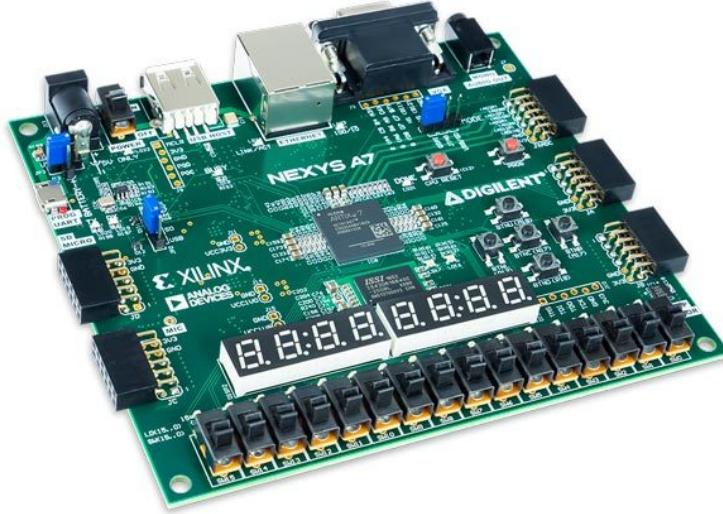


Figura 4.1: Placa Nexys A7

Para el montaje de prototipo, se va a utilizar la placa proporcionada por el departamento: Nexys A7 de Digilent, mostrada en la figura 4.1. Esta placa monta una FPGA de la familia Artix-7 modelo *XC7A100T-1CSG324C* junto con múltiples añadidos para conectividad, entrada y salida de datos, sensor de temperatura y acelerómetro. Además incluye varios diodos LED, que junto a los botones y conmutadores, permiten un cómodo manejo del conjunto de la placa. El conjunto de las prestaciones será más que suficiente para probar más adelante el funcionamiento del prototipo.

De especial trascendencia para la puesta en marcha del conjunto, serán las bahías de pines que se ubican en ambos laterales de la Nexys, puesto que permiten conectar y leer los voltajes de entrada y salida de las señales que se utilizarán. La documentación

proporcionada por Digilent [9] ha resultado muchas veces insuficiente, pero fundamental a la hora de poner en marcha el prototipo.

4.1. Gestión entrada-salida

Siguiendo el flujo de datos desde la etapa analógica, el primer paso consiste en introducir los mismos en la FPGA. Para ello, es imprescindible convertir el voltaje analógico en la señal digital mediante el uso de un *Conversor Analógico a Digital*, en lo sucesivo ADC. La Nexys A7 no incorpora ninguno integrado por lo que habrá que conectarlo externamente. Análogamente, será necesario también un *Conversor Digital a Analógico*, o DAC, para reconvertir la salida de audio procesado en tensión analógica. Por ello, se decide buscar ambos conversores conjuntamente, para simplificar su puesta en marcha.

Existen infinidad de conversores de este tipo en el mercado, todos ellos ideados para operar en diferentes condiciones de trabajo, en diferentes formatos y a un precio muy asequible. En el caso de las señales de audio, la mayor especificación que deben de cumplir es el compromiso de la tasa de muestreo t_s (o más frecuentemente su inversa, la *frecuencia de muestreo F_s*) para no corromper la señal entrante y preservar su calidad [14]. Típicamente se utilizan algunos valores ya estandarizados por las grandes empresas a lo largo del siglo XX:

- $F_s = 8 \text{ kHz}$: Utilizada especialmente para telefonía, aunque no tan común en componentes de audio musical.
- $F_s = 22050 \text{ Hz}$: Frecuencia de muestreo típica de radio que permite reproducir señales con componentes máximas de hasta 10kHz.
- $F_s = 32 \text{ kHz}$: Se utiliza escasamente en algunos formatos de vídeo digital, como el miniDV.
- $F_s = 44,1 \text{ kHz}$: La más extendida en formatos como MP3, MPEG y CD por razones tanto históricas como prácticas. Como un oído joven es capaz de percibir tonos de hasta 20kHz, se estableció esta cifra tras aplicar el criterio de Nyquist junto con un pequeño margen.
- $F_s = 48 \text{ kHz}$: También muy utilizada en televisión digital, DVD y audio profesional.
- $F_s = 96 \text{ o } 192,4 \text{ kHz}$: Pensada especialmente para audio de alta definición en formatos como HD-DVD y Blue-Ray Disc

Tras analizar las posibilidades, resulta evidente que para una aplicación musical conviene establecer la frecuencia de muestro en 44,1 o 48 kHz de forma que conserve cierta similitud con los equipos comerciales y profesionales del mercado [17].

Así, aprovechando su reciente adquisición por parte del departamento, se va utilizar un componente que integre tanto ADC como DAC y que permita trabajar a estas velocidades: el *Pmod i2-s2*, también de Digilent.

4.1.1. Pmod i2-s2

Este componente contiene todo lo necesario para este proyecto: junto a los ADC y DAC incorpora dos puertos para mini-jack estéreo hembra¹ y una serie de pines dispuestos de tal forma que la conexión con la placa es inmediata. Se pueden observar estas características en la imagen 4.2.

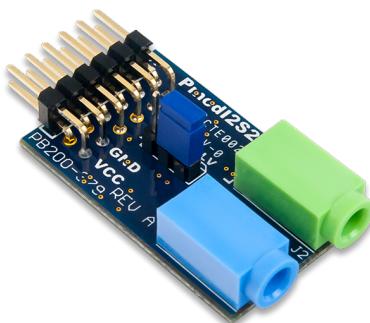


Figura 4.2: Detalle del Pmod i2-s2

Como tanto la placa como el *plug-in* son productos de Digilent, no hay ningún problema a la hora de interconectarlos, basta con insertarlo en una de las bahías de pines que tiene la FPGA. Además, las conexiones de alimentación: Vcc y GND, se encuentran hechas de serie en la Nexys, por lo que no es necesario añadirlas en el fichero de conexiones o *constraints*.

Tanto el modelo de ADC, *Cirrus CS5345* [7], como el de DAC, *Cirrus CS4344* [6] realizan las conversiones con hasta 24 bit por muestra e incorporan un filtro paso bajo anti-aliasing que por tanto, no hará falta implementar. Además tienen 98 dB de rango dinámico y baja distorsión armónica, haciéndolos ideales para las aplicaciones de audio, aún más cuando están conectados a las entradas de *mini-jack*.

Aunque los conversores soportan hasta 24 bit, el tamaño de cada palabra de datos será de 16 bit, debido a que es la longitud que utiliza el formato CD y aumentar este tamaño resulta costoso en recursos. La otra diferencia con respecto a este formato será que el procesado será *mono* en lugar de *estéreo*. Evidentemente, la existencia de dos canales dobla el número de operaciones y la complejidad del algoritmo al tener que duplicar el flujo de datos en todos los puntos del diseño. Por ello, no se suelen utilizar conexiones ni procesados estéreo en aplicaciones de audio en tiempo real, especialmente en instrumentos monofónicos². Normalmente, estos instrumentos tienen un lugar fijo en la panorámica de la producción, siendo esta la razón por la cual el procesado estéreo no se suele emplear. En la práctica, se muestrarán los datos del canal izquierdo, que por convenio es el principal cuando se opera solo con un canal, y tras el procesado, se escribirá el resultado en ambos canales.

¹Lo más extendido para audio, puesto que son los conectores que llevan móviles, auriculares, ordenadores, etc...

²Instrumentos que solo pueden producir una nota al mismo tiempo

4.1.2. Consideraciones de la implementación del Pmod

Generalmente, se ha recurrido a la documentación del fabricante, *Cirrus Logic* en el caso de los conversores o *Artix* en el caso de la FPGA en lugar de la proporcionada por Digilent, que resulta ser en muchos casos poco concisa e incompleta.

Tras insertar el Pmod en la bahía de pines, se debe modificar el archivo de conexiones *.xdc*. Es recomendable buscar este archivo en la página de la Nexys A7³. Hay que recordar que el lenguaje de este tipo de archivos es *case sensitive*, es decir, que a diferencia del VHDL distingue entre minúsculas y mayúsculas. En este tipo de ficheros tampoco puede haber dos señales con el mismo nombre, si es necesario duplicar una señal (que lo será en varios relojes) debe hacerse desde el código principal VHDL para posteriormente nombrarlo diferente de cara a las conexiones.

```
##Pmod Header JA

set_property -dict {PACKAGE_PIN C17 IOSTANDARD LVCMOS33} [get_ports MCLK_DAC]
set_property -dict {PACKAGE_PIN D18 IOSTANDARD LVCMOS33} [get_ports LR_W_SEL_DAC]
set_property -dict {PACKAGE_PIN E18 IOSTANDARD LVCMOS33} [get_ports SCLK_DAC]
set_property -dict {PACKAGE_PIN G17 IOSTANDARD LVCMOS33} [get_ports DATA_OUT]
set_property -dict {PACKAGE_PIN D17 IOSTANDARD LVCMOS33} [get_ports MCLK_ADC]
set_property -dict {PACKAGE_PIN E17 IOSTANDARD LVCMOS33} [get_ports LR_W_SEL_ADC]
set_property -dict {PACKAGE_PIN F18 IOSTANDARD LVCMOS33} [get_ports SCLK_ADC]
set_property -dict {PACKAGE_PIN G18 IOSTANDARD LVCMOS33} [get_ports DATA_IN]
```

Figura 4.3: Conexiones del Pmod con la FPGA en el fichero *.xdc*

La comunicación entre ambos módulos sigue un protocolo *I²S* [16] el cual está extendido en este tipo de componentes. Este protocolo separa la señal de datos del resto de señales de reloj, de forma que se garantiza su funcionamiento síncrono. Así, la FPGA actúa como dispositivo maestro generando estas señales y tanto el ADC como el DAC lo hacen como esclavos. Los relojes utilizados serán los siguientes:

- **MCLK:** Es el reloj maestro del sistema a partir del cual se van a generar el resto. De este reloj depende el funcionamiento de la lógica interna del Pmod.
- **SCLK:** Corresponde al reloj de bit, es decir, cada periodo equivale a la duración de un bit al leer o escribir. Este reloj está en contrafase con el resto.
- **LRCK:** Conmuta con la variación canal, es decir, cuando LRCK = 1 se realizan operaciones sobre el canal derecho y cuando LRCK = 0 sobre el izquierdo, de forma que su lectura es alterna. Como se lee una palabra en cada semiperiodo del reloj, su frecuencia será siempre el doble de la frecuencia de muestreo. Para evitar confusiones en el proyecto de Vivado se ha cambiado su nombre a LR_W_SEL.

Es fundamental entender que las relaciones entre estos tres relojes aseguran tanto la correcta interpretación de los datos y su reconstrucción como la comunicación con el módulo maestro, la FPGA. Para este proyecto se ha fijado una frecuencia de **50 MHz** para MCLK, ya que de esta forma es más sencilla su relación con el reloj de sistema de 100 MHz.

³En algunos ejemplos de Digilent en GitHub el numerado de los pines es incorrecto, por lo que hay que actuar con cautela.

Para el resto de relojes, hay que consultar la tabla de equivalencias proporcionada por el fabricante, imagen 4.4.

Speed Mode	MCLK/LRCK Ratio	SCLK/LRCK Ratio	Input Sample Rate Range (kHz)
Single-Speed Mode	256x	64	4 - 24, 43 - 54
	512x	64	43 - 54
	384x	64	4 - 24, 43 - 54
	768x	64	43 - 54
Double-Speed Mode	128x	64	86 - 108
	256x	64	86 - 108
	192x	64	86 - 108
	384x	64	86 - 108

Figura 4.4: Tabla de las relaciones entre los relojes

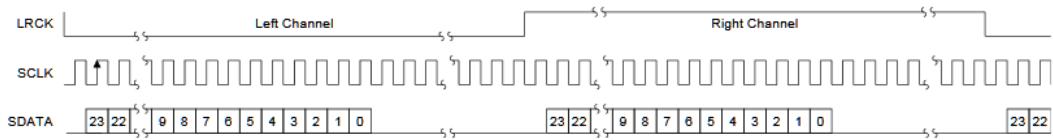


Figura 4.5: Funcionamiento de los relojes de control con flujo de datos

Como la frecuencia de MCLK es fija dada por el sistema, para obtener la frecuencia de muestreo deseada en torno a los 44,1 kHz, hay que aplicar el factor 512. De esta forma si $F_{MCLK} = 50 \text{ MHz}$ entonces $F_{LRCK} = F_{MCLK}/512 = 97,656,25 \text{ Hz}$ pero compartida entre los dos canales. Por tanto F_s de un canal resulta $F_s = 97,656,25/2 = 48,828,125 \text{ Hz} \approx 48,8 \text{ kHz}$. Finalmente, como el factor $F_{SCLK}/F_{LRCK} = 64$ obtenemos $F_{SCLK} = 3,135 \text{ MHz}$. Como se puede ver, la frecuencia de muestreo obtenida es mayor que la buscada y necesaria, pero se asemeja al estándar de alta calidad, por lo que tampoco se decide modificarla. Para generar estas frecuencias se barajó la opción de utilizar el *clocking wizard*, un módulo IP de Vivado que garantiza la correcta generación de varias señales de reloj salientes a partir de una entrante. Sin embargo, este no admite valores inferiores a las decenas de megahercios. Por ello se ha tenido que implementar un contador que permitiera dividir la frecuencia en múltiplos de 2 usando los bits de esa señal. El pseudocódigo se muestra en la figura 4.6.

```
-- Frequency divider to generate ADC/DAC clocks
LRCK_next <= count(9); -- /1024
SCLK_next <= count(3); -- /16
OTRAS_SENALES_DE_CONTROL_next <= count(2);
MCLK_next <= count(0); -- /2
```

Figura 4.6: Esquema de la generación de los relojes

La adquisición de los datos se completa usando un registro de desplazamiento propuesto por P.Chu [5]. El primer bit, que será el más significativo o MSB, se lee un ciclo después de que cambie LRCK, como se muestra en la figura 4.5. **El resultado son palabras de 16 bit en complemento a 2 normalizadas en el intervalo (-1,1).**

4.2. Controlador de datos

Una vez establecido el flujo de entrada/salida, se puede comenzar el desarrollo del procesado. Primeramente es necesario adecuar las palabras recibidas para su correcta interpretación y transformarlas al dominio de la frecuencia. Una vez realizadas todas las operaciones sobre ellas, se realiza la transformada inversa y su reconstrucción. La figura 4.7 muestra estas etapas junto con el tamaño de palabra asociado en cada punto. Hay que notar que debido a las diferentes frecuencias de funcionamiento los módulos funcionan a velocidades distintas, por lo que la escala temporal está distorsionada en esa figura. Esa es la razón por la cual el número de flechas de entrada y salida no coincide entre todos los bloques.

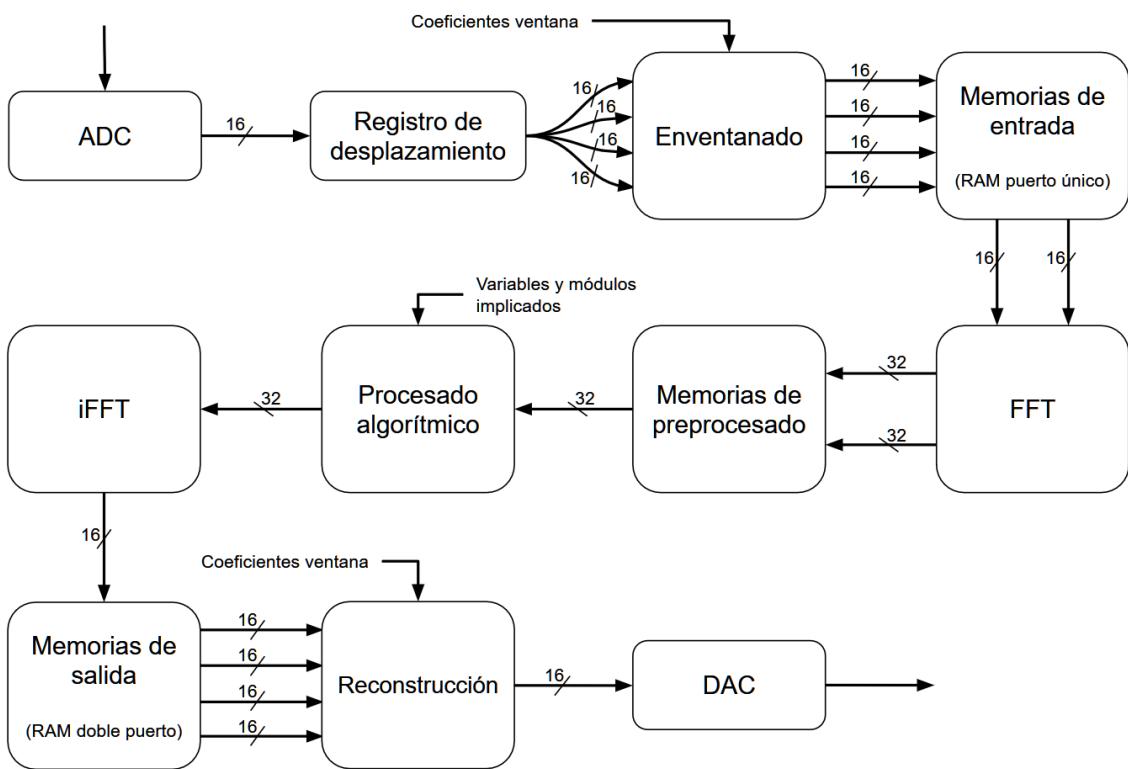


Figura 4.7: Diagrama de bloques del procesado en la FPGA

El encargado de gestionar el flujo de datos de entrada/salida es el llamado *master_controller*. Este se encarga de generar las señales necesarias para el correcto funcionamiento de Pmod y de instanciar el resto de componentes que se utilizarán, a excepción de los displays. Para garantizar el flujo de datos apropiado, se modela mediante una máquina de estados cuyo esquema corresponde a la figura 4.8. La lógica del cambio de estados se ubica en un fichero aparte llamado *fsm_control*. Los cambios de estado los produce la variable *frame_number*, la cual se encarga de llevar la cuenta de los ciclos de reloj de bit (SCLK) que se producen en cada flanco del reloj de canal (LRCK) hasta un total de 32. En principio se cumple este diagrama solo para el canal izquierdo, es decir, cuando LR_W_SEL = 0, aunque veremos que esto podrá variar. A continuación se describen cada uno de los estados:

- **IDLE:** Es el estado fundamental aunque no de reposo, ya que se activa el registro de desplazamiento para leer el dato proveniente del ADC quedando almacenado en un búfer.
- **WRITE_INPUT:** Lee el búfer donde está escrito el dato de entrada proveniente del ADC, le aplica el factor de enventanado que le corresponda y lo almacena en memoria.
- **LOAD_FREQ:** Traslada el dato de la memoria donde estaba almacenado al módulo que realiza la FFT. Como la velocidad de procesado es mucho más rápida que la frecuencia de muestreo, se evitarán las colisiones.
- **UNLOAD_FREQ:** Devuelve el dato procesado proveniente del módulo que realiza la iFFT y lo almacena en una memoria de salida.
- **READ_OUTPUT:** Lee los cuatro datos de las memorias de salida y los almacena en otros registros temporales.
- **READ_SUM:** Aplica la ventana de salida a las cuatro muestras ubicadas en los registros y las suma entre sí para obtener el dato que se va a escribir en el siguiente ciclo, el cuál se guarda nuevamente en un registro.

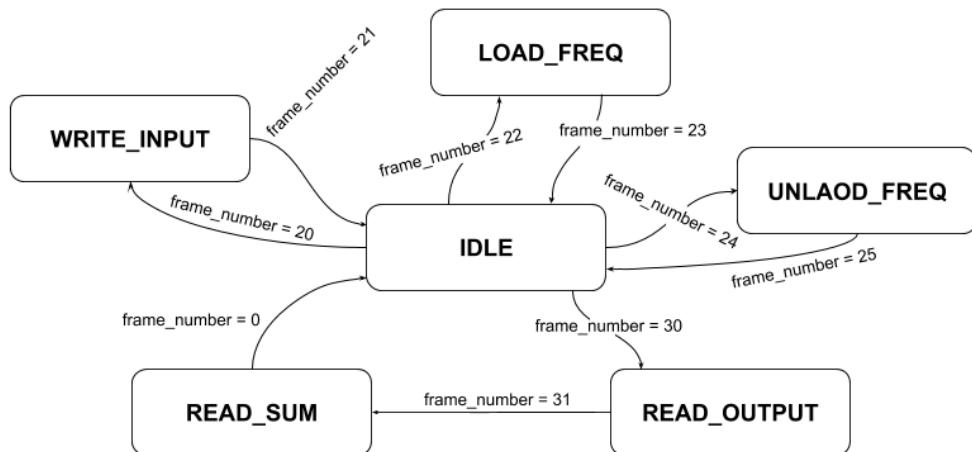


Figura 4.8: Diagrama de estados del controlador de datos

Salta a la vista que son necesarios varios bancos de memorias, cada uno ubicado en un punto diferente del flujo de datos. La implementación de cada uno de ellos varía en función de su propósito, como se verá a continuación en 4.2.2. Sin embargo, conviene primero aclarar cómo se lleva a cabo el proceso de enventanado, ya que condiciona en gran medida el diseño del flujo de datos.

4.2.1. Implementación del solapamiento

Como se ha descrito anteriormente, el factor de solapamiento que se va utilizar es del 75 %. Esto quiere decir que serán necesarias al menos cuatro memorias para poder almacenar toda la información, tal y como se muestra en la figura 4.9.

Cada una de las barras horizontales representa una de las cuatro ventanas que se almacenarán en espacios de memoria diferentes. Es importante asegurar que la muestra entrante se multiplica por el coeficiente de enventanado adecuado, que depende exclusivamente de la posición de la muestra respecto a la ventana. De esta forma, la palabra entrante se registra una vez para luego ser multiplicada por la constante de enventanado de cada ventana antes de ser almacenada en su memoria correspondiente.

En el momento de la reconstrucción, es necesario hacer cuatro accesos a memoria que recuperen cada uno de los cuatro datos procesados para multiplicar nuevamente por la constante apropiada y sumarlos todos entre sí. Los valores de las constantes de ambas ventanas garantizan que no haya desbordamiento en esta suma.

Los coeficientes de enventanado que se utilizan durante el procesado se almacenan en dos memorias no volátiles, una para la etapa previa a la FFT y otra para la posterior la iFFT, las cuales se corresponden con las etapas de enventanado y reconstrucción respectivamente. Estas memorias se generan utilizando únicamente lógica combinacional en VHDL para minimizar la latencia de las mismas. Para no introducir todos los valores de forma manual se ha creado un script de Matlab que escribe los ficheros automáticamente, cuyo código se encuentra en el apéndice C.

4.2.2. Bancos de memorias volátiles

Todas las memorias volátiles del proyecto están implementadas utilizando los módulos de Vivado *Block Memory Generator*. Estos asistentes permiten crear memorias RAM y ROM e inicializarlas a un valor determinado usando un fichero de texto. Únicamente se van a utilizar dos tipos de RAM generadas de este manera cuyo esquema se muestra en la figura 4.10.

En el caso de las memorias de entrada, está garantizado que no existan colisiones tanto en el proceso de lectura como en el de escritura, por lo que se puede utilizar el diseño más simple que propone Vivado: las memorias RAM de puerto único [20]. Este tipo de memorias poseen un puerto *WEA* que conmuta la posición de lectura y escritura. Como hay un estado para escribir y otro para leer, el valor de *WEA* va a conmutar como una señal de salida de tipo Moore de la máquina de estados asociada y consecuentemente nunca

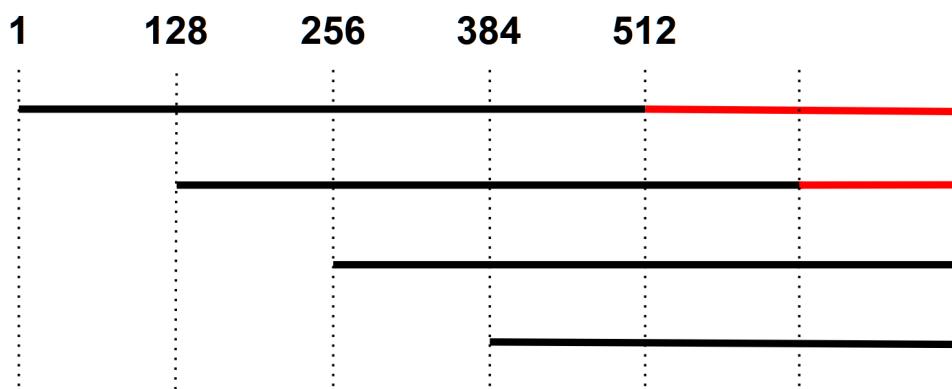


Figura 4.9: Esquema del solapamiento entre las memorias

se van a corromper los datos. En general, se utilizarán este tipo de memorias siempre que se pueda debido a su simpleza, aunque se debe asegurar que no se produzcan colisiones.

Las memorias RAM de puerto doble (derecha en la figura 4.10) funcionan prácticamente de la misma manera, salvo que poseen los puertos de entrada y salida duplicados. De esta forma se puede leer y escribir en el mismo instante, siempre que sea en direcciones diferentes. Este tipo de memorias serán adecuadas cuando la lectura y escritura no estén claramente diferenciadas en el tiempo, por ejemplo, en el caso de la salida. Tras el procesado inverso de la iFFT, las muestras deben almacenarse en memoria de forma continua, sin interrupción, lo que podría causar problemas si justo en ese momento fuera necesario leer una muestra para escribirla en el búfer de salida. Esto es la consecuencia de no realizar todas las operaciones bajo el mismo periodo de reloj: mientras que el muestreo se realiza a frecuencia $F_s = 48,8kHz$, el reloj general del sistema tiene una frecuencia $F_{clk} = 100MHz$. La razón para realizar esta conversión es que si todo el sistema trabajase a la frecuencia de muestreo, el procesado FFT requeriría mucho más tiempo para llevarse a cabo, aumentando en gran medida la latencia.

Ambos tipos de memorias poseen las mismas características, salvo que dependiendo del lugar que ocupen en la cadena de datos, el tamaño de la palabra y el número de direcciones que poseen varían. Es importante tener en cuenta que estas memorias tienen un *pipeline* interno que provoca un retraso de 2 ciclos en la lectura, ciclos durante los cuales es necesario mantener la señal de *enable* activa. La documentación que proporciona Xilinx y los ficheros auxiliares que se generan al sintetizar estos módulos facilitan el correcto tratamiento y configuración de los bloques de memorias.

4.2.3. Core FFT e iFFT

Análogamente, se ha utilizado el módulo de Vivado *Fast Fourier Transform v9.0* [21] para realizar las transformaciones de Fourier al dominio de la frecuencia. Aunque existe la posibilidad de utilizar más de un canal para llevar a cabo varias operaciones de forma simultánea, he preferido utilizar varios módulos monocanal ya que de esta forma se pueden configurar más fácilmente y simplificar el flujo de datos, a cambio de área en la FPGA.

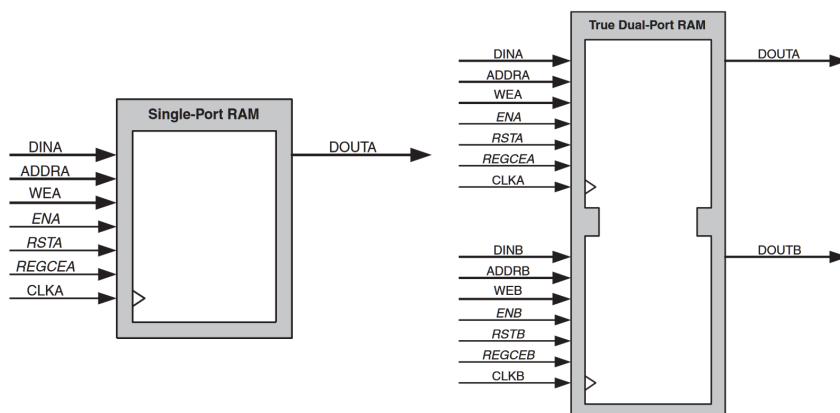


Figura 4.10: Tipos de memorias RAM implementadas

Dado que existen 4 ventanas, parece que serán necesarios 4 pares FFT-iFFT, sin embargo, debido a la diferencia de velocidad en el procesado podemos utilizar únicamente 2 de estos pares. Así las ventanas pares compartirán un módulo FFT y las impares el otro. La figura 4.11 muestra el esquema temporal que se va a implementar para permitir esta operativa donde el azul representa la carga del módulo par y el amarillo del impar.

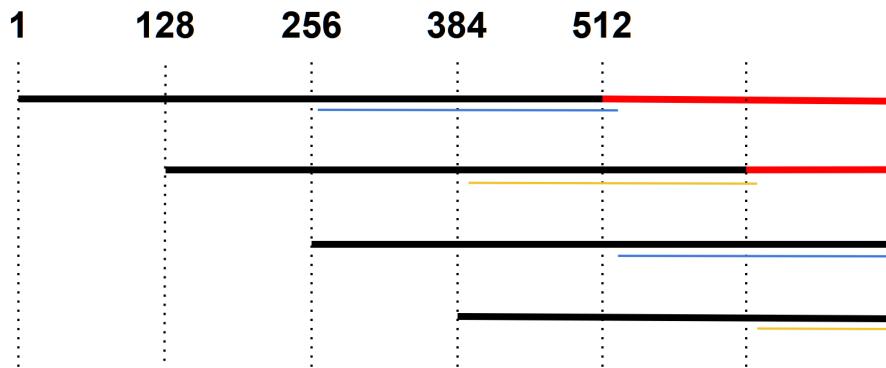


Figura 4.11: Esquema temporal de la carga de las FFT

En este esquema, la carga de los módulos FFT ocupa exactamente la mitad del tiempo que se emplea para leer los datos de una ventana. Esto se garantiza en la máquina de estados utilizando los ciclos del canal que no está en uso de forma que hay dos ciclos de carga FFT por cada uno de los de llenado de memoria.

La configuración de cada uno de estos módulos se orienta a mejorar lo más posible la latencia, para lo que se establece una longitud de transformada de 512 puntos y una arquitectura con *pipeline* en formato de punto fijo. Se utilizarán 16 bit para parte real y otros 16 para parte imaginaria de forma que al final de cada etapa interna se redondeen los valores, en lugar de truncarlos. Además, para asegurar que no se produzca *overflow* o desbordamiento en ninguna de estas etapas se va a realizar un escalado por un factor 8, aunque el valor de este se puede configurar en tiempo real por medio de una señal destinada a tal fin: *s_axis_config_tdata* (ver figura 4.12). Esta misma señal es la que contiene la información sobre el sentido de la transformada. Para intercambiar satisfactoriamente la información de las señales de datos se sigue siempre el mismo protocolo. Cuando el maestro va a proceder a enviar el dato, comuta la señal *VALID* a 1, si el esclavo está listo para recibir el dato, entonces la señal *READY* se establece a 1. Solo cuando ambas señales *VALID* y *READY* valgan 1, se producirá el intercambio. Este protocolo se aplica a la entrada y salida de datos, a la entrada de configuración y a la salida de *estatus*, la cual no va a ser utilizada debido a la configuración de nuestros módulos.

Las señales de eventos son de gran utilidad especialmente para hacer *debug* del código y ponerlo a punto. En concreto, la señal de comienzo de trama (*event.frame_started*) y las de final (*event.tlast_unexpected* y *event.tlast_missing*) son las que hay que tener en cuenta para comprobar si se ha introducido la trama correctamente. Aún así, pueden inducir a error debido a que el módulo tiene un *pipeline* interno de $n = \log_2(l_{FFT})/2$ etapas, redondeando hacia arriba.⁴ Consecuentemente, las señales de los diferentes eventos se observan en la salida con 5 ciclos de retraso respecto al momento en el que se produce dicho evento, lo

⁴En este caso el *pipeline* tiene $\log_2(512)/2 = 4,5 \rightarrow 5$ etapas

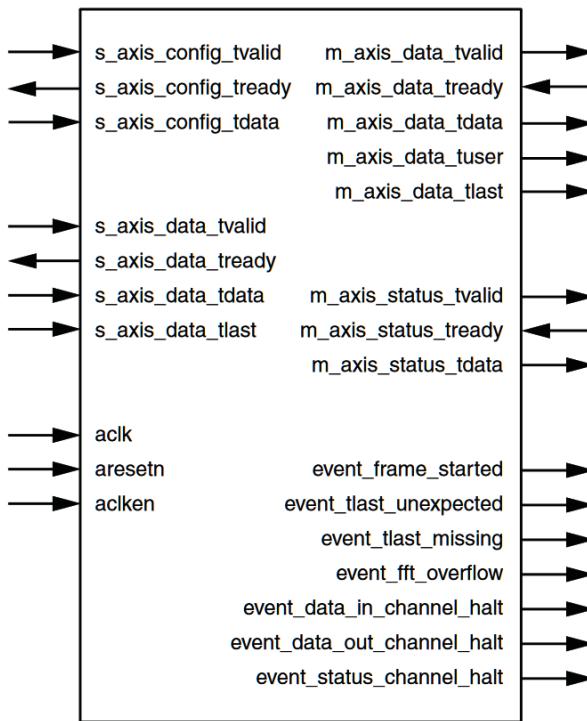


Figura 4.12: Diagrama de señales del módulo FFT de Vivado

que es importante a la hora de validarlas.

Cabe destacar que los núcleos FFT poseen memorias internas de entrada y salida que favorecen el intercambio de datos con facilidad. Mediante el correcto uso de la señal *VALID* de la entrada de datos se va a pausar la carga de los mismos de forma que únicamente se vuelquen muestras durante los estados destinados a tal fin. El procesado, en cambio, se lleva a cabo sin tener en cuenta el estado de *master_controller*. Contrariamente, el resultado de la transformación se va a guardar en memoria inmediatamente después de haberse generado.

Debido a que la operativa sobre la fase no se ha implementado en VHDL, no se ha estudiado con detenimiento cómo resulta más conveniente hacer la carga de datos al módulo iFFT, pero como el tiempo de procesado va a ser elevado⁵ lo más sencillo sería ir introduciendo los datos de salida según se fueran generando. Así los núcleos FFT e iFFT funcionarían de la misma manera.

Por último, la salida de las muestras procesadas por la iFFT en el dominio del tiempo, se almacenan en la memoria de doble puerto mencionada con anterioridad para que este disponible en el momento en que la escritura debe llevarse a cabo.

⁵Conviene recordar que para procesar una muestra es necesario conocer la trama siguiente

4.3. Controlador global

El controlador global del sistema recibe el nombre de *fsm_global* y también utiliza una máquina de estados para gestionar el estado del dispositivo. Para esta versión inicial del prototipo, la máquina implementada es muy sencilla ya que únicamente dispone de una función de *pause*. De cara al futuro, el resto de mejoras del pedal en cuanto a manejo y control deberían implementarse en esta máquina de estados. Un ejemplo puede ser la incorporación de un control de volumen o de panorámica estéreo. Si se deseara habilitar más opciones de control como un botón o vincular alguna información a los LED, también habría que realizarlo aquí.

Este fichero es el *TOP* de la arquitectura, por lo que las señales entrantes y salientes que precise, deberán provenir o dirigirse únicamente a la FPGA, utilizando para ello el fichero de conexiones *.xdc*. Los archivos de *testbench* que se empleen con el objetivo de probar el conjunto del sistema también deben referirse a este archivo, especialmente para la simulación post-síntesis o post-implementación.

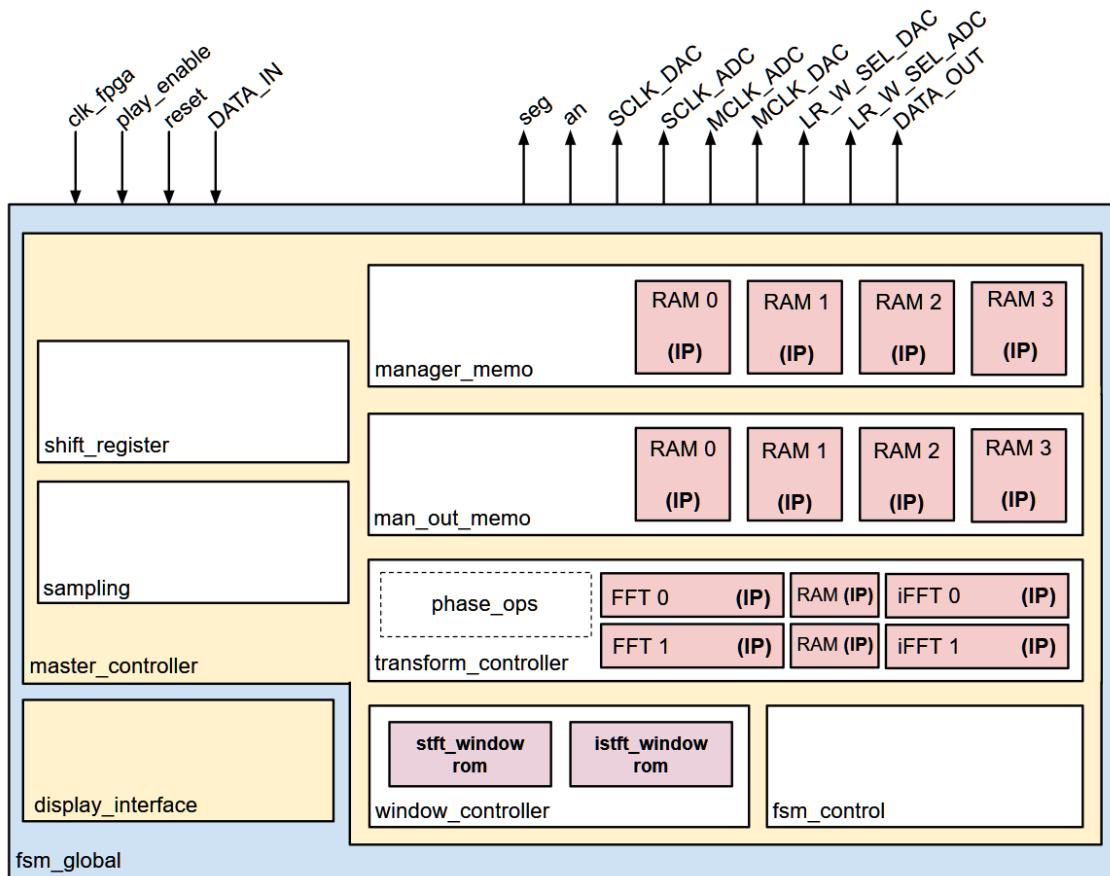


Figura 4.13: Estructura del código implementado

La figura 4.13 describe los ficheros VHDL empleados para implementar la operativa descrita previamente en la figura 4.7 a excepción del módulo de operaciones sobre la fase, *phase_ops*, que se encuentra punteado. En ella se aprecian las señales de entrada y

de salida globales, usando mayúsculas para las utilizadas por el Pmod i2-s2. Todos los ficheros utilizados se encuentran en su apéndice correspondiente al final del documento. Finalmente se detalla el funcionamiento de los displays utilizados.

4.3.1. Displays

El fichero *display_interface* será el que controle el manejo de los *displays*. En este caso, únicamente va a leer la variable del estado global y modificar el mensaje de las pantallas en consecuencia entre *play* y *pause*. Existe gran documentación y ejemplos de como es el funcionamiento de estos *displays* pero se ha utilizado la literatura proporcionada por Xillinx [9] para su implementación. Esta utiliza dos señales principales:

- **an:** Tiene 8 bits y controla qué display se va a emplear en cada instante. Por ello, debe ir comutando entre todos ellos a una frecuencia lo suficientemente alta como para que resulte imperceptible al ojo. En este caso, se ha implementado un contador de 250000 que a una frecuencia de $100MHz$ se corresponde con un periodo de refresco de $20ms$.
- **seg:** Es la señal que contiene la información sobre los segmentos que se van a iluminar, por lo que posee 7 bits, uno para cada segmento. Debe variar junto con la señal *an* para que se muestre la información correcta en el display apropiado en cada momento.

Las asignaciones a estas señales siguen una lógica combinacional que solo dependen del estado global y de la cuenta que se realiza para actualizar el valor de *an*. Cuando se alcanza el final de dicha cuenta, se actualiza el valor de *an* de forma que se desplaza hacia la izquierda. El valor de *seg* se almacena en 8 registros de forma que se utilice el correspondiente a cada valor de *an*, tal y como muestra la figura 4.14.

```
-- Switches between different displays
with curr_display select
    an <= "01111111" when "000",
    "10111111" when "001",
    "11011111" when "010",
    "11101111" when "011",
    "11110111" when "100",
    "11111011" when "101",
    "11111101" when "110",
    "11111110" when "111",
    "11111111" when others;

-- Shows the right letter depending on the current display
with curr_display select
    seg <= display1 when "000",
    display2 when "001",
    display3 when "010",
    display4 when "011",
    display5 when "100",
    display6 when "101",
    display7 when "110",
    display8 when others;
```

Figura 4.14: Asignación de valores a las señales *an* (derecha) y *seg* (izquierda).

En esta figura también se puede ver su funcionamiento activo a nivel bajo. Finalmente, basta con asignar el valor adecuado a la señal de cada uno de los *displays* que se va a introducir en *seg*. Para ello se han creado una serie de constantes en el fichero de librería del proyecto, *project_trunk*, que contienen la codificación de cada uno de los caracteres que se van a emplear, para facilitar su programación.

Capítulo 5

Pruebas, depuración y medidas

5.1. Circuito analógico de entrada

La caracterización del circuito de entrada serán básicas puesto que existe un gran número de referencias y de opciones que se pueden comprar a un precio reducido ya montadas. El montaje del módulo para este proyecto ha surgido únicamente de la curiosidad académica y no de conveniencia técnica.

5.1.1. Saturación y dependencia con la tensión de entrada

Es necesario tener en cuenta que para realizar todas las medidas que se presentan a continuación ha sido necesario generar dos señales de entrada para el circuito. No hay que olvidar que al tratarse de una entrada balanceada, se debe introducir un señal en un canal y otra duplicada pero en contrafase en el otro. Afortunadamente, el generador de funciones disponible en laboratorio ofrece esta posibilidad únicamente pulsando un botón. En lo sucesivo, cuando me refiero a la señal de entrada siempre me referiré al conjunto de la señal en cuestión más el duplicado en contrafase.

En este apartado se pretende caracterizar la respuesta del circuito frente a entradas de diferentes tensiones, utilizando diferentes posiciones del potenciómetro y permitiendo hallar el punto de saturación del conjunto para una entrada sinusoidal. Las medidas tomadas son las mostradas en la tabla siguiente.

$V_{in}(mV)$	Pos. Pot.	$V_{out}(mV)$	$G(V)$	$G(dB)$	Rango Pot.(dB)
518,8	Min.	365,6	0,7047	-3,04	8,12
	Max.	931,3	1,7951	5,08	
1018	Min.	643,8	0,6319	-3,99	12,05
	Max.	2578	2,5304	8,06	
2016	Min.	1243	0,6170	-4,19	9,10
	Max.	3547	1,7594	4,91	
3313	Min.	1875	0,5660	-4,94	9,59
	Max.	5656	1,7072	4,65	

En esta tabla, la columna *Pos. Pot* hace referencia la posición del potenciómetro, es decir, si estaba ajustado al mínimo o al máximo, lo que permite estimar el rango de la ganancia en la columna *Rango. Pot*. Se puede apreciar como la mayor sensibilidad se obtiene para la entrada de $1V_{pp}$ aproximadamente. La última medida corresponde al punto de saturación, el cual se ha obtenido empíricamente aumentando la tensión de entrada hasta que se observase recorte en la salida (con ganancia máxima), el proceso se muestra en la figura 5.1.

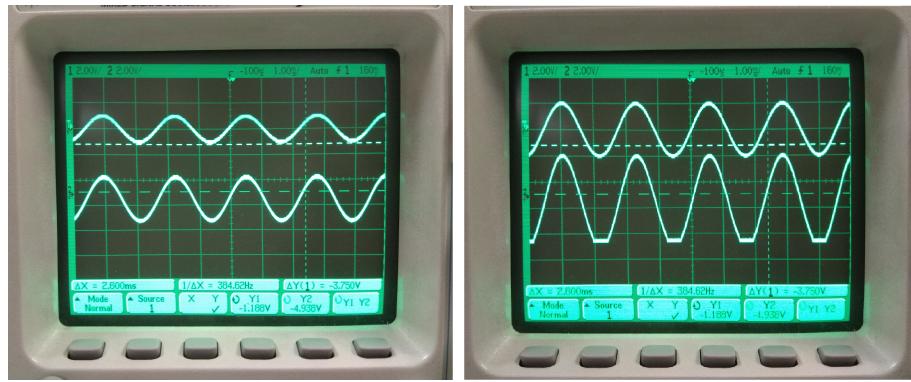


Figura 5.1: Fotografías de las medidas realizadas buscando el punto de saturación

5.1.2. Dependencia de la ganancia con la frecuencia

5.1.3. Otros parámetros

5.2. Algoritmo de *Vocoder de fase*

Para probar el algoritmo que se va a implementar se ha utilizado un fragmento breve interpretado con saxofón alto del tema *Billie's Bounce* de *Charlie Parker*. La grabación de este fragmento está realizada con un micrófono corriente no profesional y se ha guardado en formato *.wav*, es decir, sin compresión. Este formato, al igual que Matlab utiliza una frecuencia de muestro $F_s = 44,1\text{ kHz}$.

El fichero de prueba se encuentra en el Apéndice B. Esta función precisa de una ruta de archivo de audio que se va a octavar y de un entero que se corresponde con un cierto número de muestras de retraso. Este último parámetro permite cuantificar los efectos de la latencia en la FPGA.

El resultado de la prueba son dos archivos de audio: uno contiene el fichero de entrada octavado y el otro mezcla la entrada con el fichero octavado en una proporción de 35 % y 65 % respectivamente. Lógicamente, es en este último en el que la latencia tiene efecto. Adicionalmente se ha graficado tanto la suma de las componentes en frecuencia (figura 5.2) como en el dominio del tiempo (figura 5.3) ambos con latencia nula. En la primera podemos observar claramente como los picos se encuentran en frecuencias de la mitad de valor debido a la octavación. Aunque pueda parecer que el resultado muestra la serie armónica, hay que recordar que en este fragmento se han interpretado múltiples notas, cada una con sus correspondientes frecuencias y armónicos asociados, por lo que la similitud que vemos

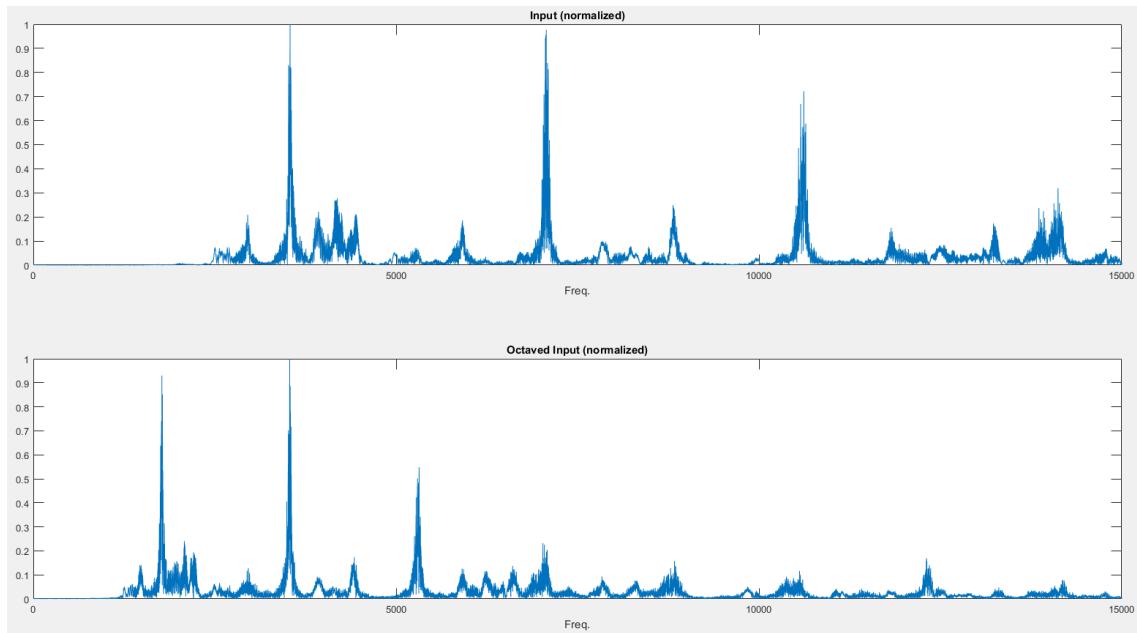


Figura 5.2: Suma de las componentes en frecuencia

entre esta gráfica y la serie armónica se debe a que las notas de la serie se utilizan más frecuentemente en la melodía.

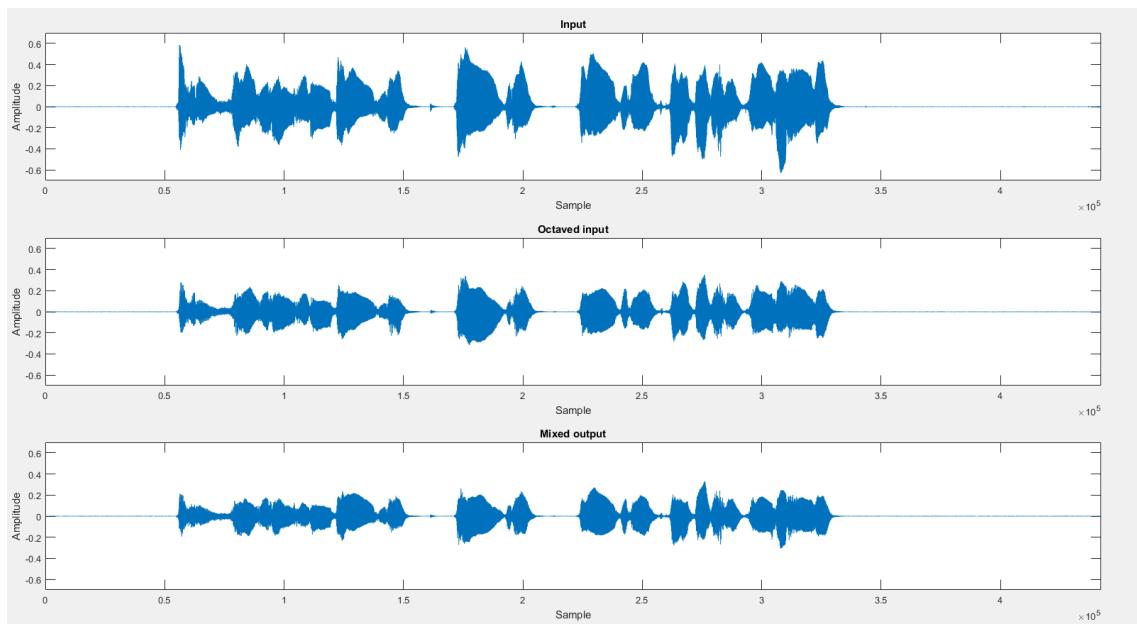


Figura 5.3: Comparativa de los ficheros resultantes

En la gráfica correspondiente al dominio del tiempo se observa una ligera pérdida de amplitud en la señal debido al procesado. Para cuantificar el valor de estas pérdidas se utiliza el cociente entre el valor medio de la señal de salida entre el de la entrada, resultando

en **1,5586 dB** para la señal octavada y **2,3890 dB** para la mezcla de ambas. También se puede apreciar como la forma de onda no se ha distorsionado en exceso; la calidad del audio resultante resulta razonablemente buena teniendo en cuenta que se ha pretendido simular el comportamiento de la FPGA con una longitud de transformada de 512 muestras de 16 bit. [UNA VEZ HALLADA LA LATENCIA DE LA FPGA, INTRODUCIR ESE VALOR EN MATLAB Y COMPARAR]

5.3. Sobre la implementación en FPGA

Capítulo 6

Conclusiones y trabajo futuro

La realización de este proyecto ha permitido alcanzar los objetivos que se planteaban al comienzo del mismo. En primer lugar se han investigado y evaluado diferentes formas y algoritmos para abordar la problemática inicial. Como consecuencia de ello se seleccionó la aproximación de Ellis al Vocoder de Fase. Posteriormente, trabajar en la implementación del prototipo ha permitido generar una versión parcialmente funcional del mismo en la placa proporcionada, en la que se han estudiado diferentes formas de lidiar con la problemática derivada del diseño, siendo necesario añadir modificaciones en el mismo a lo largo del periodo de trabajo.

Es evidente que todo ello ha significado la consecución del objetivo último consistente en sintetizar y afianzar los conocimientos utilizados dentro de los diferentes ámbitos del proyecto: teoría de la señal y procesado de la misma, programación hardware y lenguaje VHDL, ciclo de desarrollo de un producto y finalmente electrónica analógica de adecuación de señal.

Durante el periodo de trabajo en este proyecto no ha sido posible completar la implementación del prototipo, debido a que el ciclo de desarrollo de un aparato de estas características supera con creces la longitud del curso académico. Teniendo en cuenta la enorme cantidad de dificultades que han ido surgiendo a lo largo de este tiempo debido a lo novedoso del tema para mí y la dificultad para encontrar literatura útil para solucionarlos.

De cara al futuro sería interesante culminar la realización del prototipo para permitir realmente medir su latencia real y su calidad en el procesado. Una vez obtenidos estos datos, se podría mejorar el diseño optimizando algunas de las etapas existentes en el mismo, como usando módulos FFT específicos, experimentando con diferentes longitudes de palabra, de las ventanas, de las transformadas y tomando medidas en todos estos casos. Unas vez realizadas se podría establecer una comparación útil con otros algoritmos e implementaciones para permitir evaluar las fortalezas y debilidades de la implementación realizada.

Apéndices

Apéndice A

Código del algoritmo en MATLAB

Fichero: algoritmo_final.m

```
function out = algoritmo_final( input )  
  
%ALGORITMO_FINAL Algoritmo de Ellis ajustado y tal como se va a introducir  
% en la FPGA  
% Utiliza el algoritmo de Ellis para implementar un octavador. Requiere  
% de varias operaciones complejas que hay que buscar para implementarlas en  
% FPGA:  
% - Coseno  
% - Exponencial  
% - Arctan  
% - FFT/iFFT  
  
r = 2;  
transform_length = 512;  
hop = transform_length/4;  
  
% Espera la entrada como una fila, si no, la transpone  
if size(input,1) < 1  
input = input';  
end  
  
s = length(input);  
% Enventanado para la STFT  
half = transform_length/2; halfwin = 0.5 * ( 1 + cos( pi * (0:half)/half ));  
win_stft = zeros(1, transform_length);  
win_stft((half+1):(half+half)) = halfwin(1:half);  
win_stft((half+1):-1:(half-half+2)) = halfwin(1:half);  
  
% Inicializa el array de salida y las variables necesarias  
stft_signal = zeros((1+transform_length/2),1+fix((s-transform_length)/hop));  
c_stft = 1;  
% Realiza la FFT utilizando una columna por cada ventana  
for i = 0:hop:(s-transform_length)
```

```
frame = win_stft.*input((i+1):(i+transform_length));
tot = fft(frame);
stft_signal(:,c_stft) = tot(1:(1+transform_length/2))';
c_stft = c_stft+1;
end;

[rows, cols] = size(stft_signal);
t = 0:r:(cols-2);
% Tiene cuidado con las columnas porque se inicializa en 0 y hace falta la
% ventana siguiente para el solapamiento

num = 2*(rows-1);
% Inicializa el array de salida
pv_signal = zeros(rows, length(t));

% Calcula la fase a esperar
dphi = zeros(1,num/2+1);
dphi(2:(1 + num/2)) = (2*pi*hop)./(num./(1:(num/2)));
% Acumulador de fase
% Se inicializa la fase de todas las muestras de la primera trama
ph = angle(stft_signal(:,1));
% Incluye una columna de seguridad para evitar problemas
stft_signal = [stft_signal,zeros(rows,1)];
ocol = 1;
for i = t
% Coge dos coulmnas
bcols = stft_signal(:,floor(i)+[1 2]);
bmag = abs(bcols(:,1));
% Calcula la fase siguiente
dp = angle(bcols(:,2)) - angle(bcols(:,1)) - dphi';
% La reduce a rango de -pi, pi
dp = dp - 2 * pi * round(dp/(2*pi));
% Reconstruye la muestra
pv_signal(:,ocol) = bmag.*cos(ph) + j.*bmag.*sin(ph);
% La fase calculada corresponde a la siguiente trama
ph = ph + dphi' + dp;
ocol = ocol+1;
end

% inicializa las variables necesarias
s = size(pv_signal);
cols = s(2);
win_istft = 2/3*win_stft;
```

```
% Inicializa las variables de salida y procede a hacer transformada % inversa xlen = trans-
form_length + (cols-1)*hop;
istft_signal = zeros(1,xlen);
for i = 0:hop:(hop*(cols-1))
    ft = pv_signal(:,1+i/hop)';
    ft = [ft, conj(ft([(transform_length/2):-1:2]))];
    px = real(ifft(ft));
    istft_signal((i+1):(i+transform_length)) = istft_signal((i+1):(i+transform_length))+px.*win_istft;
end

% PREPARACION FINAL ++++++
% Remuestreo (interpolacion) tras filtrado antialiasing
pre_out = resample(istft_signal,2,1);
% Fuerza a la salida a tener la misma longitud que a la entrada
if length(pre_out) == length(input)
    out = pre_out';
else
    out = zeros(size(input));
    out(1,1:end) = pre_out(1,1:length(input))';
end
end
```

Apéndice B

Código prueba del algoritmo en MATLAB

Fichero: testEllis.m

```
function max_lat = testEllis(FILE_IN, samples_delayed)
%Testea el algoritmo de Ellis con los parámetros prefijados
% Emula el comportamiento real de la implementacion montando el fragmento
% octavado sobre el original, que se pasa como parametro. Tambien
% devuelve la latencia maxima (ms) segun el numero de muestras de retardo.

% LATENCIA MAXIMA SOPORTABLE
close all;
% Configuracion de las graficas
f1 = figure('Name','Input and Output','NumberTitle','off');
f2 = figure('Name','Frequency domain','NumberTitle','off');
% Configuracion de los archivos de escritura
FILE_OUT = 'Audio/Octaved audio.wav';
FILE_MIX = 'Audio/Mixed audio.wav';

% Begin
[in, Fs] = audioread(FILE_IN);
in_mono = in(:,1);
out = algoritmo_final(in_mono)';
t = 1:1:length(in_mono);

mix = zeros(size(out));
mix(1:samples_delayed) = 0.5.*in_mono(1:samples_delayed);
mix(samples_delayed+1:end) = 0.35.*in_mono(samples_delayed+1:end) + 0.65.*out(1:end-samples_delayed);
max_lat = 1/105*samples_delayed;

% Gestión de las FFT para las graficas
```

```
in_fft = fft(in_mono);
out_fft = fft(out);
in_fft_plot = 2*abs(in_fft(1:length(in_fft)/2+1));
in_fft_plot(1) = in_fft(1);
in_fft_plot = in_fft_plot./max(in_fft_plot);
out_fft_plot = 2*abs(out_fft(1:length(out_fft)/2+1));
out_fft_plot(1) = out_fft(1);
out_fft_plot = out_fft_plot./max(out_fft_plot);

f = 1:1:length(out_fft)/2+1;
% Graficas
f_lim = 0.7;
figure(f1)
subplot(3,1,1), plot(t,in_mono), title('Input'), xlabel('Sample'), ylabel('Amplitude');
xlim([0 length(t)]);
ylim([-f_lim f_lim]);
subplot(3,1,2), plot(t,out), title('Octaved input'), xlabel('Sample'), ylabel('Amplitude');
xlim([0 length(t)]);
ylim([-f_lim f_lim]);
subplot(3,1,3), plot(t,mix), title('Mixed output'), xlabel('Sample'), ylabel('Amplitude');
xlim([0 length(t)]);
ylim([-f_lim f_lim]);

f_lim = 1;
x_lim = 15000;
figure(f2)
subplot(2,1,1), plot(f,in_fft_plot), title('Input (normalized)'), xlabel('Freq.');
xlim([0 x_lim]);
ylim([0 f_lim]);
subplot(2,1,2), plot(f,out_fft_plot), title('Octaved Input (normalized)'), xlabel('Freq.');
xlim([0 x_lim]);
ylim([0 f_lim]);

audiowrite(FILE_OUT,out,Fs);
audiowrite(FILE_MIX,mix,Fs);
end
```

Apéndice C

Script de generación de valores para las ROM

codigo aqui

Bibliografía

- [1] P.ALLISON, *Low Noise Balanced Microphone Preamp*. Edited by R.Elliot for Elliot Sound Products ESP, 2008. <http://sound.whsites.net/project66.htm>
- [2] AUDIO TECHNICA, *What's the pattern?* <https://www.audio-technica.com/cms/site/aa901ccabf1dfc6b/index.html>
- [3] P.BALL, *El instinto musical: escuchar, pensay y vivir la música*. Capítulo III: Los átomos de la música. Editorial Turner Noema, 2010.
- [4] T.CARNEY, *The Vocoder*. 2012 University of Sydney. Versión digital: <https://ses.library.usyd.edu.au/bitstream/2123/8296/2/311107435%20Technology%20Review%20-%20Vocoder.pdf>
- [5] P.P.CHU, *RTL Hardware Design Using VHDL*. Ed. Wiley Interscience, 2006
- [6] CIRRUS LOGIC, *CS4344/45/48*. https://www.cirrus.com/products/cs4344-45-48/?_ga=2.264355407.117367827.1557737954-871020677.1554307733
- [7] CIRRUS LOGIC, *CS5343-44*. https://www.cirrus.com/products/cs5343-44/?_ga=2.264355407.117367827.1557737954-871020677.1554307733
- [8] I. COOPER'S: VISUAL PHYSICS ONLINE, *Waves: Musical Instruments, Strings*. University of Sydney. En http://www.physics.usyd.edu.au/teach_res/hsp/sp/mod31/m31_strings.htm
- [9] DIGILENT, *Nexys A7 Reference Manual*. <https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual>
- [10] D.ELLIS, *A Phase Vocoder in Matlab*, LabROSA at Columbia University, Marzo 2003: <http://www.ee.columbia.edu/~dpwe/LabROSA/matlab/pvoc/>
- [11] D.GALE, *The Vocoder: History and deployment*. Artículo en MusicTech a julio de 2018. Versión digital: <https://www.musictech.net/features/the-history-of-the-vocoder/>
- [12] A.HAGHPARAST, H.PENTTINEN, V.VÄLIMÄKI, *Real-Time Pitch-Shifting Of Musical Signals By A Time-Varying Factor Using Normalized Filtered Correlation Time-Scale Modification (NFC-TSM)*. Proc. of the 10th Int. Conference on Digital Audio Effects (DAFx-07), Bordeaux, France, September 10-15, 2007

- [13] N.JUILLERAT, S.SCHUBIGER-BANZ, S.MÜLLER ARISONA, *Low Latency Audio Pitch Shifting in the Time Domain* ICALIP 2008 - Proc. of the IEEE International Conference on Audio, Language and Image Processing, Shanghai, China; pp.29 - 35.
- [14] D.LAVRY, *The Optimal Sample Rate for Quality Audio*. Lavry Engineering Inc, 2012. Edición digital: http://www.lavryengineering.com/pdfs/lavry-white-paper-the_optimal_sample_rate_for_quality_audio.pdf
- [15] A.V.OPPENHEIM, R.W.SCHAFER, *Discrete-Time Signal Processing*. Pearson Education, 2011
- [16] PHILIPS SEMICONDUCTORS, *I2S Bus Specifcaion* https://www.sparkfun.com/data_sheets/BreakoutBoards/I2SBUS.pdf
- [17] J.R.STUART, *Coding High Quality Digital Audio*. Meridian Audio Ltd. Edición digital: <https://tams.informatik.uni-hamburg.de/lehre/2000ws/vorlesung/audio-verarbeitung/high-quality-audio-coding.pdf>
- [18] UNIVERSITY NEW SOUTH WALES, *How harmonic are harmonics?*. 2005. En <http://newt.phys.unsw.edu.au/jw/harmonics.html>
- [19] UNIVERSITY NEW SOUTH WALES, *What is a sound spectrum?*. 2005. En <http://newt.phys.unsw.edu.au/jw/sound.spectrum.html>
- [20] XILLINX, *Block Memory Generator v8.3*. https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf
- [21] XILLINX, *Fast Fourier Transform v9.0*. https://www.xilinx.com/support/documentation/ip_documentation/xfft/v9_0/pg109-xfft.pdf -REUBICAR
- [ojo] SHURE, *Difference between a dynamic and condenser microphone*. <https://www.shure.com/en-US/support/find-an-answer/difference-between-a-dynamic-and-condenser-microphone>
- [ojo] T.BONE, *The T.Bone MB60 SET*. <https://www.tbone-mics.com/en/product/information/details/mb-60-dynamisches-mikrofon-set/>
- [ojo] TEXAS INSTRUMENTS, *TL071*. Información completa detallada en <http://www.ti.com/product/TL071/description>