Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Trabajo Fin de Grado

Implementación de algoritmos en FPGA para tratamiento de audio y reproducción de música 3D

Autor: César Cabanillas Núñez

Tutor/es: Ignacio Bravo Muñoz

2021

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Trabajo Fin de Grado

Implementación de algoritmos en FPGA para tratamiento de audio y reproducción de música 3D

Autor: César Cabanillas Núñez

Tutor: Ignacio Bravo Muñoz

TRIBUNAL:

Presidente: JOSÉ MANUEL VILLADANGOS CARRIZO

Vocal 1°: Mª DEL CARMEN PÉREZ RUBIO

Vocal 2°: IGNACIO BRAVO MUÑOZ

FECHA: 07/10/2021

“La inteligencia consiste no sólo en el conocimiento, sino también en la destreza de aplicar los conocimientos en la práctica”

Aristóteles

Agradecimientos

Este TFG termina una etapa de mi vida y empieza otra completamente nueva. Aún recuerdo cuando de pequeño me preguntaban en el colegio qué quería estudiar de mayor, a lo que yo siempre respondía: «No se … alguna ingeniería». Quién me iba a decir que 15 años más tarde esa afirmación se convertiría en realidad.

Quiero pensar que lo que de verdad importa es el viaje y no el destino. Que son los años que pasas yendo a un sin fin de clases, las tardes de estudio en la biblioteca de la Politécnica y los fines de semana preparando exámenes en la biblioteca de Medicina, lo que de verdad deja una huella imborrable en mí. Quiero pensar que el destino es un papel muy grande con letras bonitas con el que adornar el salón, mientras que el viaje son todas las experiencias que he vivido por el camino, y sobre todo, las personas que han caminado a mi lado.

En primer lugar, agradecer a mis padres Manuel y Juana María por el apoyo que han supuesto para mi todos estos años, pues son, sin lugar a duda, los responsables en la sombra de que haya sido capaz de terminar la carrera. Han supuesto un enorme pilar donde apoyarme cuando más lo he necesitado.

A mi hermana Elvira, pues sin duda ver cómo superaba uno a uno todos mis logros académicos ha resultado un empuje para terminar la carrera antes de que me alcanzara. Ojo, habiendo empezado a estudiar yo con tres años de ventaja.

A mi hermano Rodrigo, que siempre ha estado ahí para ayudarme con los trabajos más tediosos, ya sea para ordenar la caja de resistencias o para editar videos. Muchas veces a costa de sacrificar su tiempo sin pedir nada a cambio y también superando con creces todas mis expectativas.

Al resto de mi familia, a mis primos Andrés, Elena y Miguel por recordarme que estaba haciendo algo grande y que tenía que seguir siempre hacia delante. A mis tíos Emilio, Emilia, Paqui y Antoñito por todo el cariño que me han mostrado. A mi abuela Mercedes por preguntarme cómo iban mis estudios y decirme lo buena que era esta carrera cada vez que iba a visitarla.

A mi amigos y compañeros de carrera Jorge, Sergio y Dani, por ser la mejores personas con las que me podría haber topado en mi primer día de universidad. No me puedo imaginar mi paso por la carrera sin ellos a mi lado. En especial a Jorge, que ha sido mi compañero de prácticas en casi todas las asignaturas y que ha demostrado tener una paciencia infinita para ayudarme en todas las dudas que he tenido durante estos años.

A mis compañeros de INDRA, Jose Luis, Luis, Nur, Álvaro, Juan, Miguel, Rodrigo, Barredo, Patricia, Aitor, Manu, Vanessa, Soler, Atienza, María, Fernando, Roberto, Jessica, Robert, La Roda y muchísimos más que no puedo mencionar porque entonces los agradecimientos ocuparían más que el propio proyecto. Muchas gracias a todos por tener que aguantarme 8 horas al día. Que sepáis que lo vuestro no está pagado.

Y por último y no menos importante, a mi tutor, Nacho Bravo, por aceptarme la idea loca que tuve un día como TFG y por motivarme a realizarla. Me ha ayudado a darme cuenta de que cualquier cosa es posible siempre que uno esté dispuesto a poner su esfuerzo en ello.

.

Índice de Contenido

Índice de figuras

Índice de tablas

Resumen

Este TFG detalla la elaboración de un sistema que tenga la capacidad de generar Música 3D para ser reproducida a través de unos auriculares. La dirección del estímulo sonoro podrá ser elegida por el usuario entre una selección de las direcciones más importantes de una esfera centrada en él mismo. Para ello, se diseñará un filtro FIR (*Finite Impulse Response*) que pueda ser implementado en una FPGA para realizar la convolución de la señal de audio con las respuestas al impulso de la base de datos CIPIC HRTF.

Palabras Clave

Música 3D

Sonido Espacial

FPGA (*Matriz de Puertas Lógicas Programable en Campo*)

HRTF (*Función de Transferencia Relacionada con la Cabeza*)

Reproducción biaural

Abstract

This Bachelor's Thesis details the elaboration of a system with the ability of generating 3D Music to be reproduced throw earphones. The direction of the sound stimulus shall be chosen by the user between a selection of the most important directions of a sphere whose center is themselves. For that purpose, a FIR filter will be designed so that it can be implemented in a FPGA to make the convolution of the sound signal with the impulse responses of the CIPIC HRTF database.

Keywords

3D Music

Spatial Sound

FPGA (*Field Programmable Gate Array*)

HRTF (*Head Related Transfer Function*)

Biaural recording

Resumen extendido

En el mundo de la música se demandan continuamente nuevas tecnologías y aplicaciones para sorprender a un público cada vez más difícil de asombrar. Las innovaciones pioneras en alguna técnica que salen al mercado son oportunidades de conseguir beneficios y repercusión. Hoy en día, todavía existen técnicas que han sido poco o nada explotadas en la creación de nuevos sonidos, tanto musicales como de cualquier otro tipo. Este es el caso de la Música 3D (*3 Dimensiones*), generalmente conocida como Música 8D (*8 Dimensiones*) en la jerga de internet.

Aunque el procedimiento y los algoritmos necesarios para crear Música 3D llevan publicados abiertamente en internet casi tres décadas, no ha sido hasta los últimos diez años cuando las compañías de música y de videojuegos han comenzado a poner su atención en ellos. El motivo no es otro que la creación de productos más elaborados e innovadores que puedan tener mayor impacto en las reproducciones para el caso de la música, o ventas para el caso de los videojuegos.

Un sonido 3D se crea mediante la operación matemática de la convolución entre un señal de audio mono y dos respuestas al impulso, una para el canal derecho y otra para el canal izquierdo. Las respuestas al impulso se obtienen de bases de datos publicadas en internet, siendo la base de datos CIPIC HRTF de los laboratorios CIPIC (*Center for Image Processing and Integrated Computing*) la más famosa y completa de todas. Incluye las respuestas al impulso de 45 sujetos entre los que hay modelos humanos y modelos de maniquíes. También proporciona toda documentación que detalla cómo está distribuida la base de datos y cómo se operan las respuestas al impulso. Asimismo, incluye diferentes scripts de MATLAB® y GUIs (*Graphical User Interface)* para profundizar en el funcionamiento de la Música 3D.

Una vez adquirido el conocimiento acerca de la creación de un sonido 3D, el siguiente paso es elaborar un script en Matlab® que sea capaz de generarlo a partir de una entrada de audio. La finalidad del script es validar la base de datos creando sonido espacial y reproduciéndolo a través de auriculares para verificar que se consigue el efecto 3D. La orientación del sonido vendrá dada por dos ángulos: acimut y elevación, y en función del ángulo, se selecciona la respuesta al impulso más cercana a la orientación deseada.

Los resultados de la primera aplicación se tomarán como ideales ya que sus cálculos estarán hechos en coma flotante y el error será mínimo.

Sin embargo, el objetivo de este proyecto es implementar el sistema de Audio 3D en una FPGA (*Field-Programmable Gate Array*). Eso conlleva realizar un estudio en coma fija del sistema en coma flotante. y comparar los resultados para establecer el número de bits de las señales internas. Una FPGA tiene unos recursos limitados que hay que gestionar bien o se corre el riesgo de malgastar recursos. Por ello, la decisión del tamaño de los buses se tomará en base al ECM (Error Cuadrático Medio) que en todo momento deberá ser menor al 1%, valor para el que se considera que el error es suficientemente pequeño.

Tras el estudio en MATLAB® en coma fija, el sistema diseñado se implementa en VIVADO™ usando el lenguaje de descripción hardware VHDL (*Very High Speed Integrated Circuit Hardware Description Language*). Para la convolución se empleará un filtro FIR modificado que realiza dos convoluciones paralelas: canal derecho y canal izquierdo. Previamente, la elaboración del diseño habrá tenido en cuenta los recursos disponibles en la FPGA para asegurar que tanto los tiempos como los recursos son suficientes. Al finalizar la implementación, se probará mediante *testbenches* y simulaciones de implementación temporal. Los resultados en VIVADO™ obtenidos se compararán con los resultados que previamente se habían tomado como ideales, los resultados en coma flotante de MATLAB®.

Por último, se extraen las conclusiones de la comparativa de los resultados para diferentes ángulos. El trabajo finaliza presentando los resultados para demostrar que el sistema cumpliría su función de crear música 3D si se implementara en una FPGA. También incluye un apartado de trabajos futuros donde se detalla cómo conectar la entrada y salida de audio en la FPGA y el sistema de *JoySticks* para seleccionar los ángulos de la orientación de la fuente del sonido.

Glosario de acrónimos y abreviaturas

3D	3 Dimensiones
8D	8 Dimensiones
A/D	Conversor Analógico Digital
A7	Artic-7
BRAM	Buffer Random Access Memory
CIPIC	Center for Image Processing and Integrated Computing
D/A	Conversor Digital Analógico
DSP	Digital Signal Processor
E/S	Entrada y Salida
ECM	Error Cuadrático Medio
FIR	Finite Impulse Responses
FFT	Transformada Rápida de Fourier
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
HRIR	Head Related Impulse Response
HRTF	Head Related Transfer Function
IFFT	Transformada Rápida Inversa de Fourier
IP	Intellectual Property
kHz	Kiloherzio
MB	Megabyte
MEM	Sistema electromecánico
MiB	Mebibyte
MSB	Most Significant Bit
NS	Nanosegundos
LSB	Less Significant Bit
PMOD	Peripheral Module
RAM	Random Access Memory
RBF	Radial Basis Functions
RGB	Red-Green-Blue
ROM	Read-Only Memory
SPI	Serial Peripheral Interface
SCK	Serial Clock
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1 Introducción

El ser humano ha heredado capacidad de oír en tres dimensiones a través de cientos de miles de años de evolución. Esta capacidad ha sido perfeccionada gracias a la selección natural por un motivo: la supervivencia. Para sobrevivir, un mamífero depende en gran parte en escuchar los peligros y las amenazas que le rodean, así como de localizar su origen. [1]

Cuando la ingeniería de sonido trató de trasladar estas sensaciones a un entorno de reproducción, se encontró con el problema de que eran necesarios dos canales para recrear sonido espacial o 3D. [2] Así como los mamíferos contamos con dos canales auditivos para localizar el origen de un estímulo auditivo, también en ingeniería se requieren dos canales para recrear el efecto del sonido espacial.

La combinación de oídos, cuerpo y cerebro permite recibir un conjunto de estímulos auditivos y extrapolarlos a información útil: de qué sonido se trata, de dónde proviene, a cuanta distancia se encuentra, etc. Nuestro cuerpo actúa como un gran filtro, atenuando unas frecuencias y acentuando otras. [3] Si una persona nos habla cara a cara, nuestra oreja actuará como un receptor donde el sonido rebotará y entrará al conducto auditivo. De la misma manera, si alguien nos habla desde detrás, nuestra oreja actuará como una membrana que bloqueará el sonido. Así es como nuestro cerebro es capaz de diferenciar la procedencia de éste. Además, el cerebro realiza esta tarea de forma rápida y generalmente precisa.

Pero no solo los oídos intervienen en la tarea de extrapolar la información. El cuerpo humano en su conjunto se encuentra continuamente filtrando los estímulos auditivos. Siguiendo con el ejemplo anterior, un sonido que provenga desde arriba revotará en los hombros antes de entrar en el canal auditivo. De la misma forma que el oído derecho captará un sonido proveniente del lado contrario con un retardo respecto al oído izquierdo. Y en este último caso, nuestra propia cabeza habría actuado de filtro al ser atravesada por el sonido.

A todo lo anterior se añade que cada individuo es único. La variedad en la fisionomía de las personas provoca que un modelo de sonido espacial creado para un sujeto A no tiene garantías de funcionar en un sujeto B. El objetivo final de crear un modelo único que funcione en todos por igual es, sencillamente, imposible. Por ese motivo, hoy en día existen una gran variedad de modelos para que cada usuario pueda elegir el que mejor le resulte. Asimismo, existen unos modelos genéricos creados a partir de maniquíes que funcionan generalmente bien para el público global. [4]

Todos los modelos de sonido espacial siguen la respuesta de la HRTF *(Head Related Transfer Function)* o *Función de Transferencia Relacionada con la Cabeza***.** La HRTF es una función de transferencia que permite convertir una muestra de audio monoaural en una muestra de sonido espacial o 3D aplicando una serie de algoritmos matemáticos. El efecto que se logra después de aplicar los algoritmos matemáticos es «engañar» al

cerebro, y de esa manera, hacerle creer que el sonido viene de cualquier dirección en una esfera de 360 grados.

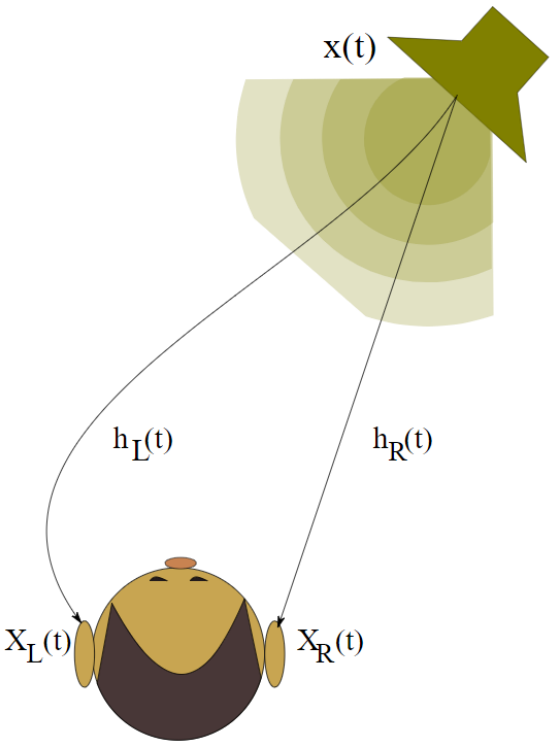


Figura 1. Efecto del filtrado de la HRTF. [1]

El efecto solo puede conseguirse a través de unos auriculares. Como ya se ha mencionado previamente, son necesarios dos canales auditivos para reproducir muestras de audio diferentes. El motivo de usar dos canales es imitar el efecto natural que ocurre cuando nuestros oídos reciben un estímulo auditivo. Al encontrarse éstos en lados opuestos de la cabeza, las ondas de sonido no tienen por qué llegar al mismo tiempo ni ser idéntico en los ambos oídos. Por el contrario, lo que sucede es que nuestro cerebro identifica las ligeras diferencias entre lo captado por un oído y el otro para localizar la fuente emisora. La Figura 1 ejemplifica este efecto.

En este TFG (Trabajo de Fin de Grado) se busca la creación de un sistema de tiempo real que transforme una muestra de audio monoaural en una muestra biaural. El audio resultante debe provocar la sensación de sonido espacial al ser reproducido por auriculares. Debido a tipo de algoritmos matemáticos que se necesitan para producir el efecto, se usará una FPGA. Los motivos de la elección de una FPGA frente a otras tecnologías son: lógica programable, recursos ofrecidos y velocidad en el procesado de señales digitales.

El modelo seleccionado para la FPGA es una Nexys 4 DDR de la familia Artix-7™. Se emplea esta FPGA ya que posee la tarjeta de evaluación y dispone de todas las

[1] https://upload.wikimedia.org/wikipedia/commons/thumb/5/50/HRTF.svg/250px-HRTF.svg.png

prestaciones necesarias para este TFG. Además, dado que es la empleada en ámbitos docentes, existe accesibilidad a ella.

Para los primeros diseños y simulaciones del filtraje de audio se utilizará el programa de cómputo MATLAB® de la empresa MathWorks®. Posteriormente, para la validación del sistema diseñado en condiciones reales se emplearán las simulaciones temporales de la herramienta VIVADO™ de la empresa Xilinx®. El objetivo final es conseguir un error suficientemente pequeño entre los resultados arrojados por MATLAB® y VIVADO™.

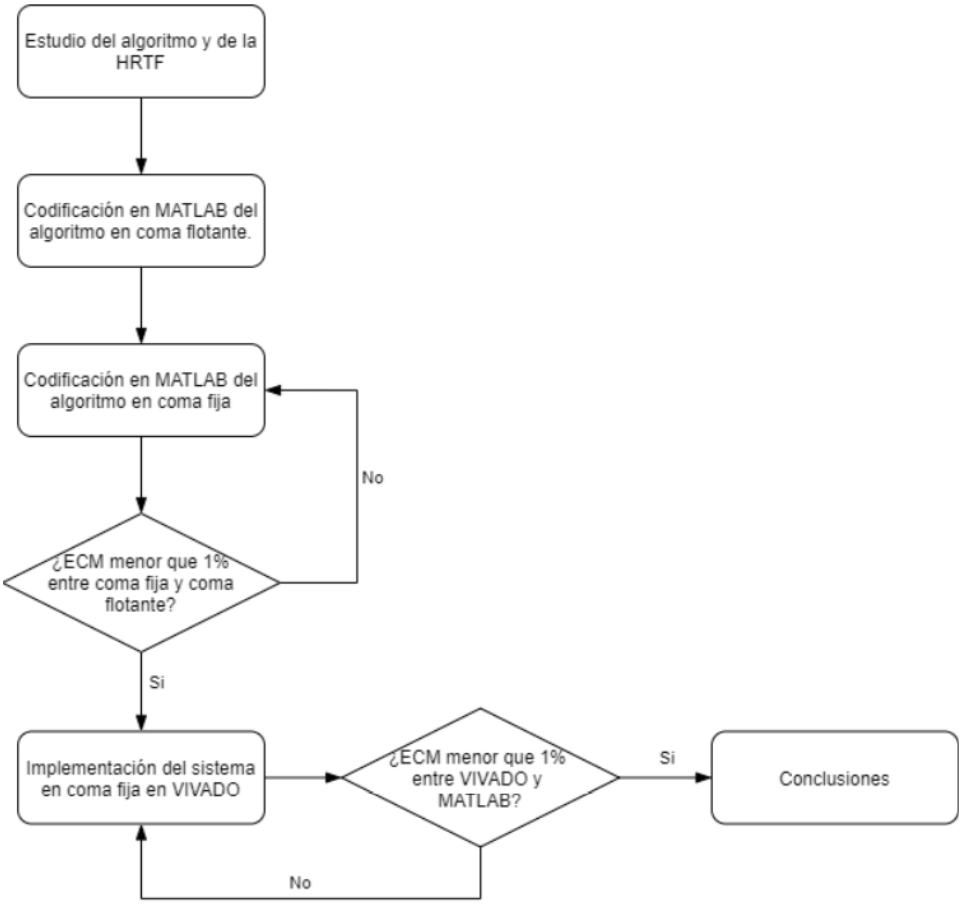


Figura 2. Flujograma del proceso seguido en este TFG.

En la Figura 2 se puede observar el proceso que se ha seguido para la realización de este TFG.

1.1 Contexto

El mundo de la música se encuentra en constante expansión y es uno de los que más dinero invierte en crear nuevas técnicas que puedan ser usadas para innovar en las canciones. A diario, se buscan innovaciones para hacer que los efectos de sonido sean mejores y de más calidad. Cualquier tecnología que mejore la experiencia del usuario es un posible mercado potencial. Este Proyecto de Fin de Grado se centrará en la música,

pero es extensible a cualquier ámbito que use algún tipo de sonido, como videojuegos, musicoterapias o efectos sonoros cinematográficos.

Figura 3. Esfera de direcciones de la Música 3D, 8D o espacial.[2]

En la jerga de internet, la Música 3D se conoce comúnmente por el nombre comercial «Música 8D», aludiendo a la esfera de 360 grados centrada en uno mismo de las direcciones desde las que puede provenir el sonido, tal y como se puede observar en la Figura 3. Sin embargo, lo más correcto es denominarla música o sonido espacial, aunque a veces también se la conoce como Música 3D.

Hoy por hoy existen multitud de programas informáticos que generan audio espacial de gran calidad a partir de muestras pregrabadas. Sin embargo, no existe ningún dispositivo comercial que genere audio espacial en tiempo real y que además permita al usuario controlar la dirección de la fuente del sonido. Bajo este pretexto, el presente documento pretende exponer un sistema en el que un usuario genérico pueda crear Música 3D sin necesitar conocimientos previos.

1.2 Objetivos

El objetivo principal que persigue este TFG es el diseño de un sistema basado en FPGA que convierta sonido monoaural en sonido espacial o 3D. A través de la lectura e investigación de artículos publicados en internet se buscarán los algoritmos que, aplicados a una entrada de audio, permitan generar una salida donde el usuario pueda controlar la ubicación de la fuente. Para realización del proyecto se han fijado los siguientes objetivos parciales:

- Investigación de la teoría matemática detrás del sonido espacial y búsqueda de los algoritmos que permitan la creación de sonido espacial.

[2] https://fiverr-res.cloudinary.com/videos/t_main1,q_auto,f_auto/hciqtlvwaahyxm3hxjvz/8d-music-8d-audio.png

- Implementación en MATLAB® de los algoritmos encontrados y comprobación de que el sonido espacial cumple con su tarea cuando es reproducido a través de auriculares. Este primer diseño se realizará en coma flotante.

- Haciendo uso del diseño en coma flotante, se rediseñará el sistema para que funcione en coma fija.

- Comparación de los resultados entre ambos sistemas. Los resultados del sistema en representación de coma flotante se tomarán como ideales y serán a los que se habrá de acercar el diseño en coma fija.

- Selección tamaño en bits de las señales del sistema cuando el error cuadrático medio entre los resultados del modelo en coma flotante y del modelo en coma fija sean inferiores al 1%.

- Implementación del sistema en coma fija en VIVADO™ usando el lenguaje de modulación hardware VHDL.

- Comparativa entre el modelo ideal en coma flotante y el modelo en coma fija implementado en VIVADO™. Se tomarán los resultados obtenidos a partir de simulaciones de implementación temporal

- Conclusiones de los resultados obtenidos.

1.3 Estructura del documento

En esta sección se define la estructura del documento que se ha dividido en 7 partes. Cada apartado está enumerado en orden ascendente e incluye una breve descripción a continuación.

1) **Introducción:** Se presenta el TFG y se explican los conceptos más importantes. Se proporciona una explicación detallada del contexto, de los objetivos y de la estructura de proyecto.

2) **Marco teórico:** Explicación en profundidad de cómo funciona la base de datos CIPIC HRTF y cómo se consigue un sonido espacial a través de la convolución de señales de audio.

3) **Diseño del modelo teórico:** Exposición de la solución adoptada para crear un sistema de procesamiento de audio espacial en FPGA. Se profundiza en los motivos que han llevado a la solución y en las limitación de los recursos *hardware* disponibles.

4) **Pruebas del modelo teórico:** Comparación de los resultados teóricos con los resultados prácticos obtenidos a partir de simulaciones de implementación temporales en VIVADO™. Se comprueba que la solución elegida es suficientemente buena usando el error cuadrático medio (ECM).

5) **Conclusiones:** Descripción del trabajo realizado y de las conclusiones extraídas.

6) **Trabajos futuros:** Aportación de ideas para continuar el desarrollo de este Proyecto de Fin de Grado. Se exploran las opciones para llegar a implementar el proyecto en una FPGA.

7) **Resultados de la implementación en placa:** Estimación de otros aspecto relacionados con la FPGA como la utilización de recursos o el consumo de potencia.

8) **Anexos:** Documentación auxiliar del proyecto.

2 Marco teórico

Para comprender el alcance de este TFG, se va a comenzar por detallar aspectos teóricos necesarios para el desarrollo de este trabajo. Así, «la HRTF describe matemáticamente el efecto combinado del pabellón auricular, cabeza y torso sobre los sonidos procedentes de una determinada posición espacial. Dependen en gran medida de la anatomía de cada persona y son por lo tanto diferentes para cada sujeto. (…) Como cada individuo es morfológicamente diferente, estas funciones son particularmente difíciles de trasponer a otros individuos sin artefactos audibles». [5]

Existen trabajos previos que han profundizado en la personalización de la HRTF a través de diferentes métodos, como la asignación de importancia a las características antropométricas [6] o la utilización de redes neurales RBF (*Radial Basis Function*) [7]. Todos estos métodos han fomentado la creación de bases de datos personalizables que resultan más precisas para los usuarios. Sin embargo, aunque los programas comerciales puedan hacer usos de esas bases de datos personalizadas, la realidad es que ninguno de ellos permite al usuario crear sonido espacial en tiempo real. Hoy en día, todos hacen uso de sonidos pregrabados.

Y aunque esos trabajos previos sean capaces de crear unas bases de datos personalizadas que resultan mucho más precisas para cada usuario, el problema viene de que no importa el grado de personalización que tenga una base de datos si el error introducido por el sistema se superpone a los resultados. Además, la base de datos siempre puede ser sustituida por otra que ofrezca una actuación mejor. Por eso, el sistema que se ha diseñado en este TFG ha priorizado la funcionalidad de poder crear Audio 3D reconocible en unas pocas direcciones espaciales frente a una base de datos personalizada con muchas direcciones espaciales.

El objetivo que persigue este TFG es diseñar un sistema capaz de usar una base de datos genérica para **crear sonido espacial en tiempo real**. La novedad recae en que el usuario del sistema elegiría un archivo de sonido desde su dispositivo móvil personal y controlaría la dirección del sonido mediante unos *JoySticks* conectados a la FPGA. También, mediante el uso de auriculares, el usuario podría de comprobar cómo la dirección de la fuente del sonido respondería al movimiento de los *JoySticks.* Cabe destacar que la parte del control de la dirección mediante *JoySticks* y la parte de la reproducción mediante auriculares no aplican para este TFG y se estudian en detalle en el apartado 6 de trabajos futuros.

Para la elaboración del sistema, este TFG emplea la base de datos de domino público CIPIC HRTF que contiene mediciones de alta resolución espacial, es decir, mediciones realizadas en incrementos entre 5° y 10° grados dependiendo de la zona de la esfera en la que se haya medido la HRIR. Gracias a su resolución, es posible replicar casi cualquier dirección espacial con una precisión decente.

Figura 4. Maniquí KEMAR durante la grabación de una HRTF. [3]

En cuanto a los sujetos de prueba, se tomaron muestras para 45 sujetos diferentes, incluido el maniquí KEMAR [8] de la Figura 4. Esta base de datos incluye las respuestas al impulso medidas desde 25 acimuts y 50 elevaciones diferentes para cada sujeto. En total, se dispone de 1250 direcciones en incrementos de 5° aproximadamente [9]. Además, la base de datos contiene medidas antropométricas de cada sujeto, aunque éstas no van a emplearse en el desarrollo del sistema de sonido espacial, ya que, como se ha mencionado anteriormente, este TFG no busca el uso de bases de datos personalizadas sino la creación de un sistema funcional que pueda usar una base de datos genérica.

Está claro que tener una base de datos personalizada para cada usuario sería la mejor opción, al fin y al cabo, la base de datos se estaría elaborando conforme a las propias características antropomórficas de dicho usuario. Ninguna otra base de datos podría jamás ofrecer la misma experiencia que aquella que se ha personalizado para un individuo en concreto. Más esta opción no es la más habitual pese a ser sin duda la óptima.

Hoy en día, ninguna empresa que haga uso de la HRTF en sus productos proporciona una base de datos personalizada para cada usuario, ya que sería imposible ante la magnitud de usuarios. Tampoco se proporcionan medios para que aquellos usuarios que lo deseen puedan fabricarse sus propias bases de datos. En su lugar, las empresas han entendido que las bases de datos genéricas como la del maniquí KEMAR son suficientemente buenas para adaptarse a un usuario genérico. Por este motivo, este TFG no emplea las medidas antropométricas para elaborar una base de datos y en su lugar emplea la base de datos genérica del maniquí KEMAR.

[3] https://www.intechopen.com/media/chapter/45612/media/image6.png

2.1 Base de datos CIPIC HRTF

Toda la información oficial de la base de datos puede descargarse libremente desde la página web de la UC Davis (Universidad de California en Davis).[4] El tamaño de la base de datos completa es de aproximadamente 180 MB (Megabytes). La medidas se hicieron en el *CIPIC Interface Laboratory* de la Universidad de California en Davis en 2001. A partir de diferentes sujetos y maniquís, se midieron las respuestas al impulso con micrófonos introducidos en el conducto auditivo en una sala anecoica, una habitación capaz de absorber las ondas sonoras sin reflejarlas.

Por definición, «una respuesta al impulso relacionada con la cabeza h depende del acimut, de la elevación y del tiempo. En los ficheros de datos, h es muestreada en espacio y tiempo. Los valores de acimut, elevación y tiempo se especifican con índices discretos: *naz*, nel y nt, y h(*naz, nel, nt*) es una matriz de dimensiones 25x50x200. Asimismo, los valores de HRIR se dan para 25 acimuts diferentes, 50 elevaciones diferentes y 200 instantes de tiempo». [9]

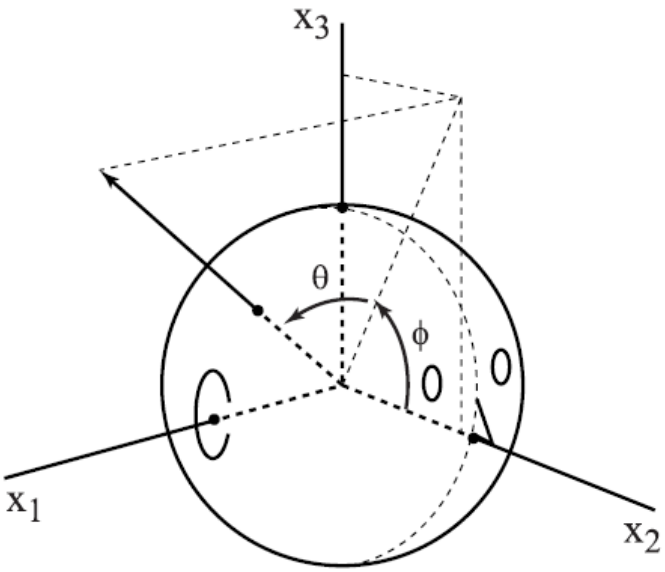


Figura 5. Sistema de coordenadas interaural-polar. [5]

«El ángulo de acimut Θ y el de elevación Φ se miden desde el centro de la cabeza en el sistema de coordenadas interaural-polar de la Figura 5. El acimut es un ángulo entre el vector de la fuente de sonido y el plano medio sagital, y varía entre -90° y +90°. La elevación es el ángulo entre el plano horizontal y la proyección de la fuente de sonido en el plano medio sagital, y varía entre -90° y +270°. Con estas coordenadas:

(0°, 0°) corresponde con el punto directamente delante

(0°, 90°) corresponde con el punto directamente encima

[4] https://www.ece.ucdavis.edu/cipic/spatial-sound/hrtf-data/
[5] https://ucdavis.app.box.com/s/lt5atmsr4hxo6yrim0fwmtfjvpbf2g7e/file/245793696045

(0°, 180°) corresponde con el punto directamente detrás

(0°, 270°) corresponde con el punto directamente debajo

(90°, 0°) corresponde con el punto directamente a la derecha

(-90°, 0°) corresponde con el punto directamente a la izquierda» [9]

Los valores del ángulo de acimut no están muestreados de manera uniforme, sino que se agrupan con mayor intensidad cerca del plano medio sagital. La relación entre el índice *naz* y el ángulo de acimut Θ puede verse en la siguiente tabla:

Tabla 1. Relación entre el índice naz y el ángulo de acimut Θ

naz	Θ	naz	Θ	naz	Θ	naz	Θ	naz	Θ
1	-80°	6	-35°	11	-10°	16	15°	21	40°
2	-65°	7	-30°	12	-5°	17	20°	22	45°
3	-55°	8	-25°	13	0°	18	25°	23	55°
4	-45°	9	-20°	14	5°	19	30°	24	65°
5	-40°	10	-15°	15	10°	20	35°	25	80°

En cambio, la elevación está muestreada de manera uniforme. El rango de la elevación va desde -45° a +230.625° en pasos de 5.625°. Este incremento angular divide el círculo completo de 360° en 64 partes iguales. Sin embargo, solo se usan 50 de esas particiones para hacer las mediciones.

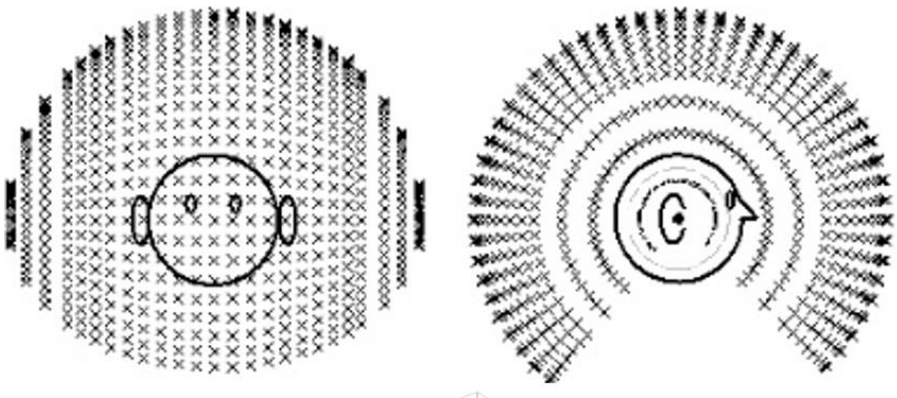


Figura 6. Alzado y perfil de las direcciones espaciales [5]

La razón es que un sonido no puede provenir de la dirección inmediatamente debajo de la cabeza, es decir, en el espacio que ocuparía nuestro torso. El hueco que se puede observar en la vista de perfil de la Figura 6 representa el hueco de un cuerpo humano. Desde esas direcciones no se ha medido ninguna respuesta al impulso ya que sería totalmente incoherente con la forma que tiene de escuchar un cuerpo humano.

La relación entre el índice *nel* y el ángulo de elevación Φ puede verse en esta otra tabla:

Tabla 2. Relación entre el índice nel y el ángulo de elevación Φ

nel	Φ	nel	Φ	nel	Φ	nel	Φ	nel	Φ
1	-45°	11	11,25°	21	67,5°	31	123,75°	41	180°
2	-39,375°	12	16,875°	22	73,125°	32	129,375°	42	185,625°
3	-33,75°	13	22,5°	23	78,75°	33	135°	43	191,25°
4	-28,125°	14	28,125°	24	84,375°	34	140,625°	44	196,875°
5	-22,5°	15	33,75°	25	90°	35	146,25°	45	202,5°
6	-16,875°	16	39,375°	26	95,625°	36	151,875°	46	208,125°
7	-11,25°	17	45°	27	101,25°	37	157,5°	47	213,75°
8	-5,625°	18	50,625°	28	106,875	38	163,125°	48	219,375°
9	0	19	56,25°	29	112,5°	39	168,75°	49	225°
10	5,625°	20	61,875°	30	118,125°	40	174,375°	50	230,625°

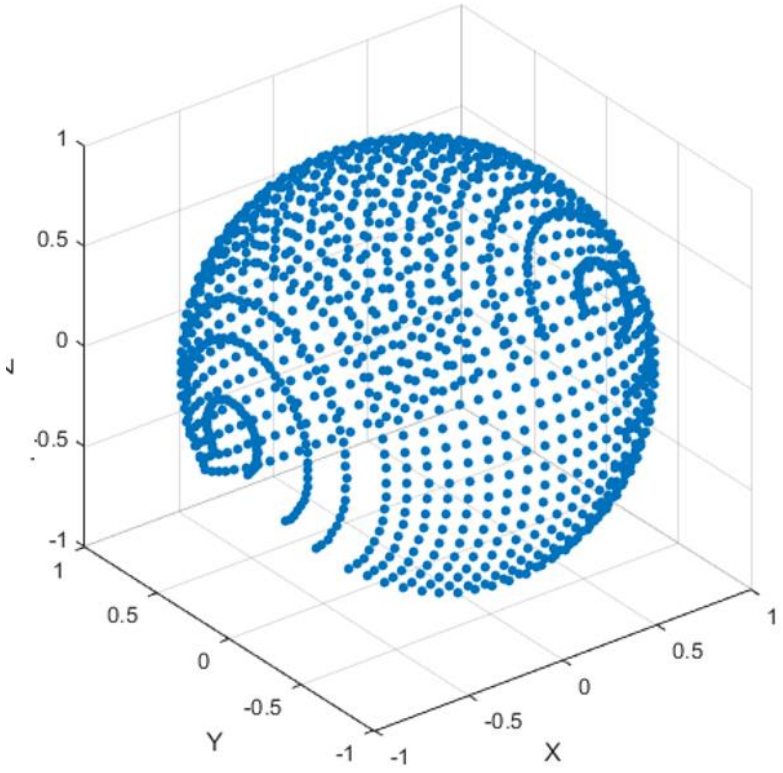


Figura 7. Representación en tres dimensiones de las direcciones espaciales.

Por último, la frecuencia de muestreo con la que se grabaron las HRIRs (*Head Related Impulse Response*) es $f_s = 44.1\ kHz$ $(Kiloherzios)$. Esta frecuencia no es casual dado que se corresponde con una frecuencia de muestreo muy utilizada en diferentes formatos de sonido como el formato'*.mp3*' [10]. Por este motivo, este TFG emplea la misma frecuencia de muestreo tanto en la entrada como en la salida de audio de la FPGA.

Cada respuesta al impulso se compone de 200 muestras tomadas a la frecuencia de muestreo $f_s = 44.1\ kHz$. Por tanto, la duración total de una HRIR es de 4.5ms.

Para lograr el efecto 3D en una muestra de audio, se realiza la operación matemática de la convolución entre una HRIR y una muestra de audio. La señal resultante será una muestra de audio 3D que se escuchará desde la misma dirección en la que se encontrara la fuente emisora en el momento de la grabación, es decir, la dirección en la que se encontrara el altavoz cuando se grabó la HRIR. Es necesario recordar que todas las 1250 respuestas al impulso se han grabado desde una dirección diferente cada una. Al realizarse una convolución con una HRIR, el audio resultante adquiere las propiedades 3D de dicha HRIR.

Normalmente el audio con el que se va a realizar la convolución ya se encuentra pregrabado. Sin embargo, este TFG pretende que sea posible obtenerlo en tiempo real para controlar la HRIR si se dispone de unos *JoyStick* que controlen los ángulos de azimut y de elevación.

Por último, cabe resaltar la importancia de la operación matemática de la convolución. Dicha operación es vital para el funcionamiento de este TFG ya que permite crear una señal a partir de dos o más señales y que la señal resultante adquiera las propiedades de las señales originarias.

2.2 Convolución

La convolución es una operación matemática formal, como lo son la suma, la multiplicación o la integración. En la suma se toman dos números para producir un tercero, mientras que en la convolución se toman dos señales para producir una tercera. Uno de los usos más extendidos de la convolución consiste en describir la relación entre las dos señales más importantes de un sistema lineal: la señal de entrada y la señal de salida. [11]

Esta propiedad hace que sea posible calcular la salida de un sistema a partir de cualquier tipo de entrada. Para ello, se emplean la convolución y la respuesta al impulso tal que $y(n) = h(k) * x(n)$, siendo y(n) la salida, h(k) la respuesta al impulso y x(n) la entrada. Aplicado al sistema de audio de este TFG, y(n) será el audio 3D, h(k) será la HRIR y x(n) será el audio en formato monoaural.

Realizando la convolución entre la entrada y la HRIR se consigue que la señal de salida adquiera las características de las dos señales. Por parte de la señal de entrada, la señal de salida hereda el propio sonido, canción o tono. Por parte de la HRIR, la señal de salida hereda la dirección de la fuente del sonido. De esta forma, cuando la señal de salida se reproduzca a través de unos auriculares se escuchará el sonido original como si la fuente del sonido estuviera localizada en una dirección, concretamente, la dirección intrínseca de la HRIR.

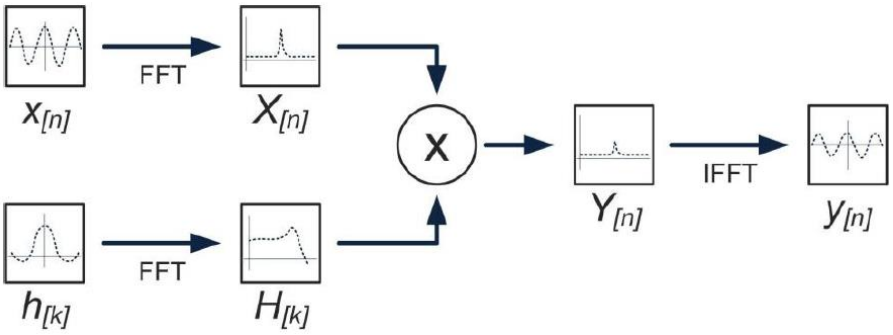


Figura 8. Convolución entre dos señales.

Por regla general, cuando se quiere implementar una convolución se utiliza la FFT (*Transformada Rápida de Fourier*) para reducir la complejidad del cálculo. En el domino de la frecuencia, una convolución se convierte en una multiplicación. Esto viene representado en la Figura 8 donde las señales discretas *x[n]* y *h[k]* se transforman al domino de la frecuencia mediante la FFT. Después, ambas señales de multiplican y producen una señal de salida en el domino de la frecuencia. Finalmente, se aplica la IFFT (*Transformada Rápida Inversa de Fourier)* a la señal de salida y el resultado final es una convolución.

Sin embargo, este TFG no emplea la FFT, sino que realiza la convolución a través de un método alternativo cuya principal característica es la utilización de un gran número de recursos de la FPGA. Hay que recordar que el objetivo final es desarrollar una aplicación que funcione en tiempo real. En este caso, la opción más sencilla resulta ser la implementación de un filtro de convolución de 200 coeficientes. Así, la entrada y la salida del filtro se podrían conectar directamente al PMOD Audio que se explica en el apartado 6.1 sin tener que realizar ninguna transformación entre el dominio del tiempo y el dominio de la frecuencia.

El principal inconveniente del filtro de convolución es el número de multiplicadores que se utiliza. Es necesario un multiplicador por cada una de las 200 muestras que componen una HRIR. Pese a todo, la FPGA de este TFG posee suficientes multiplicadores para implementar dicho comportamiento. El principal problema que se deriva de esto es que si se quisiera dedicar la FPGA a más tareas aparte del sistema de audio 3D quizá sno sería posible o habría que cambiar la FPGA por otra con más recursos. Esto se expone con más detalle en apartado 3.1.1.

Y así, el método empleado en este TFG consiste en la multiplicación polinómica de señales o circunvolución. «La circunvolución de dos vectores, *u* y *v*, representa el área de solapamiento bajo los puntos de *v* a través de *u*. Algebraicamente, la circunvolución es la misma operación que multiplicar polinomios cuyos coeficientes son los elementos de *u* y *v*.

Siendo ***m = tamaño(u)*** y ***n = tamaño(v)***, el **vector de salida *w*** es el vector de **longitud *m+n-1*** cuyo elemento ***k*** es:

$$\boldsymbol{w(k) = \sum_{j} u(j)v(k-j+1)}$$

La suma se resume en todos los valores de *j* que conducen a subíndices legales para ***u(j)*** y ***v(k-j+1),*** específicamente ***j = max(1,k+1-n):1:min(k,m)***. Cuando ***m = n***, esto da:

$$\begin{aligned} w(0) &= u(0) * v(0) \\ w(1) &= u(0) * v(1) + u(1) * v(0) \\ w(2) &= u(0) * v(2) + u(1) * v(1) + u(2) * v(0) \\ &\dots \\ w(n) &= u(0) * v(n) + u(1) * v(n-1) + \dots + u(n) * v(0) \\ &\dots \\ w(2 * n - 1) &= u(n) * v(n) \end{aligned}$$» [12]

En MATLAB®, la circunvolución de dos señales se realiza mediante la función *conv* y en este TFG es la función encargada de proporcionar los resultados ideales que se toman como modelo a seguir.

Por otro lado, el sistema debe producir una salida idéntica a la que se ha desarrollado matemáticamente en este apartado. La parte que corre en VHDL se ha diseñado para que seguir el desarrollo matemático de la convolución y se expone con más detalle en el apartado 3.2.1.4.

3 Diseño del modelo teórico

En este apartado se explica el esquema del sistema completo para generar audio 3D a partir de la HRTF y una señal de audio. La realización de este TFG ha seguido un modelo *Top-down*, es decir, de arriba abajo, comenzando por el resumen del sistema y definiendo cada parte nueva con la ayuda de «cajas negras».

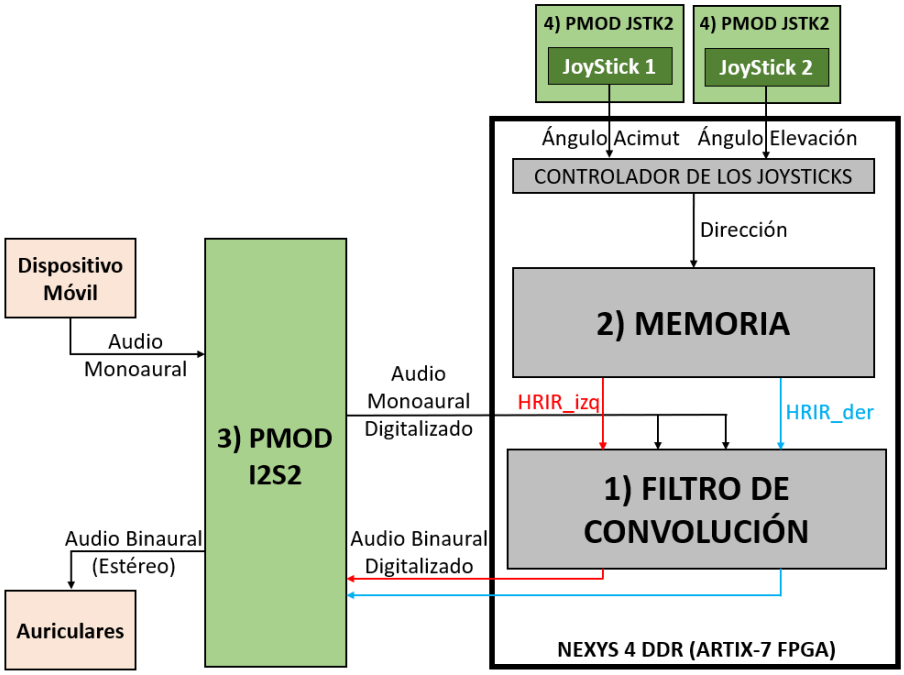


Figura 9. Diagrama de bloques del sistema de Audio 3D

El sistema consta de cuatro módulos bien diferenciados como se puede ver en la Figura 9, aunque solo dos aplican para este TFG:

1) **Filtro de convolución:** Se trata de la parte más importante de todo el sistema ya que se encarga de realizar la convolución a partir de las señales que recibe desde el resto de los módulos. Recibe una muestra de audio monoaural y proporciona una muestra de audio biaural cada 22.675 µs ($f_s = 44.1\ kHz$.). Para la convolución se emplean los coeficientes que el bloque de memoria carga en los registros del filtro. Contiene la mayor parte de la lógica programable de todo el sistema.

2) **Memoria:** Almacena en *flash* las muestras de las HRIRs que se cargan en el filtro de convolución. Cada vez que la dirección espacial del sonido cambia se envía una nueva HRIR. La memoria recibe la dirección binaria, no los ángulos de acimut y elevación, a partir de la cual se encuentran almacenados los nuevos coeficientes que habría que cargar.

3) **PMOD I2S2 (NO APLICA):** Introduciría la señal de audio monoaural en el filtro de convolución y reproduciría la señal de audio biaural obtenida como

resultado de la convolución. Se trata de un circuito integrado con dos convertidores, uno A/D y otro D/A, cada uno unido a un conector *jack* de 3.5 milímetros, la medida estándar de la mayoría de los auriculares comerciales. Está preparado para conectarse directamente a la FPGA.

Este TFG no incluye la implementación este módulo. Sin embargo, el resto de los módulos han sido diseñados para funcionar en conjunto con él. Este periférico se estudiará en detalle en el apartado 6) de trabajos futuros.

4) **PMOD JSTK2 (NO APLICA):** Permitiría al usuario controlar los ángulos de acimut y elevación a través de dos *JoySticks* conectados a la FPGA. Se trata de un circuito integrado que incorpora un *JoyStick* de dos ejes y que está preparado para conectarse directamente a la FPGA. Dependiendo de la posición de los *JoySticks*, un controlador traduciría los ángulos de acimut y elevación en una dirección binaria que se enviaría a la memoria.

   Este TFG no incluye la implementación este módulo. Sin embargo, el resto de los módulos han sido diseñados para funcionar en conjunto con él. Este periférico se estudiará en detalle en el apartado 6) de trabajos futuros.

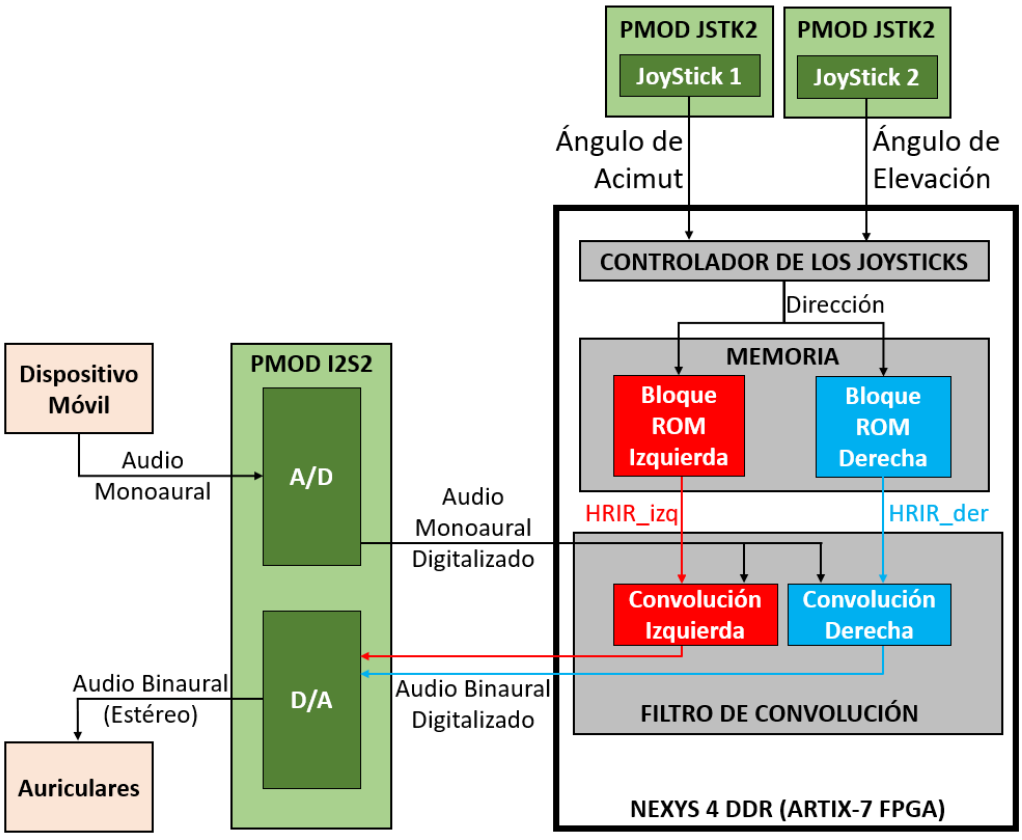


Figura 10. Diagrama de bloques del sistema de Audio 3D detallado

En la Figura 10 se puede observar el mismo diagrama de bloques de la Figura 9 pero incluyendo los detalles del interior de cada bloque. Cabe resaltar cómo se diferencia entre canal izquierdo y canal derecho en los bloques Filtro de Convolución y Memoria.

Las muestras de ambos canales siempre se mantienen debidamente separadas. Debido a que el número de multiplicadores es limitado, es imperativo que ambos canales compartan algunas secciones dentro del filtro de convolución. Por ello es muy importante y siempre se debe evitar que una muestra acabe en un registro indebido y contamine la operación de la convolución.

Para el diseño general se han tenido en cuenta los recursos hardware de la FPGA ya que las limitaciones del diseño vienen dadas por el número de recursos disponibles, en concreto, el número de multiplicadores y el tamaño de la memoria. El filtro de convolución requiere de 200 multiplicadores, uno por cada muestra de la HRIR, si se desea multiplicar todos los coeficientes de una vez.

3.1 Recursos hardware de la FPGA

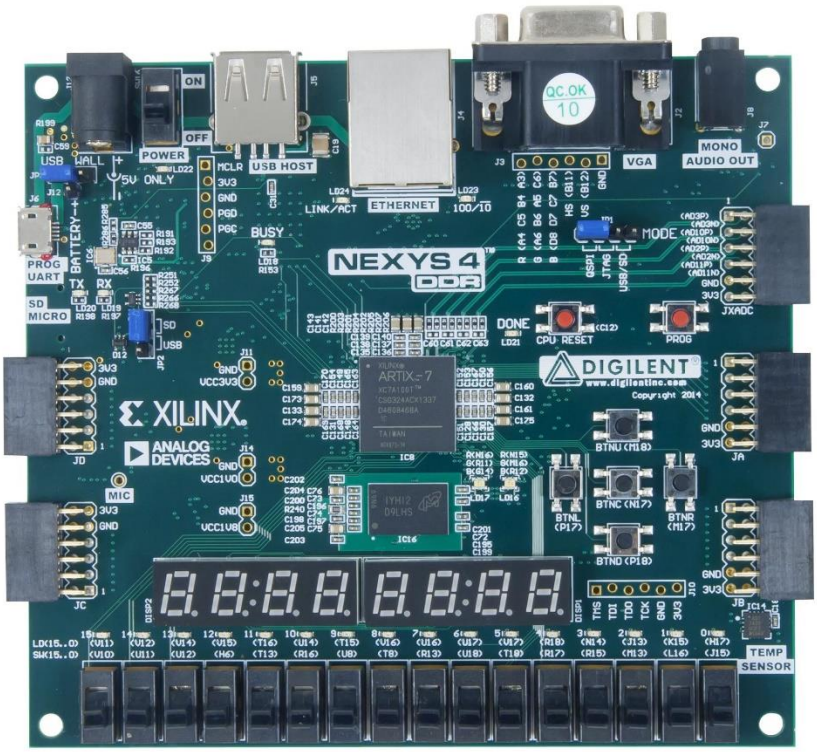


Figura 11. Fotografía de la *FPGA Nexys 4 DDR*

La placa seleccionada para la realización de este TFG es la **FPGA Nexys 4 DDR de la familia A7-100T (Artix-7™)** y se puede observar en la Figura 11**.** «Se trata de una plataforma de desarrollo de circuitos digitales completa y lista para usarse basada en la matriz de puerta programable en campo (FPGA) Artix-7 ™ de Xilinx®. Con su FPGA de gran capacidad, generosas memorias externas y una colección de puertos USB, Ethernet y otros, el Nexys A7 puede albergar diseños que van desde circuitos combinacionales introductorios hasta potentes procesadores integrados. Dispone de varios periféricos incorporados, que incluyen un acelerómetro, sensor de temperatura, micrófono digital MEMs, un amplificador de altavoz y varios dispositivos de E/S permiten que el Nexys A7 se utilice para una amplia gama de diseños sin necesidad de otros componentes». [13]

Sin embargo, las tres características más importantes de la FPGA para este proyecto son sus 240 DSP (*Digital Signal Processor*) slices, su bloque de memoria ROM (*Read Only Memory*) de 16 MiB y su reloj interno de 100MHz.

3.1.1 Bloques DSP48E1

Una FPGA está compuesta de diferentes recursos internos, entre ellos, los bloques DSP. Un bloque DSP se encarga de realizar operaciones matemáticas basadas en multiplicaciones a alta velocidad. «El bloque DSP consta de un multiplicador seguido de un acumulador. Se requieren al menos tres registros segmentados para que las operaciones de la multiplicación y multiplicación-acumulada se ejecuten a la máxima velocidad. La operación de la multiplicación de la primera etapa genera dos productos parciales que necesitan sumarse en la segunda etapa». [14]

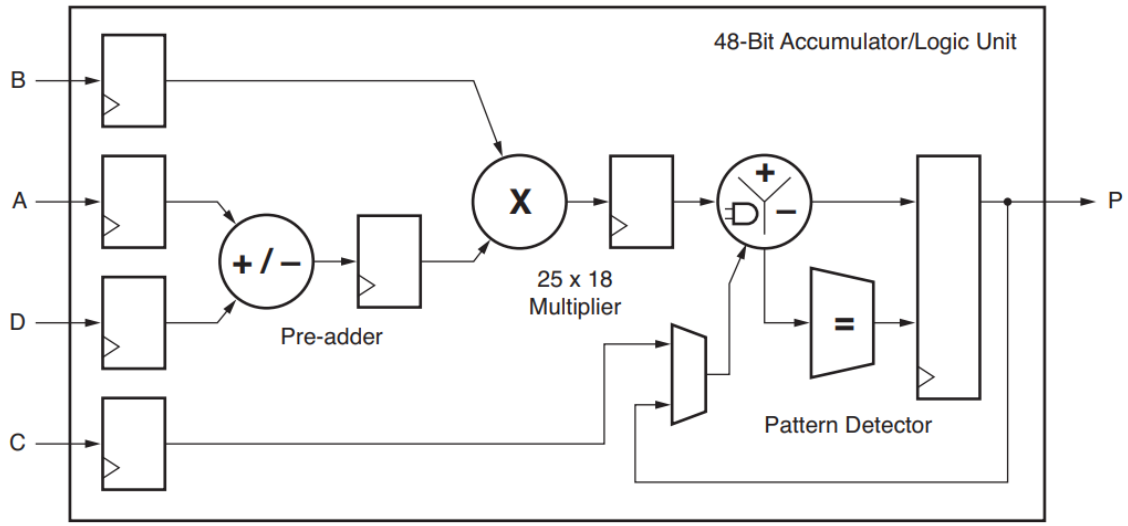


Figura 12. Funcionalidad básica de un DSP48E1 Slice

Las partes principales de un bloque DSP son el *Pre-adder,* el multiplicador, la unidad lógica y la lógica de detección de patrón.

«El bloque DSP incluye un *Pre-adder* de 25 bits, que se inserta en la ruta del registro A como se muestra en la Figura 12. En el *pre-adder, las* pre-sumas o pre-restas son posibles antes de alimentar el multiplicador. Dado que el *pre-adder* no contiene lógica de saturación, los diseñadores deben limitar los operandos de entrada a 24 bits extendidos con signo de complemento a dos para evitar el desbordamiento o el subdesbordamiento durante la operaciones aritméticas. Opcionalmente, el pre-sumador se puede omitir, lo que convierte a D en la nueva ruta de entrada al multiplicador. Cuando no se utiliza la ruta D, la salida de la tubería A se puede negar antes de conducirla el multiplicador. Hay hasta 10 modos de funcionamiento, lo que hace que este bloque de *pre-adder* sea muy flexible». [14]

Un bloque DSP contiene un solo multiplicador 25x18 capaz de realizar operaciones con dos entradas en complemento a 2, una de 25 bits y otra de 18 bits como máximo. El multiplicador produce dos productos parciales de 43 bits.

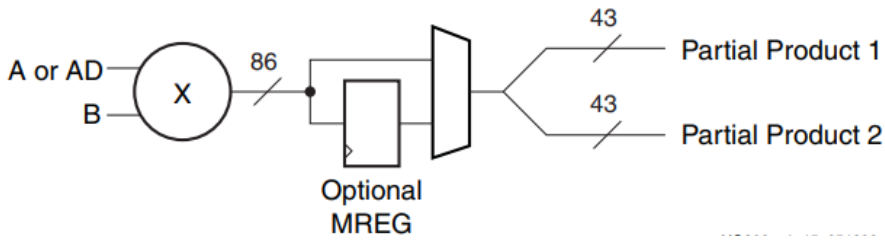


Figura 13. Multiplicador de complemento a 2 seguido de un registro opcional

Ambos productos parciales dan como resultado final un producto final de 86 bits a la salida del multiplicador, como muestra la Figura 13. La figura también muestra un registro opcional a la salida del multiplicador. Usar este registro incrementa el rendimiento con un incremento de la latencia de un ciclo de reloj.

En cuanto a la unidad lógica, «en los dispositivos de la serie 7, la capacidad de realizar sumas, restas y funciones de lógica simple en el bloque DSP existen mediante el uso de un sumador de tres entradas de segunda etapa». [14]

Por último, la lógica de detección de patrón se conecta a la salida de la unidad lógica como se observa en la Figura 12. Sus funciones principales son realizar redondeos convergentes o simétricos y funciones de 96 bits de ancho de bus cuando se actúa en conjunción con la unidad lógica.

En cada ciclo de muestreo se debe multiplicar una muestra de la HRIR con una muestra de la señal de entrada del sistema. Son necesarios, al menos, 200 multiplicadores para realizar todas las multiplicaciones en el mismo ciclo de trabajo. Dado que la FPGA de la tarjeta de evaluación Nexys 4 DDR dispone de 240 DSP Slices, se cumple el número mínimo de multiplicadores que necesita el sistema para realizar todas las multiplicaciones de un solo canal en el mismo ciclo de reloj.

Sin embargo, no es posible realizar las multiplicaciones de los dos canales de audio al mismo tiempo dado que la FPGA necesitaría de 400 multiplicadores como mínimo. Por este motivo, el filtro de convolución realiza primero las multiplicaciones de un canal y a continuación las del otro, haciendo que la convolución de los dos canales se realice de forma segmentada.

Si otra FPGA sin suficientes DSP Slices hubiera sido empleada, aun así, habría sido posible implementar el filtro de convolución. En ese caso, el problema vendría en que la dificultad de la gestión interna del filtro de convolución aumentaría considerablemente. Las multiplicaciones se deberían hacer por partes. Por ejemplo, una FPGA con 50 DSP Slices necesitaría de cuatro ciclos de reloj y acumular los resultados en registros que serían sumados posteriormente. Se realizaría la multiplicación de las 50 primeras muestras, y a continuación, la multiplicación de las 50 siguientes. Así hasta finalizar con las 200 muestras. Disponer de 200 DSP Slices simplifica la gestión interna haciendo posible que todos los multiplicadores actúen con las 200 muestras de una HRIR a la vez.

3.1.2 Memoria flash

La FPGA Nexys 4 DDR dispone de 16 MiB (Mebibyte) de memoria ROM interna. El modelo que se emplea en este TFG (*FPGA Artix-7 100T*) requiere menos de 4 MiB de memoria para almacenar los archivos de configuración, dejando libre el 77% de la memoria flash para el usuario. Asimismo, si la FPGA de configura desde otra fuente que no sean los archivos de configuración, toda la memoria puede ser usada para almacenar datos personalizados.

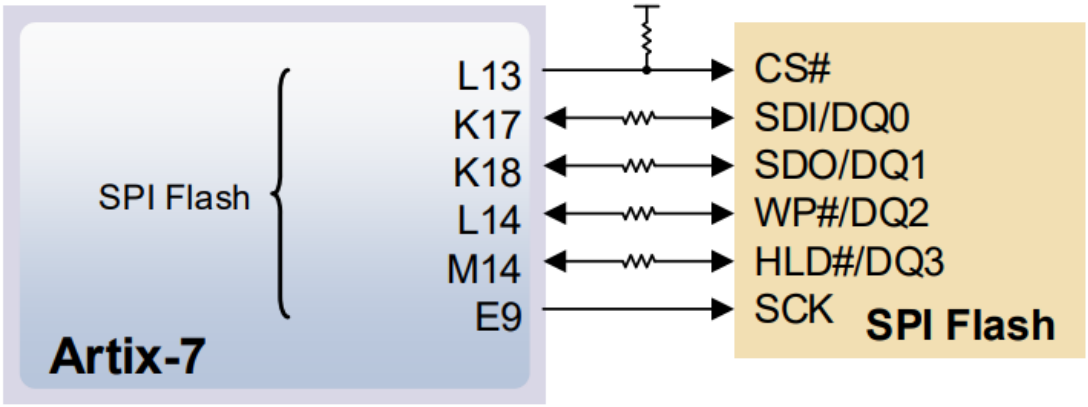


Figura 14. Pin-out de la SPI flash de la Nexys 4 DDR [6]

La Figura 14 muestra la interfaz que conecta la Artix-7 con la memoria ROM. «Los contenidos de la memoria se pueden manipular a partir de ciertos comandos sobre el bus SPI (*Serial Peripheral Interface*) … Todas las señales del bus SPI excepto SCK (*Serial Clock*) son de propósito general para E/S de usuario después de la configuración de la FPGA. SCK es una excepción porque mantiene un pin dedicado incluso después de la configuración». [15]

Para comprobar si la base de datos cabría dentro de la FPGA es necesario calcular el tamaño de todas las respuestas al impulso. Cuando se grabaron las HRIR, la salida del micrófono se digitalizó a 44.1 kHz con una resolución de 16 bits. [4] Si cada muestra por tanto son 16 bits, y una HRIR se compone de 200 muestras, el tamaño de una HRIR es:

$$\boldsymbol{HRIR = 16 * 200 = 3200\ bits = 400\ Bytes}$$

Por otro lado, el número total de direcciones espaciales, y por tanto, de HRIR es:

$$\boldsymbol{Nº\ HRIR = 25\ [Acimuts] * 50\ [Elevaciones] = 1250}$$

Para calcular el tamaño total que ocuparía la base de datos, hay que tener en cuenta ambos dos canales: izquierdo y derecho, y multiplicar por un factor de dos:

[6] https://docs.rs-online.com/2bb6/0900766b81592f18.pdf

$$\boldsymbol{Tamaño\ Total = 400 * 1250 * 2\ [canales] = 1000000\ Bytes = 1MB}$$

Un MiB equivale a 2^{20} (1048576) bytes, es decir, aproximadamente 1 MB. Todas las respuestas al impulso de la base de datos HRTF de un sujeto ocupan 1MB, siempre y cuando se almacenen todas y cada una de las respuestas al impulso. El tamaño disponible de memoria interna ROM es de 16MB aproximadamente, así que la base de datos de un sujeto cabría por completo sin ninguna limitación de memoria ya que sobrarían 15MB aproximadamente (sin usar ficheros de configuración). Si se quisiera emplear una base de datos completa con todas las direcciones no sería necesario emplear memoria *flash* externa.

El bloque de memoria BRAM (*Buffer Random Access Memory*) permite almacenar datos como si se tratase de una memoria ROM a partir del uso de un IP (*Intelectual Property*) de Xilinx®. Concretamente, se emplean bloques generadores de memoria para crear memorias ROM de un solo puerto de lectura. Se utilizan dos bloques de memoria ROM, una para guardar las HRIR del canal derecho y otra para las del canal izquierdo

3.1.3 Oscilador/Reloj

La tarjeta Nexys A7 incluye un único oscilador de cristal de 100 MHz. Este oscilador permite generar varias frecuencias de reloj con desfases conocidos para la elaboración de diseños. Sin embargo, este TFG emplea una sola frecuencia de reloj de 100MHz. La velocidad del filtro de convolución viene limitada por la frecuencia de muestreo y no por la velocidad de la FPGA. De hecho, la FPGA realiza las operaciones tan rápido que el resto del tiempo permanece esperando nuevas muestras de entrada.

La frecuencia de muestreo del audio es $f_s = 44.1\ kHz$. Esta frecuencia se corresponde con la frecuencia con la que se grabaron las HRIR y será la frecuencia de muestreo que se emplee para digitalizar el audio que entra a la FPGA a través del PMOD.

El tiempo que pasa entre cada muestra es la inversa de la frecuencia de muestreo

$$f_s = \frac{1}{44100\ s^{-1}} = 22.67\text{us}$$

Sea **Audio[N]** un array de muestras de audio digitalizado donde *n* significa la posición en la que muestra fue digitalizada en la FPGA.

Figura 15. Línea temporal del periodo de muestreo

La Figura 15 muestra el tiempo que pasa entre cada muestra y muestra. Como el reloj del sistema funciona a 100MHz:

$$\frac{100\ Mhz}{44.1\ kHz} = \frac{22.67\ us}{10\ ns} = 2267\ ciclos\ de\ reloj$$

El sistema dispone de 2267 ciclos de reloj para realizar todas las operaciones y obtener el resultado de salida antes de la siguiente muestra. En apartados posteriores se comprobará cómo estos ciclos de reloj son suficientes para realizar la convolución.

3.2 Módulos del sistema de Audio 3D

En este apartado se estudiarán en detalle dos de los cuatro módulos que conforman el sistema de Audio 3D: el filtro de convolución y la memoria. Los dos módulos restantes se estudian en el apartado 6 de trabajos futuros. Siguiendo el modelo *Top-down*, primero se analizarán los módulos partir de sus entradas y salidas, sin importar cómo esté organizada la lógica interior.

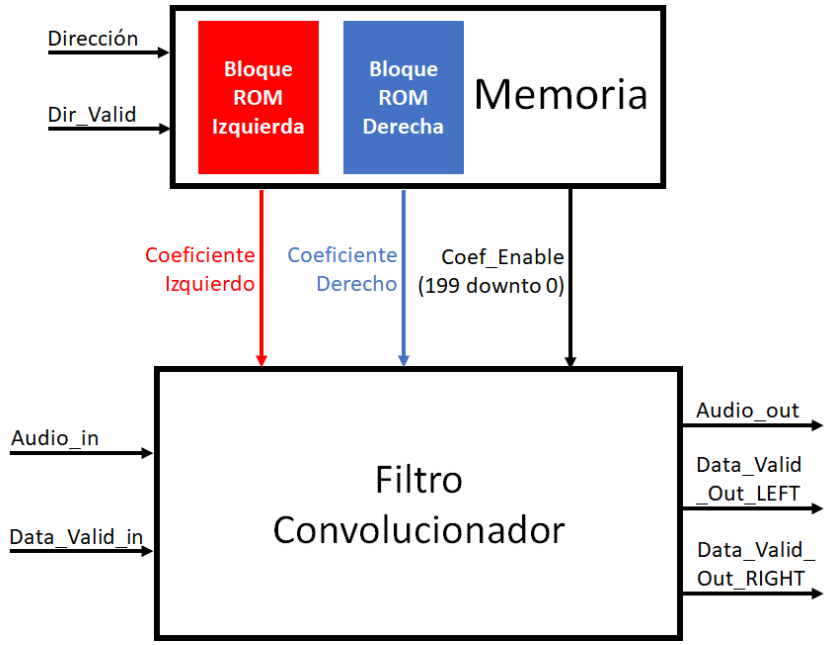


Figura 16. Diagrama de bloques con las E/S del sistema de Audio 3D

En la Figura 16, se observa que las señales E/S están acompañadas de una señal de validación o *enable*. Estas señales actúan como disparador o *trigger* de las máquinas de estado que hay implementadas dentro de los módulos.

3.2.1 Filtro de convolución

El filtro de convolución genera audio biaural a través de la convolución segmentada del canal derecho y del canal izquierdo. Para llevarla a cabo, necesita de una señal de audio y de una HRIR que son obtenidas de otros módulos, ya que dentro del filtro solo se haya la lógica de convolución (multiplexores, sumadores, registros, etc …).

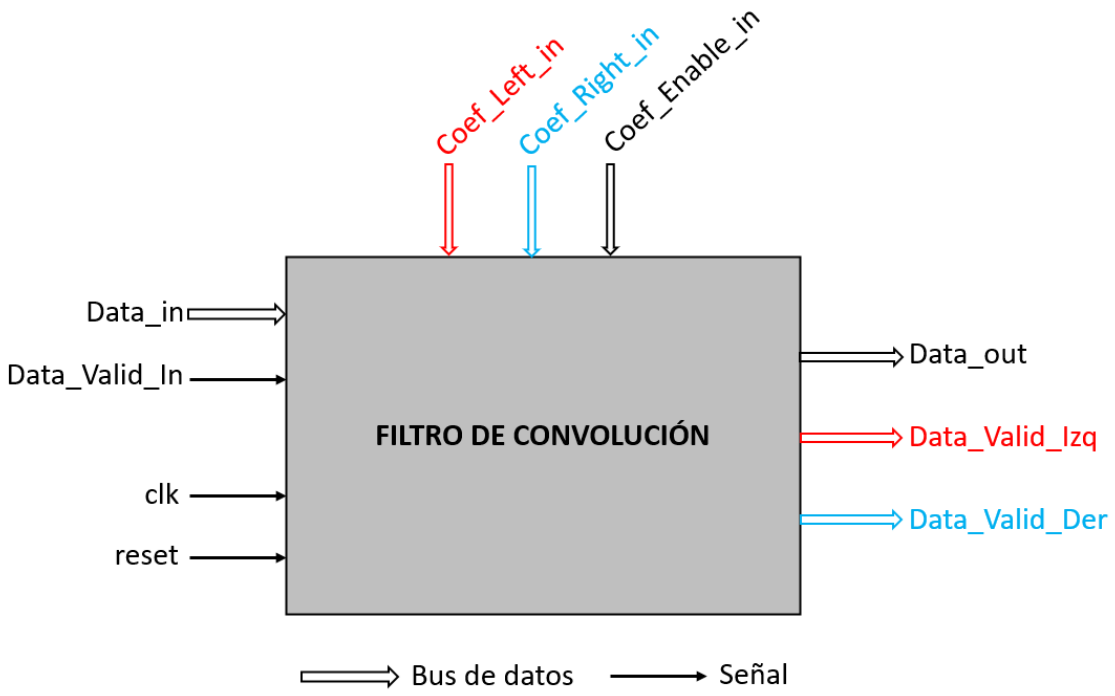


Figura 17. Entradas y salidas del filtro de convolución

3.2.1.1 Entradas

Existen dos entradas que son comunes a todos los módulos dado que su función es de vital importancia para el correcto funcionamiento del sistema. Estas señales son la **señal de reloj** y la **señal de reset**.

La **señal de reloj** o **clk** es una señal binaria que sirve para coordinar las acciones de varios circuitos. Se genera a partir del reloj interno de la FPGA y en este TFG funciona a 100MHz. Se trata de una señal común a todos los módulos para asegurar que se mantiene la sincronía cuando ocurre un evento que afecta a módulos diferentes. También realiza las particiones de tiempo o ciclos de trabajo que los componentes toman como referencia para realizar sus funciones.

La **señal de reset** es una señal binaria asíncrona que funciona a nivel alto y posibilita reinicializar todo el sistema al instante inicial. En este módulo, se encarga de poner todos los registros a cero como si todavía no se hubiera cargado ninguna señal y reinicia la máquina de estados que realiza la convolución.

Las muestras de la señal de audio monoaural se reciben a través del bus **Data_in** a una frecuencia de 44.1 MHz. El bus **Data_in** estaría conectado al módulo PMOD de audio que muestrearía la señal de audio que se recibiría de una fuente externa. Cada vez se haya obtenido una nueva muestra a través del módulo PMOD de audio, éste indicará al filtro de convolución que existe una nueva muestra con la que se debe realizar una operación de la convolución. Esto se indica mediante la activación de la señal **Data_valid_in,** que, por tanto, funciona a la misma frecuencia que la frecuencia de muestreo 44.1 MHz y además solo tiene efecto en los flancos ascendentes de la señal de reloj.

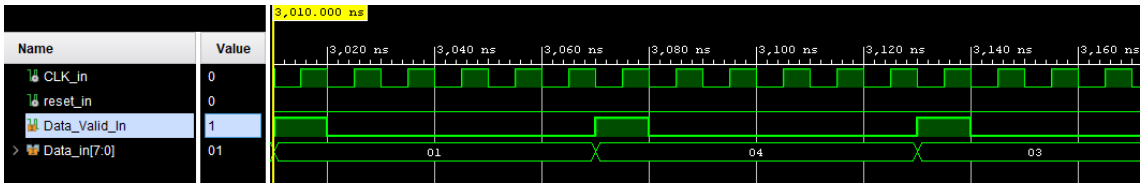


Figura 18. Cronograma de la señal de entrada al filtro de convolución

Como se puede observar en la Figura 18, entre ciclos de muestreo representados por la señal **Data_Valid_in** el dato se mantiene inalterado para que la lógica interna del filtro de convolución pueda disponer de él cuando se precise. Este dato solo cambia cuando es sustituido por la nueva muestra que llega en el siguiente ciclo de muestreo.

Cuando ocurre un flanco ascendente de la **señal de reloj** y **Data_Valid_In = '1'** se inicia la máquina de estados que ejecuta las operaciones de la convolución. Esta máquina de estados tomará el dato que se haya disponible en el bus **Data_in** y lo multiplicará con la respuesta al impulso que se haya cargado previamente en el filtro

Las respuestas al impulso almacenadas en la memoria ROM se reciben a través de los buses **Coef_Left_in** y **Coef_Right_in**. Estos buses sirven para enviar uno a uno los 200 coeficientes que conforman una HRIR. Cuando una nueva HRIR tiene que ser cargada, la señal **Coef_Enable_in** es la encargada de dirigir cada coeficiente al registro correspondiente para almacenarla y que pueda ser empleada en las operaciones.

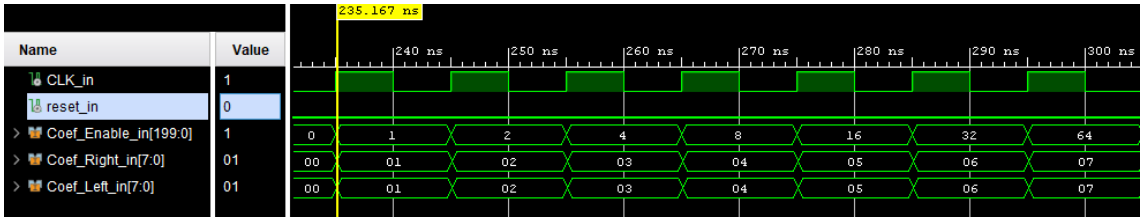


Figura 19. Cronograma de los buses de memoria del filtro de convolución

La Figura 19 muestra cómo la señal **Coef_Enable_in** activa en cada ciclo de reloj un bit diferente, por eso sus valores son del tipo 2^n, ya que los bits son puestos a nivel alto en orden de menor peso a mayor peso. Esta señal nunca tendrá dos bits simultáneamente a nivel alto. Esto se debe a que cada bit activa un registro diferente del filtro convolucionador y no se deben activar dos o más para evitar que los registros tomen muestras iguales. Los buses de entrada **Coef_Right_In** y **Coef_Left_In** llevan las muestras de la memoria ROM a los registros del filtro convolucionador. Todos los registros toman las muestras de estos buses, solo que lo hacen cuando se ha activado su bit correspondiente de la señal **Coef_Enable_In.**

3.2.1.2 Salidas

La señal de audio biaural se conforma de dos muestras, la del canal derecho y la del canal izquierdo. La muestra del audio 3D se extrae del filtro en dos ciclos de trabajo dado que se emplea el bus mismo **Data_out** para ambos canales. Por este motivo, son

necesarias dos señales auxiliares **Data_Valid_izq** y **Data_Valid_der** para indicar si la muestra que se encuentra en el bus

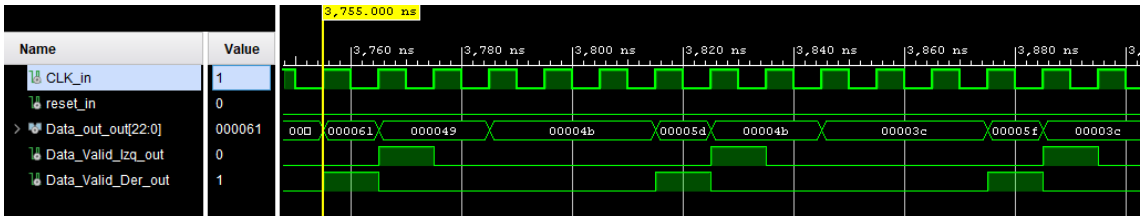


Figura 20. Cronograma de los buses de salida del filtro de convolución

Data_out se corresponde con el canal izquierdo o el canal derecho respectivamente. Al igual que sucede con la entrada, la frecuencia de salida es 44.1 MHz porque cuando una muestra entra al filtro otra debe salir para dejar hueco en los registros. Cabe resaltar que para mostrar los máximos resultados posibles, en la Figura 20 no se ha simulado a dicha frecuencia. Así, en la figura, se aprecia cómo las señales **Data_valid_Izq_out** y **Data_valid_Der_out** son las encargadas de determinar si la muestra de **Data_out** se corresponde con el canal derecho o con el canal izquierdo.

3.2.1.3 Lógica interna

Debido a la limitación del número de DPS Slices, y por tanto, de multiplicadores, no es posible crear dos filtros de convolución para los dos canales. La convolución debe de realizarse en el mismo filtro de convolución alternando las muestras del canal izquierdo y las muestra del canal derecho.

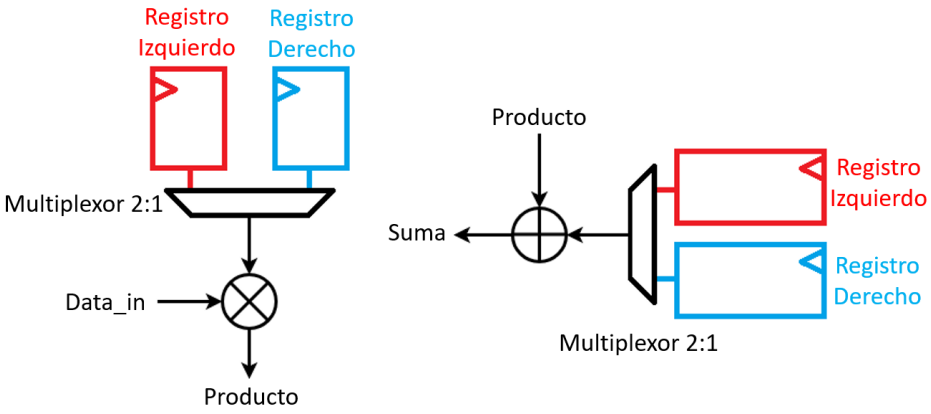


Figura 21. Disposición de los multiplexores y los elementos aritméticos

La Figura 21 muestra cómo se colocan los multiplexores antes de los elementos aritméticos, es decir, de los sumadores y de los multiplicadores. Los multiplexores son de tipo 2:1, dos entradas y una salida. Así es posible realizar la convolución de manera segmentada ya que, de esta forma, se consigue que todas las partes del circuito puedan alternar entre las muestras de ambos canales.

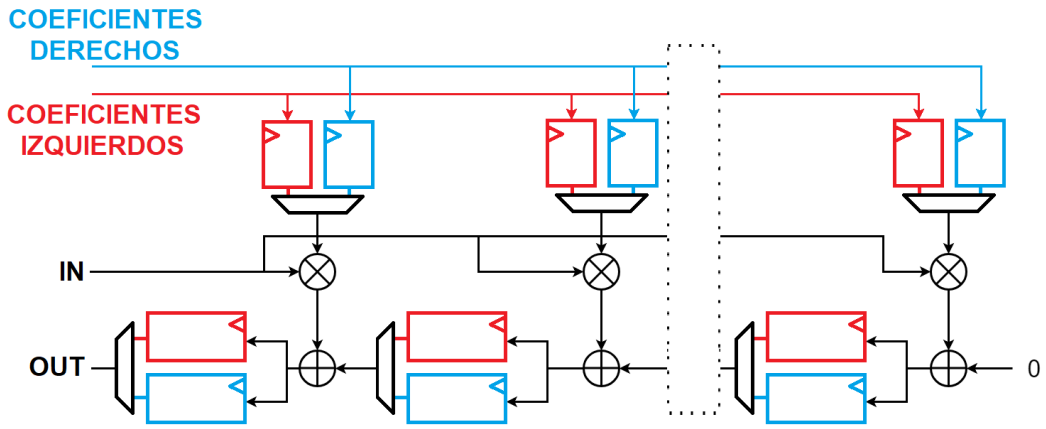


Figura 22. Esquema en detalle del circuito de convolución

Asimismo, se coloca un registro por canal antes de cada multiplexor para almacenar las muestras hasta que deban usarse en el siguiente ciclo de muestreo. El esquema general queda como en la Figura 22.

Todos los elementos del circuito, desde los registros hasta los multiplicadores están conectados a la **señal de reloj**, por lo que son elementos síncronos que actúan en cada flanco ascendente de **clk**. Por este motivo, algunos elementos como los sumadores y los multiplicadores están operando continuamente los datos que tienen en sus entradas sin que los resultados se estén guardando en ningún registro. Esto se debe a que lo registros tienen una señal de control que indica cuando se deben guardar las muestras. Todo se regula mediante una máquina de seis estados.

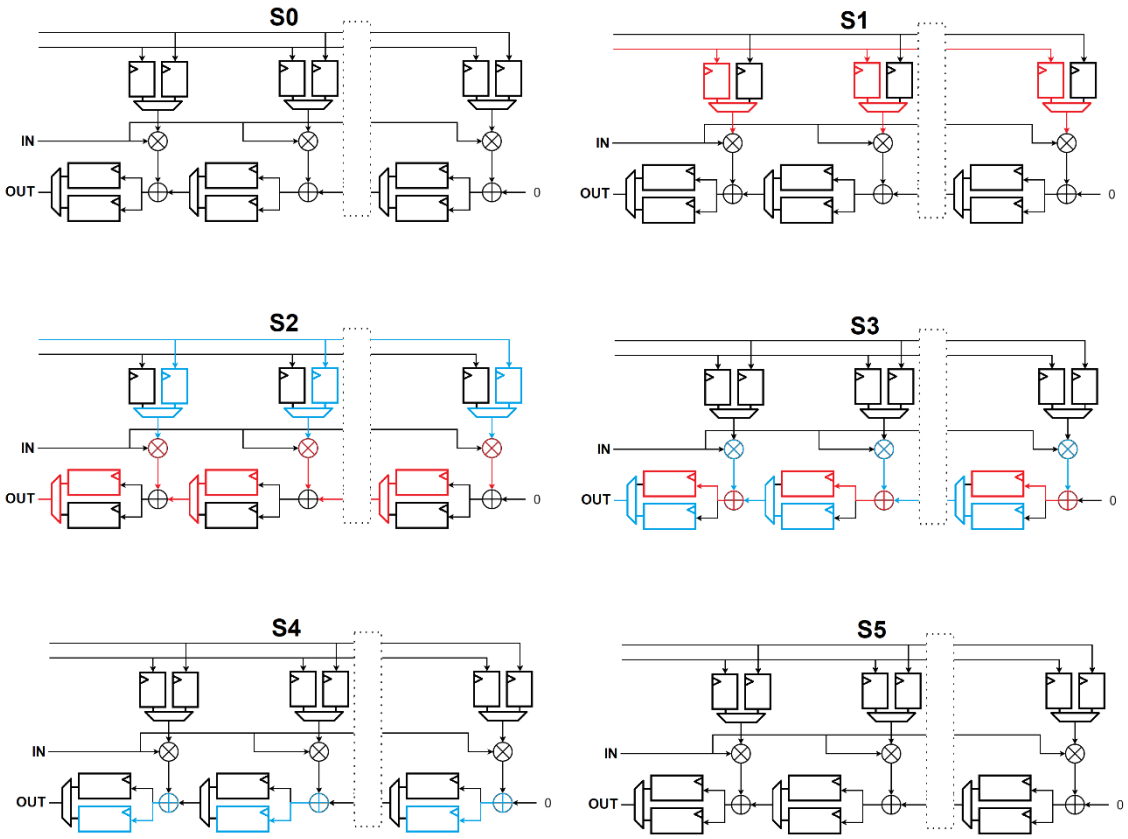


Figura 23. Representación gráfica de la máquina de estados del filtro de convolución

Estos son los estados de la máquina de estados dibujada en la Figura 23:

1) **Estado S0:** Estado de reposo o *standby.* Espera a que la señal **Data_Valid_in** se ponga a nivel alto para pasar al estado S1 y comenzar las operaciones aritméticas. Esto supondría que la muestra en el bus de entrada **Data_in** es nueva y hay que realizar la convolución con ella.

2) **Estado S1:** CANAL IZQUIERDO, Se activa la señal que selecciona los registros izquierdos en los multiplexores de forma que se dejen pasar los coeficientes hacia el multiplicador. CANAL DERECHO, no se realizan cambios.

3) **Estado S2:** CANAL IZQUIERDO, los multiplexores de los *datapath* seleccionan el canal izquierdo para que los sumadores realicen la suma de los productos de los multiplicadores (cuyo resultado se ha obtenido de la multiplicación de la señal de entrada y los coeficientes izquierdos en el anterior ciclo de reloj) con las muestras almacenadas en los registros desde el anterior ciclo de muestreo. En la salida del filtro se saca la muestra de audio 3D fuera del filtro a través del *datapath[0]* izquierdo, a la vez que se pone a nivel alto la señal validadora **Data_valid_izq**.

   CANAL DERECHO, la señal que en el estado anterior había seleccionado los coeficientes izquierdos en el multiplexor ahora conmuta y selecciona los coeficientes derechos para dejarlos pasar hasta el multiplicador.

4) **Estado S3:** CANAL IZQUIERDO, se almacena el resultado obtenido de los sumadores en los registros. Así quedan almacenados hasta usarse en el próximo ciclo de muestreo. La señal **Data_valid_izq se pone a nivel bajo.**

   CANAL DERECHO, los multiplexores de los *datapath* seleccionan el canal derecho para que los sumadores realicen la suma de los productos de los multiplicadores (cuyo resultado se ha obtenido de la multiplicación de la señal de entrada y los coeficientes derechos en el anterior ciclo de reloj) con las muestras almacenadas en los registros desde el anterior ciclo de muestreo. En la salida del filtro se saca la muestra de audio 3D del *datapath[0]* derecho a la vez que se pone a nivel alto la señal validadora **Data_valid_der**.

5) **Estado S4:** CANAL IZQUIERDO, se apaga la señal de refresco de los registros izquierdos para que no se pierdan las muestras que se han guardado en el estado anterior.

   CANAL DERECHO, se almacena el resultado obtenido de los sumadores en los registros. Así quedan almacenados hasta usarse en el próximo ciclo de muestreo. La señal **Data_valid_der** se pone a nivel bajo.

6) **Estado S5:** CANAL IZQUIERDO, no se realizan cambios.

   CANAL DERECHO, se apaga la señal de refresco de los registros derechos para que no se pierdan las muestras que se han guardado en el estado anterior.

En total se emplean 6 ciclos de reloj desde que se activa la señal **Data_valid_in** que inicia la máquina de estados hasta que se han almacenado todas las nuevas muestras de las dos convoluciones en sus respectivos registros.

3.2.1.4 Lógica de convolución

La lógica interna de la **FPGA Nexys 4 DDR de la familia A7-100T (Artix-7™)** aporta un beneficio respecto al resto de tecnologías. Permite disponer todos sus elementos de manera que se obtiene un circuito que proporciona los mismos resultados que una circunvolución y que además realiza las operaciones en tiempo real.

La convolución entre una muestra de audio monoaural y la HRIR da como resultado sonido 3D. Esta convolución se logra mediante una larga cadena de registros, conteniendo 200 registros *datapath* para cada uno de los canales de audio izquierdo y derecho. La entrada introduce las muestras de la señal de audio en formato mono que entran al circuito con una frecuencia de 44.1 kHz. Se disponen 200 multiplicadores y 200 sumadores para ejecutar el método de la multiplicación polinómica. Cada uno de los multiplicadores recibe la misma señal de entrada mono y una de las 200 muestras de la HRIR almacenadas en memoria. Los sumadores reciben la salida de un multiplicador y la suma acumulada de los sumadores anteriores.

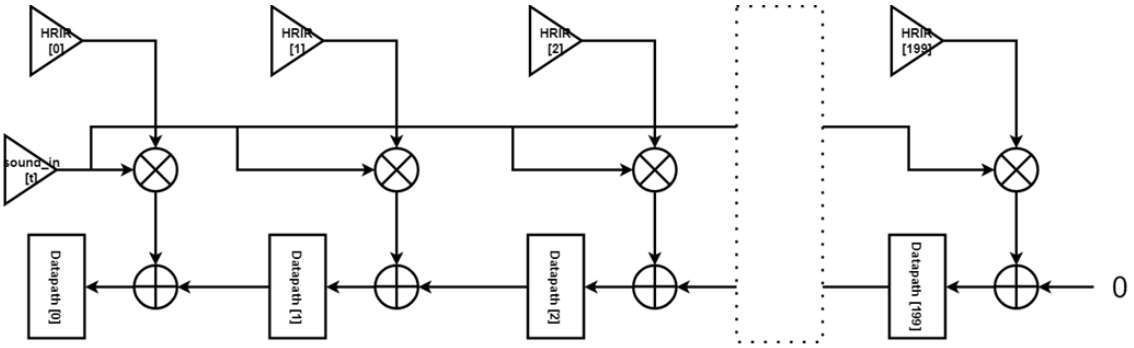


Figura 24. Lógica del circuito convolucionador. [7]

A partir de la Figura 24 se va a estudiar la tabla de entradas y salidas. Por simplicidad, solo se ha representado uno de los canales. Los resultados del circuito deben ser idénticos a la teoría de la función *conv* de MATLAB® ya que las primeras simulaciones se hacen mediante dicha función, la cual realiza las operaciones descritas en el apartado 2.2. Como se verá a continuación, el circuito de la Figura 24 arrojará la misma tabla de resultados que la función *conv* de MATLAB®, por lo que se considerará que el diseño es válido.

Realizando el estudio del circuito para cada ciclo de muestreo se obtiene la siguiente tabla:

Tabla 3. Resultados del circuito convolucionador en tiempo discreto

t	Datapath[0] (Output)	Datapath[1]	Datapath[2]
0	S[0]H[0]	S[0]H[1]	S[0]H[2]
1	S[0]H[1] + S[1]H[0]	S[0]H[2] + S[1]H[1]	S[0]H[3] + S[1]H[2]
2	S[0]H[2] + S[1]H[1] +S[2]H[0]	S[0]H[3] + S[1]H[2] + S[2]H[1]	S[0]H[4] + S[1]H[3] + S[2]H[2]
3	S[0]H[3] + S[1]H[2] + S[2]H[1] + S[3]H[0]	S[0]H[4] + S[1]H[3] + S[2]H[2] + S[3]H[1]	S[0]H[5] + S[1]H[4] + S[2]H[3] + S[3]H[2]

El registro *Datapath[0]* de la Figura 24 se corresponde con la salida del circuito. En cada ciclo de muestreo el filtro de convolución saca hacia fuera uno de los coeficientes de la señal resultante de la convolución. De esta forma, la salida del circuito valdrá S[0]H[0] en el instante cero, esto es, la multiplicación entre la primera muestra de la señal de audio y la primera muestra de la HRIR. Al comprobar el desarrollo matemático del apartado 2.2 con la Tabla 3, se observa que las salidas en el primer instante son idénticas, al igual que para los instantes posteriores:

$$t = 0 \rightarrow S[0]H[0] = u(0) * v(0)$$

$$t = 1 \rightarrow S[0]H[1] \; + \; S[1]H[0] = u(0) * v(1) + u(1) * v(0)$$

$$t = 2 \rightarrow \mathrm{S[0]H[2]} + \; \mathrm{S[1]H[1]} + \mathrm{S[2]H[0]} = u(0) * v(2) + u(1) * v(1) + u(2) * v(0)$$

[7] http://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2018/jgc232_mmm389_jw2299/hrtf_explorer/hrtf_explorer/ECE%205760%20Final%20Project.htm

De esta manera, se deduce que el circuito convolucionador es correcto dado que su comportamiento es el mismo que el de la función *conv* de MATLAB®.

3.2.2 Memoria

La memoria almacena las muestras de las HRIRs en dos bloques ROM, uno para el canal derecho y otro para el canal izquierdo. Los bloques de memoria se configuran como Memorias ROM de un solo puerto y se encuentra distribuidas de forma que todas las respuestas al impulso estén almacenadas una detrás de otra. Cada 200 posiciones comienzan los coeficientes de una nueva HRIR. Cada vez que ocurre un cambio en la dirección proveniente de los JoySticks se cargan nuevos coeficientes en el filtro de convolución.

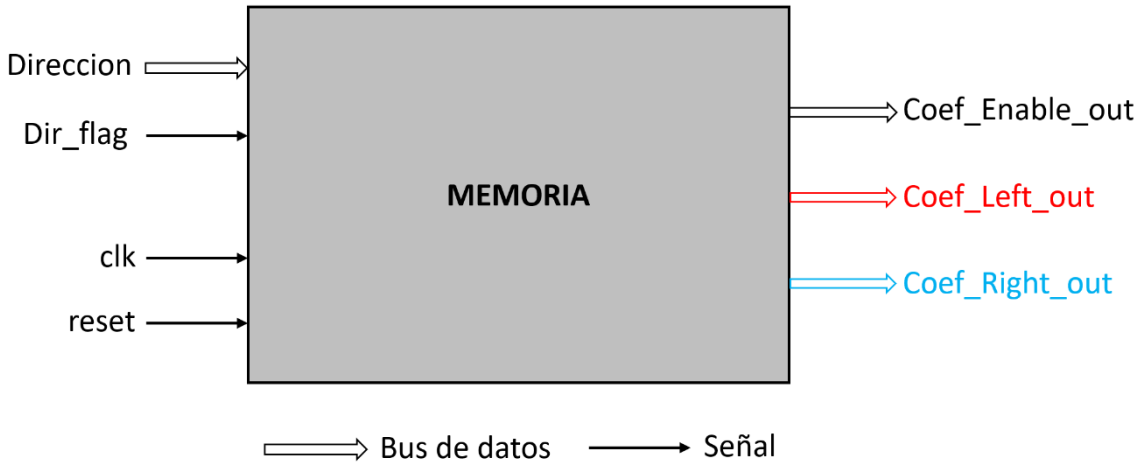


Figura 25. Entradas y salidas de la memoria

La Figura 25 muestra el bloque de memoria como una caja negra donde están representadas sus entradas y salidas, así cómo las señales de validación correspondientes.

3.2.2.1 Entradas

Existen dos entradas que son comunes a todos los módulos dado que su función es de vital importancia para el correcto funcionamiento del sistema. Estas señales son la **señal de reloj** y la **señal de reset**.

La **señal de reloj** o **clk** es una señal binaria que sirve para coordinar las acciones de varios circuitos. Se genera a partir del reloj interno de la FPGA y en este TFG funciona a 100MHz. Se trata de una señal común a todos los módulos para asegurar que se mantiene la sincronía cuando ocurre un evento que afecta a módulos diferentes. También realiza las particiones de tiempo o ciclos de trabajo que los componentes toman como referencia para realizar sus funciones.

La **señal de reset** es una señal binaria asíncrona que funciona a nivel alto y posibilita reinicializar todo el sistema al instante inicial. En este módulo reinicia al estado inicial la máquina de estados que carga los coeficientes en el filtro de convolución.

La dirección binaria donde se encuentra localizada la HRIR que se debe cargar viene dada desde el bus **Direccion.** Los cambios en este bus no iniciarían la máquina de estados, sería necesario que la señal binaria **Dir_flag** se activara a nivel alto para validar que el dato que hay en el bus **Direccion** es correcto.

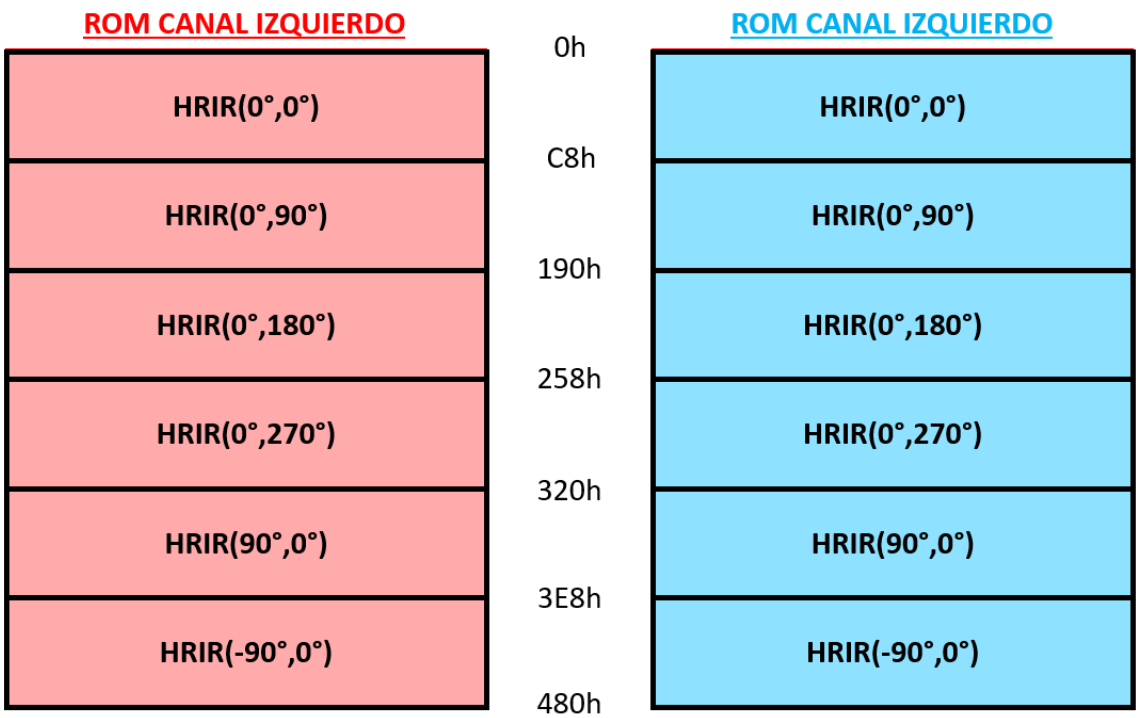
3.2.2.2 Salidas

Para cargar los coeficientes de las HRIRs en el filtro de convolución se emplean dos buses de salida, uno para cada canal de audio. Dado que los bloques de memoria ROM están configurados como memorias de un solo puerto, los 200 coeficientes que conforman la HRIR se mandan de uno en uno a través de **Coef_Left_out** y **Coef_right_out**. Por tanto se necesitarán 200 ciclos de reloj para enviar todos los coeficientes al filtro.

Para indicar a los registros del filtro cuando el coeficiente que se está enviando es el que deben cargar se utiliza el bus de señales **Coef_Enable_out**. Se trata de un bus de 200 bits donde cada bit se conecta a uno de los 200 pares de registros que guardan los coeficientes que van a los multiplicadores. De esta señal solo uno de los 200 bits puede estar a nivel alto.

3.2.2.3 Lógica interna

El bloque de la memoria recibe directamente la dirección donde se aloja la HRIR que tiene que cargar, **no recibe las coordenadas en acimut y elevación**. Debería existir un bloque previo que tradujera las coordenadas a la dirección de la memoria, aunque ese bloque no aplica para este Trabajo de Fin de Grado. Se considera que la dirección de la HRIR que se debe cargar viene por el bus **Dirección** y apunta directamente a la celda de memoria donde se encuentra el primer coeficiente. El mapa de memoria es el siguiente:



(Θ , Φ) = (ACIMUT , ELEVACIÓN)

Figura 26 Mapa de los bloques de memorias ROM [8] [9]

Uno de los objetivos que perseguía este TFG era la creación de un sistema que fuera funcional con una selección de las direcciones espaciales más importantes. En la Figura 26 se observa cómo se almacenan las HRIR en la memoria y en qué dirección binaria comienza cada una. El direccionamiento de la memoria es de 8 bits. Este tamaño se justifica en el apartado 3.3.

Para generar las memorias ROM se genera un archivo *.coe* desde Matlab® y se carga como parámetros iniciales en la IP de Vivado™.

[8] Nota 1) El sistema de coordenadas interaural-polar puede consultarse en la Figura 5.

[9] Nota 2) El tamaño de un coeficiente son 8 bits. Cada bloque de memoria ocupa 1200 bytes en total.

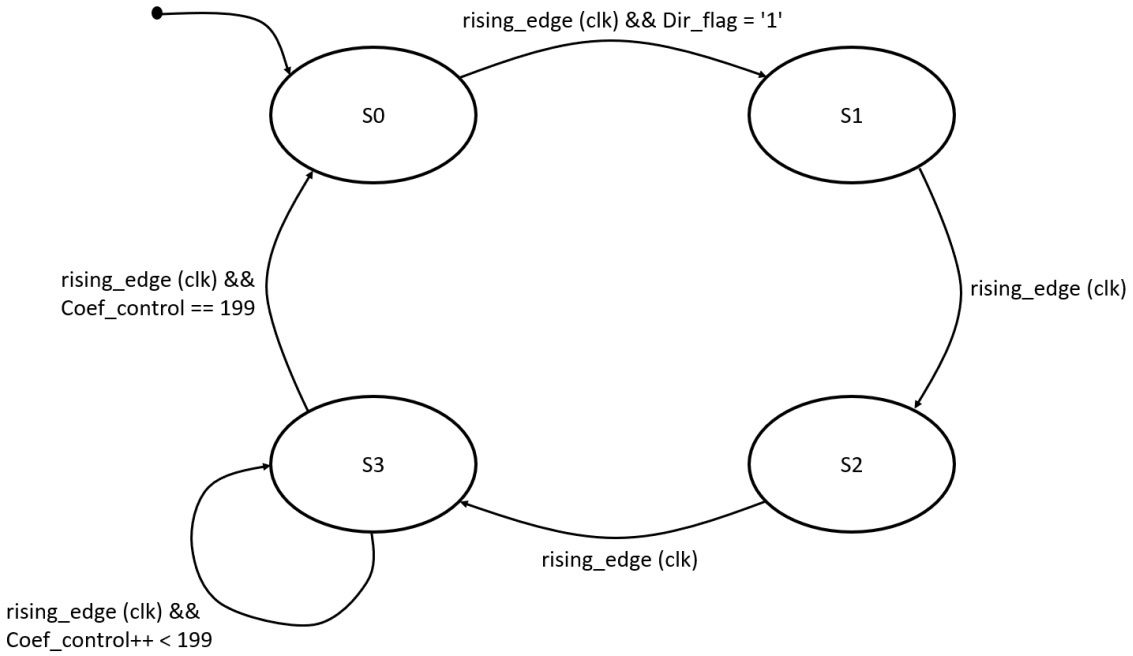


Figura 27. Máquina de estados para cargar los coeficientes en memoria

Para cargar los coeficientes en el filtro de convolución se emplea la máquina de cuatro estados de la Figura 27:

7) **Estado S0:** Estado de reposo o *standby*. Resetea las señales de control y espera el *trigger* de la señal **Dir_flag** para comenzar la carga de coeficientes en el filtro de convolución. Cuando se detecta un cambio en la dirección espacial a través de dicha señal, se pasa al estado S1.

8) **Estado S1:** Carga la nueva dirección del bus **Direccion** en un bux auxiliar que irá aumentando progresivamente su dirección. Este bus se encuentra conectado directamente a los bloques de memoria ROM. De esta forma, se indica a los bloques de memoria ROM qué celda deben sacar al exterior. Como esta dirección es binaria, no hay que convertir las coordenadas de los *JoySticks* puesto que se debería de haber hecho ya en el módulo PMOD de los *JoySticks*.

   Los bloques de memoria no sacan al exterior el coeficiente de la dirección inmediatamente, sino que tardan dos ciclos desde que se carga la nueva dirección. El retardo viene dado por tratarse de un circuito síncrono, que espera los flancos ascendentes de la señal de reloj. Dicho retardo de 2 ciclos de reloj es lo que tarda la muestra que se ha pedido a través del bus **Direccion** en ser estable a la salida del bloque de memoria.

9) **Estado S2:** La dirección del bus auxiliar aumenta en uno dado que la dirección anterior ya ha sido cargada en el bloque de memoria justo al entrar en este estado, en concreto, en el flanco ascendente de reloj. En el siguiente flanco de reloj los bloques ROM sacarán el coeficiente de la dirección que se cargó en el estado S1.

10) **Estado S3:** Estado recurrente que permanece hasta que se hayan cargado los 200 coeficientes en el filtro de convolución. Cada vez que se entra, se aumentan en una unidad las señales de control que llevan la cuenta de los coeficientes que se han cargado hasta el momento. También se continúa aumentando la dirección del bus auxiliar hasta en 198 ocasiones, ya que se deben restar los aumentos que se han realizado en los estado S1 y S2.

A continuación, se muestra un datagrama que explica gráficamente cómo se cargan los coeficientes en el filtro de convolución. Para el ejemplo se emplean las siguientes posiciones de memoria con sus respectivos coeficientes:

Tabla 4. Ejemplo de la disposición de la memoria ROM

Dirección	00h	01h	02h	03h	04h	05h
Coeficiente	0	1	2	3	4	5

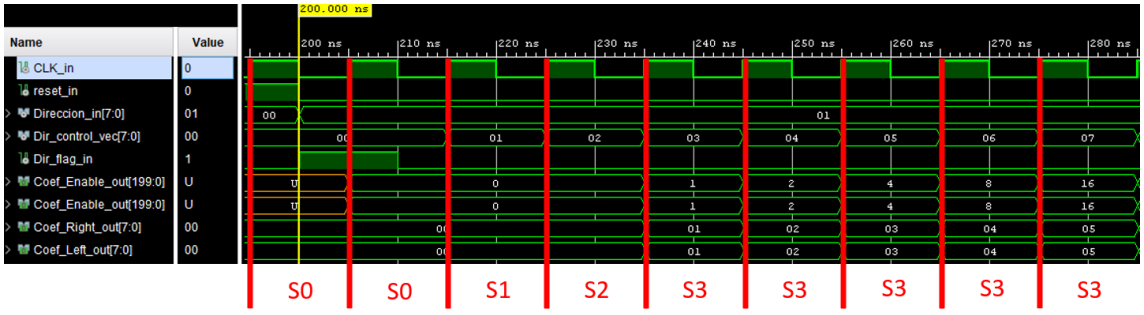


Figura 28. Representación gráfica de la máquina de estados de la memoria ROM

En la Figura 28, en el instante 200ns marcado en amarillo, se activa **Dir_flag** indicando que el bus **Dirección** tiene una nueva dirección, en este caso, la celda 01h. Así pues, la máquina de estados programa que en el siguiente flanco ascendente de reloj se pase al estado S1, que a su vez programará que en el siguiente flanco ascendente de reloj se pase al estado S2. Todo esto mientras no se deja de aumentar el valor del bus auxiliar, el bus que indica a las memorias qué celda deben sacar hacia fuera.

Una vez se llega al estado S3, los bloques de memoria ROM ya tendrán preparado el coeficiente de la dirección introducida en el estado S1. Y así, el estado S3 seguirá sacando coeficientes ininterrumpidamente hasta llegar a los 200 ya que durante los estados S1 y S2 no ha dejado de aumentar el bus auxiliar. En total son necesarios 202 ciclos de reloj para completar el proceso de cargado de coeficientes, un ciclo de S1, un ciclo de S2 y 200 ciclos de S3.

Por último, cabe resaltar el funcionamiento del bus **Coef_Enable**. Se trata de un bus que agrupa 200 bits donde solo uno de ellos puede estar a nivel alto mientras que el resto deben permanecer a nivel bajo. Esto se debe a que cada uno de esos bits se conecta a uno de los 200 pares de registros que almacenan los coeficientes en el filtro convolucionador. Si el bus **Coef_Enable** tiene un valor decimal de '1', significa que el LSB (Bit de Menor Peso) se encuentra a nivel alto (…0001b). En cambio, si tuviera un valor decimal de '2', '4' u '8', significaría que se encuentran a nivel alto los bits LSB+1, LSB+2 y LSB+3:

$$CoefEnable = 1 \rightarrow \ \ldots 00001b$$

$$CoefEnable = 2 \rightarrow \ \ldots 00010b$$

$$CoefEnable = 4 \rightarrow \ \ldots 00100b$$

$$CoefEnable = 8 \rightarrow \ \ldots 01000b$$

$$CoefEnable = 16 \rightarrow \ \ldots 10000b$$

…

Se observa cómo el valor de **Coef_Enable** aumenta siempre en potencias de 2^n.

3.3 Diseño de la representación en coma fija

El primer paso en cualquier diseño que funcione en una FPGA es definir el tipo de precisión a emplear. Cuando se realiza un diseño en FPGA se ha de tener en cuenta el número de bits que van a constituir las señales internas del circuito. Dado que los recursos son limitados, es importante realizar un estudio teórico para encontrar una precisión que cumpla con los requisitos del sistema.

Para realizar el estudio teórico, este TFG emplea MATLAB® y hace uso de los objetos *fi*. Dichos objetos son numéricos y se organizan como una estructura donde se configura el número de bits que se designan para representar el signo, la parte entera y la parte fraccionaria. Para crear un objeto *fi* se utiliza la función *fi:*

$$\boldsymbol{fi}\ (\boldsymbol{Número},\ \boldsymbol{Signo},\ \boldsymbol{Tamaño\ Palabra},\ \boldsymbol{Parte\ Fraccionaria})$$

Número
→ *El número que se desea codificar en coma fija.*

Signo
→ *En caso de haberlo, el bit de signo es el bit de mayor peso.*

Tamaño Palabra
→ *Bits totales que ocupa el número codificado.*

Parte Fraccionaria

$\rightarrow$ *Bits dedicados a la parte decimal.*

Figura 29. Representación de un número en coma fija

$$\boldsymbol{fi}\,(\boldsymbol{N\acute{u}mero},\ \mathbf{1},\ \mathbf{8},\ \mathbf{4})$$

La Figura 29 muestra cómo sería la representación de un número codificado en coma fija con 1 bit de signo, 4 bits de parte fraccionaria y 8 bits de tamaño de palabra. El cálculo de los bits de la parte entera se obtiene substrayendo al tamaño de la palabra los bits dedicados al signo (máximo 1) y los bits dedicados a la parte fraccionaria, pudiéndose dar el caso de representar números que no tengan bits dedicados a la parte entera.

Cabe destacar que el bit de signo siempre será el MSB (bit de mayor peso) y podrá ser un bit de parte entera o de parte fraccionaria, dependiendo de la distribución que se haya indicado en la función *fj.* Por ejemplo, si se estipula que una señal va a tener un tamaño de palabra de 8 bits, con signo y 7 bits de parte fraccionaria, entonces el bit de signo pertenecerá a la parte entera. El bit de signo ocuparía ese bit de parte entera que surge de la resta del tamaño de palabra menos la parte fraccionaria.

Con estos objetos se pretende encontrar el número de bits que proporcionen unos resultados cuyo error sea suficientemente pequeño, es decir, menor que 1%. El error se obtendrá comparando los resultados del modelo en coma flotante frente al modelo en coma fija mediante el ECM.

Dado que la base de datos CIPIC se encuentra normalizada, no es necesario destinar bits para representar la parte entera. En principio, ninguna HRIR debería exceder la unidad como valor absoluto, lo que permite destinar un mayor número de bits a la parte fraccionaria.

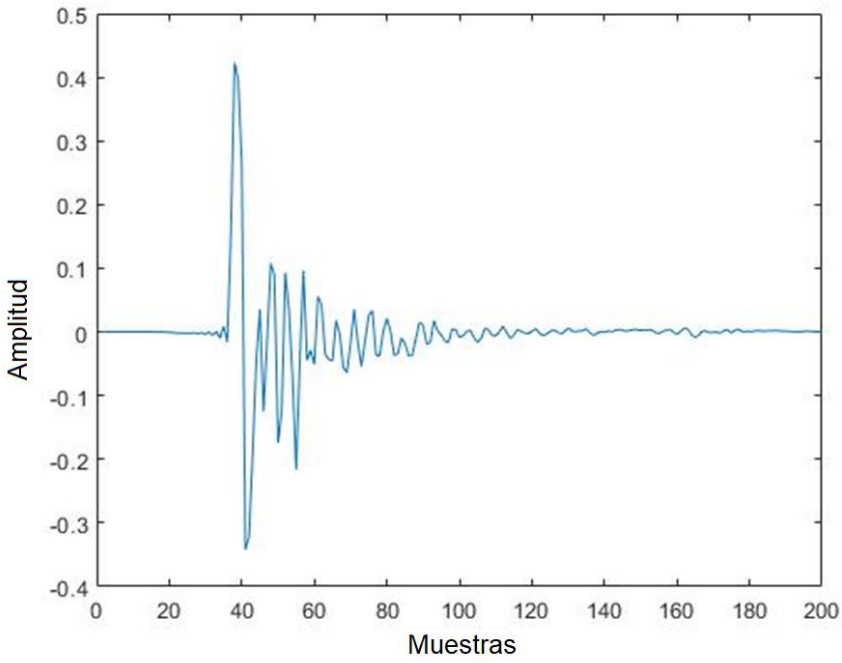


Figura 30. HRIR del canal izquierdo para un acimut de -5° y una elevación de 132°

El bit de signo si es necesario, ya que las respuestas al impulso alcanzan tanto valores positivos como valores negativos como se aprecia en la Figura 30. Por tanto, la codificación empleada tendrá la siguiente forma.

$$\boldsymbol{N\acute{u}mero} \rightarrow 1.0.\mathrm{X}$$

$$1\ bit\ de\ signo, 0\ bits\ de\ parte\ entera\ y\ X\ bits\ de\ parte\ fraccionaria$$

Para encontrar el número X de bits de parte fraccionaria se ha calculado el ECM de diez HRIRs codificadas en coma fija respecto a las mismas diez HRIRs codificadas en coma flotante. Los anchos de palabra que se han utilizado en la simulación van desde 16 bits a 6 bits. Las HRIRs que se han empleado se corresponden con los siguientes ángulos:

```
azimuth  = [-80, -65, -50, -35, -20, -5, 10, 25, 40, -10];
elevation = [ 45, 60, -45, 235, 78, 132, 84, 94, -14, 15];
```

Para todas las direcciones anteriores se ha simulado una precisión de coma fija con un tamaño de palabra variable, de 16 a 6 bits. Las direcciones escogidas son totalmente arbitrarias y podrían haberse escogido cualquier otras entre las 1250 direcciones que contiene la base de datos CIPIC.

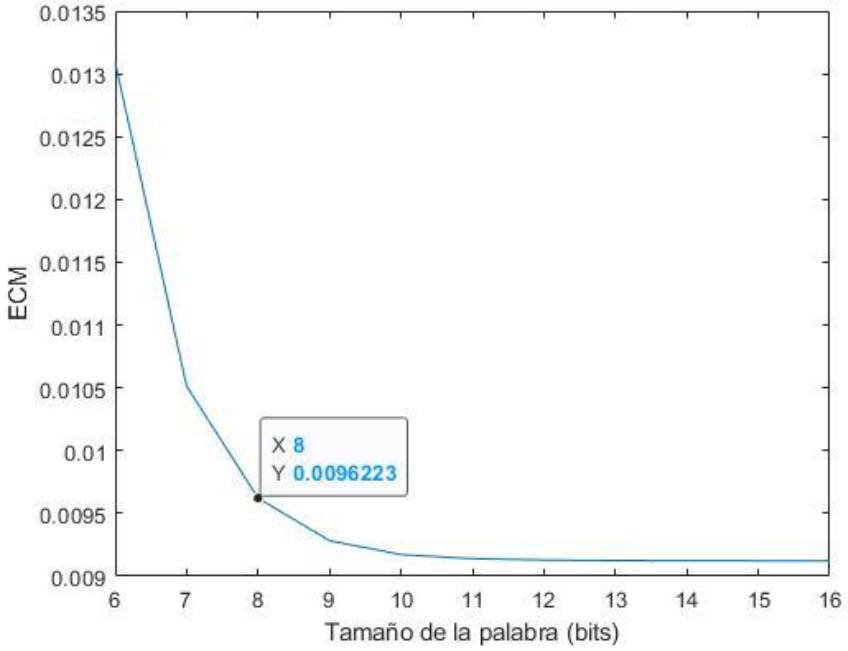


Figura 31. Simulación del ECM en función del tamaño de la palabra

En la Figura 31 se muestra el EMC entre los modelos de coma fija y coma flotante para diferentes tamaños de palabra. El error mostrado se ha obtenido como la media de los errores de las 10 direcciones arbitrarias para cada número de bits. Cuando el tamaño de la palabra, que incluye el bit de signo, es menor que 8 bits, el error crece de manera exponencial, superando el 1% que se ha impuesto como requisito para este TFG.

En consecuencia, se estima que el número mínimo de bits para codificar las HRIR ha de ser 8 bits, y tal y como se ha mencionado previamente, se destinarán 7 bits para la parte fraccionaria, 1 bit para el signo y 0 bits para la parte entera dadoo que la base de datos se encuentra normalizada.

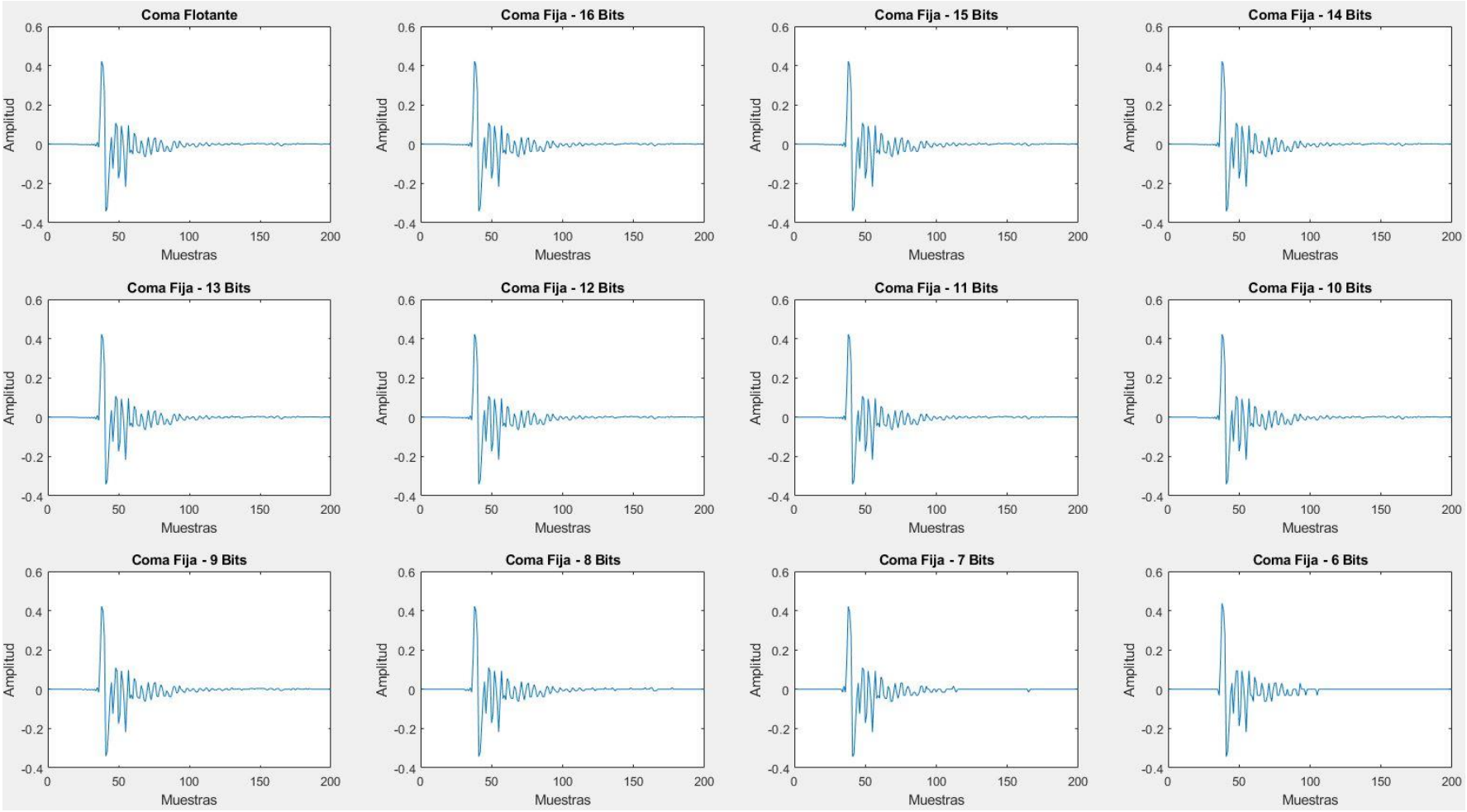


Figura 32. Comparación de una HRIR del canal izquierdo (acimut de -5° y elevación de 132°)

para diferentes anchos de palabra.

Para ejemplificar el caso de los 8 bits, la Figura 32 muestra la representación de la HRIR de la Figura 30 cuando se codifica para diferentes anchos de palabra. En los casos donde el ECM es mayor que el 1%, estos son para 6 y 7 bits, se observa cómo a partir de la muestra 120 las pequeñas oscilaciones desaparecen ante la falta de precisión.

Ya fijado el tamaño de la codificación en coma fija de las HRIR en 8 bits, el tamaño del resto de señales del filtro de convolución puede calcularse a partir de dicha señal.

3.3.1 Multiplicadores

Primero, se establece que el ancho de palabra de los datos de entrada **(*Data_in)*** será también de 8 bits. De esta forma se mantiene la misma precisión que tienen las HRIRs. Ambas señales son las entradas de los multiplicadores que hay en el filtro de convolución. **Data_in** es común a todos los multiplicadores mientras que **HRIR** es diferente dependiendo del canal y de la posición del multiplicador. Aun así, ambas señales tienen siempre el mismo tamaño: 8 bits.

Por regla general, el tamaño de la salida de un multiplicador debe ser la suma del tamaño de sus entradas. Cuando lo anterior se cumple, se asegura que no ocurre desbordamiento, y por tanto, que no se pierden bits. Siendo A y B las entradas y siendo C la salida:

$$A\ [N\ bits] * B\ [M\ bits] = C\ [N + M\ bits]$$

N y M son los tamaños en bits de las entradas. Se observa cómo el tamaño de la salida es M+N. Siempre y cuando se cumpla esta condición jamás habrá desbordamiento. Para el caso particular de este TFG donde A y B tienen el mismo tamaño la condición queda así:

$$A\ [N\ bits] * B\ [N\ bits] = C\ [2N\ bits]$$

Es decir, el producto del multiplicador debería tener un tamaño de 16 bits para asegurar que no ocurre desbordamiento.

Tabla 5. Estudio de los casos límites del multiplicador

Casos Límite	**Multiplicación**	**Producto**
Nº más Positivo $- * -$	$-1d * -1d$ $1000\ 0000b * 1000\ 0000b$	$= 1d$ $= 0100\ 0000\ 0000\ 0000b$

Nº más Negativo $- * +$	$-1d * 0.9922d$ $1000\ 0000b * 0111\ 1111b$	$= -0.9922d$ $= 1100\ 0000\ 1000\ 0000b$

En la Tabla 5 se estudian los del mayor número positivo y el menor número negativo que son posibles conseguir mediante una multiplicación. Con esto se aprecian dos cosas principalmente:

1) El mayor posible número (positivo) tiene un bit de parte entera.

2) El menor posible número (negativo) tiene sus dos MSBs siempre a 1.

En conclusión, queda demostrada la necesidad de que la salida del multiplicador tenga un tamaño de 16 bits. No tanto por el menor número posible, dado que el tener los dos MSB a uno es redundante y se podría quitar uno sin que el número cambiara, sino porque el mayor posible número necesita 1 bit de parte entera para mostrar todo el rango. Es decir, contando el bit de signo se necesitarían 16 bits.

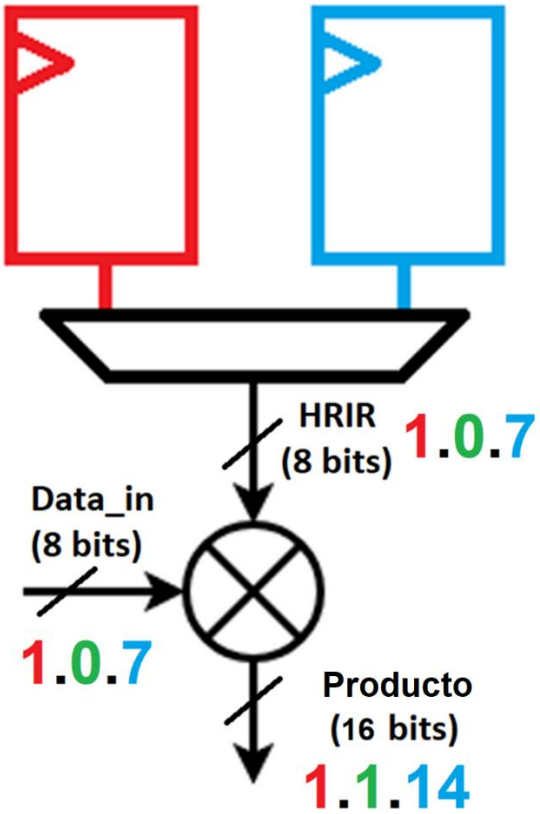


Figura 33. Tamaño de las señales de los multiplicadores

En la Figura 33 se representa gráficamente el ancho de las señales de los multiplicadores del filtro. También se distingue entre qué bits corresponden a según qué parte: signo, entera o fraccionaria.

3.3.2 Sumadores

Para calcular el tamaño de los buses de los sumadores se ha de partir del análisis realizado en apartado anterior. Los casos límites que se han estudiado van a servir para calcular el valor máximo que podría alcanzar la salida del filtro.

A continuación de los multiplicadores se encuentran 200 sumadores donde cada uno recibe el producto de la multiplicación y el resultado del sumador previo. El primer sumador no tiene ningún sumador previo, así que se limita a sumar cero al producto que recibe, por lo que, en realidad, es como si se estuviera sumando nada.

Si se supone que el resultado de las multiplicaciones es continuamente el mayor número positivo o el mayor número negativo, el máximo valor alcanzable en la salida del filtro será dicho número multiplicado por 200.

$$Caso\ mayor\ número \rightarrow 1d * 200 = 200$$

$$Caso\ mayor\ número \rightarrow -0.9922 * 200 = -198.44 \rightarrow$$

En el caso del mayor número, para calcular el número de bits de parte entera necesarios basta con realizar el logaritmo en base 2: $log_2(200) = 7.644\ bits \rightarrow 8\ bits$

En el caso del menor número, hay que coger la potencia de 2 inmediatamente mayor que el número que se quiere representar. Como el número se trata del -198.44, la siguiente potencia de 2 que permite representar este número es $2^8\ (256)$ porque si el bit signo fuera 2^7 solo se podría representar hasta el – 128. En conclusión, se necesitan 8 bits de parte entera más 1 bit de signo.

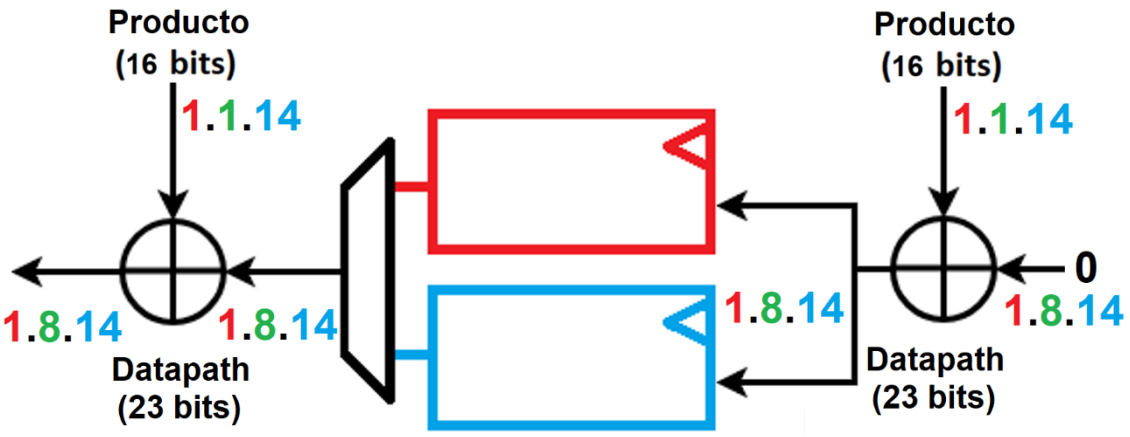


Figura 34. Tamaño de las señales de los sumadores

Como el caso del menor número es el más restrictivo ya que emplea un bit más, el bit de signo, el tamaño de las señales *datapath* quedaría 1.8.14; 1 de signo, ocho de parte entera y catorce de parte fraccionaria. Esto puede observarse gráficamente en la Figura 34.

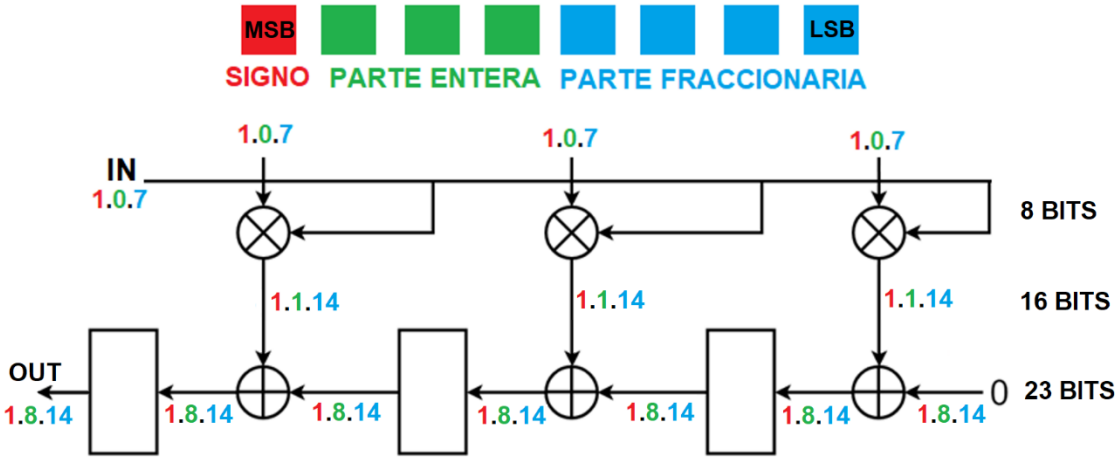


Figura 35. Tamaño de las señales internas del filtro de convolución

En la Figura 35 se observa el tamaño de todas las señales del filtro de convolución. La entrada comienza teniendo un tamaño de 8 bits y la salida después de la convolución tiene un tamaño de 23 bits. Estos tamaños son los que aseguran que no hay perdida de precisión al realizar la convolución.

4 Pruebas del modelo teórico

En este apartado se van a explicar las pruebas que se le han realizado al sistema para verificar su funcionamiento y comprobar sus resultados. Como se ha venido diciendo a lo largo de este TFG, el sistema debe arrojar unos resultados cuyo ECM suficientemente pequeño respecto a los resultados ideales. Los resultados ideales o modelos se han generado mediante simulaciones en MATLAB® con una precisión de coma flotante.

Además, también se van a comprobar que las señales y los registros internos del sistema estén tomando los valores correctos. Este es el caso de la gestión de la memoria, donde se va a probar que los coeficientes almacenados en flash se corresponden con las HRIRs de la base de datos CIPIC. Estás comprobaciones se realizarán mediante simulaciones en VIVADO™ con pruebas que realicen la carga de coeficientes correspondientes a distintas direcciones espaciales.

4.1 Carga de coeficientes el filtro de convolución

En esta prueba se comprueba el correcto funcionamiento de la carga de coeficientes en el filtro de convolución. Como se explicó en el 3.2.2.3, existe una máquina de estados que se encarga de cargar los coeficientes de la memoria ROM en los registros del filtro de convolución uno por uno. Dichos coeficientes se encuentran almacenados en la memoria ROM y han sido volcados en la FPGA mediante ficheros *.coe* generados en MATLAB.

Los ficheros *.coe* son ficheros de inicialización de coeficientes de los bloques *Xilinx BRAM*. Las IP de memoria de VIVADO™ permiten cargar estos ficheros como datos de inicialización. Como consecuencia, una vez se han volcado los coeficientes en la FPGA no es posible modificarlos a no ser que se vuelvan a volcar otros diferentes.

```
1  memory_initialization_radix=2;
2  memory_initialization_vector=
3  00000000
4  00000000
5  00000100
6  00111010
7  01000101
8  11100111
9  00011010
10 01000001
11 11001101
12 10010100
13 11010101
14 00100000
15 11011100
```

Figura 36. Estructura de un fichero .coe de coeficientes

En la Figura 36 se muestra un ejemplo de la estructura de un fichero *.coe*. Estos ficheros se conforman de una primera línea que indica la base en la que están representados los coeficientes. Los valores válidos para *memory_initialization_radix* son 2 para binario, 10

para decimal y 16 para hexadecimal. Como en este TFG los coeficientes están codificados en coma fija, la base que se va a emplear es binaria. A continuación, se encuentra la línea *memory_initialization_vector* que se utiliza para indicar el comienzo de los coeficientes. Los coeficientes que haya debajo de dicha línea serán cargados en los bloques de memoria del sistema.

El mapa de la memoria de coeficientes ya se ha detallado antes en la Figura 26. En la memoria solo se van a cargar doce HRIRs, seis para el canal derecho y seis para el canal izquierdo, éstas se corresponden con las direcciones espaciales más importantes que se han indicado en el apartado 2.1.

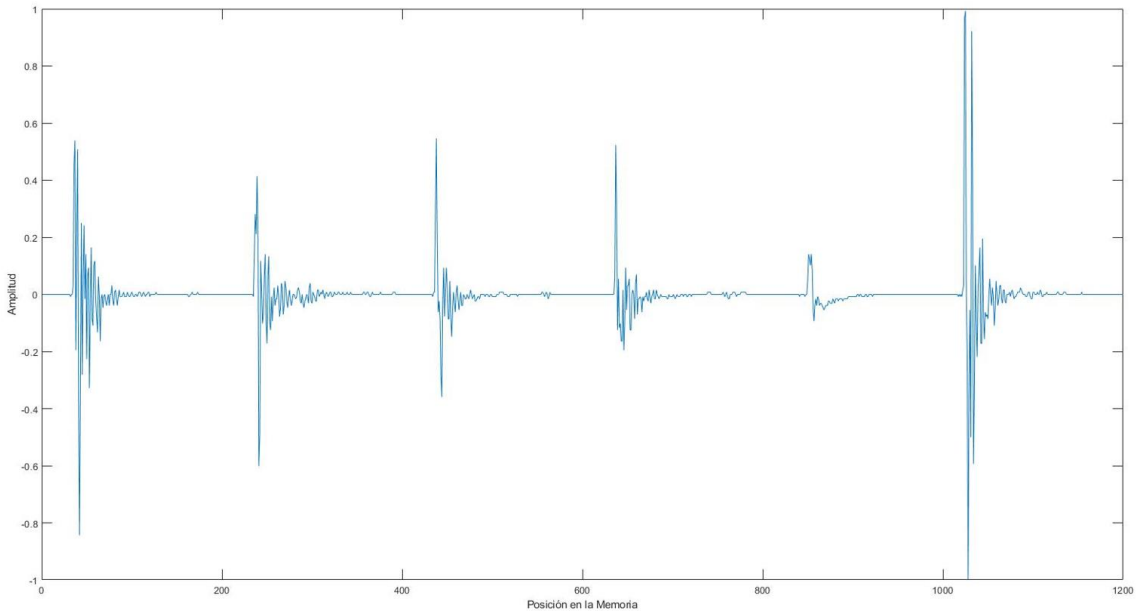


Figura 37. Representación de la concatenación de las HRIRs que se guardan en la memoria.

Dentro de la memoria todas las HRIRs se localizan de forma seguida de forma seguida. Una HRIR comienza en la posición inmediatamente siguiente a la de la HRIR anterior. Las primeras posiciones de cada HRIR están separadas por múltiplos de 200, ya que cada HRIR está conformada por 200 muestras. En caso de realizarse una representación gráfica de las muestras en la memoria, ésta tendría la forma de la Figura 37.

El *testbench* que lleva a cabo la prueba de la carga de coeficientes se ha preparado de la siguiente manera:

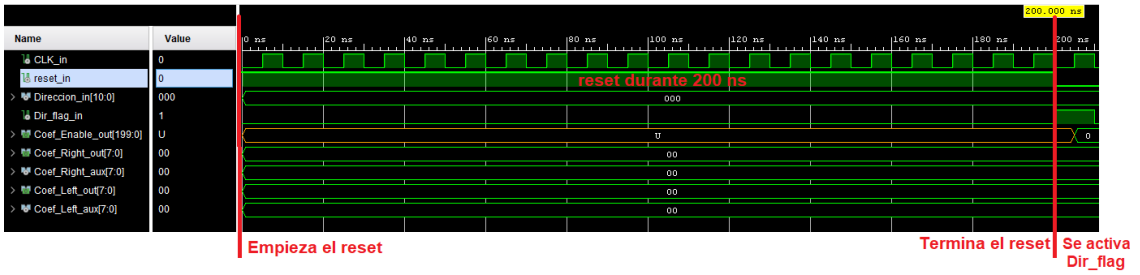


Figura 38. Simulación de la señal de reset durante la carga de coeficientes.

<u>Paso 1</u>: Se realiza un reset durante los 200 primeros ns (nanosegundos). Este reset es necesario porque la FPGA tiene un comportamiento impredecible hasta que no lleva algún tiempo funcionando. Cuando no se realiza este reset, las simulaciones no suelen funcionar y algunas señales no se comportan como deberían. Este paso se ejemplifica en la Figura 38.

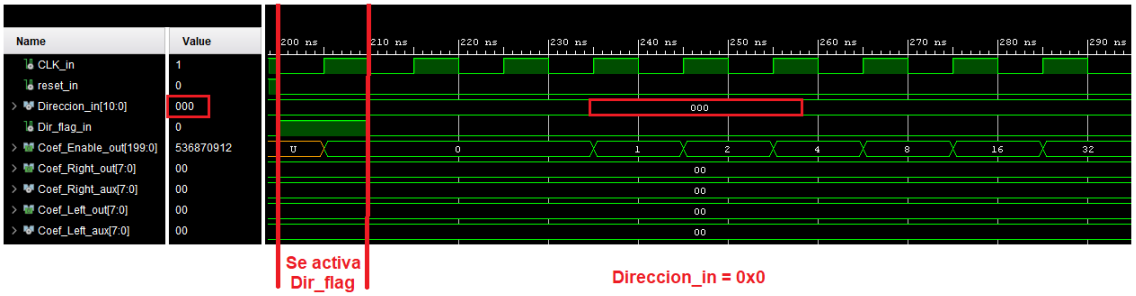


*Figura 39. Simulación de las señales **Direccion** y **Dir_flag** durante la carga de coeficientes*

1) <u>Paso 2</u>: Se le indica al sistema la dirección binaria dónde se encuentra almacenada la HRIR que se debe cargar en el sistema. Esta dirección habría sido extraída de la posición de los *JoySticks* en un módulo anterior. Además, para indicar que se trata de una nueva dirección, se activa a uno el *flag* de la dirección. El bloque de memoria comienza entonces a enviar uno a uno los coeficientes de los canales derecho e izquierdo

La dirección seleccionada se trata de la celda de memoria cuya posición es cero. En esta celda y en las 199 siguiente se encuentran alojadas las muestras de la HRIR correspondiente a los ángulos de acimut 0° y de elevación 0°. Esta HRIR se corresponde con la dirección espacial que tendríamos inmediatamente delante.

En la Figura 39 no se llega a reflejar cómo se están cargando los coeficientes en los registros porque las señales que traen los coeficientes desde la memoria (***Coef_Right_Out*** *y* ***Coef_Left_Out***) se encuentran constantemente a cero. Esto es debido a que como se pudo apreciar en la Figura 37 todas las HRIRs tienen sus primeras 20-30 muestras a cero.

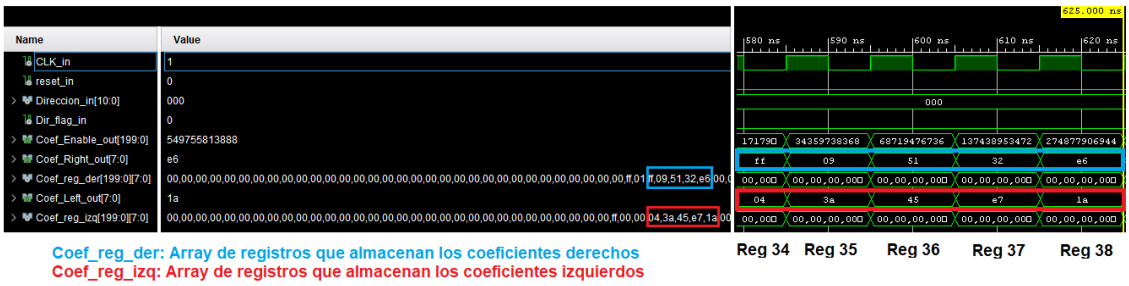


Figura 40. Simulación de los registros que almacenan las muestras de la HRIR de la dirección 0 (0x0) durante la carga de coeficientes.

Sin embargo, en la Figura 40 si se llega a apreciar que los coeficientes se están cargando. En los cuadros rojos y azules sí se observa cómo los coeficientes de los registros 34, 35, 36 y 37 están tomando las muestras que reciben de la memoria.

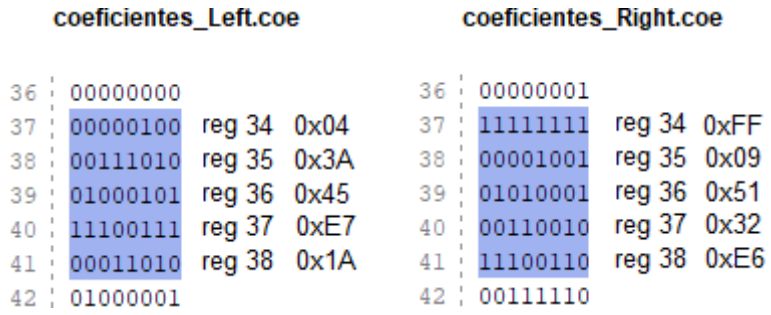


Figura 41. Muestras de la 34 a la 38 del mapa de memoria

En la Figura 41 se observan las muestras del mapa de los dos bloques de memoria, el izquierdo y el derecho. A la izquierda de ambos imágenes se encuentra el número de línea del código, por lo que ese número no es en realidad el número de registro. Los registros empiezan a contar desde cero mientras que las líneas empiezan a contar desde 1.

Además, hay que tener en cuenta que las dos primeras líneas que veíamos en la Figura 36 no eran muestras como tal, sino líneas de configuración del propio fichero. En conclusión, esas dos líneas sumado a que los índices no empiezan a contar por el mismo número explican la diferencia de tres unidades respecto al número de registro que hay a la derecha de las muestras. Aclarado esto, se puede observar cómo las muestras que se cargaban en la Figura 40 son exactamente iguales que los que hay en el mapa de los bloques de memoria.

2) **<u>Paso 3</u>**: Antes de modificar los coeficientes de nuevo, se espera un tiempo prudencial de 2500ns hasta que se han terminado de cargar todos los coeficientes. Hay que recordar que al tratarse de 200 coeficientes y al valer el periodo de la señal de reloj 10 ns, se tarda como mínimo 2000ns en terminar el proceso de cargado.

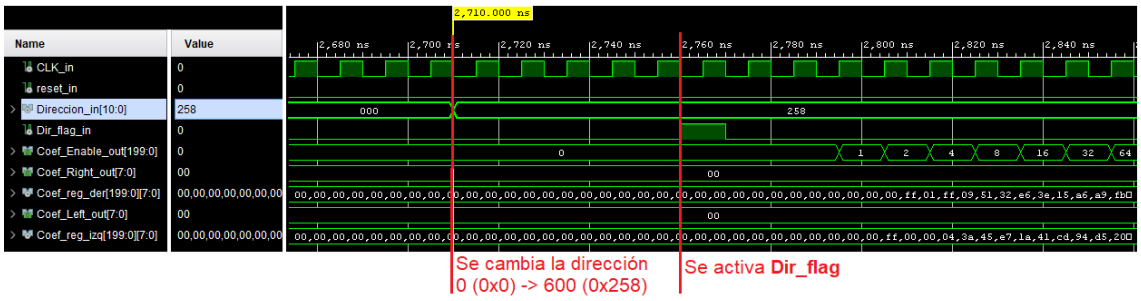


*Figura 42. Simulación de la señales **Direccion** cuando no se activa **Dir_flag** durante la carga de coeficientes*

3) **Paso 4:** Una vez ha pasado el tiempo prudencial, se cambia la dirección de los bloques de memoria, pero en este caso no se activa el *flag* de la dirección todavía. Con esto se pretende demostrar que la dirección puede cambiar las veces que se quiera que la carga de coeficientes se realizará única y exclusivamente cuando la señal del *flag* de la dirección lo indique, así como sucede en la Figura 42.

4) **Paso 5:** Después de 5 ciclos de reloj, añadidos de forma intencional, se activa finalmente el *flag* de la dirección. Esto provoca que comience un nuevo cargado de coeficientes en el filtro de convolución.

La nueva dirección es 600 (0x258), donde se encuentra alojada la HRIR que se corresponde con los ángulos (0°,90°) de acimut y elevación. Esta dirección se corresponde con el punto directamente detrás de la cabeza

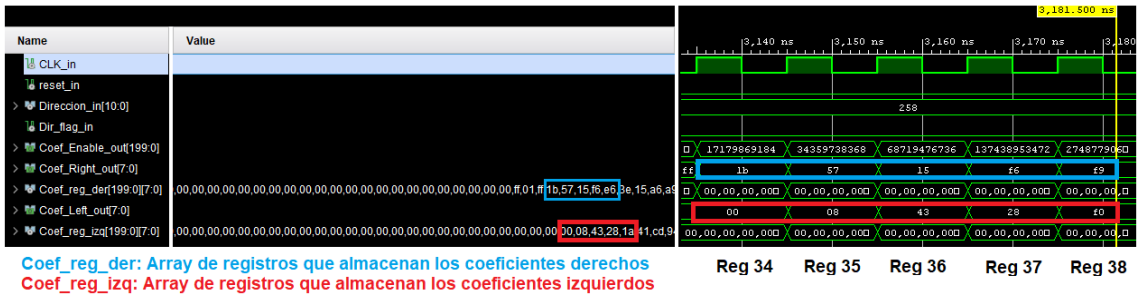


Figura 43 Simulación de los registros que almacenan las muestras de la HRIR de la dirección 600 (0x258) durante la carga de coeficientes.

La Figura 43 muestra en los cuadros rojos y azules cómo los registros están cargando las nuevas muestras que reciben de la memoria, y con ello, sustituyendo las antiguas muestras que tenían almacenadas. Para comprobar que estas muestras son las correctas hay que buscar las muestras de los ficheros *.coe* en las líneas de código que van desde la 637 a la 642. La lógica para calcular estas líneas de código es exactamente la misma que se aplicó en el cálculo de los índices de la dirección anterior.

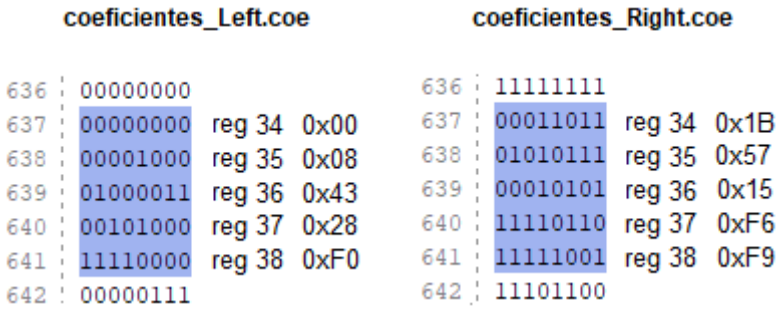


Figura 44. Muestras de la 634 a la 638 del mapa de memoria

En la Figura 44 se observan las muestras de las posiciones de memoria de la 634 a la 638 para comprobar que las muestras que están siendo cargadas en la Figura 43 son, en efecto, correctas. Sin embargo, en los registros 38 todavía no se han cargado los nuevos valores (0xF9 en el registro izquierdo y 0xF0 en el registro derecho) sino que, como la

línea de tiempo amarilla se encuentra en la mitad del ciclo de reloj, esos registros mantienen sus valores antiguos.

Dichos valores (0x1A en el registro izquierdo y 0xE6 en el registro derecho) se localizan en la posición 38 de los bloques de memoria como se puede observar en la Figura 41

En resumen, en esta prueba se ha podido comprobar cómo la gestión de la memoria funciona. La carga de los coeficientes almacenados en la memoria ROM dentro de los registros del filtro de convolución ha resultado exitosa para varias direcciones distintas. Esto prueba que el sistema es capaz de cargar una primera HRIR y cambiarla más adelante por otra que haya en la base de datos.

4.2 Convolución de una muestra de audio con diferentes HRIR

Como prueba final se va a comprobar la funcionalidad del sistema de audio 3D convirtiendo un fragmento de audio monoaural en audio 3D. Para reducir el número de simulaciones redundantes se van a realizar dos convoluciones empleando las HRIR que se usaron en el apartado 4.1. Estas HRIR se correspondían con las direcciones directamente delante (0°, 0°) y directamente detrás (0°, 180°).

El fragmento de audio que se va a utilizar dura aproximadamente un cuarto de segundo y forma parte una pequeña grabación de 1.5 segundos proporcionada por los laboratorios CIPIC como un ejemplo. Esta grabación se ha recortado a un fragmento más pequeño de 10000 muestras frente a las casi 65000 que tenía la grabación original. El motivo es reducir drásticamente el tiempo de simulación en VIVADO™.

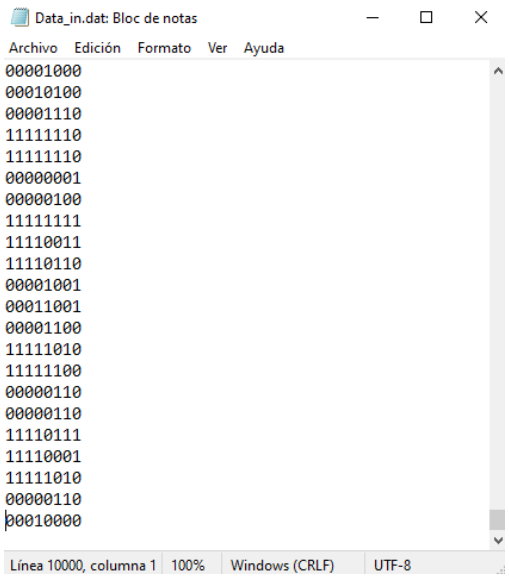


Figura 45. Fichero de datos .dat generado en MATLAB®

Para introducir el fragmento al sistema de audio 3D primero se ha de generar un fichero *.dat* con MATLAB® como el de la Figura 45. El fichero no tiene ninguna línea de configuración sobre dónde empiezan los datos o en qué base están codificadas las muestras, al contrario de lo que le sucedía a los ficheros *.coe*. Los datos se estructuran

en una columna donde cada muestra ocupa una fila y son extraídos hacia el sistema de audio en el *testbench*.

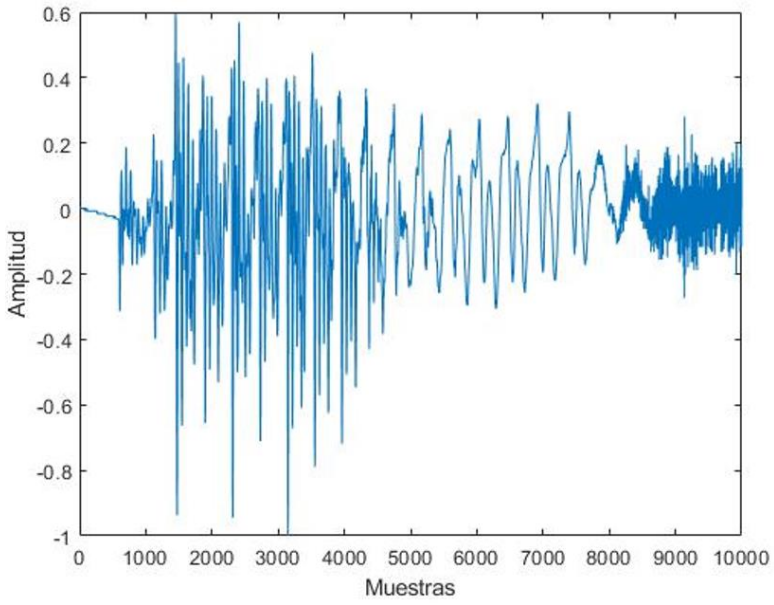


Figura 46. Representación gráfica del fragmento de audio codificado en coma fija.

En la Figura 46 se encuentra la representación gráfica de los datos de la Figura 45 antes de sufrir ninguna transformación dentro del filtro. Cuando estas muestras comiencen a ser convolucionadas dentro del filtro, se obtendrán dos representaciones gráficas que se corresponderán con los canales auditivos izquierdo y derecho.

Para poder representar gráficamente las muestras de audio 3D que arroja el sistema y también para poder calcular el EMC, el *testbench* escribirá en otros dos ficheros de datos *.dat* las muestras de ambos canales auditivos. La velocidad con la que se escribirá en estos ficheros será la misma con la que se leerá el fichero de datos de entrada. Es decir, cuando una muestra entre al sistema, otra saldrá, o dicho de otra forma, otras dos saldrán, ya que la salida del filtro arroja los canales derecho e izquierdo por separado.

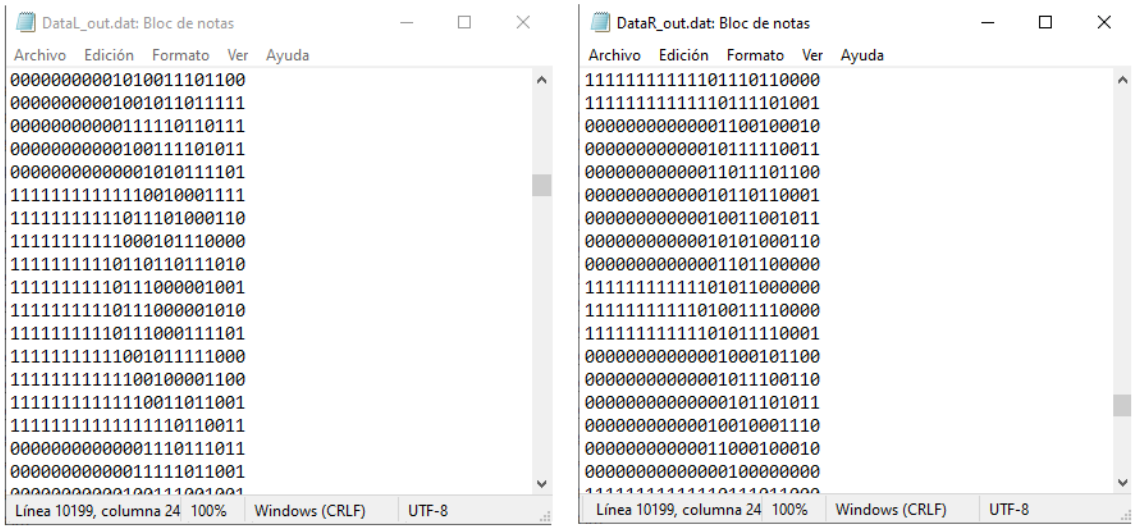


Figura 47. Muestras de audio 3D generadas por el filtro de convolución separadas en canal derecho y canal izquierdo.

Las señales resultantes de la convolución tendrán 10199 muestras de salida por el desarrollo que se hizo en el apartado 2.2 ya que la señal de entrada tiene 10000 muestras y la HRIR tiene 200. En la Figura 47 se ha puesto el cursor del ratón en la última línea, la número 10199, y en la última columna, la número 24, aunque no se llegue a mostrar en la imagen. En su lugar se están enseñando otras muestras que no son completamente cero como lo son las últimas muestras de la señal. Esto permite comprobar que el filtro ha realizado la convolución de forma correcta y que el tamaño de las muestras de salida es 23 bits tal y como se calculó en el apartado 3.3.2.

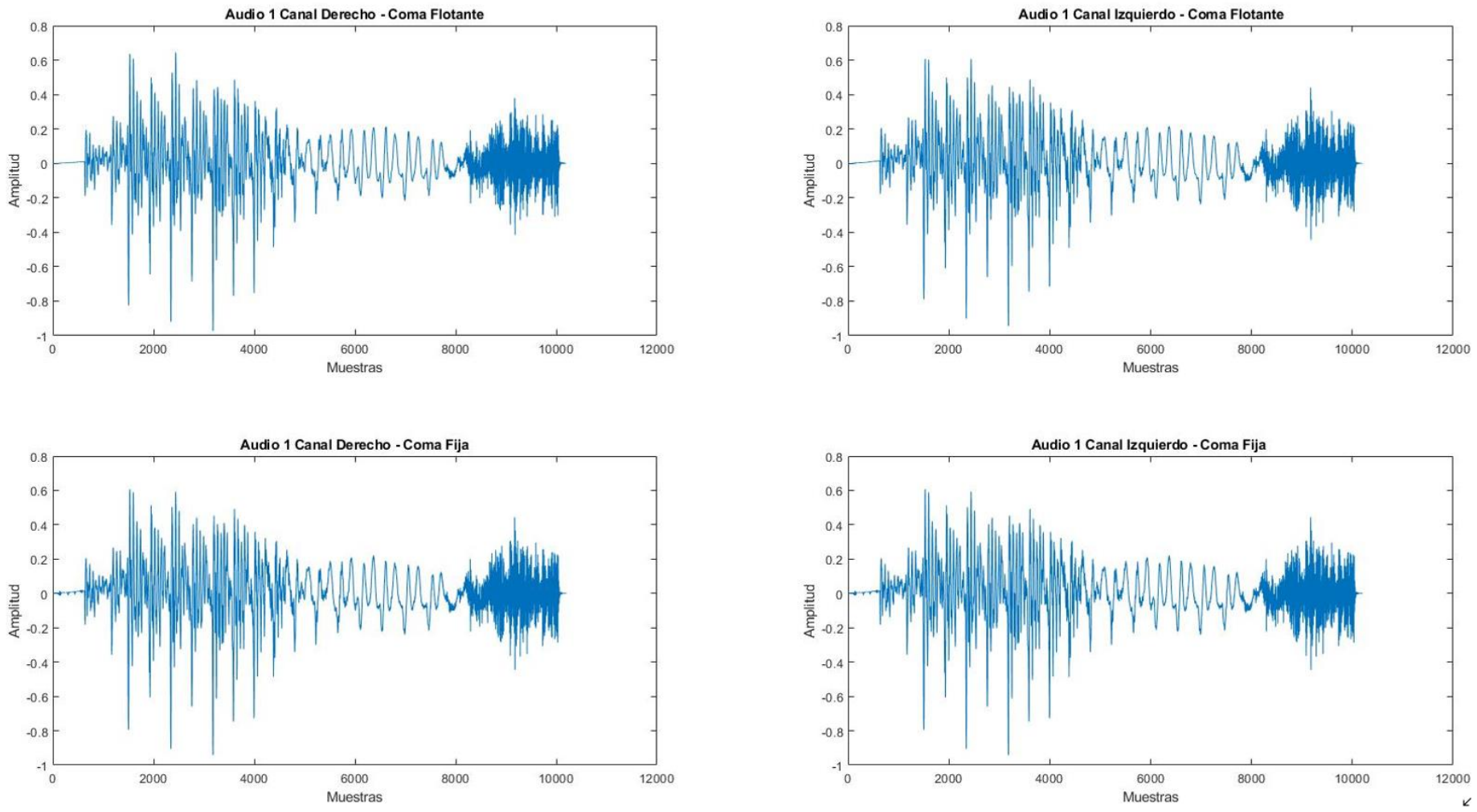


Figura 48. Comparación de los resultados del sistema de Audio 3D frente a los resultados ideales para una HRIR de ángulos (0°, 0°)

En la Figura 48 se comparan los resultados reales frente a los resultados ideales cuando se ha realizado la convolución del fragmento de audio con la HRIR (0°, 0°) cuya dirección es la inmediatamente delante. Por un lado, las dos gráficas de la parte superior muestran los resultados obtenidos de la simulación en coma flotante del sistema de Audio 3D. Por otro lado, las gráficas de la parte inferior muestran los resultados reales. A simple vista, si se comparan las imágenes de arriba con las de abajo, no se aprecian diferencias significativas.

Ahora bien, cuando se realiza el ECM:

```
ECM_1 = immse(Audio1_cf,double(Audio1_fi));
X = sprintf('El ECM es %f%%',ECM_1*100);
disp(X);

El ECM es 0.063383%
```

Se concluye que el error es suficientemente pequeño porque es menor que el 1%.

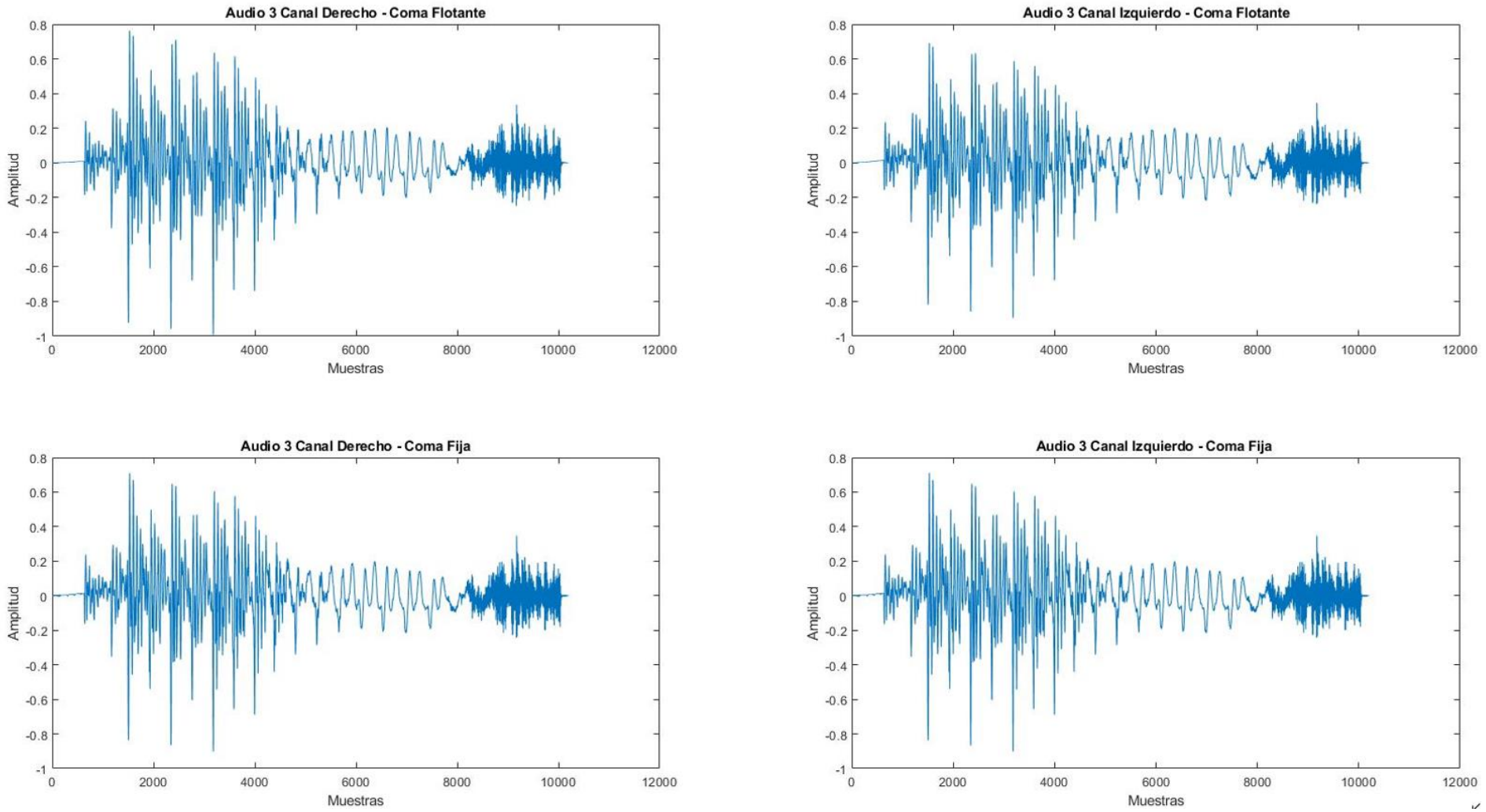


Figura 49 Comparación de los resultados del sistema de Audio 3D frente a los resultados ideales para una HRIR de ángulos (0°, 180°)

En la Figura 49 se han vuelto a comparar los resultados reales frente a los resultados ideales, solo que esta vez la convolución del fragmento de audio se ha realizado con la HRIR (0°, 180°) cuya dirección es la inmediatamente detrás. Las dos gráficas de la parte superior continúan mostrando los resultados obtenidos de la simulación en coma flotante del sistema de Audio 3D. Por otro lado, las gráficas de la parte inferior muestran los resultados reales. A simple vista, si se comparan las imágenes de arriba con las de abajo, siguen sin apreciarse cambios significativos.

En cuanto al ECM:

```
ECM_3 = immse(Audio3_cf,double(Audio3_fi));
X = sprintf('El ECM es %f%%',ECM_3*100);
disp(X);

El ECM es 0.079693%
```

Se concluye que el error es suficientemente pequeño porque es menor que el 1%.

En resumen, en esta prueba se ha podido comprobar cómo los resultados del sistema de audio 3D ofrecen unos resultados con un ECM inferior al 1%. Tanto para la HRIR (0°, 0°) como para la HRIR (0°, 180°) los resultados han cumplido las expectativas. Se puede destacar que al tratarse de direcciones espaciales tan importantes, sobre todo la dirección inmediatamente delante, la HRTF no atenúa o acentúa tantas frecuencias como lo haría en otras direcciones más complejas que vinieran desde arriba o desde abajo.

5 Conclusiones

El objetivo de este TFG ha sido la creación de un sistema que fuera capaz de generar audio 3D en tiempo real y que diera la oportunidad al usuario de elegir la dirección de la procedencia del sonido. Durante todo el documento se han explicado los conceptos teórico de cómo generar audio espacial, así como también se han proporcionado las herramientas necesarias para crearlo.

Uno de los motivos por los que ha sido posible realizar este TFG ha sido gracias a una investigación en profundidad de publicaciones del IEEE sobre la HRTF y sobre el sonido espacial. Aunque muchos de esas publicaciones tienen más de dos décadas, sus conclusiones siguen pesando tanto hasta tal punto de que hoy en día suponen la base teórica de cualquier aplicación que use HRTF para crear audio 3D, como es el caso de este proyecto.

El sistema que se ha propuesto como solución al problema de crear audio 3D en tiempo real viene motivado por la operación de la convolución. Dicha operación matemática es la artífice que sustenta todos los cálculos que se realizan para el empleo de la HRTF. Sin la operación de la convolución para generar una señal a partir de una muestra de audio y de una HRIR no se podría concebir la idea de la elaboración de sonido espacial.

Por ello el filtro de convolución que se ha diseñado tiene como requisito principal arrojar una salida idéntica a la de la operación de la convolución. Para guiar la elaboración de la lógica interna del filtro se ha empleado la documentación oficial de MATLAB® respecto a la convolución. [12] La estructura final del filtro se asemeja a la estructura típica de un filtro FIR salvo por el detalle de que es necesario el empleo de 200 coeficientes para lograr la convolución de la señal de entrada con la HRIR que se haya seleccionado. Esta limitación ha impedido el empleo de IP de Xilinx® que implementaban el filtro FIR directamente que hubiera permitido realizar la convolución.

Por ello se ha tenido que diseñar todo el filtro hasta el detalle del ancho de sus señales internas. Toda aplicación que corre en una FPGA debe realizar un estudio de la codificación y la precisión que va a emplear en sus buses de datos. Para este TFG se han llevado a cabo distintas simulaciones para encontrar la distribución de bits que ofreciera unos resultados cuyo ECM entre coma fija y coma flotante fuera menor que el 1%. Este estudio se realizó recreando la solución del sistema adoptado en MATLAB®.

Una vez fijado el tamaño de todas las señales se procedió a implementar la solución adoptada en VIVADO™ para simular en el entorno más cercano a una FPGA que hay. Las pruebas de funcionamiento han resultado exitosas puesto que se ha podido comprobar que el sistema diseñado cumple con los requisitos que se le habían impuesto. Tanto, que a la vista de los resultados todo parece indicar se iba por muy buen camino para terminar volcando el código en una FPGA.

Pese a no haberse podido probar el sistema en placa, los resultados son lo bastante buenos como para pensar que si se diseñan y se implementan los módulos PMOD según el funcionamiento que se les ha supuesto que realizaban, el sistema llegaría a ser una

aplicación en tiempo real donde un usuario generaría audio 3D con un dispositivo móvil y unos auriculares conectados a la FPGA para escuchar las direcciones espaciales que el mismo usuario seleccionaría desde los *JoySticks.*

6 Trabajos futuros

En este apartado se profundiza en los dos módulos que serían necesarios para que el sistema de sonido espacial funcionara físicamente en una FPGA. Este TFG ha simulado el comportamiento del filtro en conjunto con la memoria. Sin embargo, las entradas y salidas del filtro, así como el cambio de dirección espacial, se realizaban mediante *testbenches*. Los módulos que se van a explicar a continuación sustituyen las instrucciones de los *testbenches* que simulaban la E/S de datos en el filtro y de cambio en la dirección de la memoria.

El PMOD de audio se encarga de muestrear el sonido en formato binaural, para después convertir a monoaural, y de reproducir el audio en formato binaural una vez ha pasado a través del filtro de convolución. El PMOD de los *joystick* obtiene los ángulos de azimut y de elevación que seleccionaría el usuario del sistema. Ambos módulos han sido fabricados por Digilent® y tienen conectores preparados para conectarse directamente a la **FPGA Nexys 4 DDR** siguiendo la especificación de la interfaz de los PMOD de Digilent®. [17]

6.1 PMOD de audio

El PMOD de audio para el que se ha diseñado el sistema es el PMOD I2S2 de la empresa Digilent®.

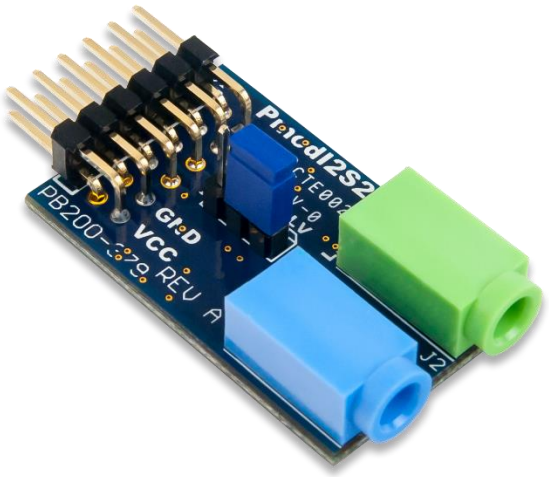


Figura 50. PMOD I2S2 de Digilent®.

En la Figura 50 se aprecian los dos conectores tipo Jack de 3.5mm para conectar una entrada como puede ser la salida de un dispositivo móvil y una salida de audio como unos auriculares. Entre sus características principales se hayan dos conversores, uno A/D y otro D/A de 24 bits cada uno, con capacidad para muestrear y reproducir audio estéreo.

Este PMOD puede funcionar con dos modos de velocidad:

- **Modo de velocidad único:** Muestrea los dos canales del audio estéreo. Debido a ello, la frecuencia de muestreo máxima es la mitad que en el modo de velocidad único.

- **Modo de velocidad doble:** Muestrea uno solo de los canales del audio estéreo. A cambio de perder la información del canal que no se ha muestreado, permite alcanzar el doble de frecuencia de muestreo.

Tabla 6. Especificaciones del PMOD I2S2 [10]

Parámetro	Mínimo	Típico	Máximo	Unidad
Voltaje de la fuente de alimentación	3.0	3.3	5.25	V
Frecuencia de muestre de entrada de audio (Modo de velocidad única)	4	-	54	kHz
Frecuencia de muestre de entrada de audio (Modo de velocidad doble)	86	-	108	kHz
Frecuencia de muestreo de salida de audio	2	-	200	kHz

En la Tabla 6 se disponen las especificaciones del *datasheet* del fabricante para el componente PMOD I2S2. [18] Se puede apreciar cómo la máxima frecuencia de muestreo en el modo de velocidad doble es 108 kHz frente a los 54 kHz del como de velocidad único. Asimismo, se observa que la frecuencia de muestreo de salida alcanza los 200 kHz.

El diseño del sistema de este TFG emplea una frecuencia de muestreo de 44.1 kHz por lo que, en principio, este PMOD cumpliría la limitación de la frecuencia de muestreo. Ahora bien, para cumplir dicha limitación el funcionamiento del PMOD debe ser en modo de velocidad única. El motivo es que la frecuencia mínima del modo de velocidad doble es 86 KHz, lo que supera los 44.1 kHz de frecuencia de muestreo que emplea este TFG.

En relación con lo anterior, el modo de funcionamiento que habría de usarse sería el de velocidad única. Éste obtendría muestras de dos canales de audio. Sin embargo, el sistema que se ha diseñado solo está preparado para tratar audio en formato monoaural, no binaural. Por tanto, habría que desprenderse de las muestras de uno de los dos canales, perdiendo la información que lleven, y quedarse con las muestras del otro canal. Dichas muestras serían transmitidas al filtro para que se realizara la convolución con ellas.

Finalmente, esa muestra que habría entrado en formato monoaural al filtro saldría convertida en formato binaural. Lo único que restaría sería normalizar los bits de la salida del filtro para puedan ser reproducidos a través de los auriculares conectados al PMOD I2S2.

6.2 PMOD de los JoySticks

El PMOD de los *JoySticks* para el que se ha diseñado el sistema es el PMOD JSTK2 de la empresa Digilent®. Se trata de un dispositivo que puede ser fácilmente incorporado a

[10] https://digilent.com/reference/pmod/pmodi2s2/reference-manual

una gran variedad de variedad de proyectos, entre ellos, este TFG. Dispone de un joystick de dos ejes, un botón de disparo y un led RGB (Reed-Green-Blue) programable capaz de mostrar 24 bits de colores. Utiliza el protocolo de comunicaciones SPI para mandar las coordenadas a la FPGA.

Figura 51. PMOD JSTK2: Two-axis Joystick de Digilent®

En la Figura 51 se observa cómo el PMOD JSTK2 solo trae incorporado un solo *joystick*. Como ya se ha explicado, este TFG utiliza dos ángulos, acimut y elevación, para elegir la dirección espacial del sonido 3D. Por este motivo hay que emplear dos PMOD JSTK2 y asignar a cada uno de ellos uno de los ángulos.

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
MOSI	COMMAND / 0	PARAM1 / DUMMY	PARAM2 / DUMMY	PARAM3 / DUMMY	PARAM4 / DUMMY
MISO	smpX (Low Byte)	smpX (High Byte)	smpY (Low Byte)	smpY (High Byte)	fsButtons

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
fsButtons	EXTPKT	0	0	0	0	0	TRIGGER	JOYSTICK

EXTPKT: Extended Packet Status Bit
1 = additional data corresponding to the command byte is available and may be retrieved after this byte
0 = standard response packet, no additional data follows this byte

TRIGGER: Trigger Button Status Bit
1 = trigger button is currently pressed
0 = trigger button is not being pressed

JOYSTICK: Joystick Center Button Status Bit
1 = joystick center button is currently pressed
0 = joystick center button is not being pressed

Figura 52. Interfaz entre el PMOD JSTK2 y la FPGA

Para mandar las coordenadas se emplea un protocolo *standard* de 5 bytes, lo que quiere decir que toda la información se manda en un solo paquete cuya estructura se encuentra representada en la Figura 52. Los byte 1 y 2 se utilizan para mandar la coordenada X. De la misma manera, los bytes 3 y 4 mandan la coordenada Y. Por último, el byte 5 manda información de si el botón de disparo o el *joystick* se encuentran presionados.

La FPGA recibiría las coordenadas cartesianas de ambos *JoySticks*. Después, de cada uno de los *JoySticks* se extraería uno de los ángulos del sistema de coordenadas interaural-polar, que son los ángulos en los que se basa la base de datos CIPIC. Una vez se tengan los ángulos de acimut y de elevación, la FPGA deberá convertir dichos ángulos en una dirección binaria para mandársela a la memoria.

En el apartado 3.2.2.3 se explicaba cómo el bloque de la memoria recibe directamente la dirección binaria de donde se encuentra almacenada la HRIR de la dirección seleccionada por *JoySticksk*. Por este motivo, la conversión de las coordenadas cartesianas a dirección binaria de la HRIR almacenada debe hacerse antes de ser enviada al bloque de memoria.

7 Resultados de la implementación del sistema en FPGA

Dentro de la placa ocurren otros datos de interés comunes a cualquier proyecto que haga uso de una FPGA. Gracias a VIVADO™ es posible realizar una estimación de estos aspectos.

7.1 Utilización de los recursos

La aplicación que se ha desarrollado en este TFG no es excesivamente compleja en cuanto a la utilización de recursos. Además de los dos bloques BRAM utilizados para generar la memoria ROM, también se usan 200 DSP Slices de un total de 240.

Resource	Utilization	Available	Utilization %
LUT	8143	63400	12.84
FF	17259	126800	13.61
BRAM	2	135	1.48
DSP	200	240	83.33
IO	45	210	21.43
BUFG	1	32	3.13

Figura 53. Utilización de los recursos totales de la FPGA Nexys 4 DDR de la familia Artix-7™

En la Utilización de los recursosFigura 53 se lista la utilización de los principales recursos que tiene la **FPGA Nexys 4 DDR de la familia Artix-7™** y la cantidad de ellos que queda disponible. El dato más preocupante es el de los DSP *Slices.* Su utilización está por encima del 80% lo que supone que no se pueda correr ninguna aplicación más que la que se ha explicado en este TFG ya que utiliza casi todos los multiplicadores de la placa. No obstante, la solución que se ha adoptado no contempla que más proyectos utilicen la FGPA al mismo tiempo por lo que en realidad todos los recursos estarían disponibles para el sistema de audio 3D.

7.2 Consumo de potencia

Las FPGA son placas muy potentes debido a principalmente a que muchos de sus recursos pueden estar funcionando de forma paralela. Esto puede acarrear altos consumos en la potencia, por lo que es preciso emplear la herramienta que proporciona Xilinx® para comprobar que el consumo de potencia es soportado por la FPGA.

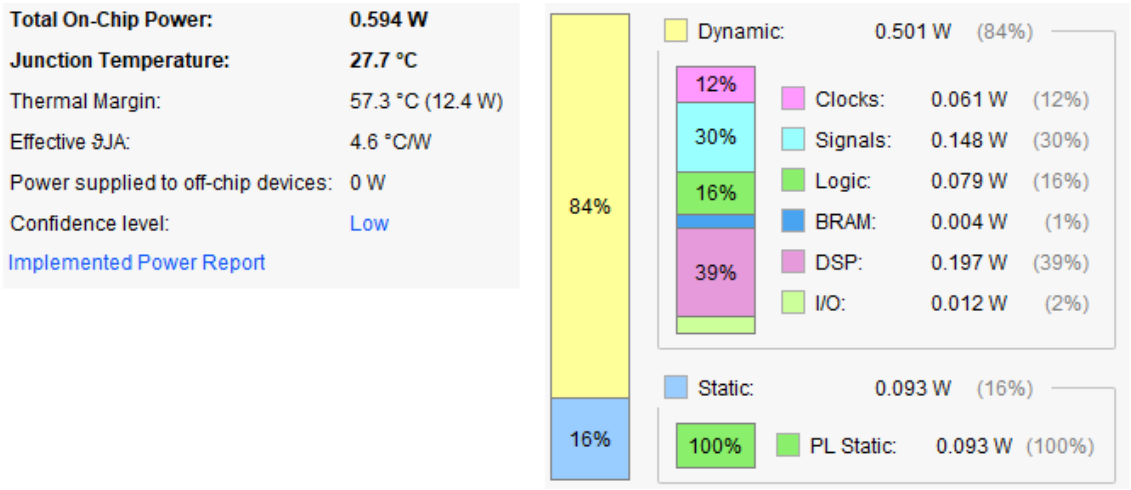


Figura 54. Consumo de potencia de la FPGA Nexys 4 DDR de la familia Artix-7™

En la Figura 54 se observa, como es lógico, que la mayor parte del cosumo de potencia es debido a los *DSP Slices*. Aún así, los consumos de potencia no son en absoluto elevados por lo que es de esperar que la placa pudiera soportar la aplicación sin ningún problema

8 Anexo I: Bibliografía

[1] M. Manrique Rodriguez y J. Marco Algarra, "*Audiología*", Ponencia Oficial de la Sociedad Española de Otorrinolaringología y Patología Cérvico-Facial, 2014.

[2] V. Parra Guisado "*El sonido biaural, una nueva experiencia de publicidad sonora*", TFG, Facultad de Comunicación, Publicidad y Relaciones Públicas. Universidad Pompeu Fabra, Barcelona, 2020.

[3] J. Blauert, "*Spatial Hearing - The Psychophysics of Human Sound Localization*" MA: MIT press, Cambridge, 1997.

[4] V. R. Algazi, R. O. Duda, D. M. Thompson and C. Avendano, "The CIPIC HRTF database," *Proceedings of the 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics (Cat. No.01TH8575)*, 2001, pp. 99-102, doi: 10.1109/ASPAA.2001.969552.

[5] A. Londoño Borja y T. Viana Alzate, "*Desarrollo de un plugin para la implementación de audio espacial en proyectos musicales y audiovisuales"*, Trabajo de grado Ingeniería de Sonido, Universidad de San Buenaventura Medellín, Facultad de Ingenierías, 2018.

[6] M. Zhu, M. Shahnawaz, S. Tubaro and A. Sarti, "HRTF personalization based on weighted sparse representation of anthropometric features," *2017 International Conference on 3D Immersion (IC3D)*, 2017, pp. 1-7, doi: 10.1109/IC3D.2017.8251901.

[7] L. Li and Q. Huang, "HRTF personalization modeling based on RBF neural network," *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 3707-3710, doi: 10.1109/ICASSP.2013.6638350.

[8] M. D. Burkhard, "Manikin Measurements", Conference Proceedings, 1976.

[9] CIPIC Interface Laboraty, "*Documentation for the UCD HRIR Files"* University of California at Davis, October 1998.

[10] https://en.wikipedia.org/wiki/44,100_Hz

[11] A. Oppenheim and R. Schafer, "*Discrete-Time Signal Processing"*. Pearson, 1989.

[12] https://es.mathworks.com/help/MATLAB®/ref/conv.html

[13] https://reference.digilentinc.com/programmable-logic/nexys-a7/reference-manual

[14] https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[15] https://docs.rs-online.com/2bb6/0900766b81592f18.pdf

[16] https://www.bocm.es/boletin/CM_Orden_BOCM/2020/07/25/BOCM-20200725- 2.PDF fff

[17] https://digilent.com/reference/_media/reference/pmod/digilent-pmod-interface-specification.pdf

[18] https://digilent.com/reference/pmod/pmodi2s2/reference-manual?redirect=1

9 Anexo II: Pliego de condiciones

A continuación, se detallan los recursos hardware y software que han hecho posible la elaboración de este TFG.

Hardware

- Procesador: AMD Ryzen 5 2600X 3.60 GHz
- Memoria RAM: 16.0 GB
- Disco duro: 512 GB SSD
- Gráfica: GeForce 1050 TI

Software

- Sistema Operativo: Windows 10 Pro N
- Vivado 2018.3
- Matlab R2019a
- Office 365:

10 Anexo III: Presupuesto

En la realización de este TFG se han utilizado recursos software y hardware con un coste asociado.

Tabla 7. Costes Hardware

HARDWARE			
Elemento	**Modelo**	**Vendedor**	**Coste**
Tarjeta gráfica	ASUS Cerberus GeForce GTX 1050 Ti	PCCOMPONENTES	194,90 €
Fuente de alimentación	Corsair RM750X V2 750W	PCCOMPONENTES	119,99 €
Caja	Nox Pax Blue Edition USB 3.0	PCCOMPONENTES	24,99 €
Disco Duro	Samsung 860 EVO Basic SSD 500GB SATA3	PCCOMPONENTES	84,99 €
Disipador	Noctua NH-D9L Perfil Bajo	PCCOMPONENTES	51,00 €
Memoria RAM	G.Skill Ripjaws V DDR4 3000	PCCOMPONENTES	139,00 €
Placa Base	Placa Base MSI X470	PCCOMPONENTES	139,90 €
Procesador	AMD Ryzen 5 2600X 4.2 Ghz	PCCOMPONENTES	213,99 €
		TOTAL	968,76 €

En la Tabla 7 se ha realizado un desglose de los componentes del ordenador que se ha utilizado en este TFG.

Tabla 8. Costes Software

SOFTWARE				
Herramienta	**Version**	**Vendedor**	**Duración**	**Coste**
Vivado Webpack	2018.3	Xilinks	-	0 €
Matlab	R2019a	MathWorks	1 Año	800,00 €
Sistema Operativo	Windows 10 Pro	Microsoft	-	145,00 €
Microsoft Office 365	2019	Microsoft	1 Año€	69,00 €
			TOTAL	1014,00 €

En la Tabla 8 se ha realizado un desglose de los costes y la duración de los programas software que se han empleado en este TFG.

Además de los gastos anteriores, también hay que añadir el gasto de la mano de obra calculado a partir de las horas trabajadas. Este TFG tiene una equivalencia de 12 créditos ECTS donde cada crédito equivale a 25 horas de trabajo. En total, 300 horas.

Según el convenio del metal [16] la retribución anual para un licenciado/grado es de 28.122,41 €. La jornada se fija en 1.764 horas anuales. Es decir, la mano de obra tendría un coste de 15.94€/hora. Como el TFG tiene una duración estimada de 300 horas, el coste por mano de obra total sería de 4782.72€.

Tabla 9. Costes totales del TFG

Concepto	Coste
Hardware	968,76 €
Software	1014,00 €
Mano de obra	4782,72 €.
Total	6765,48 €

En consecuencia, el coste final de este TFG equivaldría a **seis mil setecientos sesenta y cinco euros con cuarenta y ocho céntimos (6765,48 €)**, tal y como se dispone en la Tabla 9.

11 Anexo IV: Código

En este anexo se expone el código que ha permitido la realización de este TFG. Se pueden encontrar dos tipos de código dependiendo del lenguajes empleado. Los *script* de MATLAB® se encuentran programados en un *pseudo*-lenguaje C mientras que los ficheros de VIVADO™ se encuentran modulados en VHDL.

```
%CPPYRIGHT NOTICE: the hrir_final.mat is provided by CIPIC HRTF Database.
%All rights reserved.
%https://ses.library.usyd.edu.au/handle/2123/15038

%
% Este script pretende ser una simulación usando codificación en coma
% fija en lugar de codificación en coma flotante. Con esta simulación
% se puede observar cuan diferente es el resultado respecto al filtro
% original (el que usa coma flotante) para decidir el número de bits
% dentro de la FPGA.
%
% Este script no compara directamente. Tan solo te da el resultado final.
% Tienes que ser tu desde la consola quien ejecuta la función del error
% cuadrático medio. El valor obtenido se multiplica por 100 para obtener
% el error en porcentaje. Nosotros vamos a estimar que un error inferior al
% 1% es un error aceptable.

clear all;
%load nofi.mat;
%load nofi2.mat;
load nofi3.mat
load hrir_final;
%azimuth = input('\nWHAT ANGLE IN AZIMUTH DO YOU WANT TO ACHIVE(-90~90)?\n\n');
%elevation = input('\nWHAT ANGLE IN ELEVATION DO YOU WANT TO ACHIEVE(-45~231)?\n\n');

SAMPLES = 5000;
cambios = 10;

% Diferentes ángulos para obtener distintas muestras al impulso
azimuth  = [-80 -65 -50 -35 -20 -5 10 25 40 65];
elevation = [45 60 -45 235 78 132 84 94 -14 0];

%Normalizamos las respuestas al impulso
hrir_r_norm = hrir_r./max(max(max(hrir_r)));
hrir_l_norm = hrir_l./max(max(max(hrir_l)));

% Se digitaliza una muestra de audio mono en coma flotante
[data_cf,fs] = audioread("a.wav");

%Extraemos la HRIR ideales en coma flotante
for i = 1:cambios
pulseL_cf(:,i) = getNearestUCDpulse(azimuth(i),elevation(i),hrir_l_norm);
pulseR_cf(:,i) = getNearestUCDpulse(azimuth(i),elevation(i),hrir_r_norm);
end

j=0;
for j=1:9

word = 17-j;
frac = 16-j;

% Se convierten las HRIR ideales a coma fija
for i = 1:cambios
pulseL_fi(:,i) = fi(pulseL_cf(:,i),1,word,frac);
pulseR_fi(:,i) = fi(pulseR_cf(:,i),1,word,frac);
end

data_fi = fi(data_cf, 1,word,frac);

for i = 1:cambios
data_parts(:,i) = data_mono(1+SAMPLES*(i-1):SAMPLES*i);
end

for i = 1:cambios
signalL(:,i) = fi(conv(data_parts(:,i),pulseL(:,i)),1,word,frac);
```

```
signalR(:,i) = fi(conv(data_parts(:,i),pulseR(:,i)),1,word,frac);
end


signalL_full = signalL(:,1);
signalR_full = signalR(:,1);

for i = 2:cambios
signalL_full = vertcat (signalL_full, signalL(:,i));
signalR_full = vertcat (signalR_full, signalR(:,i));
end

output_fi = [signalL_full,signalR_full];

output_double = double(output_fi);

error(j) = immse(output_double,output_nofi);

end
```

```
% CREACION DE FICHERO COE
% Este código genera el fichero .coe de inicializacion que se carga
% en los bloques Xilinx BRAM

clear;
load hrir_final;

azimuth  = [0 0 0 0 90 -90];
elevation = [0 90 180 270 0 0];

for i = 1:6
pulseL_fi(:,i) = fi(getNearestUCDpulse(azimuth(i),elevation(i),hrir_l),1,8,7);
pulseR_fi(:,i) = fi(getNearestUCDpulse(azimuth(i),elevation(i),hrir_r),1,8,7);
end


%%% Generación del fichero de coeficientes izquierdos%%%
fid = fopen('coeficientes_Left.coe', 'wt');

% Hex value write to the txt file
fprintf(fid,'memory_initialization_radix=2;\n');
fprintf(fid,'memory_initialization_vector=\n');

for j = 1:6
    for i = 1:200
       fprintf(fid, '%s\n', bin(pulseL_fi(i,j)));
    end
end

% Close the txt file
fclose(fid);


%%% Generación del fichero de coeficientes derechos%%%
fid = fopen('coeficientes_Right.coe', 'wt');

% Hex value write to the txt file
fprintf(fid,'memory_initialization_radix=2;\n');
fprintf(fid,'memory_initialization_vector=\n');

for j = 1:6
    for i = 1:200
       fprintf(fid, '%s\n', bin(pulseL_fi(i,j)));
    end
end

% Close the txt file
fclose(fid);
```

```
%CPPYRIGHT NOTICE: the hrir_final.mat is provided by CIPIC HRTF Database.
%All rights reserved.
%https://ses.library.usyd.edu.au/handle/2123/15038

%
% Este script pretende ser una simulación usando codificación en coma
% fija en lugar de codificación en coma flotante. Con esta simulación
% se puede observar cuan diferentes son las respuestas al impulso respecto
% a las originales (que usan coma flotante) para decidir el número de bits
% dentro de la FPGA.
%


clear all;
load hrir_final;

SAMPLES = 5000;
cambios = 10;

% Diferentes ángulos para obtener distintas muestras al impulso
azimuth  = [-80 -65 -50 -35 -20 -5 10 25 40 -10];
elevation = [45 60 -45 235 78 132 84 94 -14 15];

%Normalizamos las respuestas al impulso
hrir_r_norm = hrir_r./max(max(max(hrir_r)));
hrir_l_norm = hrir_l./max(max(max(hrir_l)));

% Se digitaliza una muestra de audio mono en coma flotante
[data_cf,fs] = audioread("a.wav");

%Extraemos la HRIR ideales en coma flotante
for i = 1:cambios
pulseL_cf(:,i) = getNearestUCDpulse(azimuth(i),elevation(i),hrir_l);
pulseR_cf(:,i) = getNearestUCDpulse(azimuth(i),elevation(i),hrir_r);
end

HRIR_cf = [pulseL_cf,pulseR_cf];

j=0;
for j=1:11

word = 17-j;
frac = 16-j;

% Se convierten las HRIR ideales a coma fija
for i = 1:cambios
pulseL_fi(:,i) = fi(pulseL_cf(:,i),1,word,frac);
pulseR_fi(:,i) = fi(pulseR_cf(:,i),1,word,frac);
end

% Se convierten las HRIR de coma fija a coma flotante ya que la funcion
% immse necesita trabajar con doubles. En el proceso se ha perdido
% precisión, así que da igual que ahora convirtamos a double otra vez.

pulseL_fi = double(pulseL_fi);
pulseR_fi = double(pulseR_fi);

HRIR_fi = [pulseL_fi,pulseR_fi];


error(j) = immse(HRIR_cf,HRIR_fi);

end

N_bit_word = 16:-1:6;
plot (N_bit_word,error*100); %Error*100 para obtener el porcentaje
xlabel('Tamaño de la palabra (bits)');
ylabel('ECM')
```

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

-- Company: UAH

-- Engineer: César Cabanillas Núñez

--

-- Create Date: 28.02.2021 11:49:15

-- Design Name: Componente TOP práctica TFG

-- Module Name: top - Behavioral

-- Project Name: Musica 3D en FPGA

-- Target Devices:

-- Tool Versions:

-- Description:

--

-- Dependencies:

--

-- Revision:

-- Revision 0.01 - File Created

-- Additional Comments:

--

----------------------------------------------------------------------------------



library IEEE;

use IEEE.STD_LOGIC_1164.ALL;



-- Uncomment the following library declaration if using

-- arithmetic functions with Signed or Unsigned values

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

```
-- Uncomment the following library declaration if instantiating

-- any Xilinx leaf cells in this code.

--library UNISIM;

--use UNISIM.VComponents.all;



entity TOP is

   Port( -- ENTRADAS GENERALES

         CLK : in STD_LOGIC;

         reset : in  STD_LOGIC;

         -- ENTRADAS FIR

         Data_in : in STD_LOGIC_VECTOR (7 downto 0);

         Data_Valid_In : in STD_LOGIC;

         -- ENTRADAS BLOQUE MEMORIA

         Direccion : in STD_LOGIC_VECTOR (10 downto 0);

         Dir_flag : in STD_LOGIC; --Flag para avisar de que la dirección ha cambiado y que nuevos coeficientes deben ser cargados.

         -- SALIDAS FIR

         Data_Valid_Der : out STD_LOGIC;

         Data_Valid_Izq : out STD_LOGIC;

         Data_out : out STD_LOGIC_VECTOR (22 downto 0)

         );

end TOP;



architecture Behavioral of TOP is



   COMPONENT FIR

    PORT (

      -- ENTRADAS

      Data_in : in STD_LOGIC_VECTOR (7 downto 0);
```

```
    Data_Valid_In : in STD_LOGIC;
    CLK : in STD_LOGIC;
    reset : in  STD_LOGIC;
    -- ENTRADAS BLOQUE MEMORIA
    Coef_Left_in : in STD_LOGIC_VECTOR (7 downto 0);
    Coef_Right_in : in STD_LOGIC_VECTOR (7 downto 0);
    Coef_Enable_in : in STD_LOGIC_VECTOR (199 downto 0);
    -- SALIDAS
    Data_Valid_Der : out STD_LOGIC;
    Data_Valid_Izq : out STD_LOGIC;
    Data_out : out STD_LOGIC_VECTOR (22 downto 0)
    );
END COMPONENT;


COMPONENT MEMORIA_ROM
  PORT (
    -- Inputs
    Direccion : in STD_LOGIC_VECTOR (10 downto 0);
    Dir_flag : in STD_LOGIC; --Flag para avisar de que la dirección ha cambiado y que nuevos coeficientes deben ser cargados.
    clk : in STD_LOGIC;
    reset : in STD_LOGIC;
    -- Outputs
    Coef_Left_out : out STD_LOGIC_VECTOR (7 downto 0);
    Coef_Right_out : out STD_LOGIC_VECTOR (7 downto 0);
    Coef_Enable_out : out STD_LOGIC_VECTOR (199 downto 0)
    );
END COMPONENT;
```

```
    signal Coef_Left_aux : STD_LOGIC_VECTOR (7 downto 0);
    signal Coef_Right_aux : STD_LOGIC_VECTOR (7 downto 0);
    signal Coef_Enable_aux : STD_LOGIC_VECTOR (199 downto 0);


begin


   Filtro_FIR: FIR
    PORT MAP (
      Data_in => Data_in,
      Data_Valid_In => Data_Valid_In,
      clk => CLK,
      reset => reset,
      Coef_Left_in => Coef_Left_aux,
      Coef_Right_in => Coef_Right_aux,
      Coef_Enable_in => Coef_Enable_aux,
      Data_Valid_Der => Data_Valid_Der,
      Data_Valid_Izq => Data_Valid_Izq,
      Data_out => Data_out
      );


   Bloques_ROM: MEMORIA_ROM
    PORT MAP (
      Direccion => Direccion,
      Dir_flag => Dir_flag,
      clk => CLK,
      reset => reset,
      Coef_Left_out => Coef_Left_aux,
      Coef_Right_out => Coef_Right_aux,
```

```
        Coef_Enable_out => Coef_Enable_aux

        );



end Behavioral;
```

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.std_logic_arith.all;

use IEEE.numeric_std.all;



entity FIR is

    Port ( -- ENTRADAS

           Data_in : in STD_LOGIC_VECTOR (7 downto 0);

           Data_Valid_In : in STD_LOGIC;

           CLK : in STD_LOGIC;

           reset : in  STD_LOGIC;

           -- ENTRADAS BLOQUE MEMORIA

           Coef_Left_in : in STD_LOGIC_VECTOR (7 downto 0);

           Coef_Right_in : in STD_LOGIC_VECTOR (7 downto 0);

           Coef_Enable_in : in STD_LOGIC_VECTOR (199 downto 0);

           -- SALIDAS

           Data_Valid_Der : out STD_LOGIC;

           Data_Valid_Izq : out STD_LOGIC;

           Data_out : out STD_LOGIC_VECTOR (22 downto 0)

           );

end FIR;
```

```
architecture Behavioral of FIR is



  COMPONENT multiplicador

   PORT (

     CLK : IN STD_LOGIC;

     A : IN STD_LOGIC_VECTOR(7 DOWNTO 0);

     B : IN STD_LOGIC_VECTOR(7 DOWNTO 0);

     P : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)

     );

  END COMPONENT;



  COMPONENT sumador

   PORT (

     A : IN STD_LOGIC_VECTOR(22 DOWNTO 0);

     B : IN STD_LOGIC_VECTOR(15 DOWNTO 0);

     CLK : IN STD_LOGIC;

     S : OUT STD_LOGIC_VECTOR(22 DOWNTO 0)

     );

  END COMPONENT;



  COMPONENT Multiplexor_Coef is

   Port (

     SEL : in STD_LOGIC;

     Right_coef : in STD_LOGIC_VECTOR (7 downto 0);

     Left_coef : in STD_LOGIC_VECTOR (7 downto 0);

     Coef_Out : out STD_LOGIC_VECTOR (7 downto 0)

     );

  END COMPONENT;
```

```
COMPONENT MultiplexorData is
 Port (
  SEL : in STD_LOGIC;
  Right_Data : in STD_LOGIC_VECTOR (22 downto 0);
  Left_Data : in STD_LOGIC_VECTOR (22 downto 0);
  Data_Out : out STD_LOGIC_VECTOR (22 downto 0)
  );
END COMPONENT;

COMPONENT Registro_Coef is
 Port (
  clk : in std_logic;
  EN : in std_logic;
  rst : in std_logic;
  IN_reg : in std_logic_vector(7 downto 0);
  OUT_reg : out std_logic_vector(7 downto 0));
END COMPONENT;

COMPONENT Registro_Data is
 Port (
  clk : in std_logic;
  EN : in std_logic;
  rst : in std_logic;
  IN_reg : in std_logic_vector(22 downto 0);
  OUT_reg : out std_logic_vector(22 downto 0));
END COMPONENT;
```

```
  COMPONENT FSM_Datapath is

   PORT (

    clk : in STD_LOGIC;

    data_valid_in : in STD_LOGIC;

    reset : in STD_LOGIC;

    output : out STD_LOGIC_VECTOR (2 downto 0)

    );

  END COMPONENT FSM_Datapath;



  -- Constantes

  -- constant DataWidth : integer := 4;

  constant CoefNumber : integer := 200;



  -- Registros que almacenan los datapaths del canal derecho y del izquierdo.

  -- El filtro alterna los datos y los coeficientes para el canal derecho y

  -- para el izquierdo para cada una de las muestras. Es decir, se consigue

  -- hacer dos convoluciones en paralelo con un mismo filtro FIR.

  type Acc_Register is array (CoefNumber-1 downto 0) of std_logic_vector (22 downto 0);

  -- Interconexión entre los registros de datos y los multiplexores

  signal datapath_aux_Rreg : Acc_Register := (others=>(others=>'0')); -- Datapath auxiliar para conectar el Registro 'Right' al demux.

  signal datapath_aux_Lreg : Acc_Register := (others=>(others=>'0')); -- Datapath auxiliar para conectar el Registro 'Left' al demux.

  -- OUT: Interconecta la salida del sumador con los registros

  -- IN: Interconecta la salida del multiplexor con el siguiente sumador

  signal datapath_auxOUT : Acc_Register := (others=>(others=>'0'));   -- Datapath auxiliar para guardar el dato que SALE de los sumadores.

  signal datapath_auxIN : Acc_Register := (others=>(others=>'0'));    -- Datapath auxiliar para guardar el dato que va a ENTRAR en los sumadores.



  -- Datapath a la salida de los multiplicadores.
```

```
    type Res_Mul is array (CoefNumber-1 downto 0) of std_logic_vector (15 downto 0);
    signal Res_multi : Res_Mul := (others=>(others=>'0'));

    -- Señales que interconectan los multiplexores con los registros de coeficientes
    type Coef_reg is array (CoefNumber-1 downto 0) of std_logic_vector (7 downto 0);
    signal Coef_reg_izq : Coef_reg := (others=>(others=>'0'));
    signal Coef_reg_der : Coef_reg := (others=>(others=>'0'));

    -- Entrada de coeficientes a los multiplicadores
    type Coe_Mul is array (CoefNumber-1 downto 0) of std_logic_vector (7 downto 0);
    signal Coe_multi : Coe_Mul := (others=>(others=>'0'));

    -- Señal para elegir el bloque rom a usar
    signal ROM_select : STD_LOGIC := '0'; -- Right Coef si '1', Left_Coef si '0'

    -- Señal para elegir qué canal pasa por el Multiplexor de Datos
    signal DATA_select : STD_LOGIC := '0'; -- Right Data si '1', Left Data si '0'

    -- Señal de control que refresca el valor de los registros de los canales Left and Right
    signal ENA_Left_Reg : STD_LOGIC := '0'; -- Canal Izrecho. Cuando '1', lo que haya a la entrada lo pone a la salida
    signal ENA_Right_Reg : STD_LOGIC := '0'; -- Canal Derecho. Cuando '1', lo que haya a la entrada lo pone a la salida

    -- Señal que representa el estado en el que se encuentra el cambio de ragistros de salida (datapaths)
    signal state_data : STD_LOGIC_VECTOR (2 downto 0);

begin

    --Instanciación de los registros que almacenan los coeficientes
```

```
Gen_Reg_Coef_Right: for i in 0 to CoefNumber-1 generate

   Right_reg: Registro_coef

    PORT MAP (

      clk => clk,

      EN => Coef_Enable_in((CoefNumber-1)-i),

      rst => reset,

      IN_reg => Coef_Right_in,

      OUT_reg => Coef_reg_der(i)

      );

end generate Gen_Reg_Coef_Right;




Gen_Reg_Coef_Left: for i in 0 to CoefNumber-1 generate

   Left_reg: Registro_coef

    PORT MAP (

      clk => clk,

      EN => Coef_Enable_in((CoefNumber-1)-i),

      rst => reset,

      IN_reg => Coef_Left_in,

      OUT_reg => Coef_reg_izq(i)

      );

end generate Gen_Reg_Coef_Left;




-- Instanciación de los multiplexores de la selección de COEFICIENTES

Gen_Multiplexores_ROM: for i in 0 to CoefNumber-1 generate

   mux_coef : Multiplexor_Coef

    PORT MAP (
```

```
        SEL => ROM_select,
        Right_coef => Coef_reg_der(i),
        Left_coef => Coef_reg_izq(i),
        Coef_Out => Coe_multi(i)
        );
end generate;

-- Instanciación de los registros que guardan los datos del canal derecho e izquierdo
Gen_Registros_Datos: for i in 0 to CoefNumber-1 generate

    Right_reg: Registro_Data
     PORT MAP (
       clk => clk,
       EN => ENA_Right_Reg,
       rst => reset,
       IN_reg => datapath_auxOUT(i),
       OUT_reg => datapath_aux_Rreg(i)
       );

    Left_reg: Registro_Data
     PORT MAP (
       clk => clk,
       EN => ENA_Left_Reg,
       rst => reset,
       IN_reg => datapath_auxOUT(i),
       OUT_reg => datapath_aux_Lreg(i)
       );
```

```
end generate Gen_Registros_Datos;


-- Instanciación de los Demultiplexores del almacenado de DATOS
Gen_Multiplexores_Data: for i in 0 to CoefNumber-1 generate


   Todos_Mux : if i < (CoefNumber-1) generate
      Demux_Data : MultiplexorData
       PORT MAP (
         SEL => DATA_select,
         Right_Data => Datapath_aux_Rreg(i),
         Left_Data => Datapath_aux_Lreg(i),
         Data_Out => datapath_auxIN(i+1)
         );
   end generate Todos_Mux;


   Ultimo_Mux : if i = (CoefNumber-1) generate
      Demux_Data_Last: MultiplexorData
       PORT MAP (
         SEL => DATA_select,
         Right_Data => Datapath_aux_Rreg(CoefNumber-1),
         Left_Data => Datapath_aux_Lreg(CoefNumber-1),
         Data_Out => Data_out
         );
   end generate Ultimo_Mux;


end generate Gen_Multiplexores_data;


-- Instanciación de los multiplicadores
```

```
Gen_Multipicadores: for i in 0 to CoefNumber-1 generate

   multi : multiplicador

    PORT MAP (

      CLK => clk,

      A => Data_in,

      B => Coe_multi(i),

      P => Res_multi(i)

      );

end generate;



-- Instanciación de los sumadores

Gen_Sumadores : for i in 0 to CoefNumber-1 generate



   Primer_Sumador : if i=0 generate

      S0: sumador

       PORT MAP (

         A => datapath_auxIN(i),

         B => Res_multi(i),

         CLK => clk,

         S => datapath_auxOUT(i)

         );

   end generate Primer_Sumador;



   Resto_Sumadores : if i>0 generate

      SX: sumador

       PORT MAP (

         A => datapath_auxIN(i),

         B => Res_multi(i),
```

```
            CLK => clk,
            S => datapath_auxOUT(i)
            );
    end generate Resto_Sumadores;

end generate Gen_Sumadores;

-- Instanciación de la Maquina de Estados (FSM)
-- Esta máquina realiza en cambio de datapaths en los sumadores
Maquina_Estados : FSM_Datapath
  PORT MAP (
    clk => CLK,
    data_valid_in => Data_Valid_In,
    reset => reset,
    output => state_data
    );

cambiar_datapaths : process(clk,reset)
begin
    if reset = '1' then
        Data_Valid_Der <= '0';
        Data_Valid_Izq <= '0';
    elsif rising_edge(clk) then
        case state_data is
            when "000" =>
                -- Standby

            when "001" =>
```

```
            ROM_Select    <= '1';   -- Coef_Der


          when "010" =>

            ROM_Select    <= '0';   -- Coef_Izq

            DATA_Select   <= '1';   -- El Mux deja pasar los registros Derecha

            Data_valid_Der <= '1';   -- Datos del canal derecho válidos


          when "011" =>

            DATA_Select   <= '0';   -- El Mux deja pasar los registros Izquierda

            Data_valid_Izq <= '1';   -- Datos del canal izquierdo válidos

            Data_valid_Der <= '0';

            ENA_Right_Reg  <= '1';   -- Refrescamos registros Derecha justo antes de la siguiente operación


          when "100" =>

            Data_valid_Izq <= '0';

            ENA_Right_Reg  <= '0';

            ENA_Left_Reg   <= '1';   -- Refrescamos registros Izquierda justo antes de la siguiente operación


          when "101" =>

            ENA_Left_Reg   <= '0';


          when others =>

            Data_valid_Der <= 'X';

            Data_valid_Der <= 'X';

        end case;

      end if;

    end process;
```

```
end Behavioral;
```

```
-- Esta máquina de estados carga los datapath en las salidas de los multiplicadores

-- Carga primero los datos del canal derecho y a la vez que se realizan las operaciones

-- en los multiplicadores empieza a cargar los coeficientes del canal izquierdo.


library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


entity FSM_Datapath is

    Port ( clk : in STD_LOGIC;

           data_valid_in : in STD_LOGIC;

           reset : in STD_LOGIC;

           output : out STD_LOGIC_VECTOR (2 downto 0));

end FSM_Datapath;


architecture Behavioral of FSM_Datapath is


  type State_Type is (s0, s1, s2, s3, s4, s5);

  signal state : State_Type;


begin


    process (clk, reset)

    begin

       if reset = '1' then

          state <= s0;
```

```
    elsif (clk'event and clk = '1') then
      case state is


        when s0=>
          if data_valid_in = '1' then
            state <= s1;
          else
            state <= s0;
          end if;
        when s1=>
          state <= s2;
        when s2=>
          state <= s3;
        when s3=>
          state <= s4;
        when s4=>
          state <= s5;
        when s5=>
          state <= s0;
      end case;
    end if;
  end process;


  process (state)
  begin
    case state is
      when s0 =>
        output <= "000";
```

```
        when s1 =>
           output <= "001";
        when s2 =>
           output <= "010";
        when s3 =>
           output <= "011";
        when s4 =>
           output <= "100";
        when s5 =>
           output <= "101";
     end case;
  end process;

end Behavioral;
```

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

entity Registro_Coef is
   Port ( clk : in std_logic;
          EN : in std_logic;
          rst : in std_logic;
          IN_reg : in std_logic_vector(7 downto 0);
          OUT_reg : out std_logic_vector(7 downto 0)
          );
end Registro_Coef;
```

```
architecture Behavioral of Registro_Coef is

begin


   process (clk,rst,EN) --Aunque realmente los cambios de EN nunca harán nada _CCN *Preguntar*
   begin
      if rst = '1' then
         OUT_reg <= (others=>'0');
      elsif rising_edge(clk) then
         if EN = '1' then
            OUT_reg <= IN_reg;
         end if;
      end if;
   end process;


end Behavioral;
```

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


entity Multiplexor_Coef is
   Port ( SEL : in  STD_LOGIC;
          Right_Coef : in  STD_LOGIC_VECTOR (7 downto 0);
          Left_Coef : in  STD_LOGIC_VECTOR (7 downto 0);
          Coef_Out : out STD_LOGIC_VECTOR (7 downto 0));
end Multiplexor_Coef;
```

```
architecture Behavioral of Multiplexor_Coef is

begin



   process (SEL, Right_Coef, Left_Coef)

   begin



      case SEL is

         when '0' =>

            Coef_Out <= Left_Coef;

         when '1' =>

            Coef_Out <= Right_Coef;

         when others =>

            Coef_Out <= (others=>'X');

      end case;

   end process;

end Behavioral;
```

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;



entity Registro_data is

   Port ( clk : std_logic;

          EN : in std_logic;

          rst : in std_logic;

          IN_reg : in std_logic_vector(22 downto 0);

          OUT_reg : out std_logic_vector(22 downto 0)

          );
```

```
end Registro_data;



architecture Behavioral of Registro_data is

begin



   process (clk,rst,EN) --Aunque realmente los cambios de EN nunca harán nada _CCN *Preguntar*

   begin

      if rst = '1' then

         OUT_reg <= (others=>'0');

      elsif rising_edge(clk) then

         if EN = '1' then

            OUT_reg <= IN_reg;

         end if;

      end if;

   end process;



end Behavioral;
```

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;




entity MultiplexorData is

   Port ( SEL : in STD_LOGIC;

          Right_Data : in STD_LOGIC_VECTOR (22 downto 0);

          Left_Data : in STD_LOGIC_VECTOR (22 downto 0);

          Data_Out : out STD_LOGIC_VECTOR (22 downto 0)
```

```
        );
end MultiplexorData;


architecture Behavioral of MultiplexorData is
begin


   process (SEL, Right_Data, Left_Data)
   begin


      case SEL is
         when '0' =>
            Data_Out <= Left_Data;
         when '1' =>
            Data_Out <= Right_Data;
         when others =>
            Data_Out <= (others=>'X');
      end case;


   end process;


end Behavioral;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;


entity Memoria_ROM is
    Port (
        -- Inputs
        Direccion : in STD_LOGIC_VECTOR (10 downto 0);
        Dir_flag : in STD_LOGIC; --Flag para avisar de que la dirección ha cambiado y que nuevos coeficientes deben ser cargados.
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        -- Outputs
        Coef_Left_out : out STD_LOGIC_VECTOR (7 downto 0);
        Coef_Right_out : out STD_LOGIC_VECTOR (7 downto 0);
        Coef_Enable_out : out STD_LOGIC_VECTOR (199 downto 0)
        );
end Memoria_ROM;


architecture Behavioral of Memoria_ROM is


    COMPONENT ROM_Derecha
      PORT (
        clka : IN STD_LOGIC;
        addra : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
        douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
      );
    END COMPONENT;
```

```
COMPONENT ROM_Izquierda
 PORT (
   clka : IN STD_LOGIC;
   addra : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
   douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
 );
END COMPONENT;



COMPONENT FSM_ROM is
 PORT (
   clk : in STD_LOGIC;
   Dir_flag : in STD_LOGIC;
   reset : in STD_LOGIC;
   output : out STD_LOGIC_VECTOR (2 downto 0)
 );
END COMPONENT FSM_ROM;


-- Este número indica el número de coeficientes que tiene el filtro, y que por tanto, hay que traer de la memoria.
constant N_coef : integer := 200; -- El cero también cuenta


-- Controlador de la dirección del coeficiente que va a salir en cada ciclo de reloj
signal Dir_control_vec : STD_LOGIC_VECTOR (10 downto 0) := "00000000000";


-- Controlador del índice que lleva la cuenta de qué emable hay que activar cada vez.
signal Coef_control : integer range 0 to N_coef-1 := 0;
```

```
    -- Índice de los datos que se tienen que sacar. Limita Dir_Control_Vec para que no

    signal Data_control : integer range 0 to N_coef-2 := 0; -- Se resta 2 porque Dir_Control_Vec ya se ha aumentado 1 vez en el
estado "010"



    -- Señal que representa el estado en el que se encuentra el cambio de ragistros de salida (datapaths)

    signal state : STD_LOGIC_VECTOR (2 downto 0);




begin



    Right_ROM : ROM_Derecha

     PORT MAP (

       clka => clk,

       addra => Dir_control_vec,

       douta => Coef_Right_out

       );



    Left_ROM : ROM_Izquierda

     PORT MAP (

       clka => clk,

       addra => Dir_control_vec,

       douta => Coef_Left_out

       );



    -- Instanciación de la Maquina de Estados (FSM)

    -- Esta máquina realiza en cambio de datapaths en los sumadores

    Maquina_Estados : FSM_ROM

     PORT MAP (

       clk => clk,
```

```
    Dir_flag => Dir_flag,
    reset => reset,
    output => state
    );

cambiar_datapaths : process(clk,reset)
begin
    if reset = '1' then

    elsif rising_edge(clk) then
      case state is
        when "000" =>
          --Do nothing
          Coef_control <= 0;
          Data_control <= 0;
          Coef_Enable_out <= (others=>'0');

        when "001" =>
          Dir_Control_vec <= Direccion;

        when "010" =>
          Dir_Control_vec <= Dir_Control_vec + '1';

        when "011" =>
          -- Este IF saca N_coef datos de la ROM. Lleva un contador (Data_control) para sacar el número exacto
          if Data_control < N_coef-2 then
            Dir_Control_vec <= Dir_Control_vec + '1';
            Data_control <= Data_control + 1;
```

```
                end if;


                -- Este IF activa el ENABLE correspondiente al registro que tenga que cargar el coeficiente en ese ciclo.
                if Coef_control = 0 then
                   Coef_Enable_out(Coef_control) <= '1';
                   Coef_control <= Coef_control + 1;
                elsif Coef_control < N_coef then
                   Coef_Enable_out(Coef_control-1) <= '0';
                   Coef_Enable_out(Coef_control) <= '1';
                   Coef_control <= Coef_control + 1;
                end if;


             when others =>


          end case;
       end if;
    end process;


end Behavioral;
```

```
-- Esta máquina de estados carga los datapath en las salidas de los multiplicadores
-- Carga primero los datos del canal derecho y a la vez que se realizan las operaciones
-- en los multiplicadores empieza a cargar los coeficientes del canal izquierdo.


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

```
entity FSM_ROM is

    Port ( clk : in STD_LOGIC;

           Dir_Flag : in STD_LOGIC;

           reset : in STD_LOGIC;

           output : out STD_LOGIC_VECTOR (2 downto 0));

end FSM_ROM;



architecture Behavioral of FSM_ROM is



   type State_Type is (s0, s1, s2, s3);

   signal state : State_Type;



   --Este número indica el número de veces que se va a repetir el estado s5.

   constant N_coef : integer := 200;



   --Controlador del índice que lleva la cuenta de qué emable hay que activar cada vez.

   signal Coef_control : integer range 0 to N_coef+1 := 0;



begin



    process (clk, reset)

    begin

       if reset = '1' then

           state <= s0;

       elsif (clk'event and clk = '1') then

           case state is



              when s0=>
```

```
            if Dir_Flag = '1' then
               state <= s1;
            else
               state <= s0;
               Coef_control <= 0;
            end if;
         when s1=>
            state <= s2;
         when s2=>
            state <= s3;
         when s3=>
            if Coef_control < N_coef-1 then
               Coef_control <= Coef_control + 1;
               state <= s3;
            else
               state <= s0;
            end if;
      end case;
   end if;
end process;


process (state)
begin
   case state is
      when s0 =>
         output <= "000";
      when s1 =>
         output <= "001";
```

```
        when s2 =>
            output <= "010";
        when s3 =>
            output <= "011";
    end case;
  end process;


end Behavioral;
```

Universidad de Alcalá