Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Trabajo Fin de Grado

Procesado de datos en FPGAs mediante el empleo de HLS

Autor: Jaime Nuevo Benito

Tutor/es: Ignacio Bravo Muñoz

2022

ÍNDICE

Índice de figuras

Resumen

El presente trabajo trata sobre el desarrollo de un sistema de procesado digital de datos implementado en la tecnología FPGA (Field-Programmable Gate Array), empleando a su vez herramientas de HLS (High-Level Synthesis). El sistema se caracteriza por basarse en el procesado digital de audio, donde flujos de datos de audio serán tratados en diferentes etapas por algoritmos de procesado de señal.

El sistema a implementar constará de distintos bloques funcionales conectados entre sí, los cuales modifiquen el flujo de datos entrante según el propósito de cada uno, formando finalmente un sistema secuencial que entregará los datos generados a la salida.

Palabras clave:

FPGA

HLS

AXI

Algoritmos

Dataset

Abstract

This document is based on the development of a digital data processing system implemented on the FPGA (Field-Programmable Gate Array) technology, using HLS (High-Level Synthesis) tools. The system is characterized by being about the digital audio processing, where data streams of audio will be treated in different stages by algorithms of signal processing.

The system to be implemented will have several functional blocks connected to each other, which modify the incoming data stream according on the purpose of each, generating a sequential system that will give the generated data to the output.

Key words:

FPGA

HLS

AXI

Algorithms

Dataset

Resumen extendido

El objetivo de este trabajo fin de grado es el desarrollo de un sistema de procesado digital de datos el cual demanda el uso de una tecnología adecuada para llevar a cabo el diseño. Para ello, nos encontramos con que la mejor opción para implementar nuestro sistema es mediante el uso de un dispositivo de hardware programable como es una FPGA (Field-Programmable Gate Array).

El uso de dispositivos de lógica programable para este proyecto es debido a que se caracterizan por la posibilidad de ser programados y reprogramados para realizar una tarea en específico, haciéndolos más eficientes y flexibles con respecto a un procesador común y siendo de gran utilidad, debido al rendimiento que ofrecen y su alto grado de paralelismo, en aplicaciones de procesado de datos, consiguiendo así un gran auge en los últimos años.

Además, nos apoyaremos en el uso de herramientas de síntesis a alto nivel HLS (High-Level Synthesis) a la hora de generar algunos de los distintos bloques que compone nuestro sistema. Se trata de una herramienta interesante debido a que al trabajar con lenguaje de alto nivel (C, C++ o SystemC) podemos generar el hardware digital deseado en un modelo RTL (Resistor-Transistor Logic) de una forma optimizable y eficiente mediante la transformación de nuestro código HLS. Esto nos permite crear núcleos IP (Intellectual Property), en el entorno de desarrollo Vivado HLS™ de Xilinx, de una forma más rápida y con un mayor rendimiento (sin la necesidad de crear el modelo RTL manualmente) para posteriormente usar estos bloques en el propio software Vivado™.

Nuestro sistema de procesado de datos estará orientado en el procesado digital de audio. Dispondremos de un fichero de muestras extraídas de un archivo de audio, las cuales convertiremos en un flujo de datos que, secuencialmente, será procesado por las distintas etapas y bloques que disponemos en nuestro sistema. Las distintas etapas que componen el sistema global son: etapa de pre-procesado de audio, procesado de audio y post-procesado de audio.

Antes de mencionar los algoritmos de tratado de audio que dispone cada etapa cabe destacar el proceso de diseño de los mismos. En primer lugar, todos los algoritmos serán previamente estudiados e implementados en MATLAB®, obteniendo el resultado numérica, gráfica y sonoramente a partir de un fichero ".wav" de audio. Este estudio y desarrollo en MATLAB® nos servirá como validación para el funcionamiento de dicho algoritmo.

Posteriormente, pasaremos a desarrollar el algoritmo en nuestro entorno de síntesis de alto nivel Vivado HLS™, implementando el código C/C++ y verificando su comportamiento mediante la lectura del fichero de audio de salida generado por nuestro proceso en HLS.

Finalmente, optimizaremos el algoritmo mediante las herramientas de Vivado HLS y generaremos nuestro núcleo IP en lenguaje VHDL (Very high speed integrated circuits Hardware Description Language) preparado para ser usado como parte de nuestro sistema global en Vivado™.

Con respecto al interior de los bloques, la primera etapa del sistema es la de pre-procesado de audio. El algoritmo localizado en la misma es un Compresor de Rango Dinámico de audio o DRC (Dynamic Range Compressor). Se encarga de controlar el rango dinámico de nuestra señal de audio mediante la compresión de la misma en momentos de superación de un umbral estimado con objeto de evitar una posible saturación.

Posteriormente, en la etapa de procesado de audio, o etapa de filtrado de audio, dispondremos de un filtro de respuesta finita al impulso (FIR: Finite Impulse Response) o un filtro de respuesta infinita al impulso (IIR: Infinite Impulse Response) el cual tendrá el objetivo de reducir o eliminar el ruido que incorpore nuestra señal de entrada. Se realizará una comparativa entre ambos y se decidirá finalmente cual de los dos se implementará en nuestro sistema.

Por último, la etapa de post-procesado de audio, donde encontramos un banco de efectos de sonido que modificarán el flujo de datos según la estructura de los mismos dando lugar a particulares efectos de audio, obteniendo en ese momento los ficheros de datos de salida de cada efecto individual o de varios efectos combinados.

Todos los algoritmos serán conectados en el sistema global y se comunicarán entre ellos mediante el protocolo AXI (Advanced eXtensible Interface), el cual se trata de un bus de comunicaciones desarrollado en los chips de ARM para la transmisión y recepción de datos entre núcleos IP. Obteniendo finalmente un sistema sintetizable e implementable en FPGA al que le introduzcamos un dataset de audio y del que obtengamos el dataset de audio final deseado.

1. Introducción

El procesado digital de datos basado en la tecnología FPGA está a la orden del día debido al gran rendimiento que ofrecen en este ámbito, es por ello por lo que nuestro proyecto, dada su temática, estará enfocado en esta tecnología al encontrarnos que es la opción idónea por las características de la misma relacionadas con la flexibilidad, reprogramabilidad, paralelismo o alta eficiencia en el procesado de datos a alta velocidad.

El sistema de procesado de datos elegido se basará en un sistema de **procesado digital de audio**, el cual demandará la tecnología FPGA debido a las características y necesidades que impondremos al propio sistema. La elección de basar el proyecto en un sistema de procesado de audio es debido a que este mismo campo no está tan desarrollado en el mundo FPGA comparado con otras tecnologías, por lo que aún hay un gran campo por explorar, además de ser la mejor opción en la que implementar nuestro sistema frente a otras tecnologías como DSP (Digital Signal Processor) u otros procesadores comunes.

Bajo esta premisa, se ha seleccionado una estructura concreta para el sistema, la cual engloba un conjunto de algoritmos elegidos entre la gran variedad de posibles que podemos estudiar y aplicar en una aplicación de tratado de sonido.

Previamente a realizar el estudio y desarrollar todos los algoritmos que formarán el sistema, se esquematizará mediante un diagrama de bloques como será el sistema desde una visión general como podemos ver en la siguiente figura.

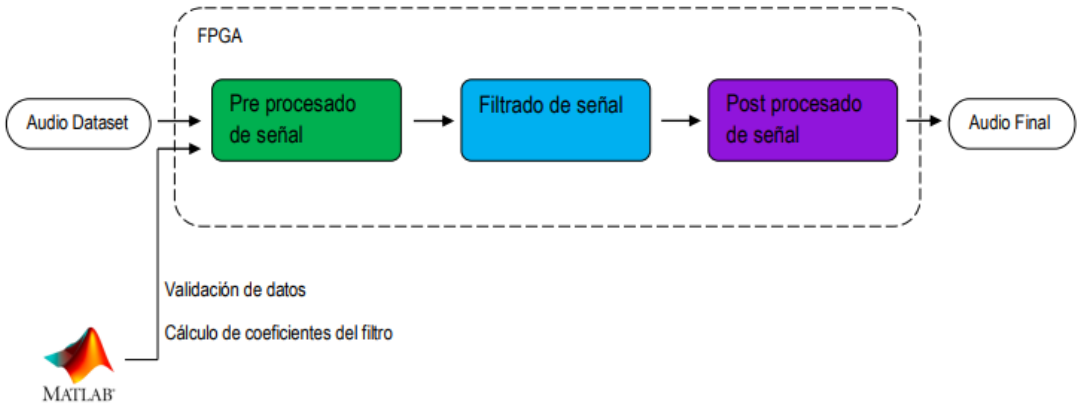


Figura 1: Esquema general del sistema

En el esquema de la figura 1 se plasma la idea general de nuestro sistema implementado en FPGA, de forma que el mismo recibirá a la entrada el dataset de audio. En nuestro caso, el dataset de audio consistirá en un fichero ".txt" al ser la mejor opción en la que plasmar todas las muestras, línea a línea, obtenidas de la lectura de un fichero ".wav". Nuestra FPGA trabajará con archivos binarios, por lo que las muestras serán datos binarios los cuales han sido codificados previamente en el tipo de dato "coma fija".

Esto será un asunto importante en nuestro trabajo, ya que, como veremos más adelante, un archivo de audio ".wav" está compuesto por datos decimales entre 1 y -1.

Esto significa que todos los bloques que implementemos deberán de estar basados en datos de tipo coma flotante o coma fija al necesitar el procesado de números decimales, por lo que, se decidirá usar datos en coma fija frente a datos en coma flotante como se verá justificado próximamente. Además, nuestros datos binarios codificados en coma fija tendrán una determinada precisión, unos bits serán los correspondiente a la parte decimal y los restantes corresponderán a la parte entera del total. Se realizará un estudio del error cometido al usar datos tipo coma fija frente a los datos en coma flotante leídos directamente del archivo original de audio, estos últimos serán denominados "golden data", es decir, los datos a conseguir con el menor error posible.

Como podemos ver en la figura 1, el primer bloque de nuestro sistema se denomina "Pre-procesado de audio", en él encontraremos un bloque basado en un algoritmo de compresión de rango dinámico de audio o DRC (Dynamic Range Compressor). Todos y cada uno de los algoritmos serán explicados y detallados en apartados posteriores, pero como introducción a este bloque podemos decir que su objetivo principal será controlar los valores máximos de la señal de audio, de modo que esta sea atenuada en momentos de saturación, todo ello basándonos en el modelo matemático del mismo compresor. Este algoritmo será implementado en HLS.

Seguidamente, nos encontramos con la etapa central o etapa de filtrado. Debido a que nuestra señal de audio original suele tener algún tipo de ruido acoplado a la misma, deberemos de filtrar la señal para eliminar en la mayor medida posible este ruido. Nuestro filtro será de tipo FIR o IIR, la decisión será tomada mediante la comparación de ambos en un apartado del trabajo, considerando sus ventajas y desventajas de forma que finalmente se diseñe el filtro y se obtengan los coeficientes, mediante MATLAB®, que usaremos en su posterior implementación en Vivado™.

Para finalizar, la etapa de "Post-procesado de audio". En ella nos encontraremos un banco de efectos de audio, donde se seleccionarán varios de ellos y se implementarán en el sistema, de forma que entre los diferentes tipos de efectos, que veremos detallados en su correspondiente apartado, se seleccionarán varios de los mismos y de distintos tipos. Estos efectos tendrán el objetivo de modificar la señal de audio que les llegue de forma que consigan generar la modificación de sonido deseado, además, tendremos la opción de obtener el audio con efectos individuales aplicados sobre él o podremos obtener el audio con efectos conectados entre sí de forma que podamos aplicar varios de forma consecutiva sobre un mismo audio. Todos los bloques que formen nuestro banco serán implementados en HLS.

Además, en el banco de efectos, el usuario tendrá la opción de seleccionar salidas de efectos individuales o salidas de efectos combinados, donde se combinen un par de efectos, generando así una mayor variedad de posibilidades y usando el paralelismo que nos permite usar una FPGA.

Cabe destacar, que bloques que realizaremos en HLS, mencionados anteriormente, seguirán un proceso similar de desarrollo, implementación y validación. La herramienta MATLAB® nos servirá como método de estudio y ejemplo de validación inicial a la hora de realizar los algoritmos o probar cualquier parte de nuestro sistema. Como se puede observar en el siguiente diagrama de flujo, MATLAB® será nuestro punto de partida, para finalmente conseguir realizar nuestro núcleo IP en HLS con el menor error posible y de la manera más eficiente posible.

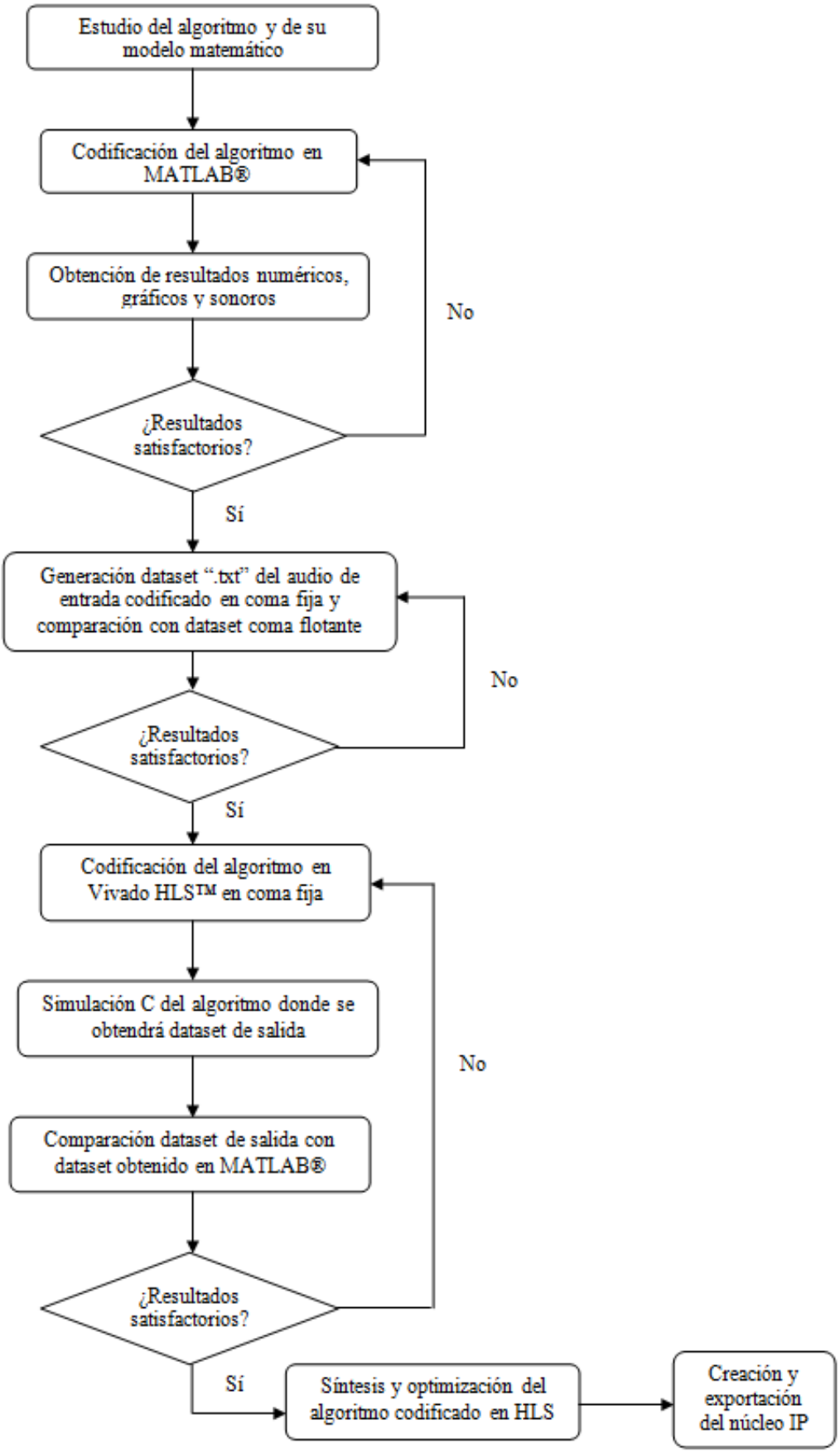


Figura 2: Diagrama de flujo proceso creación de bloques IP-core

Siguiendo los pasos mostrados en el diagrama de flujo anterior, podremos crear los bloques de nuestro sistema de una manera correcta y ordenada. Una vez tengamos nuestros bloques, y procedamos a crear nuestro sistema donde todos ellos irán conectados entre sí, deberemos de estudiar en profundidad el protocolo de comunicaciones que harán que los bloques transmitan y reciban datos de manera correcta. El protocolo usado será AXI (Advanced eXtensible Interface), el cual es un protocolo de comunicaciones albergado en los chips ARM, se profundizará en sus sub-protocolos así como en su forma de funcionamiento.

Cabe destacar, que todo sistema basado en FPGA debe de tener un dispositivo sobre el que implementar nuestro diseño, en nuestro caso, se utilizará la familia Artix7 debido a su uso en asignaturas previas así como su disponibilidad en la institución mediante la tarjeta Nexys 4. Concretamente usaremos el dispositivo "**xc7a200t**…"

1.1 Contexto

Hablar de procesado de audio es hablar de producción musical, la cual lleva en constante evolución desde mediados del siglo XX. Las diferentes revoluciones tecnológicas de la época incidieron en el procesado del sonido de forma notoria, dando lugar a nuevos formatos de grabación de audio como el disco de vinilo (1948), casete (1963)…, nuevos dispositivos de reproducción de audio como la radio portátil (1954), disco compacto (1981) o internet (1990). Además, el proceso de "electrificación" musical comenzó debido a la aparición de nuevos estilos musicales tales como el rock and roll, el cual situaba la guitarra eléctrica como un instrumento innovador [1].

Las nuevas tecnologías musicales seguían creciendo, dando lugar a múltiples dispositivos, ya sean analógicos o digitales. En este trabajo, se implementarán distintos algoritmos que emularán a dispositivos de procesado de audio hardware o software, como por ejemplo el compresor acústico, el cual fue diseñado, por primera vez, por la empresa Teletronix en 1965 [2]. Este dispositivo, que veremos detallado más adelante, tiene como objetivo reducir el rango dinámico de la señal acústica y controlar la saturación de la misma.

Figura 3: Compresor acústico Teletronix [2]

En relación con lo anterior y con el continuo crecimiento de la tecnología musical, surgieron instrumentos o efectos musicales como por ejemplo el pedal "wah-wah", el cual se creó en 1967 [1] y fue diseñado en primera instancia como complemento a la guitarra eléctrica, generando un efecto sonoro que se detallará más adelante.

Figura 4: Pedal Cry baby [1]

Los dos ejemplos mencionados son la prueba de que a lo largo de todo este tiempo, los productores musicales han usado una variedad inmensa de herramientas de producción musical con las cuales modifican una pista de audio con un objetivo final tales como comprimir el audio, ecualizar el mismo, cambiar el tono de una pista, aplicar efectos de sonido o filtrar el audio, ya sea de manera analógica mediante hardware externo o de manera digital con un software de producción musical. En la actualidad, todas las herramientas o algoritmos de procesado de audio digital están ampliamente estudiadas desde un marco teórico o matemático, de forma que en nuestro trabajo implementaremos algunos de los algoritmos más interesantes en nuestra tecnología FPGA.

Al estar, la producción musical, tan demandada en los tiempos que corren, es interesante buscar nuevas formas de conseguir métodos o aplicaciones que sean equiparables o mejores, en cuestión de rendimiento, a las herramientas tradicionales de procesado digital de audio.

1.2 Objetivos

Todo trabajo o proyecto tiene una serie de propósitos a cumplir. Objetivos los cuales al completarlos corroboran una satisfactoria realización del trabajo. En nuestro caso, la meta principal es el desarrollo y validación de un sistema de procesado de audio digital implementado en FPGA. Para ello elaboraremos la lista de objetivos principales que nos permitan conseguir la propuesta inicial, son los siguientes:

- Estudio de los algoritmos del sistema de pre-procesado y post-procesado de señal para su implementación y validación en MATLAB® y Vivado HLS™ mediante comparación de resultados y errores de los mismos.

- Optimización de los algoritmos desarrollados en Vivado HLS™ mediante el uso de directivas o directrices, así como la realización de síntesis, co-simulación VHDL y generación de los núcleos IP, para su posterior uso en el sistema global desarrollado en Vivado™.

- Estudio y comparativa entre los filtros FIR e IIR en MATLAB® para decidir cuál será el filtro central que forme parte de nuestro sistema, así como la posterior implementación en Vivado™ de dicho filtro elegido.

- Estudio y creación del sistema de bloques final, disposición final de los mismos y análisis de los bloques AXI albergados en Vivado™ extra que

necesitaremos para que el sistema funcione correctamente

- Obtención de simulaciones funcionales y temporales post síntesis y post implementación donde comprobemos los resultados obtenidos del mismo, así como elaborar unas conclusiones finales sobre el sistema completo y de los resultados obtenidos.

1.3 Desarrollo y estructura del proyecto

El presente documento estará organizado según los siguientes apartados.

- **Capítulo 1: Introducción.** Se definirá el contexto histórico y actual del procesado digital de audio, así como los objetivos a conseguir de nuestro proyecto.

- **Capítulo 2: Marco teórico.** En este capítulo se explicarán conceptos teóricos relativos al proyecto (FPGAs y sus aplicaciones, Vivado HLS, AXI, AXI-Stream y codificación coma fija).

- **Capítulo 3: Desarrollo del sistema de procesado de audio.** Apartado en el cual se desarrollarán e implementarán todos los bloques que componen el sistema, así como para el sistema completo. Se seguirán los pasos u objetivos descritos en el apartado anterior.

- **Capítulo 4: Resultados de los bloques desarrollados y del sistema completo.** Se mostrarán todos los resultados obtenidos mediante las simulaciones de cada uno de los bloques así como del sistema final.

- **Capítulo 5: Conclusiones y futuros trabajos:** Resumen final del trabajo realizado así como una enumeración y justificación de posibles añadidos que podrían haberse realizado o ideas para futuros proyectos.

- **Capítulo 6: Bibliografía:** Listado de referencias seguidas como documentación para llevar a cabo el proyecto.

2. Marco teórico

Para abordar el trabajo de la mejor manera posible, es necesario estudiar algunos conceptos relevantes que serán mencionados reiteradamente en el proyecto.

Entre los temas a tratar en este capítulo se encuentran: FPGAs y sus aplicaciones, debido a ser la tecnología principal en la que se basa todo nuestro proyecto y el por qué de su uso. Vivado HLS, ya que también se trata de una parte importante del proyecto, mostrando y explicando sus ventajas al usarlo. El protocolo AXI y su variante AXI4-Stream, donde se detallará sobre su funcionamiento y su relevancia en el sistema. Finalmente, se explicará la codificación en coma fija, así como su comparación con la coma flotante en aplicaciones sobre FPGA.

2.1 FPGA: Field-Programmable Gate Array

Las FPGAs se tratan de dispositivos programables en campo compuesto por una matriz de puertas lógicas. Esto significa que contiene bloques lógicos configurables (CLB) que pueden ser interconexionados entre sí de la manera concreta que especifique nuestro código basado en un lenguaje de descripción concreto, VHDL en nuestro caso como ya hemos visto con anterioridad [3]. Cabe destacar que la primera FPGA en aparecer en la industria tecnológica fue en 1985 por parte de Xilinx, con un dispositivo llamado XC2064 con 64 CLBs [4], siendo un importante avance en esta tecnología.

A lo largo de la historia, esta tecnología ha sido comparada con una tecnología parecida como son los ASIC (Circuitos Integrados de Aplicaciones Específicas), las cuales son usadas en aplicaciones parecidas pero la diferencia entre ambos es que las FPGAs son reprogramables mientras que las ASIC no lo son, además de que los costes y tiempos de desarrollo y producción son mucho menores. En cambio, las FPGA son más lentas y necesitan un mayor consumo de potencia, además de estar orientadas a sistemas menos complejos que una ASIC, aunque cabe decir que hoy en día es una tecnología con un gran avance y cada vez son menores estas desventajas de las FPGA con respecto a los ASIC [4] [3].

Son dispositivos que por su propia naturaleza tienen una alta flexibilidad, ya que pueden albergar desde la puerta lógica más básica hasta circuitos de alta complejidad que necesiten procesar gran cantidad de datos, de modo, que al ser programados con un diseño hardware concreto para una tarea en concreto, que desee el programador, se pueden llegar a convertir en dispositivos más eficientes que procesadores convencionales para ciertas tareas.

Como se ha mencionado anteriormente, las FPGA es una matriz de bloques lógicos configurables o programables (CLBs). Estos bloques son sistemas que albergan la siguiente estructura que podemos ver en la figura, donde nos encontramos un sistema con 4 entradas que entran a una Look Up Table (LUT) encargada de almacenar la información necesaria para generar puertas lógicas y sus valores de salida dependiendo de la combinación de bits dada por las 4 entradas.

El sistema es seguido de un Flip-Flop (FF) o también conocido como biestable,

gobernado por un reloj (CLK) y una señal de reseteo (RST) que entregará a su salida el contenido que se encuentre en la entrada, funcionando como un registro binario que guardará estados lógicos entre flancos de la señal de reloj.

Finalmente se halla un multiplexor encargado de seleccionar entre la salida de la LUT o del FF, es decir, la salida registrada en el Flip-Flop o la no registrada en el mismo.

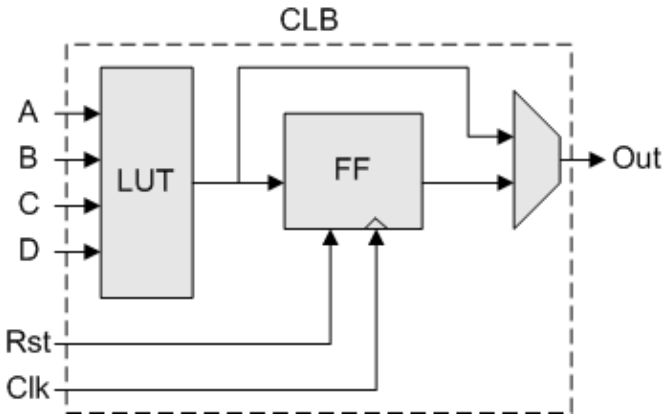


Figura 5: Diagrama bloques interno CLB [5]

La arquitectura global de una FPGA dispone de más elementos que Xilinx ha incluido en sus dispositivos, a parte de los CLBs, como bloques de entrada/salida, bloques de memoria y bloques DSP. Donde los bloques de memoria o BRAM (bloques de acceso aleatorio a memoria) serán de gran utilidad para almacenar datos en una memoria de tamaño fijo, así como los DSP, bloques lógicos de procesado digital de señal encargados de la realización de operaciones aritméticas haciendo más eficiente al dispositivo.

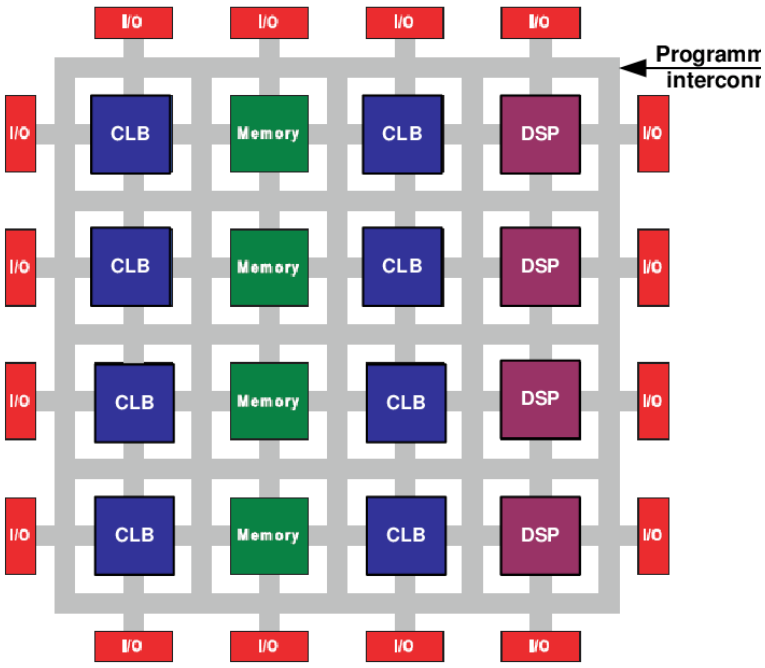


Figura 6: Arquitectura general de una FPGA [6]

2.2 FPGA y sus aplicaciones

Habiendo ya visto la teoría y contexto sobre las FPGA veremos en qué campos pueden ser aplicadas. Al ser dispositivos reprogramables y de bajo coste encajarán en gran diversidad de industrias o aplicaciones como pueden ser [3]:

- Defensa y aeroespaciales.
- Automoción.
- Electrónica de consumo.
- Industrial.
- Medicina.
- Seguridad.
- Procesado de audio, imagen y video.
- Comunicaciones inalámbricas y por cable.
- Centros de datos.

2.3 Vivado HLS

La herramienta llamada Vivado HLS™ (High Level Synthesis) de Xilinx será de gran importancia en el proyecto, como ya se ha mencionado con anterioridad. Se trata de una herramienta creada para trabajar en lenguaje de alto nivel (C, C++ o SystemC) en vez de trabajar en un lenguaje de descripción hardware como VHDL. De esta forma, Vivado HLS™ nos brinda la opción de generar algoritmos, creado a partir de código de alto nivel, y poder transformarlos a algoritmos en código de descripción hardware RTL, de forma que el usuario gane en tiempo al poder codificar un algoritmo en lenguaje de alto nivel y crear núcleos IP así mismo.

Para poder trabajar con la herramienta, debemos de estudiarla con anterioridad y conocer su estructura de trabajo. En primer lugar, al crear un proyecto con el objetivo final de generar un núcleo IP basado en un algoritmo que nosotros deseemos, elegiremos el dispositivo o familia de dispositivos al que irá orientado nuestro bloque, después, deberemos de añadir como mínimo, un fichero en C/C++ y su banco de pruebas o "testbench" en C/C++ también, además de posibles cabeceras en fichero de extensión ".h".

Nuestro fichero C/C++ albergará la codificación referente al algoritmo deseado dentro de una función "top", esta función será la que posteriormente, sea sintetizada y transformada a RTL. Una vez hayamos desarrollado nuestra función principal, pasaremos a generar nuestro "testbench", el cual solo será de utilidad para la validación de nuestro algoritmo mediante las simulaciones dentro de Vivado HLS™. Será un código en el que le explicaremos al programa que es lo que queremos que suceda al simular el mismo, de modo que en el punto del código que queramos de la forma que queramos llamemos a la función "TOP" y testeemos su funcionamiento.

El flujo de diseño de Vivado HLS™ viene representado en la siguiente figura, donde podemos seguir los pasos del primero al último en el proceso de generación de núcleos IP.

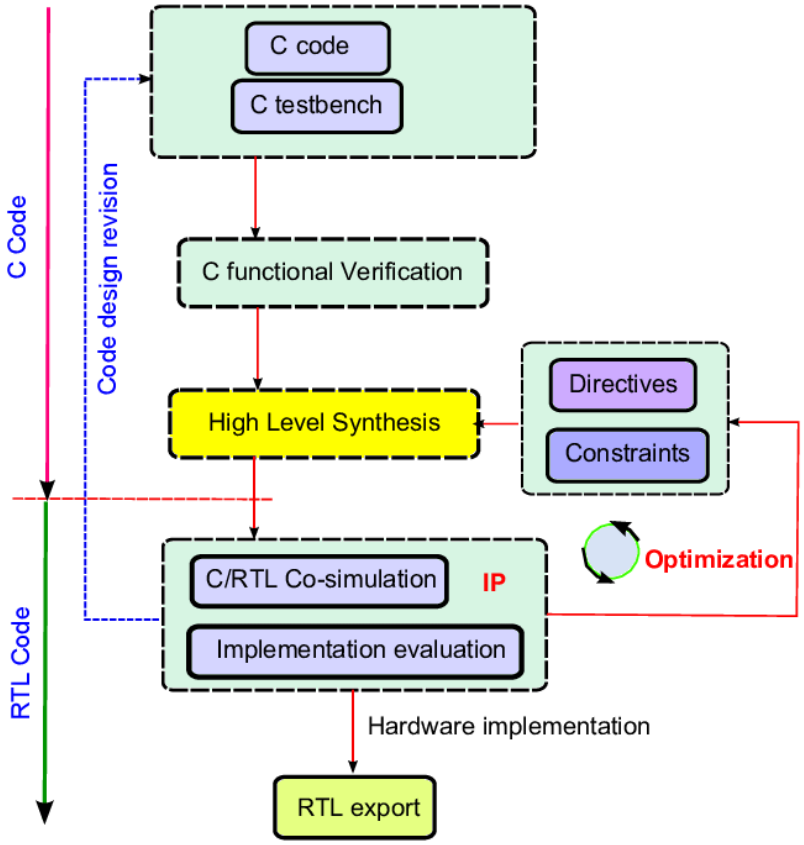


Figura 7: Diagrama de flujo de proceso de creación de núcleos IP en Vivado HLS [7]

En la figura se aprecian dos partes en el esquema, la parte basada en código C/C++ y la parte basada en código RTL, en la parte de código C vemos como al verificar nuestro código C, mediante su testbench, pasaremos a realizar la síntesis de alto nivel, aunque cabe destacar que no todo código es sintetizable. En este punto, Vivado HLS™ nos da la opción de añadir directivas de optimización a nuestro código, de modo que al implementarlas mejoraremos el rendimiento y optimizaremos los recursos.

Además, en el apartado de las directivas tendremos la posibilidad de configurar las interfaces de los puertos de entrada y salida, pudiendo implementar diferentes protocolos sobre nuestro sistema. La búsqueda de la optimización del rendimiento y recursos de nuestra FPGA será un apartado a tener en cuenta y que veremos en detalle en el siguiente capítulo al tener que usarlas en nuestro algoritmos. Las directivas podrán ser implementadas en cada una de las variables del código así como sobre la función global en sí, teniendo diferentes directivas en cada caso. Disponemos de gran variedad de directivas en Vivado HLS™, las cuales nos ayudarán a mejorar el rendimiento del algoritmo a la hora de ser sintetizado, de modo que optimicemos los recursos y el tiempo de funcionamiento, o no, según la o las directivas que empleemos. Las más destacadas e importantes son las siguientes [8]:

- **Pipeline**: Directiva aplicable sobre funciones o bucles la cual permite la ejecución concurrente de diferentes operaciones, de forma que optimizaremos nuestro código mediante la ejecución en paralelo de las

operaciones del mismo. Al usar esta directiva se mejorará la latencia a pesar de usar más recursos hardware del dispositivo.

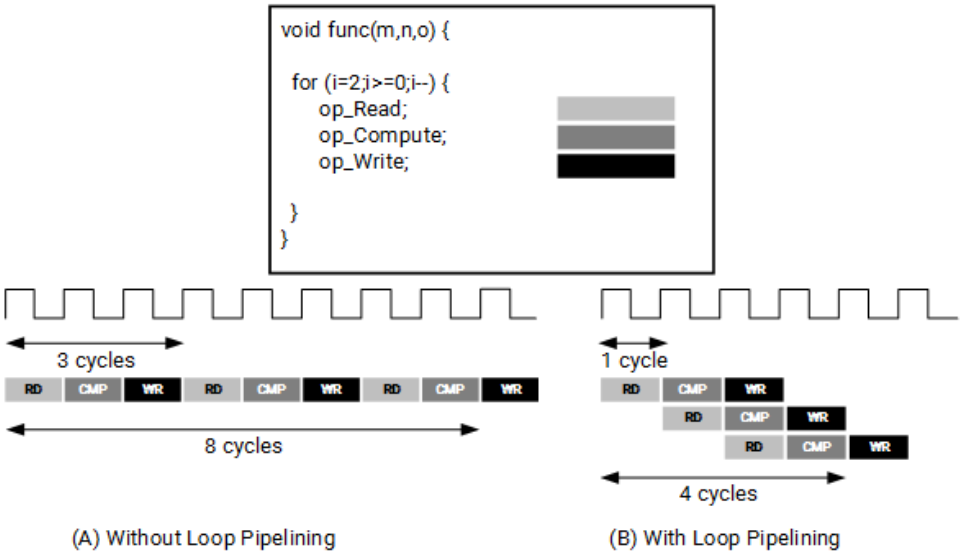


Figura 8: Esquema funcionamiento directiva Pipeline [9]

- **Dependence**: Directiva que detecta información relativa a la dependencia de datos dentro de un bucle o entre diferentes iteraciones de un bucle. Estas dependencias son de gran relevancia cuando las operaciones pueden ser programadas, por ejemplo, cuando un mismo elemento es accedido en la misma iteración o cuando un mismo elemento es accedido en una iteración de bucle diferente.

- **Dataflow**: Directiva que permite la concurrencia de tareas, donde puede ser empleada en bucles y funciones para sobreponer distintas operaciones. Mediante el incremento de la concurrencia en nuestro diseño se mejorará el rendimiento del mismo siempre que la dependencia de datos lo permita, ya que hay elementos en nuestro bucle/función que necesitan finalizar sus operaciones de lectura/escritura antes de que se haga otro tipo de operación sobre aquellos elementos leídos o escritos. A diferencia de ***Pipeline***, Dataflow permite que las operaciones de un bucle/función empiecen antes de que la anterior función o bucle haya completado todas sus operaciones.

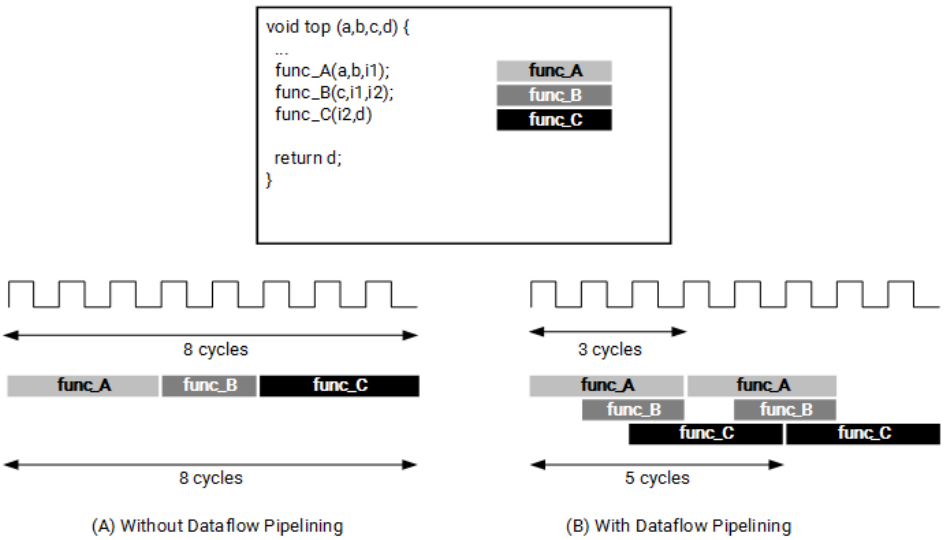


Figura 9: Esquema funcionamiento directiva Dataflow [10]

- **Resource:** Directiva que indica que una librería específica es usada para implementar una variable concreta en el diseño.

- **Inline:** Directiva que permite separar una función como entidad en la arquitectura de nuestro diseño, de forma que si disponemos de varias entidades en el mismo nivel de arquitectura, las operaciones entre ellas pueden ser más sencillas, optimizando así el sistema.

- **Allocation:** Directiva que tiene como objetivo definir y limitar el uso de recursos utilizados o elementos hardware en funciones, operaciones o bucles. Por ejemplo, cuando usamos esta directiva, los elementos se mapean de forma que con llamarlos una vez será suficiente para implementar esas instancias en vez de tener que hacerlo cada vez que ejecutamos esa función, bucle u operación. De esta manera, optimizaremos los recursos hardware.

- **Occurrence:** Indica que una región del código es ejecutada menos frecuentemente que el resto del código en una función o bucle. Esto permite emplear con una menor frecuencia la concurrencia o pipeline de esa parte del código menos usada con la demás, restando prioridad sobre esta parte del código para dársela a esas partes más ejecutadas.

- **Array_map:** Directiva que combina diferentes pequeños arrays para crear un único array más largo, reduciendo así los bloques BRAM del sistema.

- **Array_partition:** Directiva que divide un array en arrays más pequeños o incluso en elementos individuales. Esto requiere más bloques de memoria, requiere más cantidad de puertos de lecturas y escrituras, aunque mejora el rendimiento del sistema. A diferencia de la directiva anterior, consume más recursos hardware pero mejora la latencia del sistema.

- **Interface:** En nuestros diseños RTL disponemos diferentes entradas y salidas las cuales se representan mediante puertos, los cuales suelen trabajar protocolos. Esta directiva indica como nuestros puertos RTL son creados

durante la síntesis de nuestro diseño. Los puertos de nuestro diseño están especificados en los argumentos de la función top, así como en variables globales o en algún protocolo de una función, de modo que estos puertos pueden ser creados bajo una interfaz seleccionada, añadiendo a este puerto diferentes señales de control añadidas al mismo. Las distintas interfaces que podemos emplear sobre nuestros puertos son:

- **ap_none:** Interfaz que no especifica ningún protocolo, puerto de datos únicamente.

- **ap_stable:** Interfaz que no especifica ningún protocolo al igual que ap_none pero HLS asume que los datos siempre serán estables después de un reset. Esto optimiza el diseño ya que elimina registros inncesarios.

- **ap_vld:** Interfaz de datos a la que se le añade una señal de validación de datos, la cual indica cuando los datos son validos para leer/escribir.

- **ap_ack:** Interfaz de datos a la que se le añade una señal de confirmación, la cual indica cuando los datos han sido leídos/escritos.

- **ap_hs:** Interfaz que combina las dos anteriores, tanto la señal de confirmación como la señal de validación.

- **ap_ovld**: Interfaz similar a "ap_vld" pero para puertos de salida de datos.

- **ap_fifo:** Interfaz que implementa un FIFO clásico sobre el puerto deseado, añade las señales "empty" y "full" para controlar el flujo de datos del mismo.

- **ap_bus:** Interfaz que genera puertos de escritura/lectura sobre el propio bus de datos.

- **ap_memory:** Interfaz que implementa arrays de datos como interfaces RAM.

- **bram:** Interfaz similar a la anterior con la diferencia de que al usar el bloque IP sobre Vivado™, las interfaces de memoria aparecerán como puertos únicos.

- **axis:** Interfaz que genera todos los puertos bajo el protocolo AXI4-Stream.

- **s_axilite:** Interfaz que genera todos los puertos bajo el protocolo AXI4-Lite.

- **m_axi:** Interfaz que genera todos los puertos bajo el protocolo AXI4.

    - **ap_ctrl_none:** Interfaz que asegura la no generación de puertos "ap" de control.

    - **ap_ctrl_hs:** Interfaz que genera puertos de control como son: start, idle, done y ready.

    - **ap_ctrl_chain:** Interfaz que genera puertos de control como son: start, continue, idle, done y ready.

- **Unroll:** Esta directiva desenrolla los bucles de nuestro diseño, de forma que genera distintas operaciones independientes en vez de una única lista de operaciones. Esto permite que muchas iteraciones del bucle en cuestión ocurran en paralelo, mejorando así el rendimiento del sistema a costa del uso de más recursos hardware una vez más.

- **Data_pack:** Con esta directiva podemos "empaquetar" una serie de datos de una estructura en un único array o vector con un mayor ancho de palabra. De forma que el uso de memoria se verá reducido.

- **Latency:** Directiva con la cual indicamos el número máximo o mínimo de latencia que queremos en nuestro diseño para funciones, bucles o regiones, es decir, especificar el número máximo o mínimo de ciclos de reloj requeridos para generar una salida. En caso de que la latencia no satisfaga las especificaciones de nuestra directiva Vivado HLS™ intentará llegar a el valor deseado, si aún así no es posible, nos avisará mediante un "warning" y generará el diseño con mejor valor de latencia posible.

- **Stream:** Directiva que implementa arrays de nuestro diseño como RAMs, de forma que también puedan ser implementados como bloques FIFOs, esta directiva puede ser empleada para generar bloques secuenciales de datos creando un sistema de streaming de datos.

Después de tener nuestro código sintetizado con las directivas de optimización deseada y correcta pasaremos a simular el código sintetizado en lenguaje VHDL, donde podemos validar el funcionamiento del sistema ya a nivel RTL. Posteriormente, generaremos el proceso que inicie la creación del núcleo IP para posteriormente poder exportarlo. Al importar nuestro bloque IP en Vivado™ veremos cómo dispone de los puertos con las interfaces seleccionadas así como una señal de reloj y otra de reset añadidas y necesarias al bloque.

2.4 Protocolo AXI (Advanced eXtensible Interface)

Todo sistema de comunicaciones digital es regido por un protocolo, en nuestro caso se tratará del protocolo AXI. Se define como un protocolo en chip desarrollado por ARM® como parte de AMBA (Advanced Microcontroller Bus Architecture). Es el protocolo utilizado a la hora de comunicar los núcleos IP que alberga una FPGA, además de resultar muy útil a la hora de realizar transmisiones de datos en paralelo, funcionar en sistemas de baja latencia o proporcionar flexibilidad en la implementación de arquitecturas interconectadas.

Para entender el funcionamiento de este protocolo, debemos primero definir su arquitectura, conociendo sus canales de comunicación y sus métodos de transmisión/recepción de datos.

Se trata de un protocolo bidireccional con 2 interfaces, la interfaz maestra (master) y la interfaz esclava (slave), esto significa que la interfaz esclava transferirá las peticiones de lectura que haga la interfaz maestra cuando estemos en modo lectura, así como cuando estemos en modo escritura el maestro transferirá los datos a escribir al esclavo, recibiendo una confirmación de escritura por parte del esclavo a la interfaz maestra.

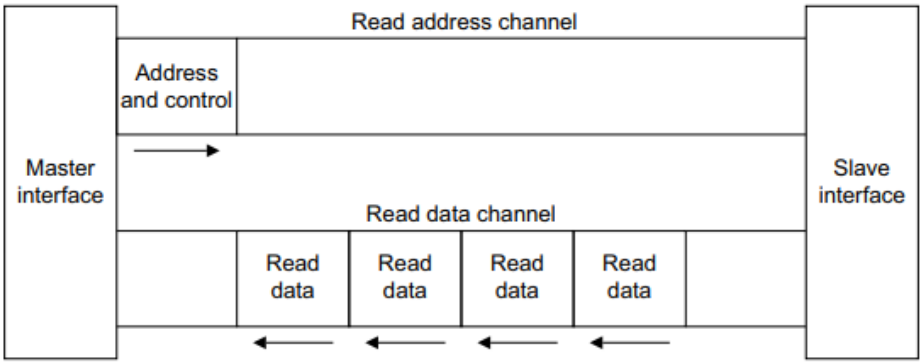


Figura 10: lectura entre maestro y esclavo protocolo AXI [11]

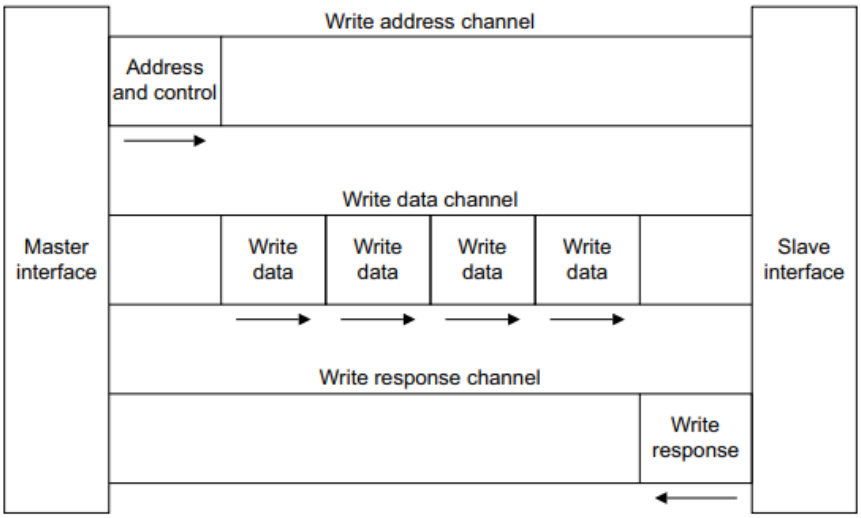


Figura 11: escritura entre maestro y esclavo protocolo AXI [11]

Como podemos ver en las figuras, disponemos de 5 canales, “read address”, “read data”, “write address”, “write data” y “write response”. Los canales de dirección (address) contienen la información relevante sobre los datos a transferir, después se escribe/lee los datos correspondientes y se envía una confirmación en el caso de la escritura por parte del esclavo.

Visto el esquema genérico de funcionamiento del protocolo pasaremos a ver las señales reales que ejecutan y controlan las transmisiones/recepciones de datos. Cada uno de los 5 canales visto antes dispone de señales de datos y de las señales "**READY**" y "**VALID**" principalmente. Las dos últimas mencionas son de gran relevancia ya que serán las señales que controlen el flujo de datos, dando la señal que de permiso a la transmisión de datos así como la validación y confirmación de si un flujo de datos enviado es correcto o no. Además, disponemos de otras señales de información adicional como la señal "LAST", la cual informa sobre cuál ha sido el último paquete de datos enviado.

La importancia de las señales READY y VALID es tal debido a que entre las dos se creará el proceso de "**apretón de manos**" o handshake, en el cual se informará de cuando tanto esclavo como maestro están listos para reciben los paquetes de datos mediante la señal READY y también, se confirmará la validez de esos mismos paquetes mediante la señal VALID.

De forma que, como vemos en la figura, los datos serán transmitidos únicamente cuando ambas señales estén a nivel alto. La fuente genera la señal VALID para indicar que la información está disponible y el destinatario genera la señal READY para indicar que puede aceptar la información. Siguiendo el cronograma, la señal VALID estará a nivel alto en el flanco T2 pero al no estarlo la señal READY, la información será transmitida en el momento que se llega al flanco T3, todo regido por una señal de reloj, ACLK en este caso.

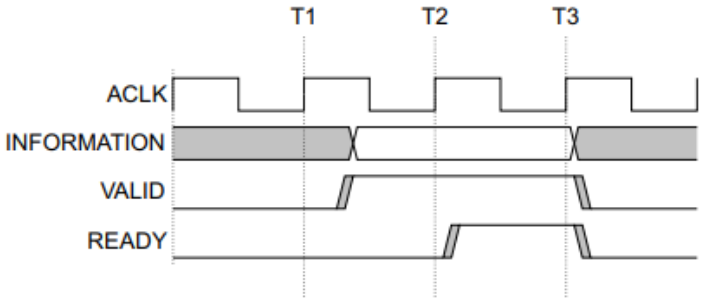


Figura 12: proceso de handshake AXI [11]

Cabe destacar, que AXI fue creado con el objetivo de disponer de un protocolo que funcionara mediante ráfagas, utilizando direcciones y una fase de control previa al intercambio de datos, pero que fuese capaz de realizar transferencias fuera de servicio o transferencias de datos no alineadas, todo a través de una interfaz de baja potencia.

Dentro del protocolo AXI versión 4 (AXI4), tenemos otras 2 sub-interfaces las cuales son: AXI4-Lite y AXI4-Stream. Donde AXI4-Lite está pensada para sistemas de comunicaciones simples sin capacidad de enviar ráfagas de datos, además de tener una interfaz más sencilla que AXI4. Mientras que AXI4-Stream se creó para trabajar con sistemas de datos de alta velocidad donde los mismos sean transmitidos de manera secuencial y unidireccional, en forma de flujos de datos (streaming). [11]

2.5 AXI4-Stream

Como hemos mencionado en el apartado anterior, uno de las sub-interfaces o sub-protocolos de AXI4 es AXI4-Stream. Esta particularidad de AXI4 tiene bastantes diferencias en cuanto a modo de funcionamiento se refiere con AXI4 o AXI4-Lite, las cuales pasaremos a comentar a continuación.

Cuando hablamos de "streaming" en procesado de datos, hablamos de transmitir los datos de forma unidireccional, secuencial y con un modelo basado en FIFOs, de modo que un dispositivo pueda enviar datos a otro de esta manera. AXI4-Stream es justo esto mismo, comparte las mismas señales de comunicación y datos mencionadas en el apartado anterior (READY, VALID, LAST y DATA) pero con la diferencia de que la comunicación entre maestro y esclavo es en una única dirección, el maestro envía los datos y el esclavo los recibe. La transmisión de datos en este caso solo se realizará cuando el esclavo esté listo para recibirlos (tready = '1') y cuando los datos a enviar del maestro sean validos (tvalid = '1').

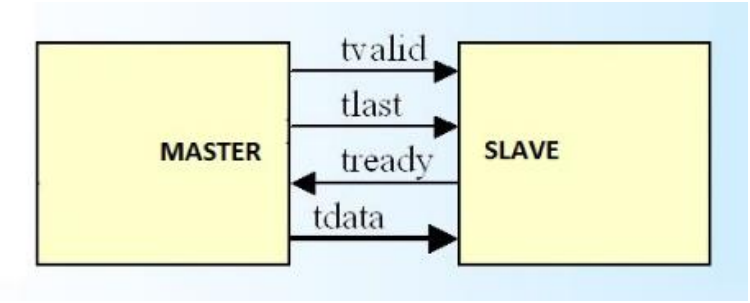


Figura 13: diagrama bloques AXI4-Stream [12]

Hablando de nuestro proyecto, este protocolo encaja perfectamente, ya que nuestro sistema está pensado para funcionar de manera secuencial con bloques consecutivos que vayan transfiriendo y recibiendo el flujo de información continuamente de manera unidireccional de bloque a bloque. Por lo que todas las interfaces de los diferentes dispositivos o bloques que albergue nuestro sistema deberán ser interfaces AXI4-Stream.

2.6 Codificación coma fija en FPGA

El tipo de codificación en un computador no debería de ser un problema para el programador ya que estos disponen de unidades aritméticas preparadas para cualquier tipo de codificación. Sin embargo, una FPGA explota características como el paralelismo o la gestión de datos a alta velocidad, es por ello que para estas aplicaciones conviene usar esta tecnología de dispositivos reconfigurables.

En primer lugar, debemos de saber qué tipo de datos tendremos en nuestro sistema, pueden ser datos enteros o datos reales. Los datos enteros son aquellos codificados en binario natural (con signo o sin él), es decir, aquellos datos sin parte fraccionaria. Los datos enteros no son problema para trabajar con HDL (High Description Lenguage), el problema viene cuando necesitamos de datos reales en aplicaciones.

Los datos reales son aquellos que poseen parte fraccionaria, estos pueden ser codificados en dos representaciones distintas: **coma fija** y **coma flotante**. La codificación en coma fija se diferencia de la coma flotante en que necesita menos recursos para su uso pero dispone de menor precisión que la coma flotante. La coma fija se caracterizada por tener un número fijo de bits tanto de parte entera como de parte decimal, además su uso es muy útil debido a que la mayoría de las FPGA disponen de un módulo dedicado a la codificación en coma flotante o FPU (Floating Point Unit), aparte de que suponen un aumento de la latencia del sistema así como de los recursos hardware necesarios para el funcionamiento.

La codificación en coma fija se basa en el uso de una coma fija decimal virtual la cual hará de separador entre la parte entera y la decimal, el donde situar esta coma y el número de bits empleados para cada parte es lo que determinará la **precisión** de un dato. En la siguiente figura observamos el paso de un número representado en coma fija a su valor decimal.

$$10101{,}110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} = 21{,}75$$

Figura 14: cálculo de valor decimal a partir de dato en coma fija [13]

En cuanto a la precisión, mencionada con anterioridad, debemos de saber que un dato puede producir "overflow" (desbordamiento), generados cuando el resultado de una operación tiene más bits que sus operandos, perdiendo información. El redondeo o truncamiento serán la solución a las pérdidas de bits, donde el redondeo consiste en la aproximación del valor de un dato a su valor más cercano, produciendo un **error** a tener en consideración más adelante.

Para decidir el número de bits para la parte entera y la parte decimal y determinar así la precisión debemos de saber el margen de valores con el que vamos a estar trabajando. De modo que en nuestro caso, trabajaremos con archivos de audio ".wav", los cuales contienen número decimales entre el 1 y el -1. Otro factor a tener en cuenta para cuando trabajemos con el protocolo AXI es que en este protocolo los buses de datos deben de tener un ancho equivalente a cualquier potencia de 2 (1, 2, 4, 8, 16, 32…) por lo que seleccionaremos uno de estos anchos y calcularemos el error en MATLAB®, seleccionando finalmente el más óptimo.

Con estas premisas pasaremos a calcular el error obtenido entre el dato original (coma flotante) y el codificado en coma fija con nuestra precisión elegida. Para ello, debemos de conocer cómo trabaja MATLAB® en coma fija. Disponemos de la función "***fi***" la cual nos convierte una serie de datos de entrada en datos codificados en coma fija con una serie de parámetros seleccionados en la función como podemos ver en la figura.

```
>> a = fi(5.7489, 1, 14, 10)

a =

    5.7490

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 14
        FractionLength: 10
```


Figura 15: Ejemplo de uso función *fi* en MATLAB®

En el ejemplo vemos como el número 5,7489 se pasará a dato "fi" con signo (1), con una longitud de 14 bits para parte decimal y entera (14) y con 10 bits de parte decimal (10), es decir, dejaremos 4 bits para la parte entera.

El error producido entre el caso ideal (coma flotante) y nuestros datos en coma fija se podrá calcular tanto mediante el error máximo (**e_{max}**) o el error cuadrático medio (**e_{sqrt}**) proporcionados mediante las siguientes fórmulas:

$$error_i(\%) = \left|V_i\right|_{Fixed\_Point} - \left|V_i\right|_{Floating\_Point} \times 100$$

Figura 16: fórmula error coma fija vs coma flotante [13]

$$e_{MAX}(\%) = \max(error_i(\%))$$

Figura 17: fórmula error máximo [13]

$$e_{SQRT}(\%) = \sqrt{\frac{\sum_{i=0}^{k}\left(error_i(\%)\right)^2}{k}}$$

Figura 18: fórmula error cuadrático medio [13]

Dadas las características de nuestros datos obtenidos de ficheros ".wav" mencionadas anteriormente, se realizará un cálculo de errores (en el capítulo 3) según el ancho de palabra en MATLAB®, donde se decidirá finalmente que precisión y cuantía de bits usar para cada parte, escogiendo la más óptima.

Habiendo visto la implementación de la coma fija en MATLAB®, pasaremos a verla el software de Xilinx™, Vivado HLS™. A diferencia de HLS, en Vivado™ se trabaja únicamente con datos binarios por lo que tendremos que plantear la codificación coma fija únicamente en HLS.

Vivado HLS™ soporta el uso de la coma fija únicamente en ficheros C++, además, disponemos de la librería "ap_fixed.h" para trabajar con este tipo de dato. Las declaraciones de variables en coma fija disponen de una serie de parámetros para concretar el ancho total del dato, los bits para la parte entera o incluso los modos de redondeo y overflow en los que actuará el software en caso de sucederse uno de estos dos casos.

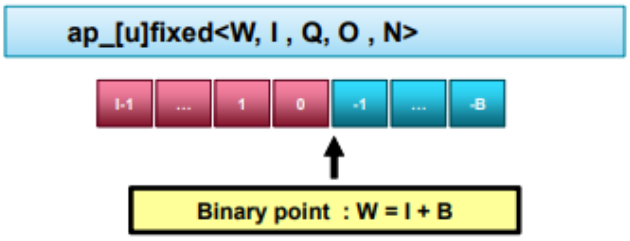


Figura 19: Estructura dato coma fija en Vivado HLS™ [14]

En la figura se ve la estructura de un dato "fixed point" en Vivado HLS™,

donde [14]:

- W: Longitud del dato en bits
- I: Número de bits para la parte entera
- Q: modo de redondeo
- O: modo de saturación
- N: Número de bits de saturación en modos “wrap”

El modo de redondeo indica cual será el tipo de redondeo a realizar en caso de que una operación genere más bits de precisión, en la parte menos significativa del dato, de los que el dato puede almacenar. Son los siguientes:

- AP_RND: Redondeo hacia +infinito
- AP_RND_ZERO: Redondeo hacia 0
- AP_RND_MIN_INF: Redondeo hacia -infinito
- AP_RND_INF: Redondeo hacia infinito
- AP_RND_CONV: Redondeo convergente
- AP_TRN: Truncamiento a -infinito
- AP_TRN_ZERO: Truncamiento a 0 (modo por defecto)

El modo “overflow” indica cual será el comportamiento cuando se requieren más bits de los que el dato contiene, es decir, un desbordamiento. Son los siguientes:

- AP_SAT: Saturación. Se convierte el valor en el máximo disponible por los bits dados, o mínimo en caso de underflow.
- AP_SAT_ZERO: Saturación a 0: Dato convertido a 0 si el resultado desborda
- AP_SAT_SYM: Saturación simétrica: Dato convertido a valor máximo en caso de overflow positivo o dato convertido en –máximo en caso de overflow negativo.
- AP_WRAP: El dato es tratado como una secuencia continua circular
- AP_WRAP_SM: (N=0): Usa signo-magnitud wrapping, si el bit más significativo es diferente del original, los restantes serán invertidos, si no, este bit será copiado hacia atrás.
- AP_WRAP_SM: (N>0): Los N bits más significativos serán convertidos a 1.

A la hora de trabajar con este tipo de datos, Vivado HLS™ nos proporciona una librería con operaciones matemáticas soportadas para coma fija llamada “hls_math.h”, la cual incluye gran variedad de funciones matemáticas, aunque puede faltar alguna.

3. Algoritmia y desarrollo de bloques

Para continuar con este capítulo 3 es fundamental conocer los puntos estudiados en el capítulo 2 previamente, donde hemos estudiado una serie de apartados de forma teórica necesarios para el desarrollo del proyecto, como son los protocolos que usaremos en el sistema, la tecnología en la que está basada el sistema, el software empleado para elaborar partes del proyecto o la codificación en la que se basarán nuestros datos numéricos.

En este capítulo 3 entraremos en el detalle de cómo se va a elaborar realmente este sistema de procesado digital de audio implementado en FPGA. Al ser un sistema de bloques o algoritmos, iremos estudiando bloque a bloque del sistema, viendo el fundamento teórico de cada uno, así como su funcionalidad y posteriormente, su implementación paso a paso hasta disponer del bloque final. Previo al estudio de bloques, se tendrán en cuenta una serie de consideraciones iniciales sobre el sistema, en relación con las características del mismo o del tipo de dato a usar.

Las bases del funcionamiento de nuestro sistema de procesado de datos serán las siguientes:

- Lectura y escritura de muestras de datos sobre ficheros ".txt" llamados "datasets".
- Transmisión de muestras de forma unidireccional y secuencial según el protocolo AXI4-Stream, así como el uso de señales de validación empleadas en la transmisión de datos.
- Uso de la herramienta MATLAB® de forma paralela como modo de validación de funcionamiento del sistema, calculo de errores o cálculo de coeficientes de nuestro filtro central.
- Uso de la herramienta Vivado™ para la obtención de bloques AXI4-Stream, nativos de este mismo software, necesarios para poder implementar nuestro sistema de forma correcta.
- Uso de la herramienta Vivado HLS™ para la generación de bloques IP basados en algoritmos de procesado de audio mediante todas las ventajas que tenemos al trabajar con HLS y con este software en concreto.

En el capítulo 1 se expuso un esquema general del sistema completo, a continuación, en este capítulo, se detallará más, concretando los bloques que engloban el sistema, de forma que más adelante se estudiará paso a paso cada uno de ellos, divididos en apartados independientes.

Como podemos observar en la figura 20, se expande el esquema visto en el capítulo 1, indicando el nombre de los algoritmos de audios empleados en cada etapa del sistema.

Inicialmente, al sistema se le es entregado un fichero ".txt" de datos binarios (características de datos mostradas en siguiente apartado), donde línea a línea tendremos las muestras obtenidas de nuestro audio de entrada ".wav", esto será llamado: "Dataset de entrada" o "Dataset In". Cabe destacar, que al tratarse de un sistema basado en FPGA, las muestras o los datos a manejar deben de tratarse de datos binarios o enteros debido a la naturaleza de la misma tecnología.

Seguidamente, tenemos el primer bloque en la etapa de pre-procesado de audio, adquiere el nombre de "**DRCompressor**", este viene de Dynamic Range Compressor, es decir, se trata de un compresor de rango dinámico de audio. Se trata de un interesante algoritmo el cual tratará con los valores de la señal de audio, de forma que según los parámetros del compresor, modificaremos el rango dinámico de la señal dada, previniendo posibles saturaciones de sonido de forma controlada, visto en parte desde la perspectiva de los decibelios y los ejes logarítmicos.

A continuación, nos encontramos con la etapa central de procesado de audio, la etapa de filtrado. Toda señal de audio o señal en general puede llevar consigo acoplado algún tipo de ruido, por lo que la misión en este caso, será la eliminación total o parcial de este acoplamiento. En nuestro caso, los audios que pasarán por nuestro sistema provienen de la misma fuente imaginaria, la cual acopla el mismo, o similar, ruido a las señales de audio que entran al sistema.

En próximos apartados de este capítulo, se estudiarán los dos posibles caso de filtros, el filtro **FIR** (filtro de respuesta finita al impulso) o el filtro **IIR** (filtro de respuesta infinita al impulso). Se realizará una comparación entre ambos, mostrando las ventajas y desventajas que nos ofrecen y finalmente se decidirá el filtro final a usar. En esta etapa será fundamental el uso de MATLAB® debido a que ofrece aplicaciones de diseño de filtros, que nos resultarán útiles a la hora de generar nuestros coeficientes en forma de fichero ".coe" o en forma de array numéricos, todo según el tipo de filtro que hayamos diseñado, paso bajo, paso alto, paso todo, paso banda, banda eliminada…

La última etapa del sistema se tratará de un banco de efectos de audio, en ella disponemos de 4 diferentes efectos como podemos ver en la figura. Cada efecto se basa en su algoritmo de funcionamiento, diferenciando varios tipos de efectos entre ellos, los cuales debido a su naturaleza modificarán el audio de una forma u otra, son los siguientes:

- **Echo**: Se trata de un efecto de audio basado en retardos o "delays", de forma que genera un efecto de eco en el audio mediante el retardo de la señal de entrada.

- **Reverb**: Consiste en un efecto de audio basado también en retardos, en este caso obtendremos un efecto de reverberación en la señal, dejando finalmente una cola de audio que irá decreciendo hasta tener el silencio absoluto. Hará uso de la señal de entrada retardada, así como de la señal de salida realimentada, la cual generará esa "cola" de sonido.

- **Wah-Wah**: Es un efecto basado en los pedales de guitarra Wah-Wah, en este caso es un efecto basado en filtros, de forma que dispondremos de un filtro paso todo al cual variaremos su frecuencia de corte mediante un oscilador de baja frecuencia (LFO), esto generará un efecto sonoro el cual hace referencia al nombre del propio efecto sobre la señal de audio.

- **Trémolo**: Basado también en efectos de guitarra eléctrica. Se trata de un efecto basado en la modulación de la amplitud de la señal de audio. La amplitud de la señal se verá afectada por una señal moduladora mediante un LFO generando un efecto de sonido interesante.

Cabe destacar, que los efectos “Echo” y ”Reverb” al estar basados en efectos de retardos, van a necesitar del uso de memorias externas al tener que guardar las diferentes señales que usan los mismos de manera retardada. En sus apartados dedicados se explicará el método para ello.

El sistema constará de dos señales de selección, permitiendo al usuario activar/desactivar las 4 salidas individuales de los efectos (bloques morados) y activar/desactivar 3 salidas de **efectos combinados** (bloques amarillos), como vemos en la figura. Finalmente, el sistema podrá entregar hasta 7 diferentes datasets de salida en forma de ficheros “.txt”, los cuales contendrán las muestras finales de salida al haber sido procesadas por todas las etapas del sistema.

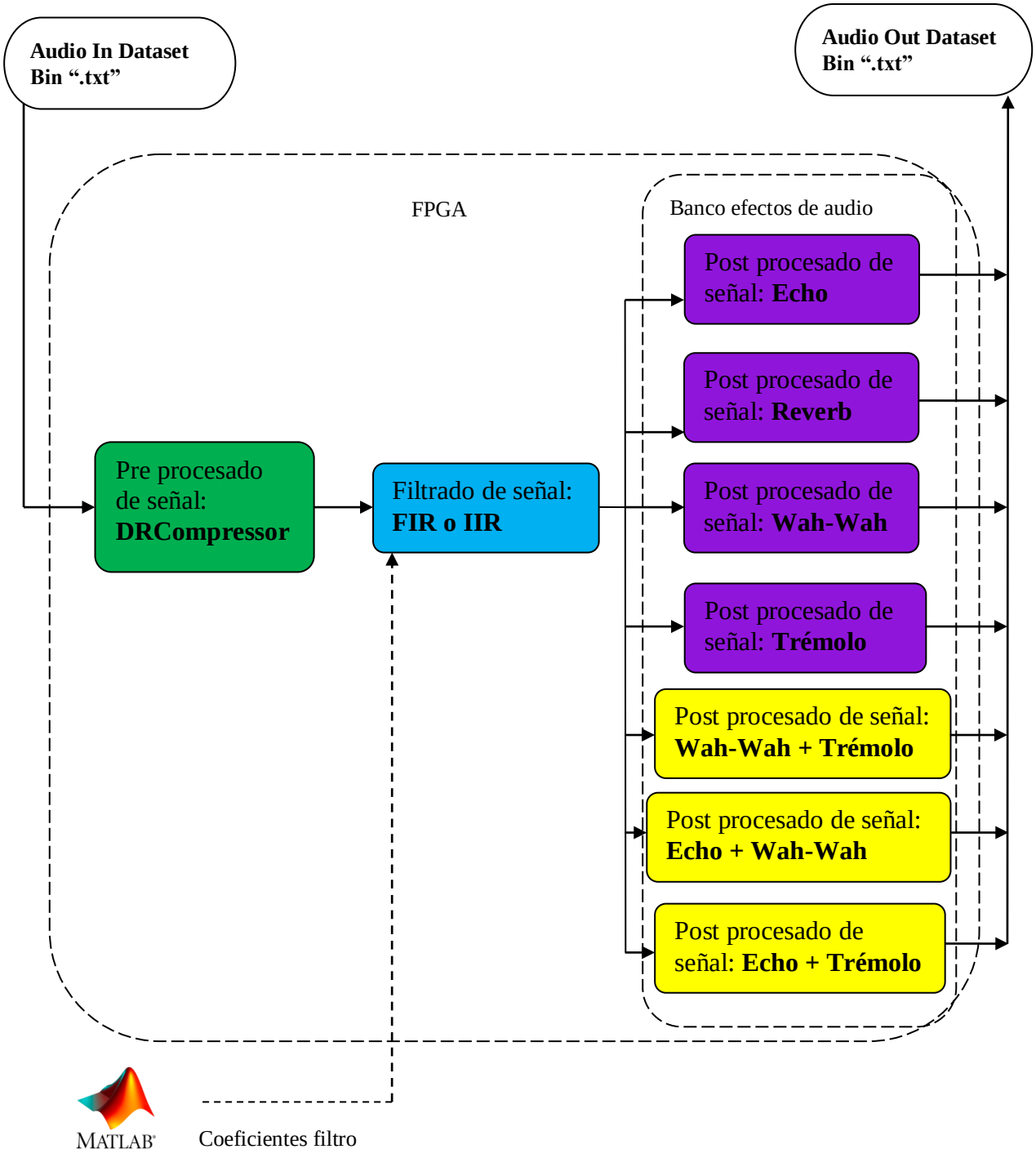


Figura 20: Esquema extendido del sistema global

3.1 Consideraciones previas

Antes de pasar a analizar y explicar cada bloque de nuestro sistema, se detallarán una serie de bases o características del mismo las cuales nos sirvan como estándares previos a la realización de estos bloques. Tales como la precisión a usar en nuestros datos, parámetros generales o el formato de los archivos empleados.

3.1.1 Características de los archivos ".wav"

Nuestro sistema de procesado digital de datos está basado en el audio, esto significa que tratará la señal de audio entrante al sistema por medio de los distintos algoritmos que albergan el mismo.

Los sistemas digitales de procesado de señales, tratan las muestras o "samples" de aquellas señales entrantes al sistema, de modo que cada una de las muestras de la señal se corresponde con el valor digital de la señal en ese determinado instante, de modo que disponiendo de la totalidad de muestras de una señal podemos construirla mediante nuestras muestras y sus momentos en el tiempo. A su vez, el número de muestras de una señal determinará la precisión de la misma, de forma que cuanto mayor sea la frecuencia de muestreo, mayor número de muestras tendremos y por lo tanto, mayor precisión a la hora de representar dicha señal.

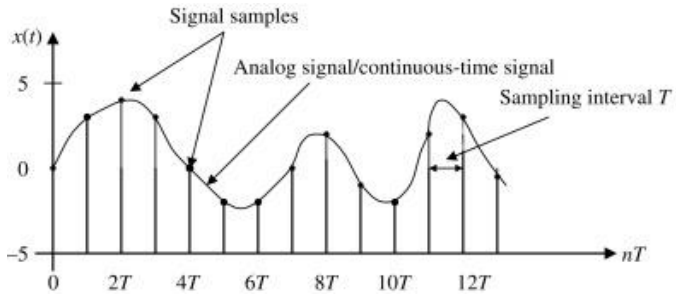


Figura 21: Señal analógica "sampleada" [15]

Una vez conocido como nuestro sistema trabajará con "samples", debemos de saber, como obtendremos nuestros "samples". A día de hoy, se conocen diversos tipos de archivos de audio, ".mp3", ".mp4", ".flac", ".alac", ".wav" … Cada uno de los formatos mencionados tienen unas características que los definen, por ejemplo, los archivos ".mp3" y ".mp4" son archivos de audio con pérdidas de datos, así como ".flac" y ".alac" son archivos sin pérdidas de datos. Sin embargo, el archivo tipo ".wav" es un tipo de formato en el cual mantenemos el mismo tamaño y los mismos datos que el fichero original, es un formato descomprimido. Esto es perfecto para el tratado digital de audio ya que al disponer de los datos originales, no se pierde información y el audio puede ser tratado de la forma más veraz posible. Por lo tanto, el tipo de archivo a usar sobre nuestro sistema serán los ".wav".

A la hora de conocer un poco más los archivos ".wav" debemos de saber sus características más importantes. En primer lugar, el nombre de ".wav" proviene de Waveform Audio File Format, además, tiene una arquitectura basada en una trama la cual tiene el formato observado en la siguiente figura, donde la misma dispondrá de su cabecera de información, donde se describen elementos relevantes al fichero de audio

en cuestión, seguido de los bytes de datos los cuales contienen todo el audio correspondiente al archivo.

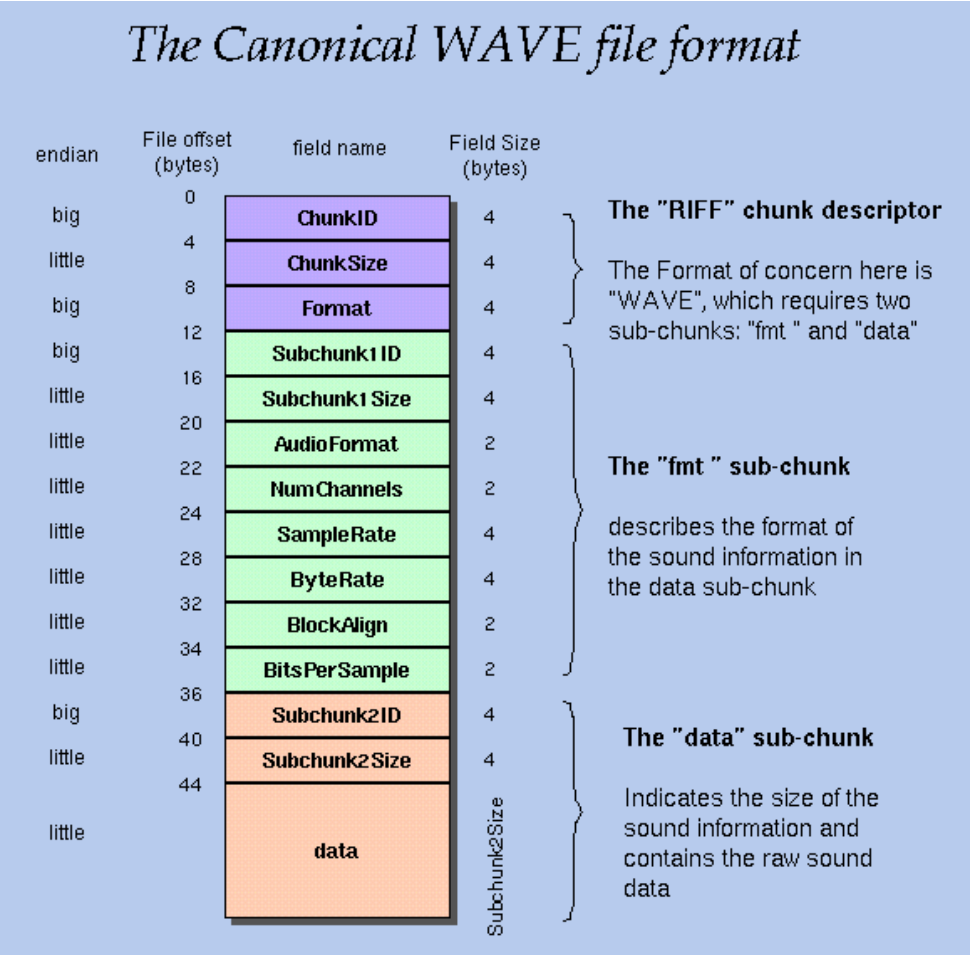


Figura 22: Estructura de un fichero ".wav" [16]

Uno de los parámetros más importantes a conocer de nuestro fichero será la **frecuencia de muestreo**, la cual será empleada en diferentes ocasiones en nuestro sistema como podremos ver más adelante.

La frecuencia de muestreo determina el número de muestras por segundo de un fichero de audio, dependiendo del archivo, esta suele ser 22050 Hz o 44100 Hz, aunque puede variar desde 1 Hz hasta los 4,3 Ghz, donde la más usada y la estandar en CDs de audio, por ejemplo, suelen ser los 44,1 kHz.

3.1.2 Parámetros generales

En nuestro caso, se seleccionó un audio de prueba para testear todo el sistema y probar todos sus bloques con el mismo llamado "**audio_in.wav**". Se trata de un audio con sonidos electrónicos con una duración perfecta, para comprobar todos los algoritmos de audio del sistema de una forma amplia y que no nos deje con ningún tipo de duda, de aproximadamente 6 segundos el cual dispone de una frecuencia de muestreo de **22050 Hz**. Esta frecuencia de muestreo será con la que configuraremos todo nuestro

sistema cada vez que sea necesario el uso de la misma.

Para saber el valor de datos como la frecuencia de muestro debemos de hacer uso de MATLAB®, el cual nos brinda la opción de usar la función "audioread", función que nos devuelve la totalidad de las muestras del archivo en formato coma flotante, así como el valor de la frecuencia de muestreo.

```
[x , Fs] = audioread('audio_in.wav');
```

Figura 23: lectura audio de entrada en MATLAB®

Con nuestro audio de prueba, obtenemos **131072 muestras** con una frecuencia de muestreo de **22050 Hz**, lo que hace un total de **5,94 segundos** de audio.

El uso real del archivo ".wav" de entrada acaba aquí, debido a que esto es únicamente realizable en MATLAB®, a la hora de trabajar con Vivado HLS y Vivado usaremos ficheros de entrada ".txt" los cuales contendrán las muestras de audio, todo ello debido a la imposibilidad de emular la función "audioread" fuera de MATLAB®.

3.1.3 Determinación de la precisión y errores

En el apartado 2.6 hablamos de las diferencias entre la codificación coma fija contra la codificación coma flotante, dándonos como resultado ganador, para el uso en nuestro sistema, a la coma fija debido al gran aumento de recursos hardware y latencia que implica usar coma flotante. Además se detalló cómo se codifica en coma fija, sus características y como se usa en Vivado HLS™.

Este apartado es una continuación a aquel apartado 2.6 debido a que también se habló en el mismo de los errores. Al disponer de un audio ".wav", donde es leído en MATLAB® y del cual se obtiene una serie de muestras en coma flotante, debemos de transformar esas muestras en muestras coma fija con las cuales trabajarán los bloques del sistema. Estas muestras en coma fija pueden contener errores si las comparamos con las muestras originales en coma flotante, lo que significa que dependiendo de la precisión que elijamos, para nuestras nuevas muestras, tendremos un error más grande o más pequeño.

Para ello, usaremos MATLAB®, donde leeremos las muestras de nuestro fichero de entrada, obteniendo así el "**golden data**" y a continuación obtendremos los arrays de muestras en coma fija según una precisión dada mediante el empleo de la función "*fi*".

Como ya mencionamos en el apartado 2.6, el número de bits para la parte entera será de 2, esto es debido a que los archivos ".wav" poseen valores entre ±1, por lo que debemos dejar 1 bit para el signo y otro para ese posible valor de 1.

Para finalmente calcular los errores proporcionados por las diferentes selecciones de precisión para nuestro array de muestras de entrada emplearemos las funciones mencionadas en el apartado 2.6, con ellas podemos calcular el valor del **error** máximo o el **error cuadrático medio**. Además, como ya se comentó con anterioridad, deben de ser datos con una longitud de palabra en potencias de 2, por lo que calcularemos la precisión para datos de 4, 8, 16, 32 y 64 bits, siempre dejando 2 bits

para la parte entera como podemos ver en la siguiente figura. Además se realizarán los cálculos con los tipos de errores posibles.

```
x_fi4 = fi(x, 1, 4, 2);
x_fi8 = fi(x, 1, 8, 6);
x_fi16 = fi(x, 1, 16, 14);
x_fi32 = fi(x, 1, 32, 30);
x_fi64 = fi(x, 1, 64, 62);
```

Figura 24: generación arrays en coma fija de diferentes longitudes

Una vez disponemos de los arrays de muestras obtenidos mediante la función "*fi*" pasaremos a calcular los errores para cada uno de los 5 arrays, de forma que emplearemos las fórmulas vistas en el apartado 2.6.

```
error4(j) = (abs(x_fi4(j))-abs(x(j)))*100;
errorSUM4 = errorSUM4 + (error4(j))^2;


errorSQRT4 = sqrt(errorSUM4/length(x));


errorMAX4 = max(abs(error4));
```

Figura 25: cálculo de distintos errores entre datos coma fija y coma flotante

Mediante estas sentencias calcularemos los errores para todas las precisiones expuestas, dando como lugar a las siguientes gráficas, donde podemos comparar el error obtenido según el tamaño de la palabra dado.

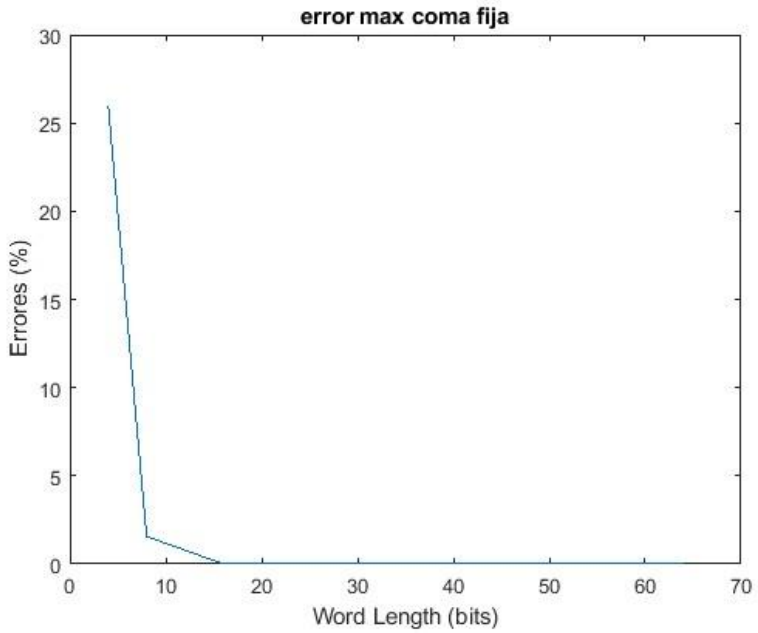


Figura 26: gráfica del error máximo dependiendo de la longitud de bits

Como podemos ver, a partir de los 8 bits de ancho de palabra, el error es despreciable al estar por debajo del 1%. Esto hace que nos descantemos por un **ancho de palabra de 16 bits** con 14 bits para la parte decimal, ya que no será necesario coger un mayor ancho de palabra al tener un error tan bajo con 16 bits, seleccionar más de 16 bits sería emplear recursos de memoria en vano.

3.2 Bloque DRCompressor

3.2.1 Introducción

A continuación, se expondrá uno a uno los bloques que emplearemos y que formarán nuestro sistema de procesado digital de audio.

En primer lugar, nos encontraremos con el primer bloque del sistema, el cual será donde el audio original de entrada sufra las primeras modificaciones, el "DRCompressor". Este es el nombre que ha adoptado, ya que se trata de un compresor de rango dinámico orientado a audio, de modo que sus siglas en inglés son: DRC (**Dynamic Range Compressor**) [17].

Antes de explicar el funcionamiento e implementación del bloque, se debe introducir el concepto teórico base en este algoritmo como es el rango dinámico de una señal de audio:

- **Rango dinámico:** Se trata de la diferencia medida en dB entre el sonido de mayor intensidad y el de menor en una señal de audio. Donde una señal con mayor rango dinámico otorga mayor calidad de sonido, mayor realismo o menor saturación que una señal con un rango dinámico pequeño o malo [18].

Bajo esta premisa podemos pasar a definir nuestro bloque. En primer lugar, como dicen sus siglas, se trata de un bloque que comprimirá el rango dinámico de nuestra señal, aunque también se le puede conocer como un bloque controlador del rango dinámico como veremos más adelante.

La búsqueda de un control del rango dinámico en nuestro audio viene dada debido a querer obtener un audio que se encuentre regido a una serie de requisitos de sonido, ya que en el "mundo real" el disponer de señales de audio que carezcan de control alguno no es lo ideal, pues se pueden obtener señales saturadas o carentes de calidad de audio. Por ejemplo, el reproducir audio en un coche, un centro comercial o en un restaurante queremos que el rango dinámico del mismo coincida con el sonido del ambiente en el que nos encontremos. Por lo tanto, el control del volumen de nuestra señal es fundamental.

Controlar el rango dinámico de una señal se basa en amplificar nuestra señal de entrada con una ganancia la cual sea controlada según el nivel de sonido que contenga la propia señal de entrada. Mediante la detección de los niveles de audio de nuestra señal podremos controlar la misma, dando lugar a una señal de salida mejorada según los requisitos que hayamos impuesto sobre nuestro sistema.

Por lo tanto, nuestro objetivo será **emular un compresor de sonido**, con el cual

conseguiremos reducir el rango dinámico de la señal sin que se note su presencia excesivamente. Donde la señal de audio se vea reducida cuando esta supere un determinado umbral que marcaremos nosotros, con el objetivo de [19]:

- Evitar posibles saturaciones de audio o picos de sonido que puedan ser dañinos para nuestro sistema de audio así como para el oído humano.

- Obtener una compresión sutil para el oído humano, ya que este es muy sensible y se debe de conseguir que no sea capaz de captar nuestra compresión.

Para ello, debemos de emular el comportamiento de un compresor tradicional, mediante el uso de un algoritmo basado en un sistema de bloques que contenga todos los parámetros característicos de un compresor de audio. Nos basaremos en el siguiente algoritmo, donde en la figura podemos ver su diagrama de bloques. Se corresponde con un controlador de rango dinámico con parámetros y características que serán explicadas a continuación.

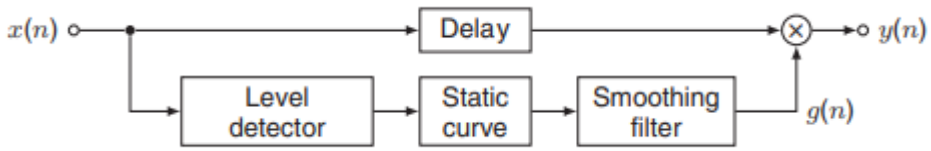


Figura 27: diagrama de bloques del compresor de audio [17]

Atendiendo al diagrama de bloques que representa nuestro algoritmo, podemos ver como es un sistema sencillo, de una única entrada y una única salida. Donde la señal de salida *y(n)* es el resultado de multiplicar nuestra señal de entrada *x(n)* retrasada una serie de muestras por una señal *g(n)*, la cual es el resultado de tratar la señal de entrada mediante una serie de bloques que veremos más adelante.

3.2.2 Parámetros del compresor

Antes de pasar a detallar el algoritmo, se deben de presentar una serie de parámetros correspondientes a un compresor de audio básico, los cuales emplearemos en nuestro algoritmo, son los siguientes [19]:

- **Umbral de Compresión o Compressor Threshold (CT):** Se trata del parámetro más importante, el cual determinará a partir de qué nivel de audio la señal pasará a ser comprimida.

- **Proporción o Ratio (R):** Se corresponde con el valor proporcional con el que se verá reducida nuestra señal, por ejemplo, una proporción 3:1 significa que por cada 3 dB nuestra señal aumentará 1 dB a la salida siempre que se vea superado del umbral.

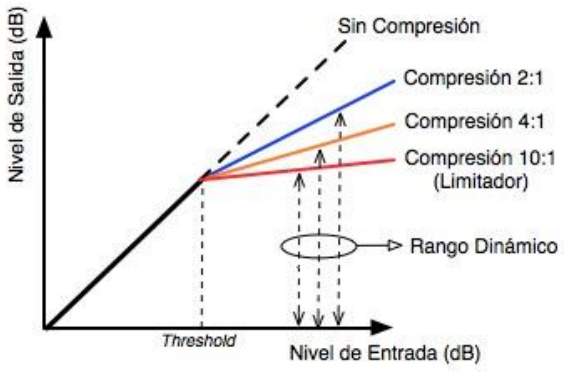


Figura 28: gráfica salida/entrada compresor para diferentes proporciones [20]

- **Pendiente o Slope (S):** Parámetro que define la pendiente de la curva de compresión cuando se supera el umbral, directamente relacionado con **R**: S = 1 – 1/R.

- **Tiempo de ataque o Attack Time (at):** Tiempo característico del compresor el cual corresponde al tiempo que se tarda desde que se supera el umbral hasta que se comprime la señal. Siendo un parámetro ajustable desde los 500 us a los 100 ms.

- **Tiempo de decaimiento o Release Time (rt):** Tiempo característico del compresor el cual corresponde al tiempo que se tarda desde que se deja de superar el umbral hasta que se deja de comprimir la señal. Siendo un parámetro ajustable desde los 100 ms a los 3 s.

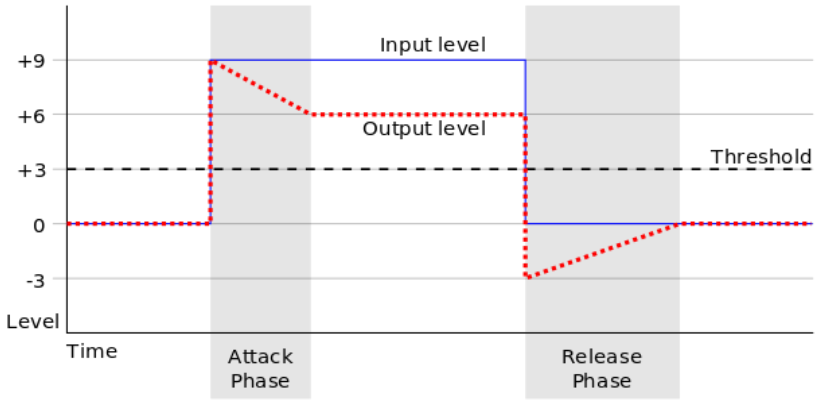


Figura 29: gráfica temporal entrada y salida compresor [19]

3.2.3 Bloques internos del compresor

Una vez vistos los parámetros generales del compresor así como su esquema general, pasaremos a entrar en detalle sobre el algoritmo. De forma, que regresando al esquema general de un DRC, veremos en detalle los bloques necesarios para obtener nuestra señal de salida *y(n)***.**

A continuación, se presenta una vista del mismo esquema visto antes pero con algo más de detalle, donde podemos ver las señales que se obtienen a la salida de cada bloque.

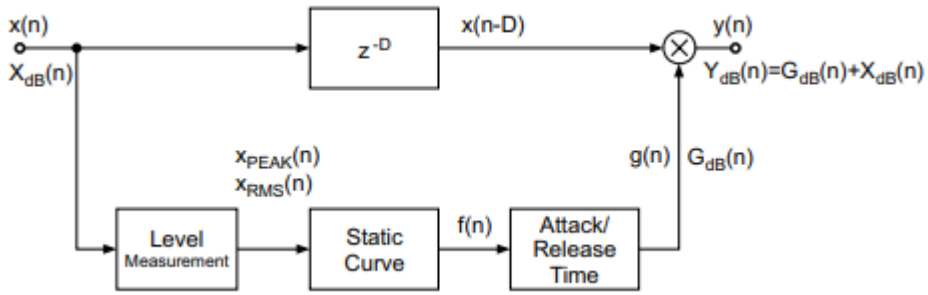


Figura 30: diagrama de bloques detallado del compresor de audio [18]

La señal de entrada se retrasa *D* muestras para conseguir un pequeño efecto de suavidad sonora en el inicio de la reproducción del audio de salida, evitando comienzos bruscos aunque el valor de estas muestras de retraso no sea muy elevado.

La señal de entrada *x(n)* se divide en dos caminos, donde por el camino de arriba esta misma es retrasada un número de muestras *D* y mientras que por el camino de abajo la señal es tratada por 3 bloques, son los siguientes.

Detector de nivel

El primero de ellos es el detector de nivel, su función es obtener el valor eficaz de la señal de entrada $x_{RMS}(n)$ el cual nos servirá para, en el siguiente bloque, comparar su valor con el umbral establecido y así operar según sea conveniente, donde usualmente la mayoría de compresores se basan en la medición de nivel según el valor eficaz de la señal dada (RMS).

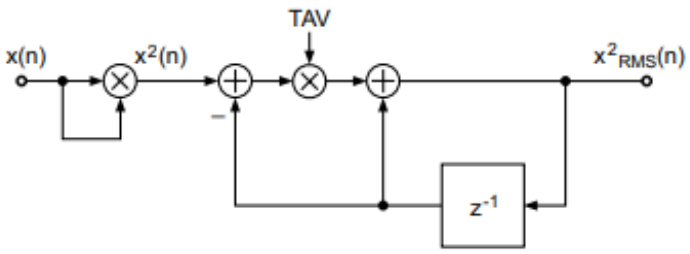


Figura 31: diagrama de bloques del detector de nivel [18]

Atendiendo a su diagrama de bloques, se puede ver como aparece el uso de una constante de tiempo llamada *TAV*, se trata de un parámetro que determina una constante

de tiempo media, que sigue la siguiente fórmula [18]:

$$TAV = 1 - e^{(\frac{-2,2*Ts}{tav/1000})}$$

En la ecuación se emplea el periodo de muestreo *Ts* (inverso a la frecuencia de muestreo vista anteriormente) y la constante temporal *tav* la cual se fija en 125 ms, valor habitual en los compresores comerciales. La ecuación en diferencias que otorga este bloque es la siguiente [18]:

$$\boldsymbol{x^2_{RMS}(n) = (1 - TAV) * x^2_{RMS}(n-1) + TAV * x^2(n)}$$

Función estática

El siguiente bloque que podemos observar, en la rama inferior de nuestro diagrama de bloques principal, es el bloque de la función estática del cual obtendremos la señal *f(n)* a partir de $x_{RMS}(n)$.

La función estática consiste en la **representación logarítmica** de la relación entre la señal de entrada y la de salida, de modo que dispondremos de una señal lineal 1:1 entre la entrada y la salida hasta llegar al momento donde el umbral (CT) se vea superado. A partir de ese momento se obtendrá una función según la proporción (R) dada por el sistema como se puede observar en la figura.

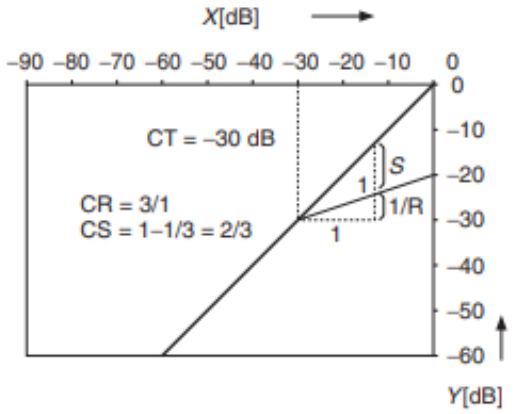


Figura 32: gráfica función estática [18]

En el ejemplo dado por la figura, tenemos un umbral de -30 dB, en el momento que se supera la señal de entrada se atenúa mediante una proporción de 3:1 dB.

La función que viene dada por esta relación es *F (dB)*, la cual depende de su entrada $x_{RMS}(n)$ en dB, del umbral en dB (CT) y de la pendiente (S) [17].

$$\boldsymbol{F = \begin{cases} -S * (X_{RMS} - CT)\ [dB] & si\ X_{RMS} > CT \\ 0\ [dB] & si\ X_{RMS} < CT \end{cases}}$$

Como podemos observar, al superar el umbral por la señal detectada X_{RMS} la señal de salida es negativa, esto es debido a que al estar en dB la señal que realmente

obtengamos de manera digital será una con un valor inferior a 1, es decir, una atenuación.

Filtro de suavizado

Este es el bloque final, del cual obtendremos la señal *g(n)* en función de la señal obtenida en el apartado anterior *f(n)*.

Se trata de un factor de suavizado el cual implementa los tiempos de ataque y de decaimiento (t_a y t_r), ya que el comportamiento de un compresor depende de estos tiempos así como de la señal *f(n)*. La obtención de nuestro factor de suavizado *g(n)* viene dado por el siguiente diagrama de bloques que podemos ver en la figura.

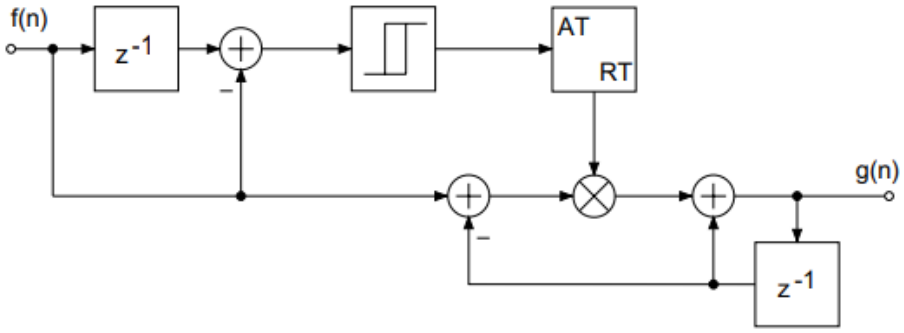


Figura 33: diagrama de bloques filtro de suavizado [17]

En el diagrama de bloques podemos observar el uso de un bloque de histéresis, el cual decide si el coeficiente a usar es el de ataque (*AT*) o si es el de decaimiento (*RT*). Estos dos coeficientes vienen dados por las siguientes funciones las cuales dependen de las propias constantes de tiempo dadas por el compresor [18].

$$AT = 1 - e^{\left(\frac{-2.2*Ts}{ta/1000}\right)}$$

$$RT = 1 - e^{\left(\frac{-2.2*Ts}{tr/1000}\right)}$$

El bloque de histéresis decide, mediante la comparación del valor de *f(n)* con el valor de *g(n-1)* en qué fase estamos, si en la de ataque o en la de decaimiento, de modo que mediante una variable llamada *k* asignaremos un el valor de la constante correspondiente (AT o RT) [18].

$$k = \begin{cases} AT & si\ f(n) < g(n-1) \\ RT & si\ f(n) \geq g(n-1) \end{cases}$$

Finalmente, obtenemos la ecuación en diferencias dada por el diagrama de la figura superior [18]:

$$\boldsymbol{g(n) = (1-k) * g(n-1) + k * f(n)}$$

Dando lugar finalmente a la señal de salida final de nuestro compresor de rango dinámico:

$$\boldsymbol{y(n) = x(n - D) * g(n)}$$

3.2.4 Implementación en MATLAB

Siguiendo el diagrama de flujo mostrado en el capítulo 1 en la introducción, el siguiente paso en el proceso de creación bloques IP será la implementación del mismo en MATLAB®, una vez estudiado el algoritmo y su modelo matemático. Por lo que, usaremos la herramienta para simular lo estudiado y comprobar la validez del compresor mediante representaciones gráficas de los resultados así como de la respuesta sonora.

Estructuraremos el código de nuestro algoritmo bloque a bloque, permitiéndonos seguir el diagrama de bloques de la manera más sencilla.

En primer lugar, leeremos el fichero de audio de entrada "***audio_in.wav***" mediante la función "audioread", de la cual obtendremos el array completo de muestras así como la frecuencia de muestreo de nuestro audio, dando lugar a 131072 muestras y una frecuencia de muestreo de 22050 Hz como ya se adelantó en el apartado 3.1.2.

Después, empezamos a elaborar el código, siendo el "bloque" o paso la creación de un array de muestras (mediante un bucle que recorra la totalidad de las muestras) que contenga la señal de entrada *x* retrasada un número de muestras D, en este caso se han seleccionado 150 muestras para la variable D.

```
[x , Fs] = audioread('audio_in.wav');
D = 150; %Muestras de retardo

%%%%%%%%%%%%%% RETARDO DE LA SEÑAL DE ENTRADA (x(n-D)) %%%%
x_D = zeros(1,D);
for i = 1:(length(x)-150)
    x_D(i+D) = x(i);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figura 34: parte 1 código DRCompressor MATLAB®

Aunque como veremos en la práctica, este pequeño retraso de 150 muestras es **despreciable y prescindible** en nuestro diseño real ya que en comparación con la totalidad de las muestras (131072) el efecto que provoca este retraso no es apreciable.

En siguiente lugar, se codificará el bloque de medición de nivel, del cual se obtiene la señal $x_{RMS}(n)$, donde previamente se declaren las constantes necesarias vistas en el apartado anterior.

```
%%%%%%%%%%%%%%% MEDICIÓN DE NIVEL (x_rms) %%%%%%%%%%%%%%%%%%
x_2 = zeros(1,length(x));
x_2rms = zeros(1,length(x));
x_rms = zeros(1,length(x));

Ts = 1/Fs;
tav_ = 125e-3; %cte de tiempo
TAV = 1 - exp((-2.2*Ts)/(tav_/1000)); %coef promedio

for i = 1:length(x)
    x_2(i) = x(i)*x(i);
    if i == 1
        x_2rms(i) = ((1-TAV)*0) + (x_2(i)*TAV);
    else
        x_2rms(i) = ((1-TAV)*x_2rms(i-1)) + (x_2(i)*TAV);
    end
    x_rms(i) = sqrt(x_2rms(i));
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figura 35: parte 2 código DRCompressor MATLAB®

Siguiendo el diagrama, codificaremos la función estática, teniendo como entrada la salida obtenida en el bloque anterior y que tendrá como objetivo obtener la señal *f(n)*. Fijaremos el **umbral** a **-10 dB**, así como una **proporción** de **2** para la elaboración del algoritmo, además, para la realización de la comparación de la señal de entrada con el nivel de umbral deberemos pasar esta a dB en primer lugar como se aprecia en el código. Una vez realizada la comparación, dada por la expresión vista en el apartado anterior, pasaremos a unidades naturales nuestra señal *F (dB)* para poder operar con ella en el siguiente bloque.

```
%%%%%%%%%%%%%%% FUNCIÓN ESTÁTICA (f(n)) %%%%%%%%%%%%%%%%%%
Xrms_dB = zeros(1,length(x));
F_dB = zeros(1,length(x));
f = zeros(1,length(x));

CT = -10;
R = 2;
S = 1 - (1/R);

for i = 1:length(x)
    Xrms_dB(i) = 10*log10(x_rms(i));
    if Xrms_dB(i) > CT
        F_dB(i) = -S*(Xrms_dB(i)-CT);
    else
        F_dB(i) = 0;
    end
    f(i) = 10^(F_dB(i)/10);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figura 36 parte 3 código DRCompressor MATLAB®

Por último, el bloque del filtro de suavizado será codificado, de manera que

previamente a realizar las operaciones dadas por las expresiones vistas en el apartado anterior, se deberán de fijar las constantes temporales del compresor como son el tiempo de ataque (*at*) y el tiempo de decaimiento (*rt*), donde serán 50 y 100 ms respectivamente, así como la obtención de las variables *AT* y *RT* mediante los tiempos fijados.

Posteriormente se hará la comparación del valor de *f(n)* con *g(n-1)*, visto en el apartado anterior, y se obtendrá el valor de *k*, la cual será empleada en la ecuación en diferencias que nos proporcionará el factor de suavizado *g(n)*.

```
%%%%%%%%%%%%%%% FACTOR DE SUAVIZADO (g(n)) %%%%%%%%%%%%%%%%%%
g = zeros(1,length(x));
k = 0;

at_ = 50e-3;
rt_ = 100e-3;
AT = 1 - exp((-2.2*Ts)/(at_/1000));
RT = 1 - exp((-2.2*Ts)/(rt_/1000));

for i = 1:length(x)
    if i == 1
        if f(i) < 0
            k = AT;
        else
            k = RT;
        end
        g(i) = ((1-k)*0) + (k*f(i));
    else
        if f(i) < g(i-1)
            k = AT;
        else
            k = RT;
        end
        g(i) = ((1-k)*g(i-1)) + (k*f(i));
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figura 37: parte 4 código DRCompressor MATLAB®

El resultado final dará como resultado la señal de salida de nuestro compresor *y(n)*, la cual vendrá dada por la multiplicación entre el factor de suavizado y la señal de entrada retrasada.

```
for i = 1:length(x)
    y(i) = g(i)*x_D(i);
end
```

Figura 38: parte 5 código DRCompressor MATLAB®

3.2.5 Resultados en MATLAB

A continuación, comprobaremos mediante resultados gráficos y sonoros si nuestro código ha sido realizado con éxito o no. Para ello, disponemos de una función en MATLAB® la cual reproduce de forma sonora, un array de muestras dada una frecuencia de muestreo, es la siguiente:

>> ***soundsc(y, Fs)***

Además, podemos hacer uso de la siguiente sentencia para guardar nuestro audio de salida en un fichero ".wav":

>> ***audiowrite('audio_out_drc.wav', y, Fs)***

Mediante la ejecución de estas sentencias podemos comprobar el audio real de salida que hemos obtenido al tratar el audio de entrada con nuestro DRCompressor, donde, el resultado audible es positivo. Al disponer de un audio casi 6 segundos de sonidos electrónicos donde la intensidad de los mismos va cambiando a medida que avanza el tiempo, podemos comprobar cómo, en los momentos donde mayor saturación se producía en nuestro audio, ahora tenemos sonidos más suaves y limpios evitando grandes picos de audio como se mostraba en los objetivos del bloque.

La representación gráfica de los resultados consistirá en una comparación entre las gráficas de la entrada y la salida medidas en decibelios, de forma que como podemos observar, *Y(dB)* ha sufrido las modificaciones dadas por la compresión de rango dinámico de nuestro algoritmo y sus parámetros.

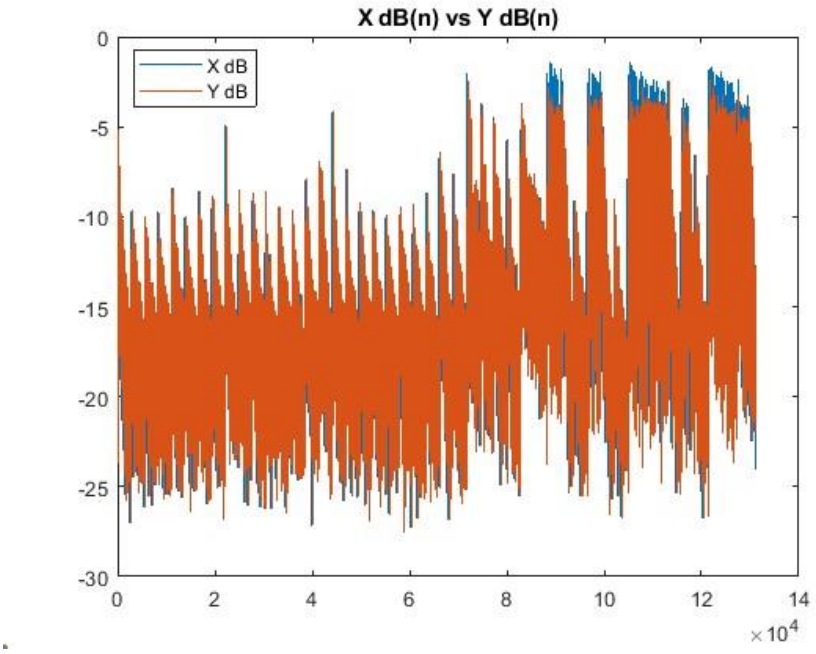


Figura 39: gráfica entrada/salida en decibelios DRCompressor MATLAB®

En la figura se aprecia como en los momentos de mayor saturación en nuestro audio de entrada, la señal de salida se ve atenuada consiguiendo los objetivos

propuestos para este bloque.

También podemos comparar la señal de entrada y de salida en unidades reales:

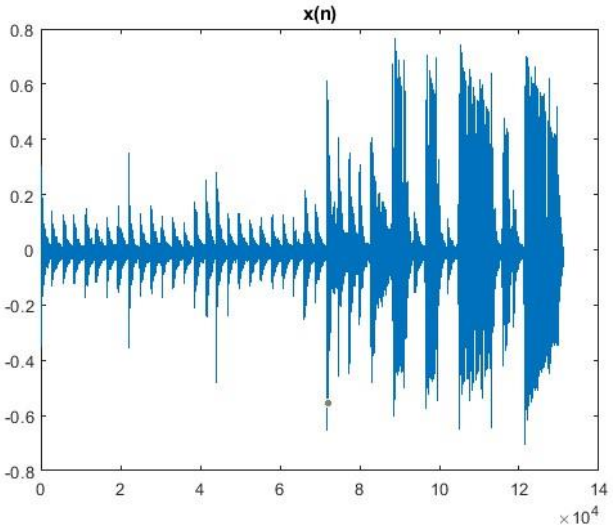


Figura 40: gráfica entrada DRCompressor en unidades reales MATLAB®

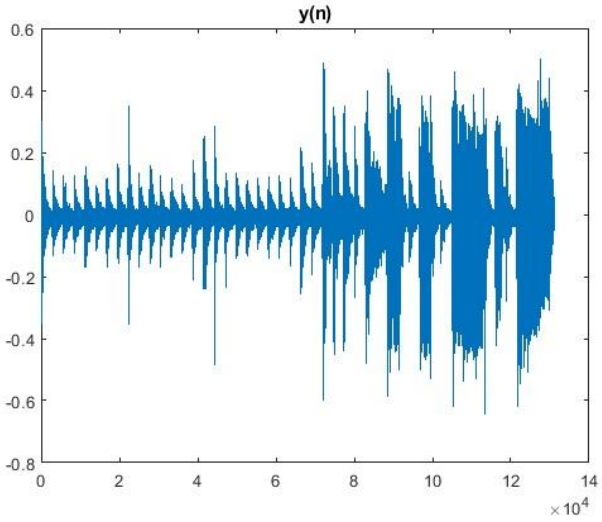


Figura 41: gráfica salida DRCompressor en unidades reales MATLAB®

En esta comparación también se aprecia la reducción de sonido dada en el audio de salida. Cabe destacar como nuestras representaciones en decibelios pueden llegar a tener un rango de valores entre 0 dB y – infinito, en cambio, como se ha repetido durante el proyecto, los audios representados en una escala real tendrán un rango entre +1 y -1.

Además, al ser MATLAB® el origen de los ficheros que usaremos más adelante, deberemos de crear nuestro ".txt", con datos en coma fija, que usaremos en Vivado HLS™, por que como ya vimos con anterioridad, será la forma de introducir los datos de entrada sobre este software. Imprimiremos línea a línea los valores de nuestro array de datos en coma fija, con 16 bits de longitud con 14 bits para la parte decimal, "*x_fi(n)*".

```
%CREACIÓN FICHERO CON ARRAY DE MUESTRAS
fid = fopen('dataset_in.txt', 'w');
for j=1:length(x)
    if(j == length(x))
            fprintf(fid, '%.6f', x_fi(j));
    else
            fprintf(fid, '%.6f\n', x_fi(j));
    end
end
fclose(fid);
```

Figura 42: código para creación fichero ".txt" con muestras de entrada

Siguiendo el diagrama de flujo dado en el capítulo 1, podemos continuar con el mismo al haber obtenido resultados positivos por parte de nuestro bloque en MATLAB®.

3.2.6 Implementación en Vivado HLS

Al ser nuestra primera implementación en Vivado HLS™ utilizaremos apartados de este proceso como ejemplo para futuros bloques. Por lo que, estos serán los pasos empleados en nuestra implementación del DRCompressor en HLS:

- Creación del proyecto
  - Seleccionar dispositivo FPGA a usar
- Creación de ficheros:
  - "DRCaudio.cpp", "DRCaudio_tb.cpp" y "DRCaudio.h"
- Añadir fichero de muestras "dataset_in.txt"
- Codificación de los 3 ficheros en C/C++
- Simulación y obtención de resultados

Según los pasos marcados, al crear nuestro proyecto en Vivado HLS™ deberemos de seleccionar nuestro dispositivo sobre el cual implementaremos nuestros diseños, como vimos en el capítulo 1, el dispositivo a elegir será el "**xc7a200t**…" de la familia Artix7. Después, añadiremos los 3 ficheros sobre los cuales implementaremos nuestro algoritmo. Como ya se mencionó anteriormente, los ficheros deben de tener el formato C++ ".cpp" ya que al emplear codificación en coma fija es el único formato soportado para ello en Vivado HLS™.

- **DRCaudio.h:** Fichero encargado de almacenar todas las constantes o variables, así como librerías necesarias para el desarrollo del bloque.

- **DRCaudio.cpp:** Fichero el cual albergará la función TOP de nuestro algoritmo, la cual será sintetizada e implementada en lenguaje de desarrollo hardware (HDL).

- **DRCaudio_tb.cpp:** Fichero utilizado como banco de pruebas. A la hora de la creación del bloque en HDL no será tenido en cuenta, solo nos servirá como validación de las simulaciones que hagamos previamente a la creación del bloque IP.

- Siguiendo el orden prescrito, codificaremos en primer lugar los bloques "**DRCaudio.h**" y "**DRCaudio.cpp**" de forma paralela.

Se debe de tener en cuenta el importante factor de que vamos a usar datos codificados en coma fija. Para ello, deberemos declarar las librerías correspondientes para ello, así como las de uso básico. Donde "ap_fixed.h" y "hls_math.h" nos permitirán el uso de datos coma fija así como operar con ellos.

```
#include <stdio.h>
#include <stdlib.h>
#include <hls_math.h>
#include "ap_fixed.h"
using namespace std;
```

Figura 43: librerías usadas en DRCaudio.h

La declaración de datos coma fija en Vivado HLS™ es: "*ap_fixed<W, I…>*" ya mencionado en el apartado 6 del capítulo 2. En nuestro caso, solo emplearemos los parámetros W (nº de bits total) e I (nº de bits parte entera), dejando los demás por defecto al no desear ningún modo de truncamiento o redondeo en concreto.

En siguiente lugar, debemos de conocer cómo funcionan las operaciones coma fija en Vivado HLS™, ya que **no** todas las operaciones matemáticas existentes están disponibles, además de tener una nomenclatura especial a la hora de usarlas. Por ejemplo:

ap_fixed<16,2> a
ap_fixed<16,2> b = 0,064

a = hls::sqrt(b);

La sentencia de una operación matemática como es la raíz cuadrada, viene dada por la expresión dada en el ejemplo, donde se trata de una función la cual debe de contener **un único argumento**, el valor a realizar la raíz. Siendo de la misma manera para las demás operaciones en coma fija.

Además, se debe de tener especial cuidado con las divisiones en coma fija, ya que para realizar una simple división que calcule el inverso de un valor (1/x) habrá que codificar este dividendo de valor entero como un dato coma fija, se pone el siguiente ejemplo al calcular el periodo de muestreo en el fichero DRCaudio.h:

```
ap_fixed<16,16> Fs = 22050;
ap_fixed<24,2> math_aux=1;
ap_fixed<22,0> Ts=math_aux/Fs;
```

Figura 44: variables DRCompressor en coma fija Vivado HLS™

Como podemos observar, cada dato tiene unos parámetros de codificación diferentes, de modo que al tratarse de constantes podemos definir previamente el número de bits necesario para cada parte.

En el caso de la frecuencia de muestreo al tratarse de un entero como es el 22050

deberemos emplear 16 bits, con 0 bits para la parte decimal.

En el caso de la declaración de nuestro dividendo, tendremos que adelantarnos al resultado, dando un número de bits para la parte decimal como tal sea ese número de bits de parte decimal del resultado deseado, además, de dejar el bit de signo y el bit del propio entero (1).

Finalmente, para obtener el resultado deseado (periodo de muestreo) debemos de saber el número de bits decimales previamente para realizar la declaración, viendo como finalmente necesitaremos 22 bits decimales. Con este número de bits podemos declarar el dividendo y el cociente, dando lugar al resultado deseado: 1/22050 = $45.35*10^{-6}$.

Este proceso lo repetiremos para todas las operaciones de constantes que necesitemos para el diseño. Cabe destacar, que el número de bits necesitado para codificar un número concreto en coma fija puede ser obtenido mediante el uso de calculadoras preparadas para ello. Además, se deberán usar numerosas variables auxiliares "***math_aux***" las cuales nos ayuden a la hora de almacenar valores previos a resultados de constantes.

Con estas premisas, el siguiente paso será llevar a cabo la codificación del algoritmo, de modo que realizaremos **un código semejante** al hecho en MATLAB®, con las modificaciones necesarias al estar trabajando con otro lenguaje de programación.

En primer lugar, la **definición de nuestra función**:

```
void DRCompressor(ap_fixed<16,2> x, ap_fixed<16,2> *y)
```

Figura 45: definición función DRCompressor

Al tratarse de un sistema secuencial de datos, nuestro bloque únicamente recibirá un dato de entrada y generará un dato de salida, además de que se tratará de una función void, ya que no necesitamos el retorno de ningún valor. Todas las señales del sistema tendrán la codificación ya mencionada: 16 bits totales con 2 bits para la parte entera. El siguiente paso será codificar el bloque de **medición de nivel**.

```
////////////// MEDICIÓN DE NIVEL (x_rms) //////////////
ap_fixed<16,2> x_2;
ap_fixed<16,2> x_2rms;
ap_fixed<16,2> x_rms;

math_aux = math_aux2*Ts;

ap_fixed<24,0> math_aux3=tav_/1000;
ap_fixed<16,1> math_aux4 = math_aux/(math_aux3);
ap_fixed<18,1> TAV = 1 - hls::exp(math_aux4); //coef promedio

    x_2 = x*x;
    x_2rms = ((1-TAV)*x_2rms_ant) + (x_2*TAV);
    x_2rms_ant = x_2rms; //x_2rms[i-1]

    x_rms = hls::sqrt(x_2rms);
////////////////////////////////////////////////
```

Figura 46: código bloque medición de nivel DRCompressor

A diferencia de nuestro código en MATLAB®, se ha eliminado la totalidad de los bucles, debido a que el bucle que recorra todo nuestro fichero de datos será generado mediante el banco de pruebas como veremos más adelante.

```
////////////// FUNCIÓN ESTÁTICA (f(n)) ///////////////
ap_fixed<12,0> ct = 0.1;
ap_fixed<16,2> f;

    //if(Xrms_dB > CT)
    if (x_rms > ct || x_rms < -ct)
        f = 0.75;
    else
        f = 1;

    //f=10^(f_dB/10);

////////////////////////////////////////////////////////
```

Figura 47: código bloque función estática DRCompressor

Llegamos a la primera modificación importante del código con respecto al original. Nos encontramos con un problema al querer usar datos en decibelios en la **función estática**.

El tratar con datos en decibelios conlleva a, una vez trabajados con ellos, volver a transformar esos decibelios en datos reales, de modo que para realizar esa conversión inversa se necesita de una expresión tal que:

$$a = 10^{(A\,(dB)/10)}$$

Se trata de una potencia de un número entero elevado a un número real. Esta operación no es posible realizarla mediante la librería matemática de coma fija en HLS, solo disponemos de exponenciales en base al número *e*. Es por ello que esta parte del código se realizó en unidades naturales a diferencia del código original.

El cambio consiste en usar, sin pasar a decibelios, números naturales, de modo que nuestro umbral de -10 dB en este casó tendrá el valor de 0.1.

La comparación en este caso, al estar en unidades naturales, se hará tanto para valores positivos como para valores negativos de nuestra señal de entrada "*x_rms*", situación que en la escala logarítmica, no sucedía.

Por último, el valor de *f(n)*. La expresión:

$$\boldsymbol{F} = \begin{cases} -\boldsymbol{S} * (\boldsymbol{X_{RMS}} - \boldsymbol{CT})\ [\boldsymbol{dB}] & \boldsymbol{si\ X_{RMS}} > CT \\ \boldsymbol{0}\ [\boldsymbol{dB}] & \boldsymbol{si\ X_{RMS}} < CT \end{cases}$$

Para transformar esta expresión a números naturales podríamos usar la siguiente expresión:

$$f(n) = \begin{cases} \left(\frac{x_{RMS}(n)}{ct}\right)^{\frac{1}{R}-1} & si\ x_{RMS}(n) > CT\ o\ x_{RMS}(n) > -CT \\ 1 & si\ x_{RMS}(n) < CT \end{cases}$$

Pero una vez más, el uso de potencias de ese modo no es posible al usar la librería de coma fija en HLS.

Por lo que se optó por lo que se puede observar en el código, seleccionar un factor medio de atenuación como es 0,75 en el caso de superar el umbral.

```
////////////// FACTOR DE SUAVIZADO (g(n)) ////////////////
ap_fixed<26,3> math_aux7 = math_aux/math_aux5; //math_aux = -2.2*Ts
ap_fixed<26,3> math_aux8 = math_aux/math_aux6;

ap_fixed<26,1> AT = 1 - hls::exp(math_aux7); //0.864049
ap_fixed<26,1> RT = 1 - hls::exp(math_aux8); //0.631285

ap_fixed<16,2> g;
ap_fixed<26,1> k;

        if(f < g_ant)
            k = AT;
        else
            k = RT;

    g = ((1-k)*g_ant) + (k*f);
    g_ant = g; //g[i-1]
////////////////////////////////////////////////////////

////////////////// AUDIO OUTPUT: y(n) /////////////////
    *y = g*x;
////////////////////////////////////////////////////////
```

Figura 48: bloque filtro de suavizado DRCompressor

Finalmente, podemos realizar un código similar al original para la última parte del algoritmo, el **filtro de suavizado**, obteniendo finalmente la salida ***y***.

- Para comprobar la validez de nuestro código, emplearemos un banco de pruebas o testbench: "**DRCaudio_tb.cpp**"

Se trata del fichero mediante el cual, seremos capaces de probar el funcionamiento de nuestra función top, siendo un fichero que no se sintetizará a la hora de transformar nuestro código a lenguaje de descripción hardware. Al realizar la "simulación C" se ejecutará el código descrito en el testbench, el cual incluirá de la forma y en el momento que nosotros queramos a la función top de nuestro diseño.

Además, mediante el testbench introduciremos las muestras de entrada mediante la lectura línea a línea de nuestro fichero "dataset_in.txt". De la misma forma, escribiremos las muestras obtenidas a la salida del algoritmo en nuestro fichero de salida "dataset_out.txt", con el cual comprobaremos la validez del bloque posteriormente en MATLAB®.

Como veremos a continuación, el testbench ejecutará una función "main", desde

la cual se hará todo lo mencionado, además de las declaraciones de librerías, de nuestra función top y del número de muestras de nuestro fichero.

El número de muestras de nuestro fichero de entrada es especificado en el testbench debido a que únicamente nos sirve para probar el algoritmo, en el funcionamiento real de bloque se leerá el fichero hasta que acabe el mismo.

En primer lugar, declaramos las librerías necesarias, donde se incluye "fstream" para trabajar con ficheros, además, se incluye el número de muestras del fichero de entrada, la declaración de la función top y dos arrays que usaremos posteriormente.

```
#include <stdio.h>
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include "ap_fixed.h"

using namespace std;

#define n_samples 131072

void DRCompressor(ap_fixed<16,2> x , ap_fixed<16,2> *y);

ap_fixed<16,2> x_n[n_samples];
ap_fixed<16,2> y_n[n_samples];
```

Figura 49: librerías y definiciones en testbench DRCompressor

Seguidamente, ejecutamos la función main, donde en primer lugar, se abrirá tanto el fichero de entrada en modo lectura (*ifstream*), como el de salida en modo escritura (*ofstream*), de modo que si alguna de estas acciones falla, saldremos de la simulación lanzando un mensaje de error.

```
int main()
{

        ifstream in_file("dataset_in.txt");
        if (!in_file) {
            printf("Can't open data file\n");
            exit(0);
        }
        ofstream out_file("dataset_out.txt");
        if (!out_file) {
            printf("Can't open data file\n");
            exit(0);
        }
```

Figura 50: lectura ficheros en testbench DRCompressor

Como ya hemos visto, nuestro bloque está diseñado de forma que recibe una muestra y saca otra, va de muestra en muestra de forma secuencial, ya que la otra opción sería introducir el array completo de muestras al bloque pero esto no es ni lógico ni posible debido a los posibles problemas de memoria de nuestro dispositivo.

Es por ello que, para comprobar su funcionamiento, el bucle donde se recorrerá

la totalidad del fichero de entrada será realizado en el testbench.

Como vemos en la figura, las muestras que contiene el fichero en "*in_file*" son introducidas al array "*x_n[]*". Según apartados anteriores, el delay **opcional** sobre la señal de entrada de 150 muestras es realizado en este momento, de forma que al superar las 150 iteraciones se ejecuta el algoritmo mediante la llamada de la función top, entregando como argumento de entrada el array "*x_n[]*" retrasado y obteniendo el resultado del algoritmo en el array "*y_n[]*".

```
for(long i=0; i<n_samples; i++){

    in_file >> x_n[i];

    if(i < 150)         //x[n-D]
        y_n[i]= 0;
    else{
        DRCompressor(x_n[i-150], &y_n[i]); //Ejecutamos algoritmo
    }

    out_file << y_n[i] << "\n";
}
in_file.close();
out_file.close();
```

Figura 51: ejecución algoritmo en testbench DRCompressor

Cabe destacar, que necesitamos declarar, en la función top, la señal "*y*" de salida como un puntero, debido a que si no lo hacemos, no se obtendrá el resultado de la señal "*y*" (albergado en "DRCaudio.cpp") en nuestro testbench.

Finalmente en el bucle, guardamos las muestras en el fichero de salida mediante la variable "*out_file*". Una vez acabado el bucle cerraremos los ficheros usados y podremos pasar a simular nuestro banco de pruebas.

3.2.7 Resultados en Vivado HLS

Siguiendo con el diagrama de flujo descrito en el capítulo1, el siguiente paso será realizar la simulación C de nuestro algoritmo.

Para ello, lanzaremos la simulación C en Vivado HLS™  , de forma que al finalizar el programa nos lanzará un mensaje informándonos sobre que no hubo errores en la simulación y que la misma ya finalizó.

```
INFO: [SIM 2] ************** CSIM start **************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
make: `csim.exe' is up to date.
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] ************** CSIM finish **************
```

Figura 52: informe de la simulación C DRCompressor

El siguiente paso, será leer en MATLAB® el fichero de salida "dataset_out.txt" obtenido, reproducir sonoramente las muestras y representar gráficamente las mismas.

```
fp1=fopen('dataset_out.txt', 'r');
y_drc=fscanf(fp1, '%f');
sound(y_drc, 22050)

figure
plot(y_drc)
title("y drc(n)")
```

Figura 53: lectura dataset de salida DRCompressor en MATLAB®

El audio reproducido es igual al reproducido en el apartado 3.2.5 y la representación gráfica comparada con la obtenida en el apartado 3.2.5 es la siguiente:

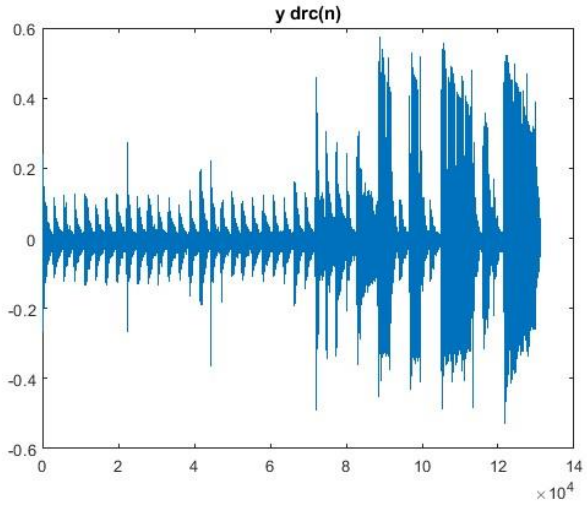


Figura 54: gráfica señal de salida DRCompressor de Vivado HLS

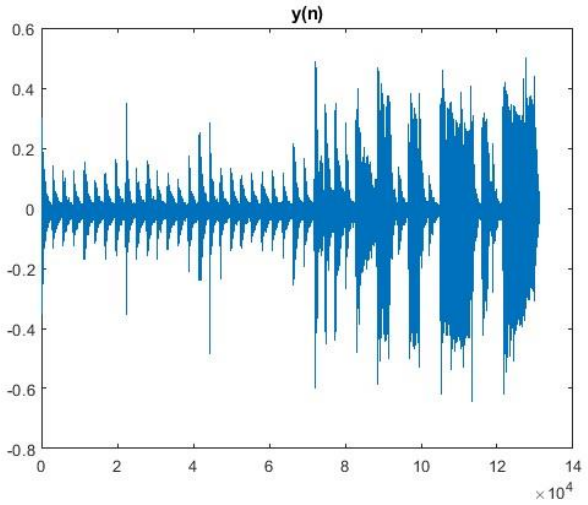


Figura 55: gráfica señal de salida DRCompressor de MATLAB

En la figura de la izquierda se muestra el audio de salida obtenido en Vivado HLS™ y en la derecha el de MATLAB®.

Se pueden observar ciertas diferencias dadas por las adaptaciones que tuvimos que realizar a la hora de implementar el algoritmo en HLS, aunque finalmente, el resultado es similar.

Para finalizar, calcularemos el **error cuadrático medio** entre el dataset de salida ideal (MATLAB®) y el real (Vivado HLS™) de la misma forma que lo hicimos en el apartado 3.1.3.

Se obtiene un error entre ambos datasets de salida del **2,8%.** Se trata de un error asumible teniendo en cuenta las modificaciones necesarias que se realizaron en la implementación del algoritmo con las librerías de coma fija en HLS.

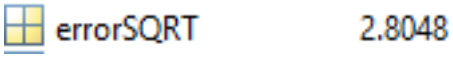


Figura 56: error cuadrático medio entre los dos datasets

3.2.7.1 Síntesis y optimización del algoritmo

Síntesis sin directivas

El siguiente paso en nuestro diagrama de flujo consistirá en sintetizar nuestra función TOP, ver los resultados obtenidos de la propia síntesis y tratar de mejorarlos mediante el uso de directivas de optimización.

Mediante la síntesis de alto nivel conseguiremos transformar nuestro código C/C++ en código basado en la lógica RTL, es decir, se interpreta el código descrito y se crea el diseño hardware digital que implementa dicho código. Lo cual nos permitirá conocer los valores de tiempo y de recursos hardware empleados por el diseño como veremos a continuación.

El informe proporcionado por Vivado HLS™ al ejecutar la síntesis  es el siguiente:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.72	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
19	19	19	19	none

Figura 57: informe de tiempos post-síntesis sin directivas DRCompressor

En él, podemos observar la velocidad obtenida por el reloj del bloque (ap_clk) la cual llega a **8,72 ns**, siendo este tiempo el mínimo periodo que podríamos usar con el bloque.

La **latencia** o los ciclos de reloj que se producen al hacer uso del algoritmo y generar un resultado a la salida, dando como resultado **19 ciclos de reloj**. Así como el **intervalo de iniciación** (*interval*) el cual indica el número de ciclos con el que podemos entregar una nueva muestra a la entrada del bloque, siendo de **19 ciclos**.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	4	-	-
Expression	-	1	0	275
FIFO	-	-	-	-
Instance	0	-	675	3068
Memory	-	-	-	-
Multiplexer	-	-	-	101
Register	-	-	218	-
Total	0	5	893	3444
Available	730	740	269200	129000
Utilization (%)	0	~0	~0	2

Figura 58: informe uso recursos hardware post-síntesis sin directivas DRCompressor

A la hora de analizar los recursos hardware empleados, podemos ver como la utilización de bloques BRAM de 18K, de DSP48E, de Flip-Flops (FF) y de Look-Up Tables (LUT) son realmente bajos, alcanzando un **2%** de LUTs como valor máximo. Esto viene debido al diseño del algoritmo que hemos empleado, en el cual no necesitamos de la necesidad de bloques de memoria por ejemplo. Estos porcentajes son en base a las especificaciones de recursos totales que dispone nuestra Artix 7 y su xc7a200t.

Síntesis con directivas

Una vez visto el informe obtenido mediante la síntesis sin directivas de optimización, pasaremos a añadir dichas directivas con el objetivo de mejorar, según nuestras necesidades, los resultados obtenidos.

Nuestros objetivos con las directivas serán los siguientes:

1. Mejorar los ciclos de latencia y del intervalo de iniciación.
2. Generar interfaz AXI4-Stream en los puertos entrada/salida

1.- En primer lugar, buscaremos la mejora en la latencia del diseño, para ello, debemos de emplear alguna de las directivas vistas y detalladas en el apartado 2.3.

Al no disponer de bucles en nuestra función TOP, las directivas orientadas a bucles serán aplicadas sobre la propia función. De forma que disponemos de las siguientes opciones de directivas que tratan de mejorar los tiempos de procesamiento:

- **Pipeline**
- **Dataflow**
- **Latency**
- **Unroll**

Con estas posibles directivas buscaremos la opción que mejores resultados nos proporcione.

- La directiva *Unroll* trata de desenrollar bucles con el fin de paralelizarlos, es una función que está implícita dentro de la directiva Pipeline por lo que no nos será útil.

- Los resultados obtenidos con la directiva *Dataflow* consisten en una pequeña reducción de la latencia y del intervalo de iniciación, pero se hacen insuficientes comparado con lo esperado.

- Los resultados obtenidos con la directiva *Pipeline* son positivos en cuanto a la reducción del intervalo de iniciación, el cual es reducido a 1 ciclo de reloj, aunque la latencia es similar.

- La directiva *Latency* tratará de conseguir la menor posible latencia para el diseño, llegando a observar que el mejor resultado de latencia son 14 ciclos de reloj. Para ello, especificaremos en la directiva un número de latencia máximo (1 ciclo) y el software intentará acercarse lo máximo posible a esta especificación.

Viendo estas opciones, la mejor opción y que mejores resultados nos proporcionará será la combinación de ***Pipeline*** y de ***Latency***. Ninguna otra combinación tendrá sentido, debido a que por ejemplo, la combinación entre *Pipeline* y *Dataflow* no es posible, emplearemos la directiva que mejor latencia nos proporcione (*Latency*) y la que mejor intervalo de iniciación nos dé (*Pipeline*).

Pero el empleo de directivas que mejoran los tiempos del diseño tiene sus consecuencias, necesitaremos más recursos hardware así como una peor velocidad de reloj. Los resultados obtenidos por nuestra combinación de directivas son los siguientes:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	30.00	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
14	14	1	1	function

Figura 59: informe de tiempos post-síntesis con directivas DRCompressor

Obtenemos una velocidad de reloj de **30 ns**, siendo 10 ns el tiempo objetivo del software, esto es un mal menor, debido a que en nuestro diseño, la frecuencia de reloj a la que será sometido el bloque, será mucho mayor, como veremos más adelante.

Por otro lado, el objetivo está cumplido, hemos reducido en 5 ciclos de reloj la latencia hasta **14 ciclos** y hemos obtenido un intervalo de iniciación de **1 ciclo**.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	4	-	-
Expression	-	1	0	347
FIFO	-	-	-	-
Instance	0	-	691	3077
Memory	-	-	-	-
Multiplexer	-	-	-	66
Register	0	-	213	32
Total	0	5	904	3522
Available	730	740	269200	129000
Utilization (%)	0	~0	~0	2

Figura 60: informe uso recursos hardware post-síntesis con directivas DRCompressor

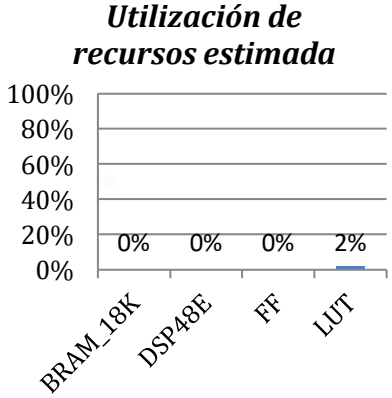


Figura 61: gráfico de la utilización de recursos DRCompressor

Además, no hemos tenido ningún tipo de empeoramiento notable en cuanto a recursos hardware, obteniendo los mismos porcentajes de uso.

2.- Nuestro segundo objetivo consistirá en crear una interfaz AXI4-Stream sobre nuestros puertos de entrada/salida.

Para ello, emplearemos la directiva ***Interface*** la cual vimos en el apartado 2.3. Es una directiva que nos permite seleccionar la interfaz deseada por el usuario sobre los diferentes puertos de entrada/salida.

Como ya hemos mencionado numerosas veces durante el proyecto, el protocolo con la que trabajaremos en nuestro sistema será el AXI4-Stream. Para ello aplicaremos la directiva *Interface* sobre las variables *x* e, indicando que nuestra interfaz será la llamada **axis** (axi stream) *y* dándoles un nombre a cada puerto. Además, aplicaremos la interfaz **ap_ctrl_none** sobre la función top al no desear la inclusión de señales "ap".

```
#pragma HLS LATENCY max=1
#pragma HLS PIPELINE
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis register depth=2 port=y name=y_drc
#pragma HLS INTERFACE axis register port=x name=x_drc
```

Figura 63: declaraciones directivas de optimización DRCompressor

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	DRCompressor	return value
ap_rst_n	in	1	ap_ctrl_none	DRCompressor	return value
x_drc_V_TDATA	in	16	axis	x_drc_V	scalar
x_drc_V_TVALID	in	1	axis	x_drc_V	scalar
x_drc_V_TREADY	out	1	axis	x_drc_V	scalar
y_drc_V_TDATA	out	16	axis	y_drc_V	pointer
y_drc_V_TVALID	out	1	axis	y_drc_V	pointer
y_drc_V_TREADY	in	1	axis	y_drc_V	pointer

Figura 62: informe interfaces post-síntesis con directivas DRCompressor

En el informe de la síntesis realizada podemos ver también el glosario de puertos de nuestro bloque, viendo cada una de sus señales.

Las señales "***x_drc***" e "***y_drc***" de desglosan en 3 señales básicas de la interfaz AXI4-Stream, las señales **TDATA**, **TVALID** y **TREADY**. Además se añaden la señal de reloj **ap_clk** y una señal de reset **ap_rst_n**.

3.2.7.2 Co-Simulación RTL y creación del núcleo IP

Finalmente, Vivado HLS nos da la opción de simular en VHDL nuestro diseño, donde veamos los resultados de las directivas todo en lenguaje hardware, con los datos en binario y con señales lógicas.

La simulación obtenida es la siguiente:

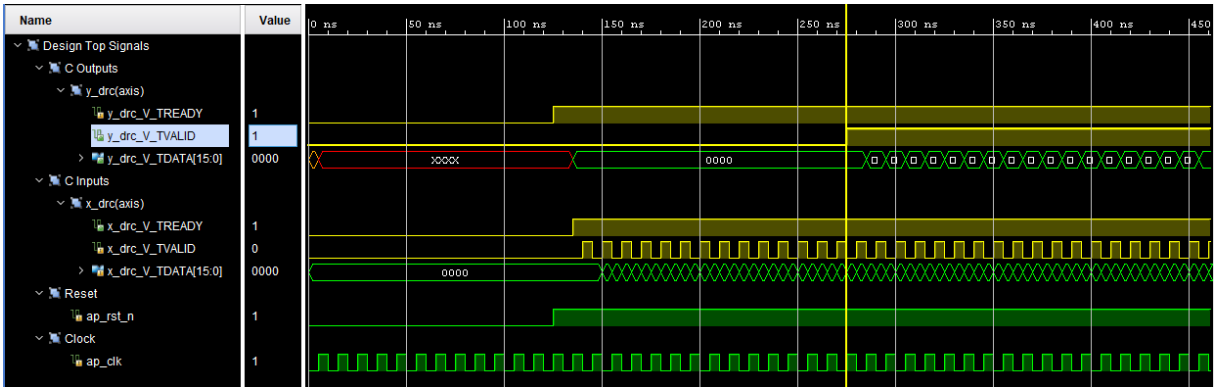


Figura 64: Co-Simulación RTL DRCompressor

Podemos comprobar cómo la latencia es de 14 ciclos de reloj al ver que las muestras de salida tardan 14 ciclos en generarse a partir de su muestra de entrada, momento a su vez, en la que la señal TVALID de la salida vale un nivel alto, indicándonos que la validez del resultado ya que sabemos que los datos de salida (TDATA) únicamente se obtiene al disponer de las señales TREADY y TVALID a nivel alto.

Además, los datos de entrada se van introduciendo al bloque secuencialmente cada vez que la señal TVALID de la entrada vale '1'.

Por último, exportamos nuestro diseño RTL obtenido a bloque IP. Una vez se hayan producido los ficheros correspondientes al mismo, importaremos en Vivado™ el bloque IP y lo añadiremos en un diseño de bloques para observarlo y ver si es lo que esperábamos obtener.

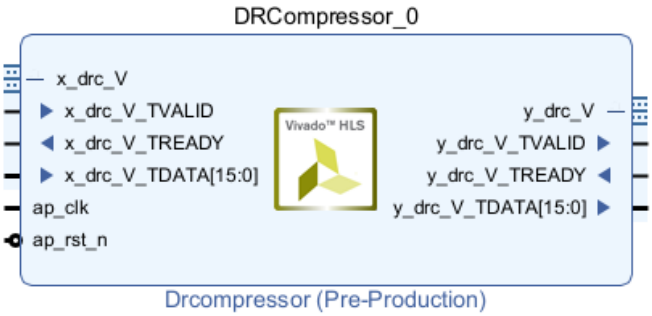


Figura 65: bloque IP-core final DRCompressor

3.3 Etapa de filtrado: FIR vs IIR

3.3.1 Introducción

Debido a que cualquier señal puede llevar acoplado distintos tipos de ruido en forma de señal parásita, es necesario en cualquier aplicación o sistema de procesado de señal el uso de filtros que puedan paliar este acoplamiento.

En esta etapa del sistema se tratará la señal, proveniente del anterior bloque, de forma que se reduzca o elimine de la mejor manera posible el ruido que nuestra señal original lleva acoplado en ella.

Nuestro filtro final realizará un procesamiento de nuestra señal que dependerá de los parámetros que apliquemos sobre el mismo, de modo que "discrimine" las componentes frecuenciales no deseadas de nuestra señal. Los filtros pueden cambiar amplitudes, frecuencias o fases según el objetivo que busquemos con el mismo, de la misma forma que existen gran variedad de filtros digitales, aunque en este trabajo solo veremos unos pocos de ellos.

Como todos los bloques de nuestro sistema, nuestro filtro se basará en un diagrama de bloques basado en un planteamiento matemático que nos proporcione una señal de salida en función de la señal de entrada. Para ello, discutiremos y compararemos dos tipos de filtros ampliamente estudiados y conocidos en el mundo del procesado de señal digital, como son los filtros **FIR** e **IIR**.

El filtro FIR (Respuesta Finita al Impulso) tiene en común con el filtro IIR (Respuesta Infinita al Impulso) que responden según la señal impulso a su entrada, pero tienen una serie de diferencias entre ellos que nos harán decantarnos por uno o por otro las cuales analizaremos en el próximo apartado, viendo sus distintas estructuras, sus comportamientos y sus ventajas y desventajas.

3.3.2 El filtro FIR

En primer lugar, un filtro FIR consiste en una estructura matemática basada en ganancias y retardos la cual consigue una respuesta al impulso de longitud finita. Además, se caracteriza por ser lineal, causal, estable e invariante, su salida viene dada por la siguiente expresión [21].

$$y_n = \sum_{k=0}^{N-1} b_k x_{n-k}$$

Figura 66: expresión general señal de salida filtro FIR [21]

La señal de entrada será retardada "*k*" veces según la iteración en la que nos encontremos, así como será multiplicada por los coeficientes del filtro b_k los cuales deben de ser calculados para la implementación del filtro.

Esta expresión matemática indica que al únicamente depender de valores previos de la señal de entrada, la respuesta al impulso será finita, al no depender de una realimentación sobre la señal de salida. La estructura básica del filtro es la siguiente:

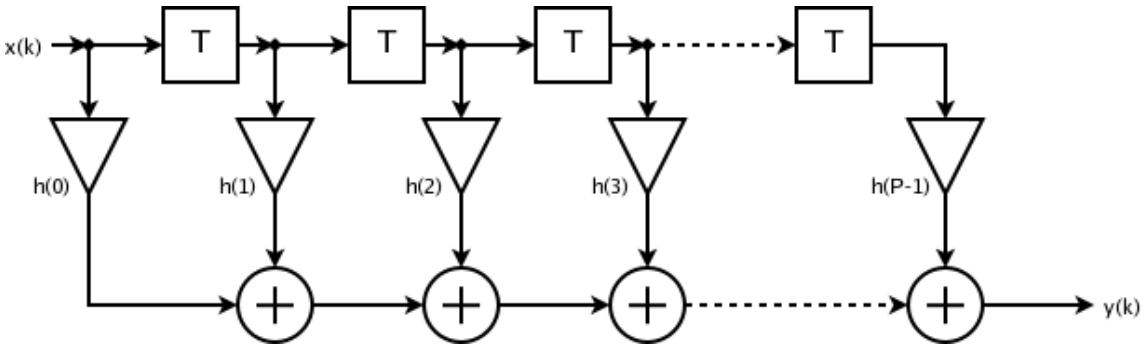


Figura 67: estructura general filtro FIR [21]

Estructura la cual se puede interpretar de forma que la salida consiste en un sumatorio de muestras retardas y amplificadas por coeficientes los cuales dan lugar a la respuesta finita del mismo.

En conclusión, un FIR se caracteriza por tener fase lineal y ser siempre estables, lo cual hace que nuestra señal mantenga su forma, aunque esto requiere un mayor orden del filtro, lo que conlleva a un mayor coste computacional.

3.3.3 El filtro IIR

En segundo lugar, los filtros IIR generan una respuesta al impulso infinita, a diferencia de los FIR. Se caracterizan, también, por no tener una fase lineal, esto permite al filtro tener un orden menor y por lo tanto, un menor coste computacional, a costa de poder distorsionar la señal de salida debido a no tener estabilidad [22].

Su salida viene dada por la siguiente expresión:

$$y_n = b_0 x_n + b_1 x_{n-1} + \cdots + b_N x_{n-N} - a_1 y_{n-1} - a_2 y_{n-2} - \cdots - a_M y_{n-M}$$

Figura 68: expresión general señal de salida filtro IIR [22]

Mediante la expresión básica de un filtro IIR podemos ver como se hace uso de la realimentación con respecto a la señal de salida retardada *M* veces además de tener la componente de la señal de entrada retardada que ya teníamos en el FIR. Esta realimentación a la salida es lo que provoca una respuesta infinita al impulso unitario.

La estructura que representa dicha expresión es la siguiente:

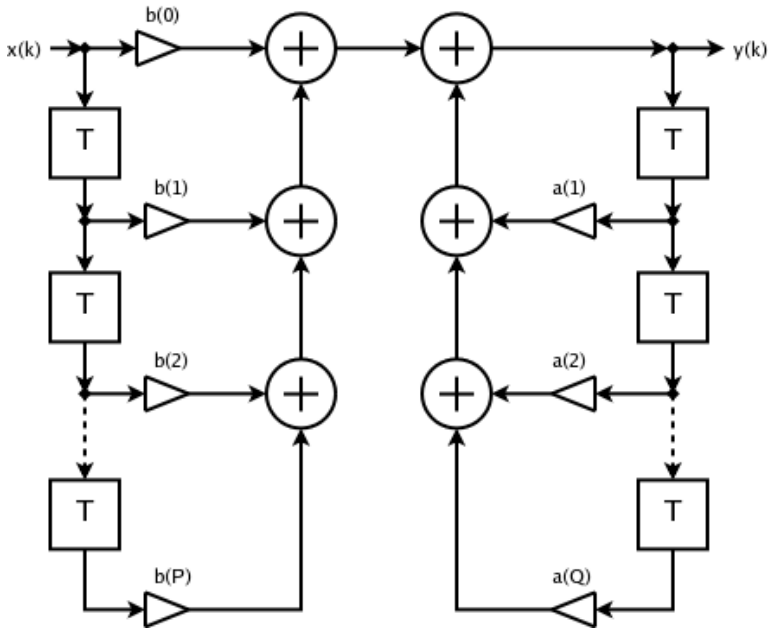


Figura 69: estructura general filtro IIR [22]

En conclusión, los filtros IIR no tienen fase lineal y son intestables, lo que puede provocar distorsiones en la señal de salida, por ello, disponen de un orden de filtro menor, lo que conlleva a un menor coste computacional.

3.3.4 Comparativa FIR vs IIR

Conociendo las principales características de ambos pasaremos a realizar una comparativa la cual decidirá que filtro diseñaremos e implementaremos sobre nuestro sistema.

Las ventajas y desventajas teóricas de ambos son las siguientes:

	Ventajas	Desventajas
FIR	Fase lineal y estable	Mayor coste computacional
IIR	Menor coste computacional	Fase no lineal y posible inestabilidad

A la comparativa teórica se les suman las siguientes ventajas y desventajas prácticas:

	Ventajas	Desventajas
FIR	IP-core nativo de Vivado™	-
IIR	-	No IP-core nativo en Vivado™ Creación de bloque en HLS con uso de memoria externa en FPGA

Con las dos comparativas sobre la mesa, tanto práctica como teórica se decide cual será el filtro a usar.

En cuanto a la **comparativa teórica** hay una clara discusión entre querer una

fase lineal o no y entre tener un mayor coste de procesado computacional o no.

En nuestro caso, el tener una fase lineal es una pequeña ventaja debido a que no se deformará la señal de salida, aunque es cierto que es posible que si se produjese una deformación de la señal el oído humano no la notaría.

Por lo tanto, analizaremos si el coste computacional proporcionado por el filtro FIR es tal como para prescindir de la linealidad del mismo.

Por otro lado, en cuanto a la **comparativa práctica** está claro que el uso de un FIR conlleva una gran ventaja con respecto al IIR. El hecho de disponer un bloque IP nativo en Vivado es una gran ventaja teniendo en cuenta que si quisiéramos emplear un IIR necesitaríamos implementarlo en HLS para posteriormente obtener el IP-core.

Además, en el caso de tener que implementar un IIR en HLS conllevaría a necesitar usar bloques de memoria de nuestro dispositivo debido a la naturaleza del algoritmo, donde se usan muestras retrasadas tanto a la entrada como a la salida.

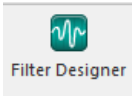
En resumen, es clara la **ventaja** (en nuestro caso) **del filtro FIR** contra el IIR, salvo un elemento, el **coste computacional**. Para ello, implementaremos los dos filtros en MATLAB en el próximo apartado y compararemos este parámetro para tomar la decisión final.

3.3.5 Implementación filtro FIR en MATLAB

Para los siguientes apartados se debe tener en cuenta que:

- Disponemos de una fuente de audio que lleva consigo acoplada una **señal parásito senoidal de 6 kHz** que interfiere en nuestros audios de entrada mediante su acoplamiento a los mismos.

El diseño de filtros en MATLAB® está bien cubierto ya que disponemos de una aplicación propia para ello.

En primer lugar, iniciaremos la aplicación de diseño de filtros: 

- Seleccionamos el tipo de filtro: FIR de **banda eliminada**. Dado que deseamos eliminar un ruido acoplado de 6 kHz.
- Seleccionamos tipo de filtro FIR: ventana tipo Hamming.
- Seleccionamos orden del filtro: 500, ya que con este orden alcanzamos los -20 dB en los 6 kHz.
- Especificamos frecuencia de muestreo y frecuencias de corte 1 y 2: 22050 Hz, 5950 Hz y 6050 Hz respectivamente.
- Aplicamos las especificaciones y observamos la respuesta en frecuencia.

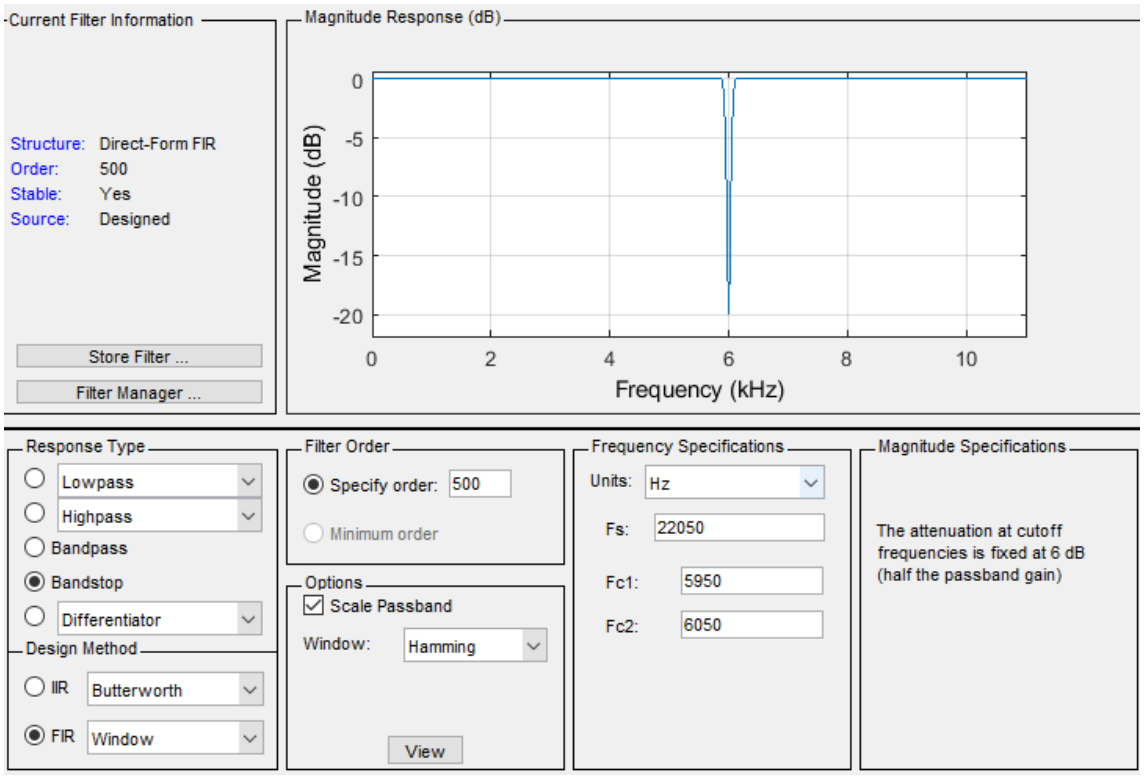


Figura 70: diseño del filtro FIR en MATLAB®

Una vez tenemos el filtro diseñado con las especificaciones deseadas, pasaremos a generar una función del mismo para poder ser empleada en un script de MATLAB®. Además, **exportaremos** al workspace el vector de **coeficientes** del filtro, los cuales serán introducidos después en el bloque de Vivado™, dándoles un ancho de bits de 24 con 23 bits de parte decimal.

El siguiente paso es comprobar el funcionamiento del filtro mediante un script, analizaremos el espectro de la señal de audio entrante observando la componente a eliminar.

Para ello, empleamos la función ***fft*** sobre nuestra señal de entrada y la representamos con respecto a la frecuencia:

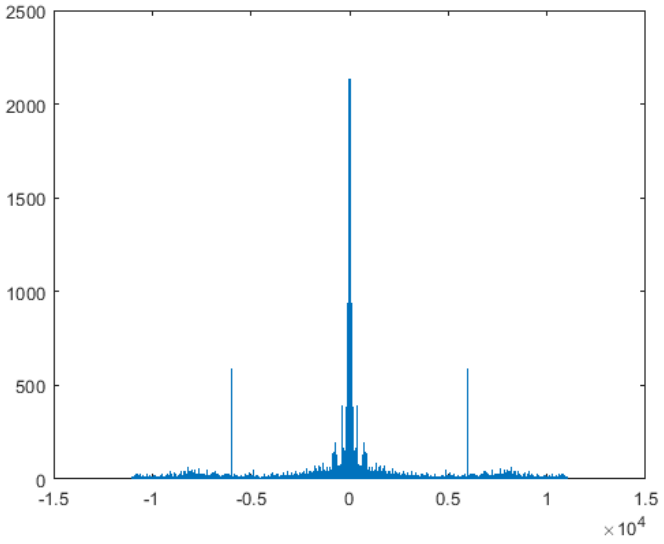


Figura 71: respuesta frecuencial de la entrada al filtro FIR

Observamos claramente que a los 6000 Hz se encuentran las dos componentes intrusas a eliminar.

Haremos uso de nuestra función generada por el diseño del filtro anterior llamada: ***bandstop***, y así como de la propia función nativa de MATLAB®: ***filter***. De modo que ejecutamos la siguiente sentencia y obtenemos la señal de salida filtrada:

>> y_fir = filter(bandstop, x_fir)

Si realizamos el mismo análisis espectral para la señal de salida obtenida observamos lo siguiente:

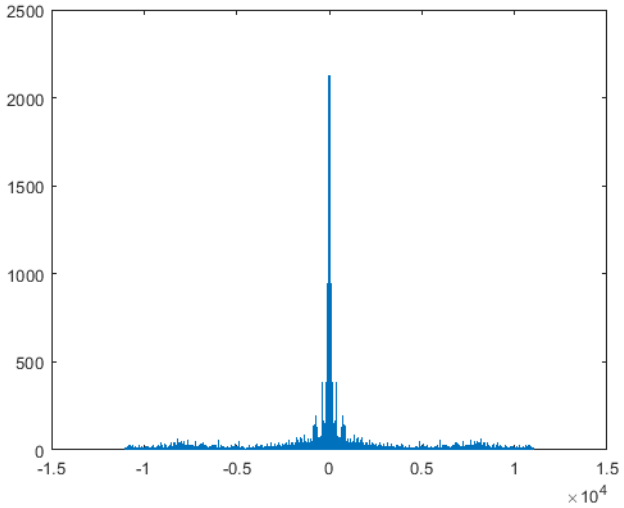


Figura 72: respuesta frecuencial a la salida del filtro FIR

Se puede ver como la componente parásita de los 6 kHz fue eliminada por completo, dando un resultado satisfactorio sobre nuestro filtro diseñado.

Por último, debemos de calcular el **coste computacional** que implica hacer uso del filtro ya que será clave en decidir que filtro usar finalmente, para ello, haremos uso de la función "*tic toc*" de la siguiente manera:

```
tic
y_fir = filter(bandstop, x_fir);
toc
```

Figura 73: función tic toc

Se trata de una función que nos devuelve el tiempo que se ha tardado en ejecutar una parte de código de nuestro script, de manera que obtenemos el siguiente resultado a comparar posteriormente con el del IIR:

```
Elapsed time is 0.131045 seconds.
```

Figura 74: tiempo computo total del filtro FIR

3.3.6 Implementación filtro IIR en MATLAB

De la misma forma que diseñamos el filtro FIR, mediante la aplicación de diseño de filtros, haremos el diseño del filtro IIR.

- Seleccionamos el tipo de filtro: IIR de **banda eliminada**. Dado que deseamos eliminar un ruido acoplado de 6 kHz.
- Seleccionamos tipo de filtro IIR: Butterworth.
- Especificamos frecuencia de muestreo, frecuencias de paso 1 y 2 y frecuencias de corte 1 y 2: 22050 Hz, 5900 Hz, 6100 Hz, 5950 Hz y 6050 Hz respectivamente.
- Aplicamos las especificaciones y observamos la respuesta en frecuencia.

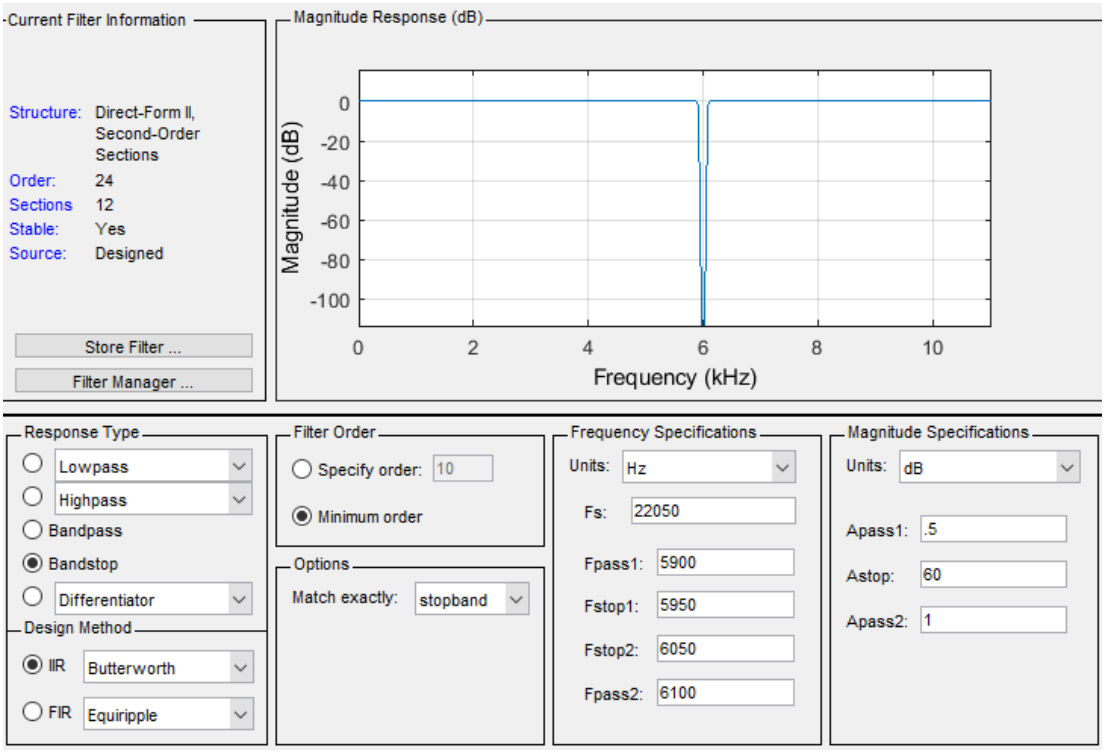


Figura 75: diseño del filtro IIR en MATLAB®

Observamos una respuesta similar a la del FIR, aunque con un orden filtro mucho menor de 24.

El siguiente paso es comprobar el funcionamiento del filtro mediante un script, analizaremos el espectro de la señal de audio entrante observando la componente a eliminar.

Para ello, empleamos la función ***fft*** sobre nuestra señal de entrada y la representamos con respecto a la frecuencia:

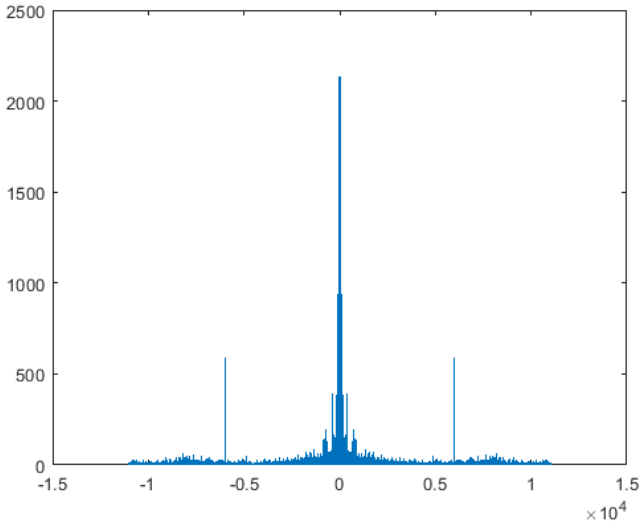


Figura 76: respuesta frecuencial de la entrada al filtro IIR

Observamos claramente que a los 6000 Hz se encuentran las dos componentes intrusas a eliminar.

Haremos uso de nuestra función generada por el diseño del filtro anterior llamada: ***bandstopIIR***, y así como de la propia función nativa de MATLAB®: ***filter***. De modo que ejecutamos la siguiente sentencia y obtenemos la señal de salida filtrada:

>> y_iir = filter(bandstopIIR, x_iir)

Si realizamos el mismo análisis espectral para la señal de salida obtenida observamos lo siguiente:

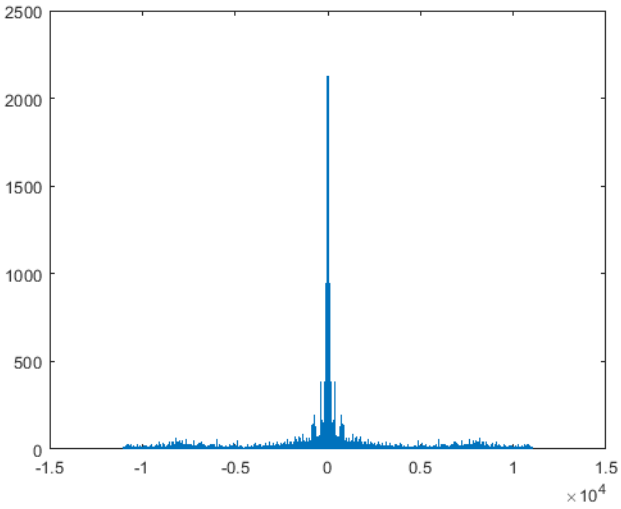


Figura 77: respuesta frecuencial a la salida del filtro IIR

Se puede ver como la componente parásita de los 6 kHz fue eliminada por completo, dando un resultado satisfactorio sobre nuestro filtro diseñado al igual que nuestro FIR.

Por último, debemos de calcular el **coste computacional** de la misma manera

que en el apartado anterior, obteniendo el siguiente resultado:

```
Elapsed time is 0.071149 seconds.
```

Figura 78: tiempo computo total del filtro IIR

La **diferencia de tiempos de cómputo** entre el FIR y el IIR es de **0,06 segundos** (60 ms). Se trata de una diferencia de tiempos muy pequeña como para ser un factor diferencial.

Es por ello, que vistas las comparativas del apartado 3.3.4 se decide elegir el filtro FIR como el filtro que se implementará en nuestro sistema.

3.3.7 Implementación filtro FIR en Vivado

Al finalmente decidir el filtro que emplearemos en nuestro sistema, tendremos que implementar el mismo en nuestro software de Vivado™, de forma que seamos capaces de llevar el modelo visto en MATLAB® a uno sintetizable e implementable en nuestro dispositivo FPGA.

Como vimos en la comparativa del apartado 3.3.4, el filtro FIR está disponible mediante un bloque IP nativo de Vivado™, esto hace que no necesitemos implementar el mismo en HLS primero como los demás bloques del sistema, y que podamos usarlo directamente en nuestro software final como es Vivado™.

Dentro del programa seleccionaremos el IP llamado: **FIR Compiler** para poder configurarlo y realizar un filtro FIR similar al diseñado en el apartado 3.3.5.

Sobre la primera pestaña de configuración deberemos seleccionar la opción de vector, y copiar desde el workspace de MATLAB® los coeficientes generados en el apartado 3.3.5.

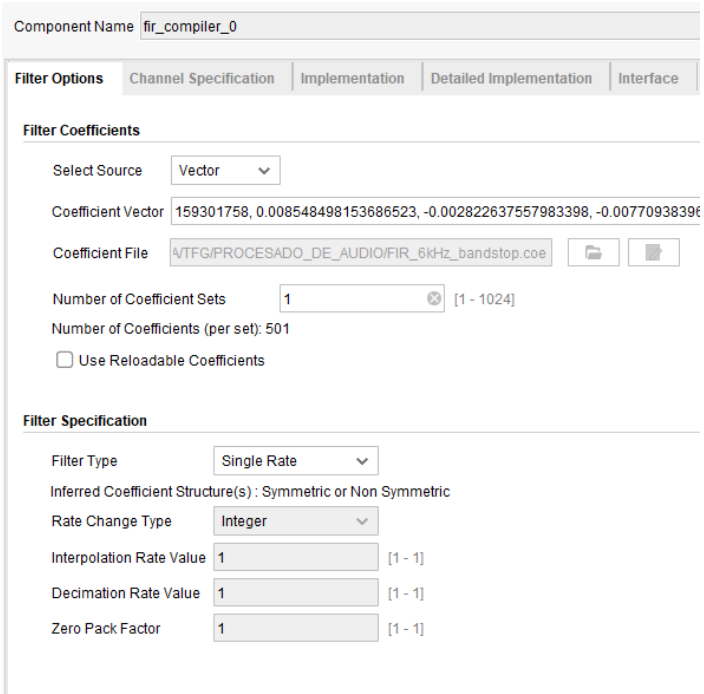


Figura 79: selección opciones filtro FIR en Vivado™

A continuación, especificamos la frecuencia de muestreo del sistema así como la del reloj del propio bloque: 22050 Hz.

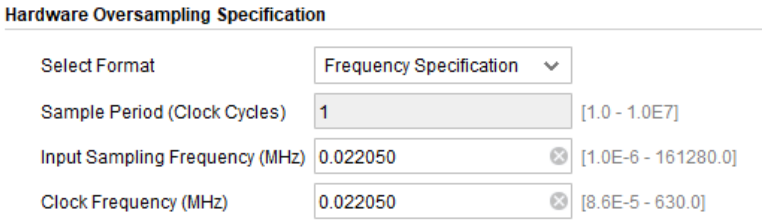


Figura 80: selección frecuencias del filtro FIR Vivado™

Después, indicaremos la anchura de bits de nuestros coeficientes: 24 y 23 de parte decimal, así como su estructura: simétrica, y las anchuras de bits de los datos de entrada: 16 y 14 para la parte decimal.

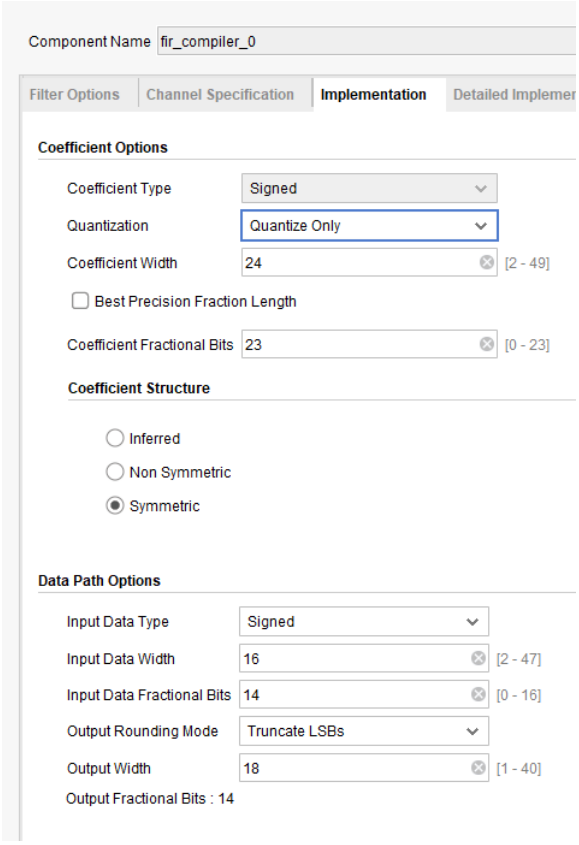


Figura 81: opciones implementación filtro FIR en Vivado™

Al desear obtener tener también a la salida 14 bits de parte decimal, el bloque FIR nos devolverá 18 bits totales en vez de 16 y tendrá 14 bits de parte decimal.

Debemos seleccionar que las señales AXI Stream dispongan de la señal TREADY, de esta manera habrá coherencia entre este bloque y los demás al disponer todos de la señal TREADY tan importante en las transferencias de datos bajo el protocolo AXI4-Stream.

Además, podemos observar una gráfica de la respuesta frecuencial del filtro,

obteniendo una similar a la que pudimos ver en el apartado 3.3.5:

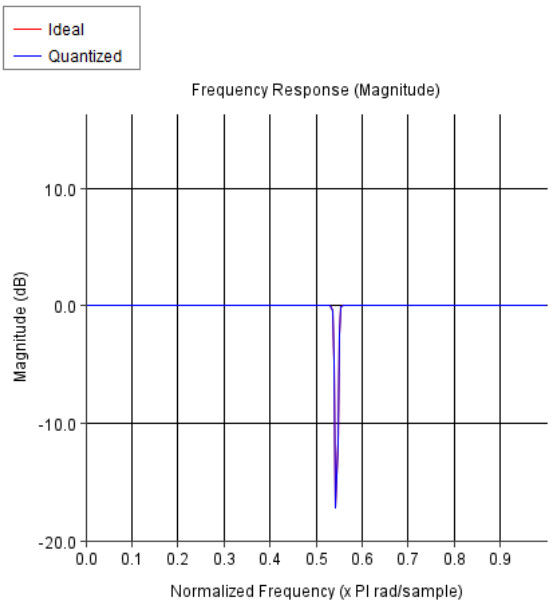


Figura 82: respuesta frecuencial filtro FIR en Vivado™

La utilización de recursos estimada que nos proporciona Vivado™ sobre este bloque se puede ver en la siguiente gráfica:

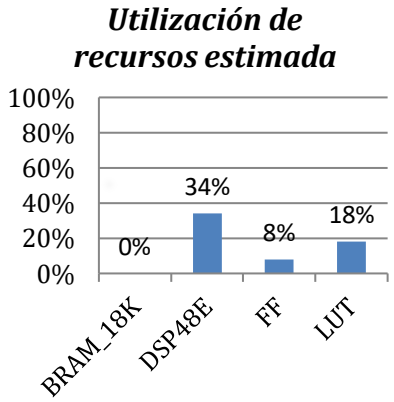


Figura 83: gráfico de la utilización de recursos estimada en el FIR IP-core

El resultado final es el siguiente bloque, disponible para hacer uso de él en nuestro sistema completo:

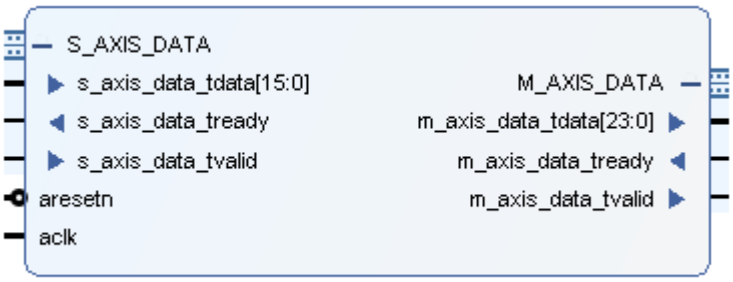


Figura 84: bloque IP-core Filtro FIR

3.4 Banco de efectos de audio

3.4.1 Introducción

Llegamos al bloque final de nuestro sistema, el banco de efectos. Se encuentra en la etapa de post-procesado de audio, la cual englobará una serie de efectos de sonido que serán detallados a continuación.

Cada efecto de audio será un bloque IP que formará parte de nuestro sistema como podemos ver en el esquema del apartado 3.

Se seguirá el mismo procedimiento que vimos en el apartado 3.2 con el bloque compresor, es decir, el diagrama de flujo del capítulo 1. Debido a que serán algoritmos implementados y creados en Vivado HLS™ de la misma forma que se creó el "DRCompressor", por lo que muchas cosas serán obviadas para no reiterar conceptos o procedimientos.

Los efectos que contendrá nuestro banco serán los siguientes como ya mencionamos en la introducción del capítulo 3:

1. **Echo**
2. **Reverb**
3. **Wah – Wah**
4. **Trémolo**

Cada uno de estos efectos está basado en un tipo de efecto de sonido distinto, de modo que se verán diferentes algoritmos de distintos comportamientos, los cuales modificaran y tratarán la señal de audio de modo que consigan generar efectos sonoros característicos de cada algoritmo.

3.4.2 Echo: algoritmo y parámetros

En el apartado 3.1 se hizo una pequeña introducción a cada uno de los efectos, en los siguientes apartados los veremos en detalle, explicando el algoritmo de funcionamiento y los parámetros de cada uno de ellos que hacen que el sistema en sí funcione y genere los efectos sonoros sobre nuestra señal entrante.

En primer lugar, el efecto "Echo". Se trata de un efecto, basado en el **retardo**. Es el efecto más básico de los posibles realizables en cuanto a efectos de retardo. El resultado que se desea obtener es un efecto similar al que se produce en el mundo real cuando hay eco, como el propio nombre del efecto indica [17].

Su algoritmo consiste en retrasar la señal de entrada un **tiempo de delay** determinado y multiplicarlas por un **factor de ganancia**, después esta señal retrasada se mezclará con la señal original dando lugar al audio de salida.

El diagrama de bloques que describe este funcionamiento es el siguiente:

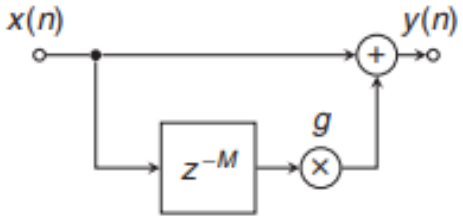


Figura 85: diagrama de bloques efecto Echo [17]

En el diagrama de bloques vemos el algoritmo descrito, donde:

- **M**: número de muestras de retardo aplicadas sobre la señal de entrada, siendo M = td*Fs.
- **Td**: tiempo de delay que queremos aplicar sobre la señal de entrada mayor de 50 ms.
- **G**: factor de ganancia. Valor entre 0 y 1 que indica la intensidad del efecto a provocar entre el 0% y el 100%.

Dando lugar a la siguiente expresión:

$$\boldsymbol{y(n) = (1 - G) * x(n) + G * x(n - M)}$$

A diferencia del diagrama de bloques original, se multiplica a la señal original por la inversa de la ganancia para mantener la señal de salida entre ±1.

3.4.3 Reverb: algoritmo y parámetros

El segundo de los efectos es "Reverb", se trata de un efecto basado en filtros y retardos del cual hay multitud de tipos de "Reverbs" aunque nosotros nos centraremos el efecto básico de ellos. Es un efecto que tiene como objetivo conseguir una **reverberación artificial** basada en la reverberación natural.

La reverberación es aquello que se produce cuando, debido a las características del entorno, el sonido tiene un efecto "espacial" debido a la multitud de reflexiones de onda sonoras que se producen en un lugar cerrado. Además, se crea una cola de sonido debido a la permanencia del mismo una vez el sonido original cesa [17].

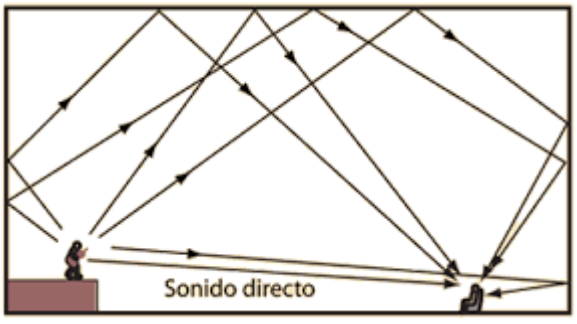


Figura 86: ilustración reverberación en espacio cerrado [22]

Su algoritmo básico consiste en un **filtro paso todo**, el cual deja pasar todas las frecuencias que no modifica el módulo de la respuesta en frecuencia pero si puede modificar la fase de la misma, dependiendo de donde esté el polo del mismo.

Este cambio de fase produce que los ecos producidos por retardos sean aleatorios. Además, sigue una estructura que genera un **delay recursivo** sobre la señal de salida, esto producirá la cola de sonido mencionada con anterioridad. Su diagrama de bloques es el siguiente:

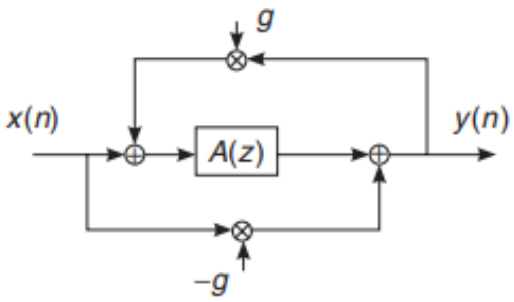


Figura 87: diagrama de bloques efecto Reverb [17]

Como en el anterior algoritmo, tendremos un número de muestras de retardo y un **factor de ganancia**. El bloque "A(z)" simplemente indica que la señal pase por él será retardada **M** muestras dadas por el **tiempo de delay** que será seleccionado por el usuario. Por lo que los parámetros de este efecto son exactamente los mismos que los del efecto anterior.

El diagrama de bloques de nuestro filtro paso todo da lugar a la siguiente expresión que emulará una reverberación natural [17]:

$$\boldsymbol{y(n) = -G * x(n) + x(n-M) + G * y(n-M)}$$

3.4.4 Wah-Wah: algoritmo y parámetros

En tercer lugar, nos encontramos el efecto "Wah-Wah", se trata de un efecto basado en filtros el cual tiene como objetivo emular un pedal de guitarra eléctrica Wah-Wah, como el visto en el apartado 1.1. Pedal el cual produce sobre nuestro audio una señal sonora que dice "wah-wah" como si alguien la estuviera produciendo con su propio habla.

Este efecto se consigue al variar la frecuencia central de un filtro paso banda, es decir, un **filtro paso banda dinámico**. Esta variación de la frecuencia central la conseguiremos mediante un **LFO** (Low Frequency Oscillator), un oscilador de baja frecuencia, el cual dará distintos valores a nuestra frecuencia central al ser este una señal senoidal [17].

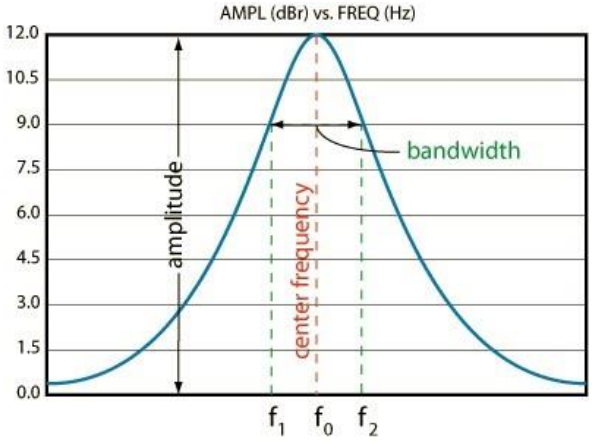


Figura 88: respuesta en frecuencia de filtro paso banda [23]

Su diagrama de bloques es el siguiente:

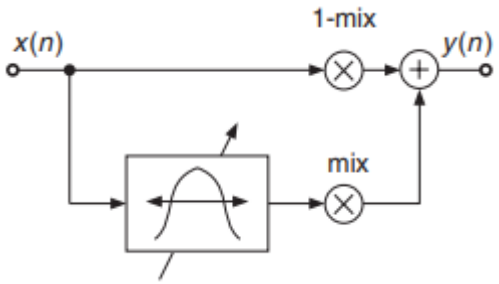


Figura 89: diagrama de bloques efecto Wah-Wah [17]

Como vemos en su diagrama de bloques, la señal obtenida de nuestro filtro paso banda dinámico se multiplica por el **factor de ganancia** ("mix" en este caso) y se mezcla con la señal original dando lugar a la señal de salida.

Para poder elaborar este algoritmo debemos de conocer también la estructura del filtro paso banda, la cual es la siguiente [17]:

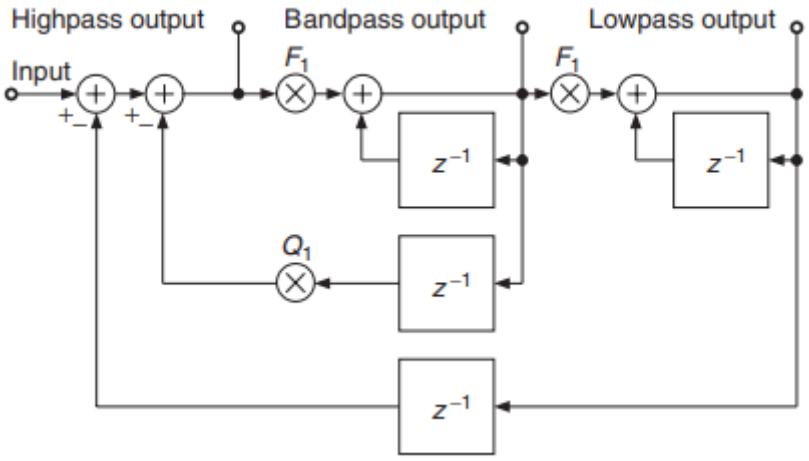


Figura 90: diagrama de bloques filtro paso banda [17]

Las ecuaciones en diferencias que obtenemos del filtro paso banda dinámico son

las siguientes [17]:

Highpass output: $yh(n) = x(n) - yl(n-1) - Q1 * yb(n-1)$

Bandpass output: $yb(n) = F1 * yh(n) + yb(n-1)$

Lowpass output: $\boldsymbol{yl(n) = y_{bandpass}(n) = F1 * yb(n) + yl(n-1)}$

Con todas las estructuras mostradas pasamos a ver los parámetros [17]:

- **G**: factor de ganancia. Valor entre 0 y 1 que indica la intensidad del efecto a provocar entre el 0% y el 100%.

- **Q1**: factor de amortiguamiento del filtro, con un valor de 0,1.

- **F1:** factor que modificará la frecuencia central del filtro:

$$F1 = 2 * sen(\frac{\pi * fc}{Fs})$$

- **fc:** frecuencia central dinámica basada en un LFO:

$$fc = f\_avg + (A0 * sen(2\pi * t * f\_lfo))$$

  - **f_avg**: frecuencia media dada por la frecuencia máxima de corte (f_max) y la mínima (f_min): *f_avg = (f_max + f_min)/2*
  - **A0**: Amplitud del LFO, controla la intensidad del efecto.
  - **f_lfo**: frecuencia del LFO, entre 0,1 Hz y 10 Hz. Controla la velocidad del efecto.

Finalmente obtenemos la señal de salida del algoritmo total a partir de los diagramas de bloques vistos:

$$\boldsymbol{y(n) = (1 - G) * x(n) + G * y_{bandpass}(n)}$$

3.4.5 Trémolo: algoritmo y parámetros

El último de los efectos de audio a desarrollar es el llamado "Trémolo". En esta ocasión, se trata de un efecto basado en la **modulación de amplitud dinámica (AM)**. Se trata de un efecto que busca emular el propio término musical trémolo, el cual indica la variación periódica de la amplitud de una señal de sonido donde la frecuencia de la misma se mantiene constante [26].

Es un efecto que también es usado como pedal para las guitarras eléctricas y que nuestro algoritmo tratará de emular. Esta modificación de la amplitud dinámica se conseguirá mediante un LFO como hemos visto en el efecto anterior. Este LFO atacará a la señal de entrada, produciendo en ella los efectos ondulatorios que generan el efecto

trémolo.

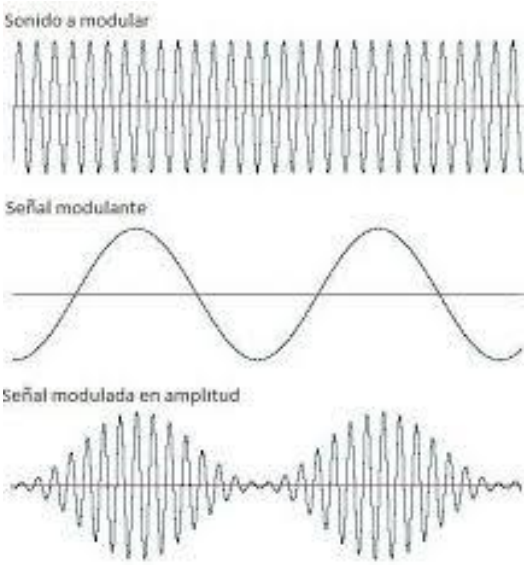


Figura 91: modulación en amplitud [25]

Su diagrama de bloques es simple, la señal de entrada es multiplicada por nuestra señal moduladora basada en un LFO, después por el factor de ganancia y finalmente será mezclada con la señal original.

Los parámetros de nuestro algoritmo son los siguientes [26]:

- **G**: factor de ganancia. Valor entre 0 y 1 que indica la intensidad del efecto a provocar entre el 0% y el 100%.
- **f_lfo**: frecuencia del LFO, entre 0,1 Hz y 10 Hz. Controla la velocidad del efecto.
- **A0**: Amplitud del LFO, controla el nivel de modulación de la señal.
- **Offset:** el inverso de A0, offset que se le añade al LFO para subir su nivel.

Dando lugar a la siguiente expresión:

$$\boldsymbol{y(n) = (1 - G) * x(n) + G * x(n) * LFO(n)}$$

$$LFO(n) = offset + (A0 * sen(2\pi * t * f\_lfo))$$

3.4.6 Implementación de efectos en MATLAB

Siguiendo el diagrama de flujo mostrado en el capítulo 1 en la introducción, el siguiente paso en el proceso de creación bloques IP será la implementación del mismo en MATLAB®, una vez estudiado el algoritmo y su modelo matemático. Por lo que, usaremos la herramienta para simular lo estudiado y comprobar la validez de los 4 efectos mediante representaciones gráficas de los resultados así como de la respuesta sonora.

3.4.6.1 Echo

En primer lugar, leeremos el fichero generado desde el script del filtro FIR, almacenando las muestras en un array, para a continuación definir los parámetros vistos en el apartado 3.4.2, donde el único cambio será que al número de muestras de delay las llamaremos "**D**" en vez de "M". Además, se establece un tiempo de delay de **400 ms** y un factor de ganancia de **0,4**, suficiente para apreciar el efecto de eco.

```
fp=fopen('dataset_out_matlabFIR.txt', 'r');
x_echo=fscanf(fp, '%f');
[N,P]=size(x_echo);

G=0.4; %Entre 0 y 1; 0 -> sin efecto; 1 -> maximo efecto
Fs=22050;
td=400e-3;

D = td*Fs;
```

Figura 92: lectura dataset entrada y declaración variables efecto Echo MATLAB®

Después generamos el bucle donde generamos la señal retrasada las D muestras rellenando un array desplazado D iteraciones con valores de la señal de entrada, como vemos en la imagen, para después obtener la señal de salida según la expresión que vimos en su teoría.

```
x_delay = zeros(1,length(x_echo));
for i = 1:(length(x_echo)-D)
    x_delay(i + D) = x_echo(i);
end
x_delay = transpose(x_delay);

%y(n) = (1-G)*x + G*x(n-D)
y_echo = (x_echo*(1-G)) + (G*x_delay);
sound(y_echo, Fs)
```

Figura 93: ejecución de algoritmo Echo MATLAB®

3.4.6.2 Reverb

Al igual que antes, leeremos el fichero generado desde el script del filtro FIR, almacenando las muestras en un array, para a continuación definir los parámetros vistos en el apartado 3.4.3.

```
fp=fopen('dataset_out_matlabFIR.txt', 'r');
x_reverb=fscanf(fp, '%f');
[N,P]=size(x_reverb);

Fs=22050;
td = 400e-3;
G = 0.4; %Entre 0 y 1; 0 -> sin efecto; 1 -> maximo efecto

D = td*Fs;
```

Figura 94: lectura dataset entrada y declaración variables efecto Reverb MATLAB®

Se ha escogido un tiempo de delay y un factor de ganancia iguales a los de Echo. Además, queremos escuchar la cola de sonido que deja nuestro efecto explicada en el apartado 3.4.3. Para ello, a partir de un número de muestras llamado "**N_stop"** silenciaremos la señal de entrada poniéndola a 0 una vez se supere N_stop.

```
N_stop = 80000;
for i = N_stop:N
    x_reverb(i) = 0;
end
```

Figura 95: silencio de la señal de entrada efecto Reverb

Para elaborar la señal de salida dada por la expresión del algoritmo vista, generaremos un bucle de muestras donde se retrasen la señal de entrada y de salida D muestras.

```
for i = 1:N
    x_delay(i + D) = x_reverb(i);

        y_reverb(i) = (-G*x_reverb(i)) + x_delay(i) + (G*y_delay(i));

    y_delay(i + D) = y_reverb(i);
end
```

Figura 96: ejecución de algoritmo Reverb MATLAB®

3.4.6.3 Wah-Wah

En tercer lugar con el Wah-Wah, leeremos las muestras del fichero y declararemos los primeros parámetros necesarios para la implementación. En esta ocasión hemos elegido un factor de ganancia del 0,75 para tener una mayor intensidad del efecto.

```
fp=fopen('dataset_out_matlabFIR.txt', 'r');
x_wah=fscanf(fp, '%f');
[N,P]=size(x_wah);

Fs=22050;
G = 0.75; %Entre 0 y 1; 0 -> sin efecto; 1 -> maximo efecto
Q1 = 0.1;
```

Figura 97: lectura dataset entrada y declaración variables efecto Wah-Wah MATLAB®

El siguiente paso será crear el LFO que implemente la variación dinámica de la frecuencia central de nuestro filtro paso banda, como vimos en el apartado 3.4.4. Se ha escogido un valor de 1 Hz para la frecuencia del LFO, 3000 Hz para la frecuencia de corte máxima y 300 Hz para la mínima, así como un valor de A0 de 1350 el cual dará un valor a la senoide suficiente para notar los efectos "wah-wah".

```
%LFO (low freq oscilator)
f_lfo=1;
A0 = 1350;

fmax=3000;
fmin=300;

favg = (fmax + fmin)/2;
fc = favg + (A0*sin(2*pi*f_lfo*t)); %frec central del filtro paso banda
```

Figura 98: creación del LFO Wah-Wah MATLAB®

Por último, podemos implementar el bucle que genere las señales dadas por el filtro paso banda, además de crear el factor F1 que irá variando según varía nuestra señal LFO de la frecuencia central.

```
for i = 1:N

        F1(i) = 2*sin(pi*(fc(i)/Fs));
        yh(i) =  x_wahwah(i)  - y_bp_delay(i) - (Q1*yb_delay(i));
        yb(i) = (F1(i)*yh(i)) + yb_delay(i);
        y_bandpass(i) = (F1(i)*yb(i)) + y_bp_delay(i);

        y_wahwah(i) = (G*y_bandpass(i)) + ((1-G)*x_wah(i));

        yb_delay(i+1) = yb(i);
        y_bp_delay(i + 1) = y_bandpass(i);
end
```

Figura 99: ejecución del algoritmo Wah-Wah MATLAB®

Se generan las señales vistas en el apartado 3.4.4 así como la señal de salida final "*y_wahwah*". En nuestro caso, se modificó el nombre de la señal "*yl*" por "*y_bandpass*" para dejar claro que esa señal es la salida del filtro paso banda.

3.4.6.4 Trémolo

Por último, implementaremos el efecto Trémolo. Leeremos las muestras del fichero y declararemos los parámetros iniciales. Habiendo elegido una vez más un factor de ganancia del 0,75 para apreciar más el efecto causado en nuestra señal de entrada.

```
fp=fopen('dataset_out_matlabFIR.txt', 'r');
x_tremolo=fscanf(fp, '%f');
[N,P]=size(x_tremolo);

G = 0.75;
Fs=22050;
```

Figura 100: lectura dataset entrada y declaración variables efecto Tremolo MATLAB®

A continuación se genera el LFO que generará la modulación de amplitud en nuestra señal original, donde se ha escogido una frecuencia un poco mayor de 3 Hz con respecto a ocasiones anteriores, así como un valor de A0 de 0,5. El valor de A0 es completamente en relación al visto en el LFO del Wah-Wah debido a que este se aplica directamente sobre la señal de entrada mientras que en el anterior se generaba el factor F1 previamente. Además, para generar el LFO usamos un array llamado "t" el cual es generado con la función "linespace", el cual contiene los valores temporales para la creación del seno.

```
%LFO (low freq oscilator) -> señal moduladora
f = 3;      %velocidad de oscilación de la señal moduladora [0.1 - 10 Hz]
A0 = 0.5;   %intensidad de la señal moduladora que determinará cuanto se modulará la señal
offset = 1 - A0;
LFO = offset + (A0*sin(2*pi*f*t)); %f = 2Hz maximo; f = 0.01 Hz mínimo
```

Figura 101: creación del LFO Tremolo MATLAB®

Finalmente ejecutamos la expresión vista en el apartado 3.4.5 la cual mezcla la señal original con el producto de la señal moduladora y la señal original.

```
y_tremolo = (LFO.*x_tremolo*G) + ((1-G)*x_tremolo);
```

Figura 102: ejecución del algoritmo Tremolo MATLAB®

3.4.7 Resultados en MATLAB

A continuación, comprobaremos mediante resultados gráficos y sonoros si nuestro código ha sido realizado con éxito o no. Seguiremos los pasos realizados en el apartado 3.2.5 donde se creó el bloque del compresor, donde se vio la función empleada para reproducir audio.

La señal de entrada que tendremos en el origen de cada efecto la tendremos como referencia para realizar comparaciones con la salida, siendo la siguiente:

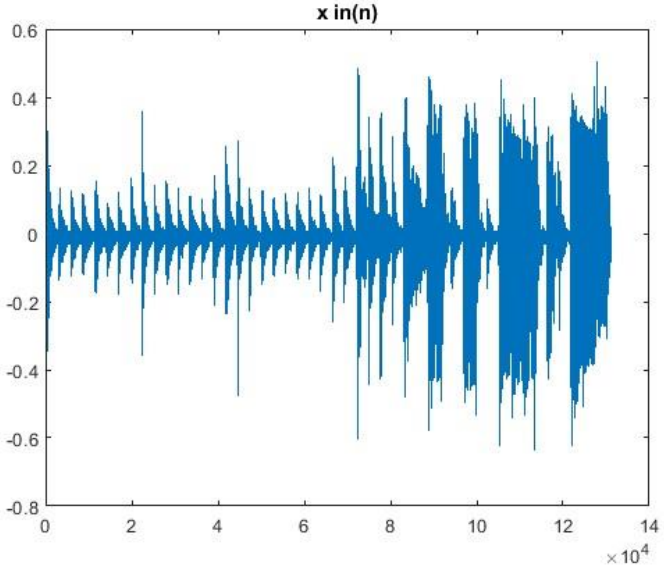


Figura 103: gráfica de la señal de entrada a los efectos MATLAB®

En primer lugar se representará la respuesta obtenida por el efecto **Echo**, además de escucharla, corroborando que es lo que esperábamos.

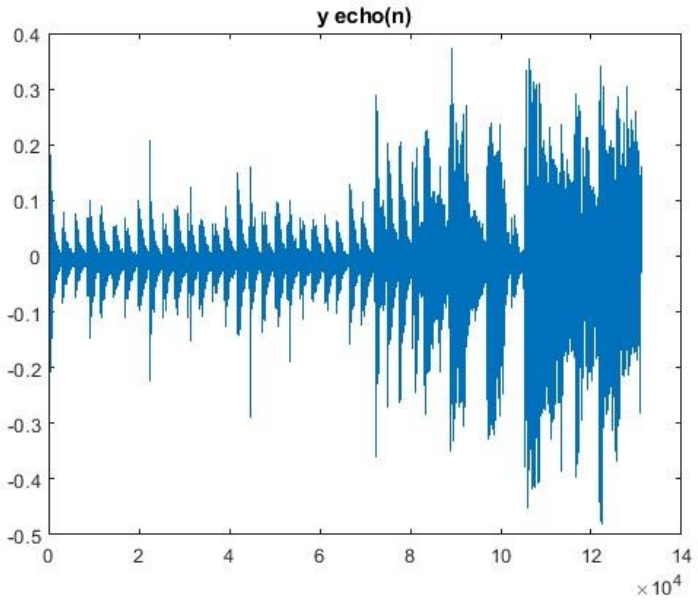


Figura 104: gráfica de la señal de salida de Echo MATLAB®

Podemos ver en la salida como al mezclarse la señal original con la misma retrasada D muestras, tenemos más grosor de sonido, es decir, menos sonidos bajos, ya que esos sonidos bajos ahora son ocupados por los ecos que produce el algoritmo.

En segundo lugar se representará la respuesta obtenida por el efecto **Reverb**, además de escucharla, corroborando que es lo que esperábamos.

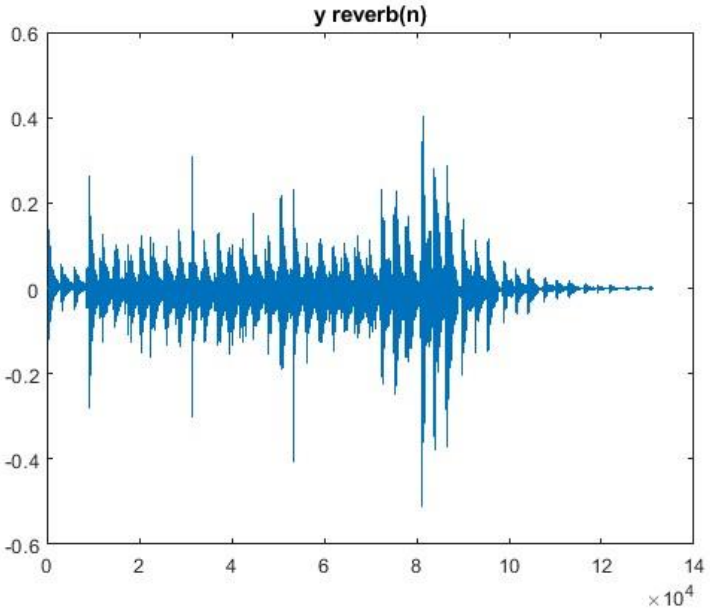


Figura 105: gráfica de la señal de salida de Reverb MATLAB®

En este caso podemos escuchar las reverberaciones creadas por el efecto hasta las 80000 muestras, momento en el cual se silencia la señal de entrada para poder apreciar la cola de sonido que produce el efecto.

Como vemos en la figura, a partir de "N_stop" (80000 muestras) únicamente se reproducen las muestras retroalimentadas en la salida, por lo que se queda una cola de sonido de los últimos momentos reproducidos, siendo cada vez menor hasta que tenemos 0 en la salida.

En tercer lugar se representará la respuesta obtenida por el efecto **Wah-Wah**, además de escucharla, corroborando que es lo que esperábamos.

Antes de ver la salida, comprobaremos si la señal LFO de la frecuencia central es correcta así como el factor F1 generado a partir de la anterior.

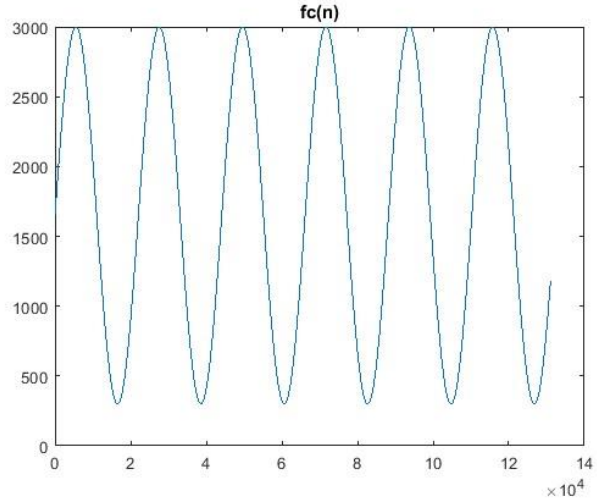


Figura 106: gráfica del LFO de la frecuencia central Wah-Wah MATLAB®

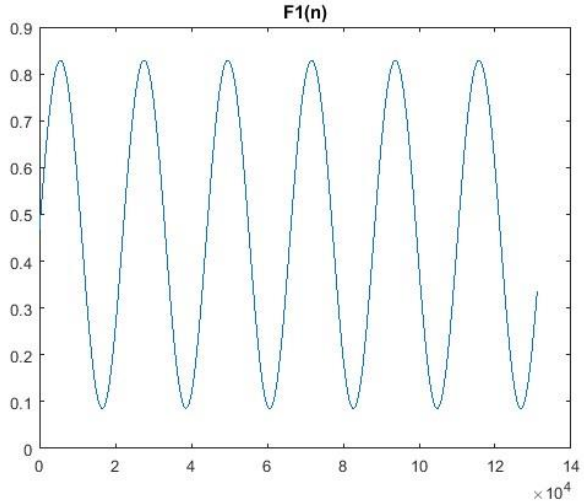


Figura 107: gráfica del factor F1 Wah-Wah MATLAB®

Ambas sinusoides son correctas y lo esperado cuando detallamos los parámetros de cada una. Mientras que la señal de salida obtenida es:

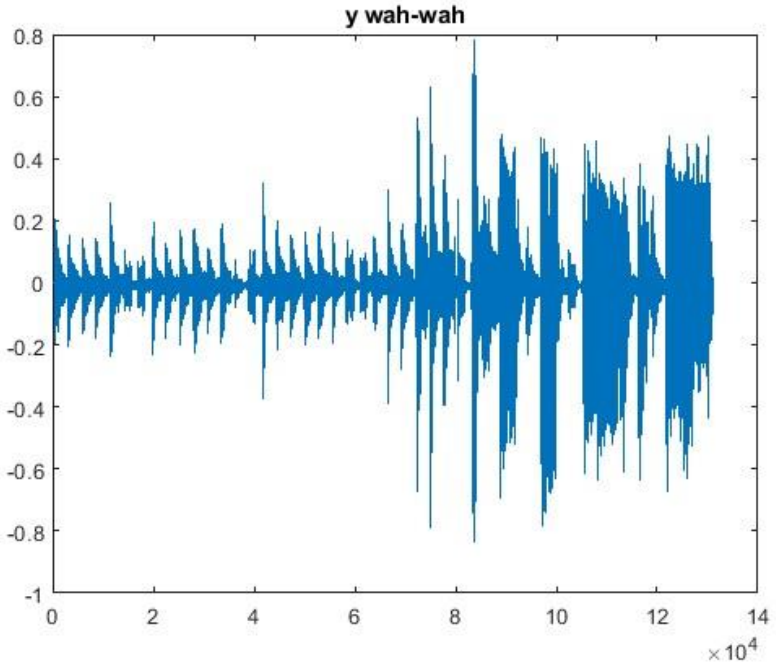


Figura 108: gráfica de la señal de salida de Wah-Wah MATLAB®

Al reproducir la salida se aprecia en su totalidad el efecto tan característico Wah-Wah, mezclándose con la señal original y generando ese sonido definido por los parámetros y expresiones vistos.

Por último, se representará la respuesta obtenida por el efecto **Trémolo,** además de escucharla, corroborando que es lo que esperábamos.

Antes de ver la salida, comprobaremos si la señal LFO que modulara la amplitud de la señal de entrada es correcta.

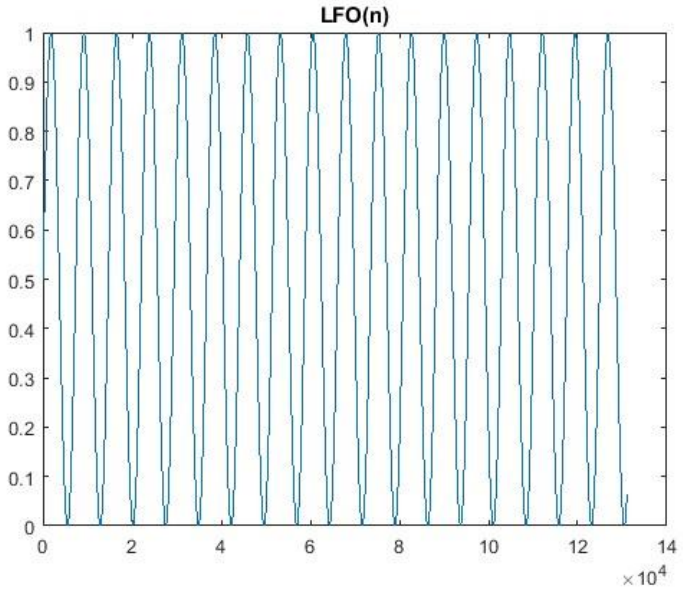


Figura 109: gráfica del LFO modulador Trémolo MATLAB®

La sinusoide es correcta y lo esperado cuando detallamos los parámetros de la misma, siendo esta más rápida que el anterior LFO al tener una frecuencia mayor (3 Hz). Mientras que la señal de salida obtenida es:

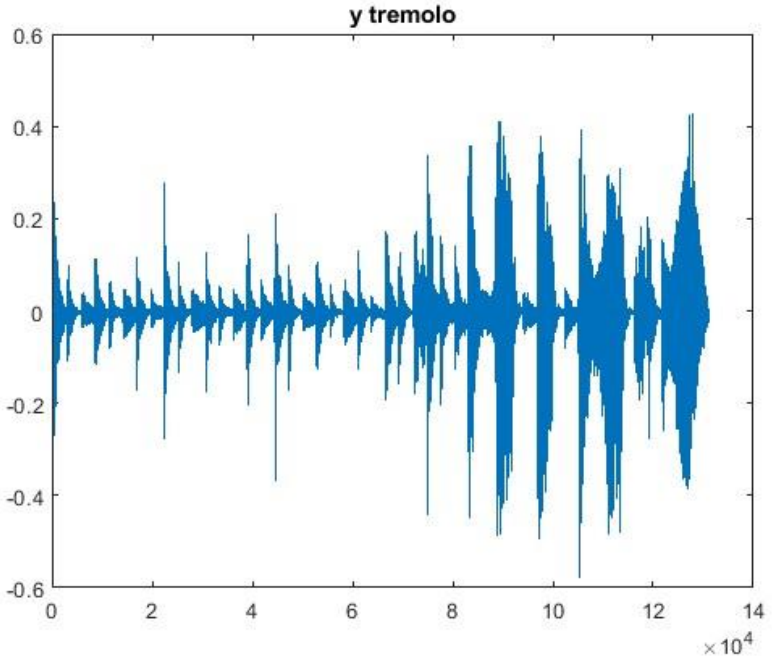


Figura 110: gráfica de la señal de salida de Trémolo MATLAB®

Se puede ver en la salida los cambios de amplitud modulados por la sinusoide de baja frecuencia en la forma de la señal, lo cual se aprecia al reproducir el audio.

Siguiendo el diagrama de flujo dado en el capítulo 1, podemos continuar con el mismo al haber obtenido resultados positivos por parte de nuestros bloques en MATLAB®.

3.4.8 Implementación de efectos en Vivado HLS

Para realizar la implementación en HLS de nuestro banco de efectos seguiremos los mismos pasos vistos en el apartado 3.2.5. Donde se creó el bloque DRCompressor en HLS explicando paso a paso todo el proceso hasta obtener el bloque final, por lo que muchas cosas ya vistas se obviarán en este apartado.

El objetivo en este apartado es conseguir los 4 bloques IP correspondientes a cada efecto visto, para ello, codificaremos los algoritmos de la forma más similar a la realizada en MATLAB®. Cada efecto o proyecto tendrá su librería, su función TOP y el testbench que ejecute la simulación deseada. Después realizaremos la síntesis, obteniendo el algoritmo codificado en VHDL para posteriormente simularlo en este mismo lenguaje y obtener el conjunto de ficheros correspondientes al bloque IP generado.

Nota importante*: Como vimos en sus apartados teóricos, los efectos **Echo** y **Reverb** hacen uso de retardos o delays, en los cuales, dado un número de muestras de retardo dado, se retrasan señales. La forma de implementar estos retardos en C es almacenando en un array la totalidad de las muestras, pudiendo acceder a ellas cuando necesitemos.

En nuestro caso disponemos de un audio de prueba de 131072 muestras de 16 bits cada una, esto da lugar a **2098 Kbits** de uso de memoria por cada array empleado y cada FPGA tiene una cantidad de memoria limitada, siendo la de nuestra xc7a200t de **2888 Kbits** de RAM máxima distribuida. El uso de estos arrays de muestras es inviable ya que consumiríamos demasiada memoria del dispositivo.

Es por ello, que se buscaron alternativas a esta implementación por medio de memorias externas como las **FIFO** (First In, First Out) como veremos más adelante, las cuales se comunicarán con nuestro bloque entregando estas muestras retrasadas cuando sea necesario.

3.4.8.1 Echo

Los archivos empleados en la implementación de Echo son: "**Echo.cpp", "Echo.h"** y **"Echo_tb.cpp"**.

En nuestra librería "**Echo.h**" declararemos los parámetros vistos anteriormente que definen el algoritmo del mismo:

```
ap_fixed<16,16> D;

      ap_fixed<47,18> n=0;
      ap_fixed<16,16> Fs = 22050;
      ap_fixed<22,1> td = 0.4; //400ms
      ap_fixed<22,1> G = 0.4; //Ganancia; G=0 -> sin efecto; G=1 -> máx efecto
```

Figura 111: parámetros Echo Vivado HLS™

Se declaran los distintos parámetros así como la variable D, que contendrá el número de muestras de retraso calculado mediante el producto de Fs y td.

A la hora de definir nuestra función TOP ("**Echo.cpp"**) debemos tener en cuenta la **Nota importante***, ya que tendremos que crear los siguientes puertos extra a parte de las señales de entrada y salida:

- ***x:*** señal que contendrá el valor del dato de entrada en cada caso.
- ***y:*** señal que contendrá el valor del dato de salida en cada caso.
- ***x_delay:*** señal de entrada que contendrá el valor del dato ***x*** retrasado ***D* muestras** en cada caso (proveniente de la FIFO).
- ***EN_delay:*** señal de salida booleana que indicará el momento en el que habilitamos la FIFO para obtener las muestras retrasadas.

```
void Echo(ap_fixed<16,2> x, ap_fixed<16,2> x_delay, bool *EN_delay, ap_fixed<16,2> *y)
```

Figura 112: declaración función Echo Vivado HLS™

El desarrollo de la función TOP consistirá en calcular el valor de *D*, y después preguntar si el número de muestras ha superado *D*, en caso de que si, se activará el Enable de delay (*EN_delay*), indicando que la señal *x(n-D)* comienza a tener valores no nulos. Con *EN_delay* activamos la FIFO que contiene la señal *x(n)* almacenada y estos valores irán entrando en el bloque para ser utilizados por la expresión de la señal de salida estudiada que vemos en la figura. El cuenteo de muestras se hará mediante un contador en la variable *n*.

```
  D = td*Fs; //delay fijo
  if(n >= D)
      *EN_delay = 1;
  else
      *EN_delay = 0;

// y(n) = (1-G)*x(n) + G*x(n-D) //
*y = ((1-G)*x) + (G*x_delay);
////////////////////////////////
  n++;
```

Figura 113: ejecución algoritmo Echo Vivado HLS™

Para corroborar el comportamiento del código descrito elaboraremos el testbench en nuestro fichero **"Echo_tb.cpp"**. En él, se empleará la misma estructura de código que el visto en el 3.2.5, donde se lea las muestras de entrada del fichero

"***dataset_in_echo.txt***" y se escriban en "***dataset_out_echo.txt***", donde posteriormente comprobaremos su validez.

"***dataset_in_echo.txt***" provendrá de la salida de nuestro bloque FIR en MATLAB®, conteniendo las muestras obtenido por esta etapa anterior.

Puesto que se trata de un banco de pruebas, la señal *x_delay* será emulada, dándole el valor de la señal *x* retrasada D muestras cuando superamos dicho número de iteraciones.

```
int main()
{
    int D = 8820;
        ifstream in_file("dataset_in_echo.txt");     //abrimos en modo lectura el dataset
        if (!in_file) {
            printf("Can't open data in file\n");
            exit(0);
        }
        ofstream out_file("dataset_out_echo.txt");  //abrimos en modo escritura el dataset
        if (!out_file) {
            printf("Can't open data out file\n");
            exit(0);
        }

        for(long i=0; i<n_samples; i++){
            in_file >> x_n[i];

            if(i >= D )
                x_n_delay[i] = x_n[i - D];
            else
                x_n_delay[i] = 0;

            Echo(x_n[i], x_n_delay[i], &EN_delay, &y_n[i]); //Ejecutamos algoritmo

            out_file << y_n[i] << "\n";
        }
        in_file.close();
        out_file.close();
}
```

Figura 114: función main testbench Echo Vivado HLS™

3.4.8.2 Reverb

Los archivos empleados en la implementación de Reverb son: "**Reverb.cpp", "Reverb.h"** y **"Reverb_tb.cpp"**.

En nuestra librería "**Reverb.h**" declararemos los parámetros vistos anteriormente que definen el algoritmo del mismo:

```
#define n_mute 80000
ap_fixed<16,16> D;

      ap_fixed<47,18> n=0;
      ap_fixed<16,16> Fs = 22050;
      ap_fixed<22,1> td = 0.4; //400ms
      ap_fixed<22,1> G = 0.4; //Ganancia; G=0 -> sin efecto; G=1 -> máx efecto
```

Figura 115: parámetros Reverb Vivado HLS™

Mismos parámetros que en Echo pero con el añadido de "***n_mute***" la cual como vimos en apartados anteriores, corresponde con el número de muestras a partir del cual la señal de entrada se silenciará para notar el efecto de la cola de sonido que deja nuestro algoritmo.

A la hora de definir nuestra función TOP ("**Reverb.cpp"**) debemos tener en cuenta la **Nota importante***, ya que tendremos que crear los siguientes puertos extra a parte de las señales de entrada y salida:

- ***x:*** señal que contendrá el valor del dato de entrada en cada caso.
- ***y:*** señal que contendrá el valor del dato de salida en cada caso.
- ***x_delay:*** señal de entrada que contendrá el valor del dato ***x*** retrasado ***D* muestras** en cada caso (proveniente de la FIFO).
- ***y_delay:*** señal de entrada que contendrá el valor del dato ***y*** retrasado ***D* muestras** en cada caso (proveniente de la FIFO).
- ***y_feedback:*** señal de salida que contendrá el valor del dato de salida inmediato.
- ***EN_delay:*** señal de salida booleana que indicará el momento en el que habilitamos la FIFO para obtener las muestras retrasadas.

```
void Reverb(ap_fixed<16,2> x, ap_fixed<16,2> x_delay, bool *EN_delay, ap_fixed<16,2> *y, ap_fixed<16,2> y_delay, ap_fixed<16,2> *y_feedback)
```

Figura 116: declaración función Reverb Vivado HLS™

En este caso, al necesitar también la señal de salida retrasada, guardaremos esos valores de salida en la FIFO a través de la señal ***y_feedback*** de modo que cuando *EN_delay = '1' y_delay* obtendrá dichos valores retrasados para ser usados en la expresión dada por el filtro paso todo en el que está basado el algoritmo.

```
D = td*Fs; //delay fijo

if(n >= D)
    *EN_delay = 1;
else
    *EN_delay = 0;

if(n > n_mute){
    x = 0;
}
if(n > (n_mute+D)){
    x_delay = 0;
}
//Filtro Paso Todo: y(n) = -G*x(n) + x(n-D) + G*y(n-D)
*y = (-G*x) + x_delay + (G*y_delay);
*y_feedback = *y;

n++;
```

Figura 117: ejecución algoritmo Reverb Vivado HLS™

Para corroborar el comportamiento del código descrito elaboraremos el testbench en nuestro fichero **"Reverb_tb.cpp"**. En él, se empleará la misma estructura de código que el visto en el 3.2.5, donde se lea las muestras de entrada del fichero

"***dataset_in_reverb.txt***" y se escriban en "***dataset_out_reverb.txt***", donde posteriormente comprobaremos su validez.

"***dataset_in_reverb.txt***" provendrá de la salida de nuestro bloque FIR en MATLAB®, conteniendo las muestras obtenido por esta etapa anterior.

Puesto que se trata de un banco de pruebas, la señal *x_delay* e *y_delay* serán emuladas, dándoles los valores de las señales *x* e *y* retrasadas D muestras cuando superamos dicho número de iteraciones.

```
int main()
{
    int D = 8820;
        ifstream in_file("dataset_in_reverb.txt");          //abrimos en modo lectura el dataset
        if (!in_file) {
            printf("Can't open data in file\n");
            exit(0);
        }
        ofstream out_file("dataset_out_reverb.txt");            //abrimos en modo escritura el dataset
        if (!out_file) {
            printf("Can't open data out file\n");
            exit(0);
        }

        for(long i=0; i<n_samples; i++){
            in_file >> x_n[i];

            if(i > 80000)
                x_n[i] = 0;

            if(i >= D ){
                x_n_delay[i] = x_n[i - D];
                y_n_delay[i] = y_n[i - D];
            }
            else{
                x_n_delay[i] = 0;
                y_n_delay[i] = 0;
            }

            Reverb(x_n[i], x_n_delay[i], &EN_delay, &y_n[i], y_n_delay[i], &y_n_feedback[i]); //Ejecutamos algoritmo

            out_file << y_n[i] << "\n";
        }
        in_file.close();
        out_file.close();
}
```

Figura 118: función main testbench Reverb Vivado HLS™

3.4.8.3 Wah-Wah

Los archivos empleados en la implementación de Wah-Wah son: "**WahWah.cpp**", "**WahWah.h**" y "**WahWah_tb.cpp**".

En nuestra librería "**WahWah.h**" declararemos los parámetros vistos anteriormente así como las señales internas que definen el algoritmo del mismo:

```
ap_fixed<24,3> pi = 3.141592;

ap_fixed<32,13> fc;
ap_fixed<24,1> F1;
ap_fixed<16,2> yh, yb, yb_delay, y_bandpass, y_bp_delay;

      ap_fixed<47,18> n=0;
      ap_fixed<16,16> Fs = 22050;
      ap_fixed<3,1> G = 0.75; //Ganancia; G=0 -> sin efecto; G=1 -> máx efecto
```

Se dará a cada señal el número de bits necesarios para garantizar obtener todo el rango de valores en el que se moverá dicha señal. Seguidamente, declaramos las variables correspondientes al LFO:

```
int f_lfo = 1; //freq del LFO (low freq oscillator)
ap_fixed<24,4> omega = 2*pi*f_lfo;

int A0 = 1350;
ap_fixed<13,13> favg = (3000 + 300)/2; //fmedia=(fmax+fmin)/2
ap_fixed<16,1> Q = 0.1;

ap_fixed<32,3> math_aux;
ap_fixed<32,1> math_aux2;
ap_fixed<32,6> alpha;
ap_fixed<32,1> beta;
```

Figura 119: parámetros Wah-Wah Vivado HLS™

Destacar que las variables *math_aux* y *math_aux2* son variables auxiliares necesarias para el cálculo de otras en la función TOP como vimos en el apartado 3.2.5.

Además, *alpha* y *beta* son las variables que actuarán como argumentos de las dos señales seno que albergan en *F1* y *fc*.

La función TOP ("**WahWah.cpp"**) únicamente tendrá las señales de entrada y salida como argumento:

- ***x:*** señal que contendrá el valor del dato de entrada en cada caso.
- ***y:*** señal que contendrá el valor del dato de salida en cada caso.

```
void WahWah(ap_fixed<16,2> x, ap_fixed<16,2> *y)
```

Figura 120: declaración función Wah-Wah Vivado HLS™

En el desarrollo de la función TOP se hará, a través de *math_aux* el cálculo de *alpha*, siendo esta variable el argumento del seno del LFO de nuestra frecuencia central dinámica. Siendo *math_aux* la señal de tiempo del seno la cual multiplicamos por 2*pi*f (*omega*).

De la misma forma con *beta*, la cual será el argumento del seno del factor *F1* ayudado por *math_aux2*. Finalmente, desarrollaremos las distintas expresiones vistas en apartados anteriores dando lugar a la señal de salida final *y*.

```
  math_aux = n/Fs;
  alpha = omega*math_aux;

  fc = favg + (A0*(hls::sin(alpha))); //LFO

  math_aux2 = fc/Fs;
  beta = pi*math_aux2;
  F1 = 2*(hls::sin(beta));

yh = x - y_bp_delay - (Q*yb_delay);
yb = (F1*yh) + yb_delay;
y_bandpass = (F1*yb) + y_bp_delay;

// y(n) = (1-G)*x(n) + G*y_bandpass(n)
*y = ((1-G)*x) + (G*y_bandpass);
////////////////////////////////////////

  yb_delay = yb;
  y_bp_delay = y_bandpass;
  n++;
```

Figura 121: ejecución algoritmo Wah-Wah Vivado HLS™

Para corroborar el comportamiento del código descrito elaboraremos el testbench en nuestro fichero **"WahWah_tb.cpp"**. En él, se empleará la misma estructura de código que el visto en el 3.2.5 por lo que no se destaca nada del mismo al no tener ninguna distinción notoria.

3.4.8.4 Trémolo

Los archivos empleados en la implementación de Trémolo son: "**Tremolo.cpp", "Tremolo.h"** y **"Tremolo_tb.cpp"**.

En nuestra librería "**Tremolo.h**" declararemos los parámetros vistos anteriormente así como las señales internas que definen el algoritmo del mismo, de la misma forma que hicimos en "**WahWah.h**" con las variables auxiliares para el LFO:

```
ap_fixed<24,3> pi = 3.141592;
ap_fixed<16,2> LFO;

      ap_fixed<47,18> n=0;
      ap_fixed<16,16> Fs = 22050;
      ap_fixed<3,1> G = 0.75; //Ganancia; G=0 -> sin efecto; G=1 -> máx efecto

      int f = 3; //freq del LFO (low freq oscillator) //0.1 - 10 Hz
      ap_fixed<2,1> A0 = 0.5;
      ap_fixed<2,1> offset = 1-A0;

      ap_fixed<24,6> omega = 2*pi*f;
      ap_fixed<32,5> math_aux;
      ap_fixed<32,9> alpha;
```

Figura 122: parámetros Trémolo Vivado HLS™

La función TOP ("**Tremolo.cpp"**) únicamente tendrá las señales de entrada y salida como argumento:

- ***x:*** señal que contendrá el valor del dato de entrada en cada caso.
- ***y:*** señal que contendrá el valor del dato de salida en cada caso.

```
void Tremolo(ap_fixed<16,2> x, ap_fixed<16,2> *y)
```

Figura 123: declaración función Trémolo Vivado HLS™

En el desarrollo de la función TOP se hará, a través de *math_aux* el cálculo de *alpha*, siendo esta variable el argumento del seno de la señal LFO moduladora. Siendo *math_aux* la señal de tiempo del seno la cual multiplicamos por 2*pi*f (*omega*).

```
    math_aux = n/Fs;
    alpha = omega*math_aux;

    LFO = (A0*(hls::sin(alpha))) + offset; //LFO -> amplitud dinámica

  *y = ((1-G)*x) + (G*x*LFO);
   n++;
```

Figura 124: ejecución algoritmo Trémolo Vivado HLS™

Para corroborar el comportamiento del código descrito elaboraremos el testbench en nuestro fichero **"Tremolo_tb.cpp"**. En él, se empleará la misma estructura de código que el visto en el 3.2.5 por lo que no se destaca nada del mismo al no tener ninguna distinción notoria.

3.4.9 Resultados en Vivado HLS

Siguiendo con el diagrama de flujo descrito en el capítulo1, el siguiente paso será realizar la simulación C de nuestros 4 algoritmos.

Para ello, lanzaremos las simulaciones C en Vivado HLS™, de forma que al finalizar el programa nos lanzará un mensaje informándonos sobre que no hubo errores en la simulación y que la misma ya finalizó.

El siguiente paso, será leer en MATLAB® los ficheros de salida "dataset_out_echo.txt", "dataset_out_reverb.txt", "dataset_out_wahwah.txt" y "dataset_out_tremolo.txt" obtenidos de Vivado HLS™, reproducir sonoramente las muestras y representar gráficamente las mismas.

Los 4 audios reproducidos son similares a los reproducidos en el apartado 3.4.6.5 y las representaciones gráficas comparadas con las obtenidas en el apartado 3.4.6.5 son las siguientes:

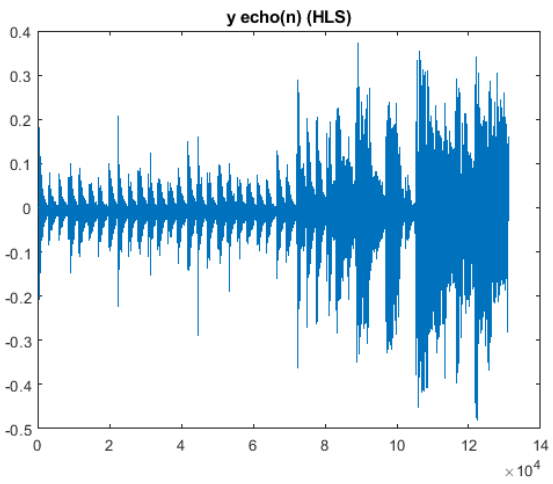


Figura 126: gráfica señal de salida Echo Vivado HLS™

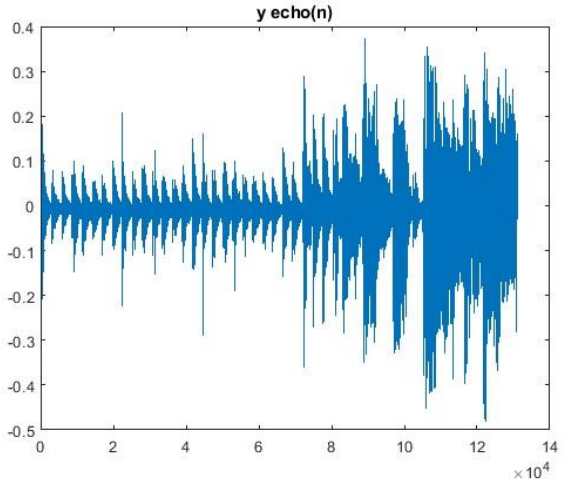


Figura 125: gráfica señal de salida Echo MATLAB®

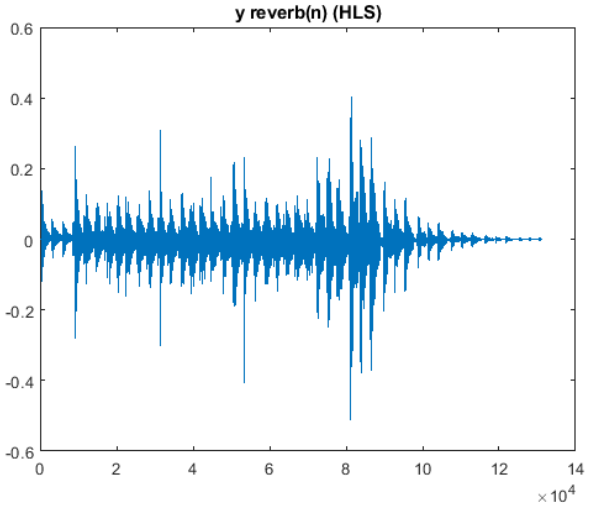


Figura 127: gráfica señal de salida Reverb Vivado HLS™

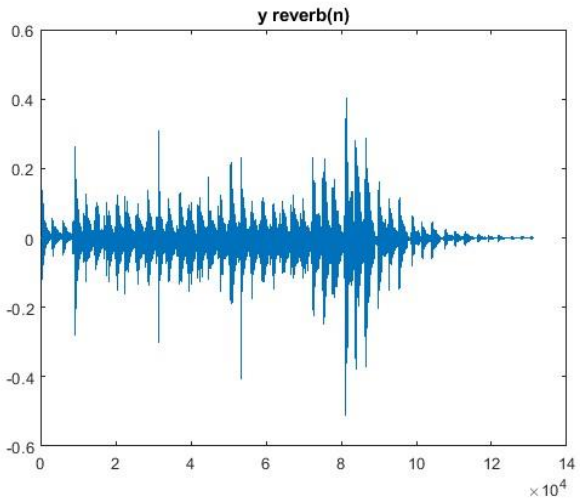


Figura 128: gráfica señal de salida Reverb MATLAB®

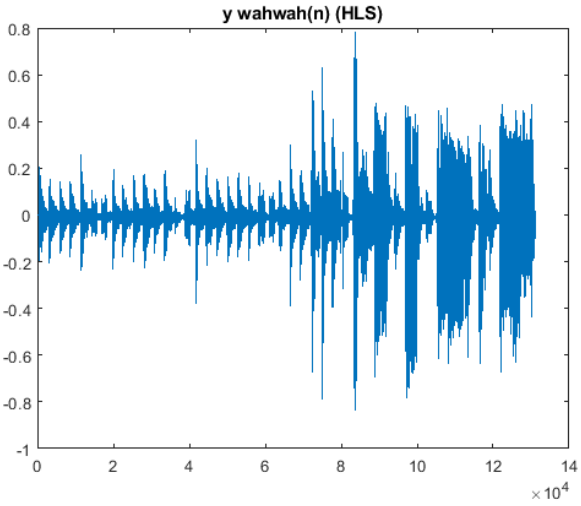


Figura 130: gráfica señal de salida Wah-Wah Vivado HLS™

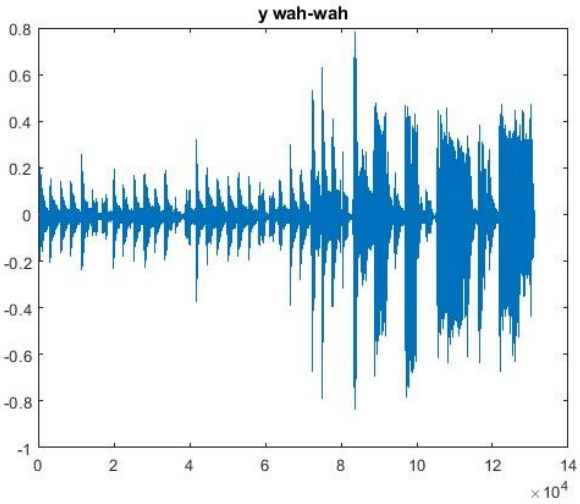


Figura 129: gráfica señal de salida Echo MATLAB®

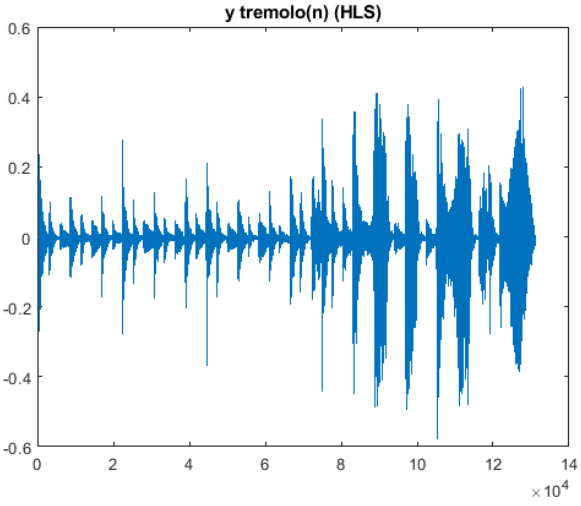


Figura 132: gráfica señal de salida Trémolo Vivado HLS™

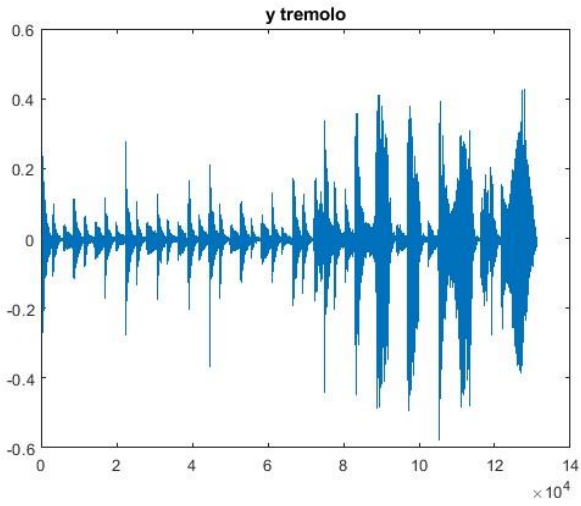


Figura 131: gráfica señal de salida Trémolo MATLAB®

En la figura de la izquierda se muestra el audio de salida obtenido en Vivado HLS™ y en la derecha el de MATLAB®.

No se observa ninguna diferencia entre ellos ya que todos los datasets son generados bajo en mismo fichero: "***dataset_out_matlabFIR.txt***", el cual como su nombre indica, es el dataset que obtenemos como salida del filtro FIR en MATLAB®.

Para finalizar, calcularemos el **error cuadrático medio** entre el dataset de salida ideal (MATLAB®) y el real (Vivado HLS™) de la misma forma que lo hicimos en el apartado 3.1.3.

- Para **Echo** se obtiene un error entre ambos datasets de salida del **0,0064%**.
- Para **Reverb** se obtiene un error entre ambos datasets de salida del **0,0079%**.
- Para **Wah-Wah** se obtiene un error entre ambos datasets de salida del **0,0304%**.
- Para **Tremolo** se obtiene un error entre ambos datasets de salida del **0,0058%**.

Errores muy cercanos a 0%, esperado al haber realizado algoritmos semejantes en ambos software y al tener como entradas los mismos datasets.

3.4.9.1 Síntesis y optimización de los algoritmos

Los objetivos y formas de actuar de este sub-apartado serán similares a los del apartado 3.2.5.2, por lo que se darán cosas por sabidas y no se entrará en tanto detalle.

Síntesis sin directivas

El siguiente paso en nuestro diagrama de flujo consistirá en sintetizar nuestras 4 funciones TOP, ver los resultados obtenidos de las síntesis y tratar de mejorarlos mediante el uso de directivas de optimización.

Los 4 informes proporcionado por Vivado HLS™ al ejecutar las síntesis son los siguientes:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.18	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	1	1	1	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	0	78
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	86	-
Total	0	2	86	93
Available	730	740	269200	129000
Utilization (%)	0	~0	~0	~0

Figura 134: informe sin directivas post-síntesis Echo

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	11.00	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	1	1	1	none

Detail

Instance

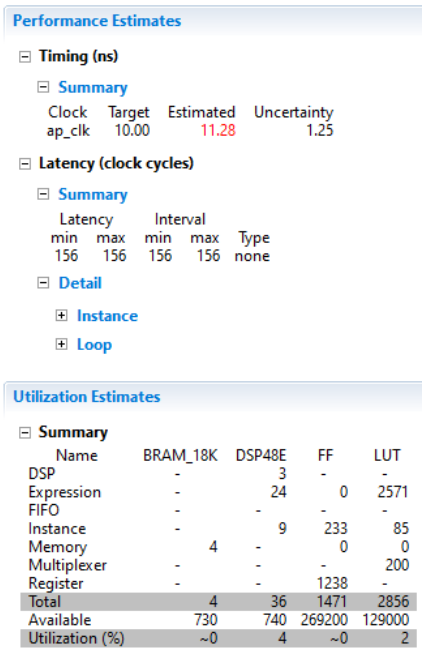
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	0	118
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	65	-
Total	0	2	65	133
Available	730	740	269200	129000
Utilization (%)	0	~0	~0	~0

Figura 133: informe sin directivas post-síntesis Reverb



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	11.28	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
156	156	156	156	none

Detail

Instance

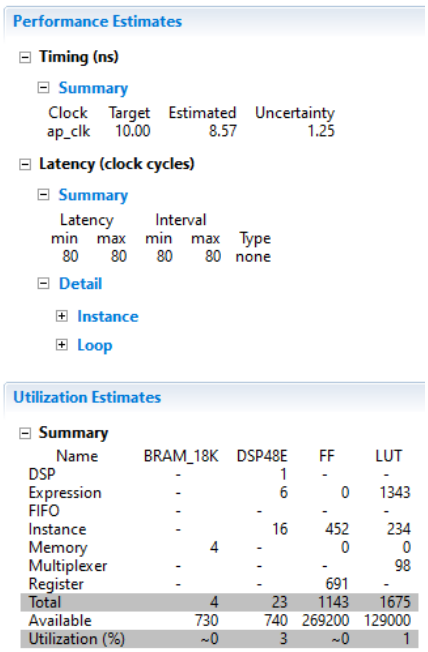
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	3	-	-
Expression	-	24	0	2571
FIFO	-	-	-	-
Instance	-	9	233	85
Memory	4	-	0	0
Multiplexer	-	-	-	200
Register	-	-	1238	-
Total	4	36	1471	2856
Available	730	740	269200	129000
Utilization (%)	~0	4	~0	2

Figura 136: informe sin directivas post-síntesis Wah-Wah



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.57	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
80	80	80	80	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	6	0	1343
FIFO	-	-	-	-
Instance	-	16	452	234
Memory	4	-	0	0
Multiplexer	-	-	-	98
Register	-	-	691	-
Total	4	23	1143	1675
Available	730	740	269200	129000
Utilization (%)	~0	3	~0	1

Figura 135: informe sin directivas post-síntesis Trémolo

En los informes vistos, los resultados obtenidos en Echo y Reverb son prácticamente perfectos, al tener la mínima latencia posible (1), mínimo intervalo de iniciación posible (1) y todos los recursos hardware con una utilización del 0% prácticamente.

Sin embargo, en los efectos Wah-Wah y Trémolo, las latencias son de **156 ciclos** y **80 ciclos** respectivamente, al igual que sus intervalos de iniciación, aunque los recursos hardware tienen una baja utilización. Esto pasará a ser mejorado con las directivas de optimización que veremos a continuación.

Síntesis con directivas

Una vez vistos los informes obtenidos mediante las síntesis sin directivas de optimización, pasaremos a añadir dichas directivas con el objetivo de mejorar, según nuestras necesidades, los resultados obtenidos.

Los objetivos serán los mismos que en 3.2.5.2 por lo que las directivas a usar serán exactamente las mismas ya vistas y explicadas.

Los resultados que obtenemos al hacer uso de **LATENCY (max=1), PIPELINE, INTERFACE (ap_ctrl_none)** e **INTERFACE** (**axis**, para cada puerto de entrada/salida con su correspondiente nombre) son los siguientes:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.18	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
3	3	1	1	function

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	0	182
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	99
Register	-	-	213	-
Total	0	2	213	281
Available	730	740	269200	129000
Utilization (%)	0	~0	~0	~0

Figura 138: informe con directivas post-síntesis Echo

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	11.00	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
3	3	1	1	function

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	0	278
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	165
Register	-	-	282	-
Total	0	2	282	443
Available	730	740	269200	129000
Utilization (%)	0	~0	~0	~0

Figura 137: informe con directivas post-síntesis Reverb

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	80.66	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
36	36	1	1	function

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	3	-	-
Expression	-	24	0	1459
FIFO	-	-	-	-
Instance	0	9	5841	19191
Memory	-	-	-	-
Multiplexer	-	-	-	66
Register	0	-	531	168
Total	0	36	6372	20884
Available	730	740	269200	129000
Utilization (%)	0	4	2	16

Figura 140: informe con directivas post-síntesis Wah-Wah

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	39.03	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
20	20	1	1	function

Detail

Instance

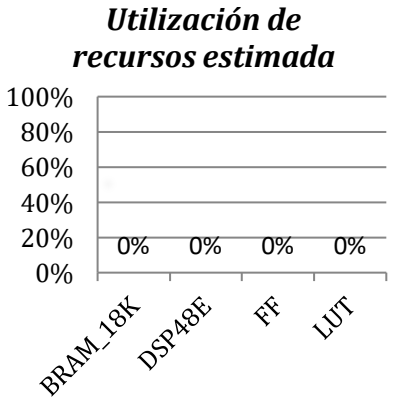
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	6	0	823
FIFO	-	-	-	-
Instance	0	16	3256	9787
Memory	-	-	-	-
Multiplexer	-	-	-	66
Register	0	-	442	128
Total	0	23	3698	10804
Available	730	740	269200	129000
Utilization (%)	0	3	1	8

Figura 139: informe con directivas post-síntesis Trémolo



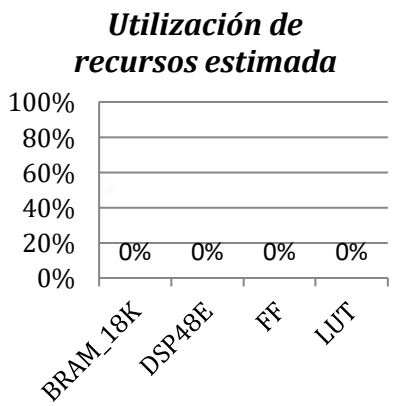


Figura 142: gráfico utilización de recursos en bloque IP Echo

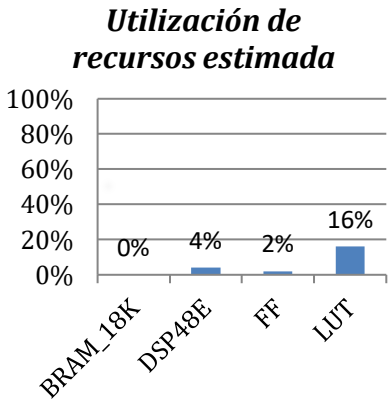


Figura 143: gráfico utilización de recursos en bloque IP WahWah

Figura 141: gráfico utilización de recursos en bloque IP Reverb

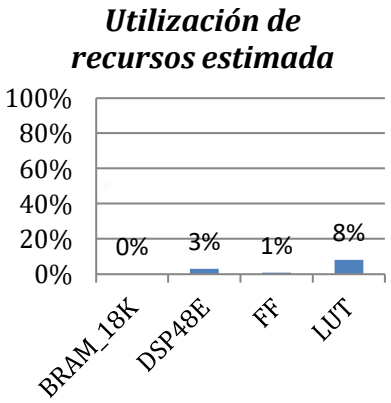


Figura 144: gráfico utilización de recursos en bloque IP Tremolo

En comparativa de latencias, intervalos de iniciación y recursos hardware tenemos que:

- **Echo**: Pasa a tener **3 ciclos de latencia** en vez de 1 debido a las directivas *Interface* añadidas. Mismo intervalo de iniciación. Utilización recursos hardware similares.

- **Reverb**: Pasa a tener **3 ciclos de latencia** en vez de 1 debido a las directivas *Interface* añadidas. Mismo intervalo de iniciación. Utilización recursos hardware similares.

- **Wah-Wah**: Pasa a tener **36 ciclos de latencia** en vez de 156 gracias a las directivas PIPELINE y LATENCY. El **intervalo de iniciación** pasa de 156 ciclos a **1 ciclo**. Utilización recursos hardware algo mayor.

- **Trémolo**: Pasa a tener **20 ciclos de latencia** en vez de 80 gracias a las directivas PIPELINE y LATENCY. El **intervalo de iniciación** pasa de 80 ciclos a **1 ciclo**. Utilización recursos hardware algo mayor.

Los resultados obtenidos tras el uso de las directivas son muy positivos, reduciendo notoriamente latencias e intervalos en Wah-Wah y Trémolo, teniendo latencias e intervalos muy bajos en Echo y Reverb, así como utilizaciones de recursos hardware despreciables en Echo y Reverb y algo mayores pero normales y esperadas en Wah-Wah y Trémolo.

Obtenemos velocidades de reloj mayores de 10 ns en Reverb, Wah-Wah y Trémolo. Siendo 10 ns el tiempo objetivo del software, esto es un mal menor, debido a que en nuestro diseño, la frecuencia de reloj ha la que será sometido el bloque, será mucho mayor, como veremos más adelante.

3.4.9.2 Co-Simulación RTL y creación de los núcleos IP

Finalmente, Vivado HLS nos da la opción de simular en VHDL nuestro diseño, donde veamos los resultados de las directivas todo en lenguaje hardware, con los datos en binario y con señales lógicas.

En las siguientes capturas de las simulaciones de cada bloque, nos fijaremos en el correcto funcionamiento de los mismos, observando la generación de datos en el momento del "apretón de manos" del protocolo AXI cuando TVALID y TREADY de la interfaz maestra de salida están a nivel alto a la vez y se produce un flanco de reloj. Así como la introducción de datos de entrada por la interfaz esclava según se produce el apretón de manos en la misma.

Las simulaciones obtenidas son:

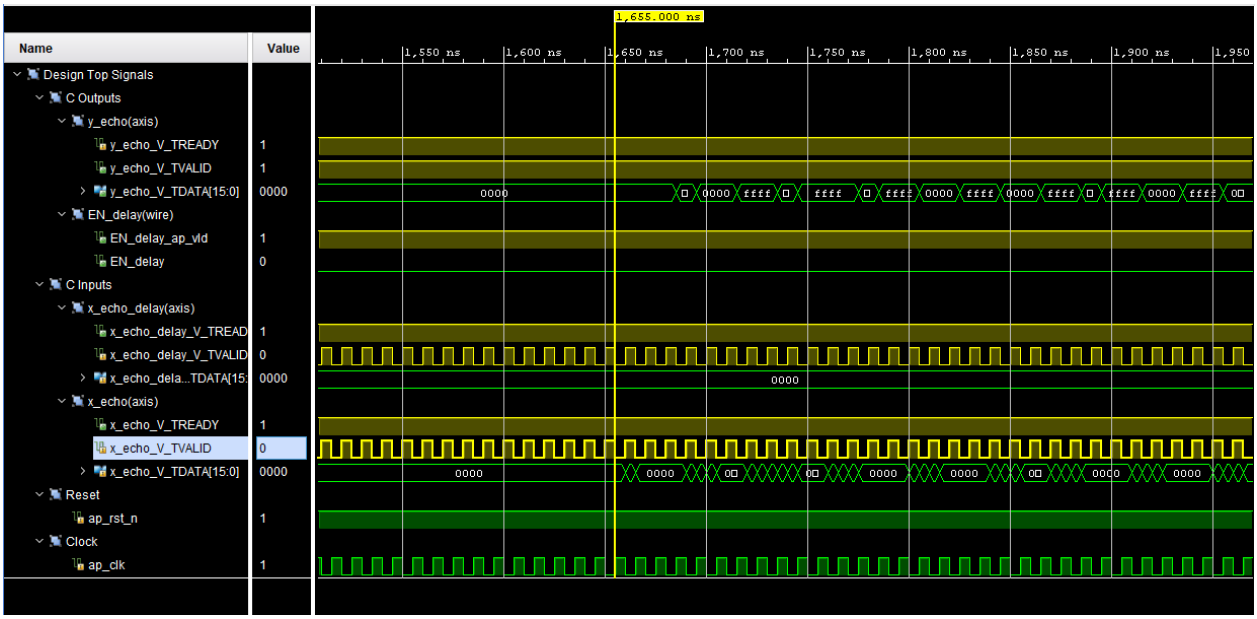


Figura 145: Co-simulación RTL Echo

Figura 146: Co-simulación RTL Reverb

Simulación de Rerverb desde una vista más general para apreciar como cuando *EN_delay = '1'* empezamos a tener valores en *y_delay* y *x_delay*. Así como cuando silenciamos la señal de entrada valiendo "0000" hasta el final de la simulación.

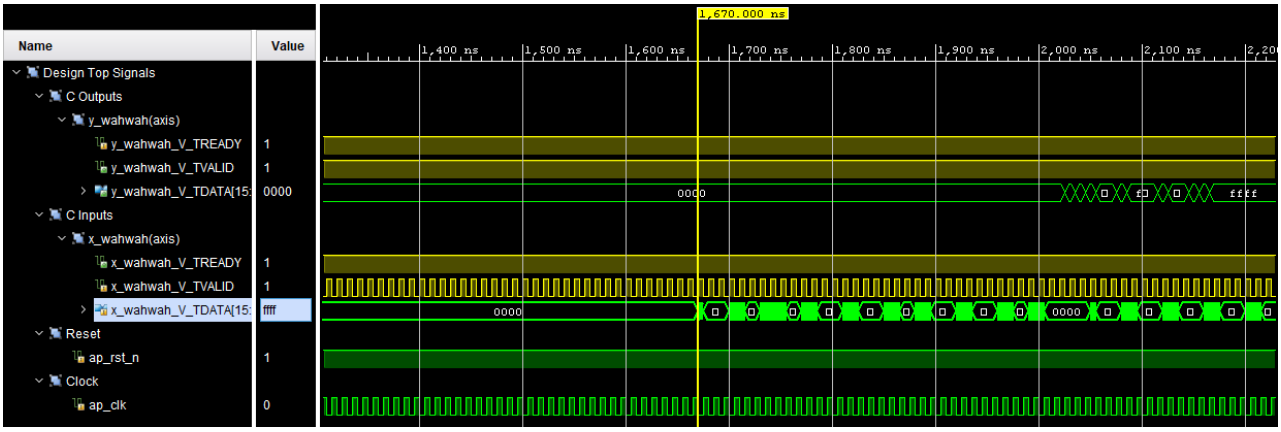


Figura 147: Co-simulación RTL Wah-Wah

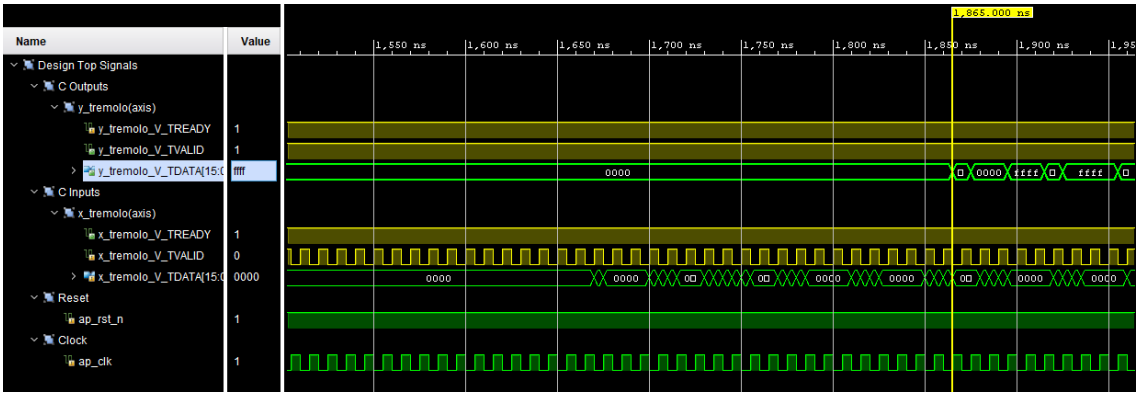


Figura 148: Co-simulación RTL Trémolo

Podemos comprobar cómo la latencia es la esperada al ver que las muestras de salida tardan los ciclos correspondientes a cada efecto en generarse a partir de su muestra de entrada, momento a su vez, en la que la señal TVALID de la salida vale un nivel alto, indicándonos que la validez del resultado ya que sabemos que los datos de salida (TDATA) únicamente se obtiene al disponer de las señales TREADY y TVALID a nivel alto.

Además, los datos de entrada se van introduciendo al bloque secuencialmente cada vez que la señal TVALID de la entrada vale '1'.

Por último, exportamos nuestros diseños RTL obtenidos a bloques IP. Una vez se hayan producido los ficheros correspondientes, importaremos en Vivado™ los bloques IP y los añadiremos en un diseño de bloques para observarlos y ver si es lo que esperábamos obtener en cuanto a interfaces de puertos salida/entrada.

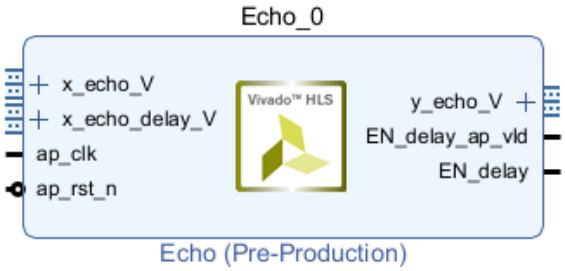


Figura 150: bloque IP-core Echo

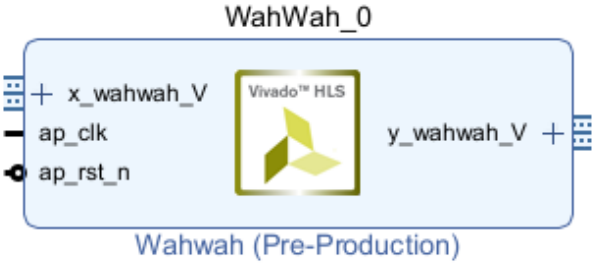


Figura 149: bloque IP-core Wah-Wah

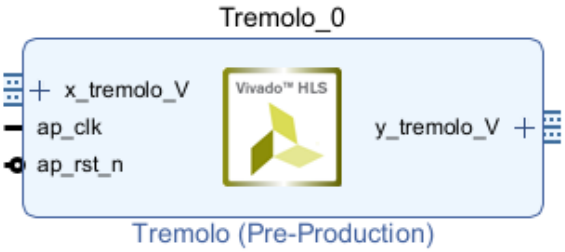


Figura 151: bloque IP-core Trémolo

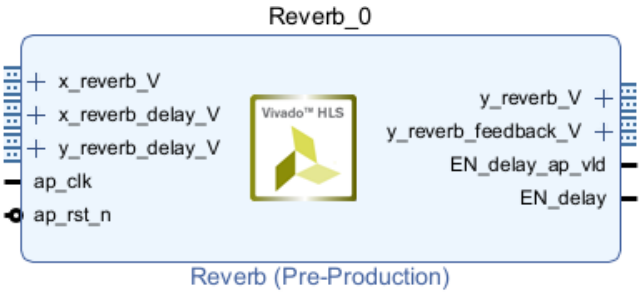


Figura 152: bloque IP-core Reverb

4. Implementación del sistema final

4.1 Introducción

Al haber estudiado, implementado y creado todos los bloques que albergan nuestro sistema, es el momento de unirlos entre ellos, creando nuestro sistema final de procesado digital de audio, llamado: **TOP AUDIO SYSTEM**.

El sistema de bloques a crear está esquematizado en el apartado 3, por lo que se seguirá ese mismo orden de bloques y organización en nuestro sistema final.

Crearemos nuestro sistema de bloques mediante la aplicación "Block Design" de Vivado™, para el cual se le asignará un código VHDL que contenga el componente TOP con los puertos entrada/salida del sistema global, así como un testbench que lea las muestras de un dataset de entrada **binario**, y escriba en los 4 datasets (1 por efecto) de salida finales las muestras de salida binarias.

Los pasos **iniciales** para crear nuestro diseño de bloques son los siguientes:

1. Importar los ficheros de la carpeta "ip" de los bloques creados (DRCompressor, Echo, Reverb, Wah-Wah y Tremolo).

2. Crear el "Block Design".

3. Añadir los bloques importados al diseño.

4. Añadir el "FIR compiler" y configurarlo tal como se hizo en el apartado 3.3.7.

5. Unir los bloques DRCompressor y FIR compiler y hacer externos los puertos de entrada de DRCompressor con la opción "make external".

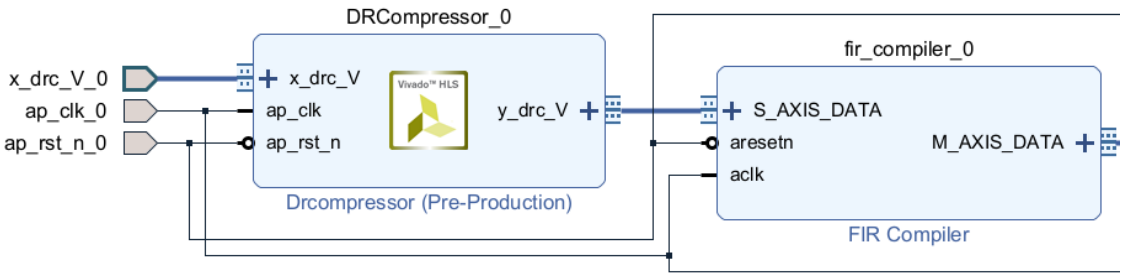


Figura 153: Unión DRCompressor y FIR compiler en TOP AUDIO SYSTEM

Una vez realizada esta unión, nos encontramos con el primer problema: en AXI4-Stream las **conexiones son punto a punto**, es decir, no es posible hacer bifurcaciones. Esto es un problema a la hora de conectar la salida del FIR compiler a las entradas de los 4 efectos creados.

Para ello, Vivado™ tiene una serie de bloques nativos de gran variedad basados en el protocolo AXI4-Stream, de los cuales buscaremos el que nos puede ayudar a solucionar este problema.

4.2 Bloques complementarios de nuestro sistema

A parte de los 6 bloques IP vistos, vamos a necesitar un par de bloques IP más para poder desarrollar el sistema, previamente, debemos plantear los problemas encontrados para seleccionar los bloques IP correctos.

- AXI4-Stream usa conexiones punto a punto, necesitamos realizar una bifurcación para conectar la salida del FIR compiler a los 4 efectos.

- Necesidad de memorias externas. Como se vio en el apartado 3.4.8, (**Nota importante***) haremos uso de memorias FIFO para retener las muestras que queremos retrasar hasta que llegue el momento de usarlas en el bloque correspondiente.

Los bloques nativos de Vivado™ que nos dan solución a los dos problemas vistos son: **AXI4-Stream Broadcaster** y **FIFO Generator**, respectivamente.

4.2.1 AXI4-Stream Broadcaster

Se trata de un bloque muy sencillo el cual su función es: **replicar** la señal de la entrada de datos en las salidas del bloque, sin realizar ninguna modificación en la misma. De modo que se consiga una retransmisión de la señal en la interfaz esclava a todas las interfaces maestras, consiguiendo la bifurcación deseada en la cantidad de interfaces maestras que deseemos (hasta 16).

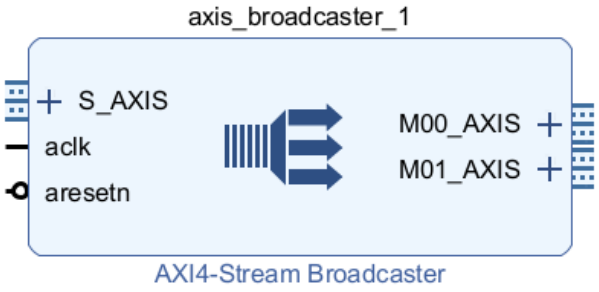


Figura 154: AXI4-Stream Broadcaster

4.2.2 FIFO Generator

Consiste en una memoria FIFO configurable, de modo que el esclavo entrega los datos a la interfaz maestra bajo la configuración FIFO, el primero que entra, el primero que sale.

En nuestro caso la usaremos a modo de **memoria caché**, es decir,

introduciremos una serie de datos a la entrada de la FIFO, y cuando queramos obtener esos datos activaremos la señal TREADY de la interfaz maestra, indicando que ya estamos listos para recibir los datos almacenados en la memoria. Recibiéndolos en el mismo orden que el que entraron a la FIFO.

Configuraremos el FIFO Generator, de modo que al abrirlo seleccionaremos:

- Interfaz AXI Stream.
- 2 Bytes de ancho bus TDATA y 0 bytes de TUSER.
- Profundidad de la FIFO de 131072 muestras

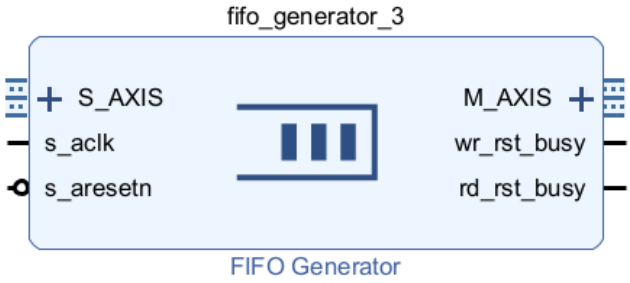


Figura 155: FIFO Generator

En total en nuestro sistema, necesitaremos **5 FIFOS**, 1 por cada señal de delay necesaria: 3 para Echo (3 bloques Echo) y 2 para Reverb (1 bloque Reverb).

4.3 Creación y codificación de sistema final

Una vez puesta solución a los dos problemas vistos, seguiremos con el apartado 4.1, completando el diseño de bloques basado en el esquema global que vimos en el apartado 3 (figura 20) y creando el código para obtener los resultados finales.

Los siguientes pasos son:

1. Configurar Broadcaster: seleccionar interfaz esclava y maestra de 2 Bytes, así como seleccionar 11 interfaces maestras o salidas, del mismo.

2. Unir la salida de FIR compiler con la entrada de AXI4-Stream Broadcaster.

3. Añadir y configurar (como vimos en el apartado 4.2.2) los 5 FIFO Generator.

4. Unir las 11 salidas del Broadcaster con las entradas de los siguientes bloques: Echo0, FIFO0, Reverb, FIFO1, Wah-Wah0, Tremolo0, Wah-Wah1, Echo1, FIFO3, Echo2 y FIFO4.

5. Unir la salida de FIFO0 con el puerto *x_echo_delay_V* (repetir para los demás Echo's) y unir la salida de FIFO1 con *x_reverb_delay_V*.

6. Unir la salida *y_reverb_feedback_V* con la entrada de FIFO2 y unir la salida

de FIFO2 con *y_reverb_delay_V*.

7. Hacer externos los puertos de salida de Echo0, Reverb, Wah-Wah0, Tremolo0, Tremolo1 (Comb0), WahWah2 (Comb1) y Tremolo2 (Comb2) con la opción “make external”.

8. Conectar las salidas de Echo1, Echo2 y WahWah1 con las entradas de WahWah2, Tremolo2 y Tremolo1 respectivamente para crear los efectos combinados.

9. Conectar todas las señales reset y señales de reloj a las externas ya creadas.

10. Conectar la salida de Reverb *EN_delay* a *m_axis_tready* de FIFO1 y FIFO2. Repetir para los bloques Echo y sus FIFO.

11. Hacer externas la señal *m_axis_tready_0 (10:0)* del Broadcaster y la señal *y_reverb_delay_V_TVALID_0* con la opción “make external”.

12. Crear puertos lógicos *EN_effects* y *EN_combs*. Unir *EN_effects* con señales TREADY de interfaz maestra de los efectos individuales (Reverb0, Echo0, WahWah0 y Tremolo0). Unir *EN_combs* con señales TREADY de interfaz maestra del primer efecto de cada combinación (Echo1, Echo2 y WahWah1).

La necesidad del paso 10 es debido a la necesidad del **forzar a nivel alto** esas señales. Necesidad dada ya que si no se fuerzan a nivel alto, son interfaces que no funcionarían, quedándose con valores lógicos desconocidos (‘U’).

Esto es debido a que en el otro lado de la comunicación entre interfaces las señales que activan a las señales mencionadas necesitan ser activadas también, es por ello el uso del paso 10.

El hecho del paso 12 es debido a que al activar la señal TREADY a nivel alto de una interfaz maestra, de uno de los efectos, estamos dando permitiendo que el bloque genere la señal de salida correspondiente, ya que como hemos visto, si esta señal está a nivel bajo la señal de salida no se obtiene al necesitar que ambas (TREADY y TVALID) estén a nivel alto a la vez para cumplir el proceso de “handshake”. De esta forma podemos seleccionar qué salidas queremos obtener del sistema.

Las salidas de las 3 combinaciones de efectos las llamaremos *y_comb_V0*, *y_comb_V1* e *y_comb_V2*, dando lugar a las combinaciones deseadas y mostradas en el apartado 3.

Habiendo seguido los pasos disponemos de nuestro sistema de bloques final, listo para codificar su banco de pruebas y obtener los resultados finales. Nuestro TOP_AUDIO_SYSTEM nos debería de haber quedado según la figura 150. En la figura 150 podemos ver todas las conexiones realizadas, los puertos externos generados y cada uno de los bloques ya estudiados. Podemos fijarnos también en el uso de las FIFOs, viendo como cobra sentido lo ideado en las implementaciones de Echo y Reverb, viendo como se crea un **lazo de realimentación** entre la salida del bloque Reverb (*y_reverb* = *y_reverb_feedback*) y la entrada *y_reverb_delay* ya estudiada.

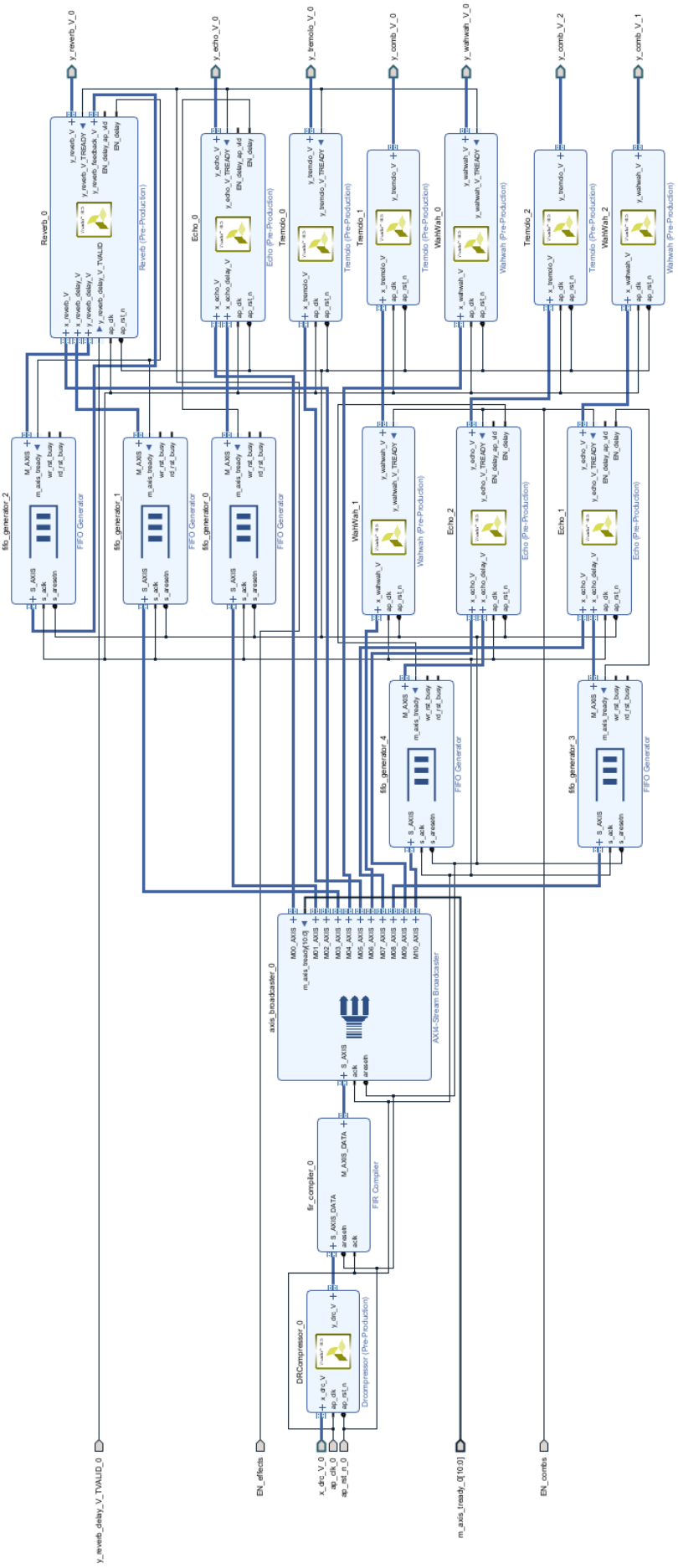


Figura 156: Diseño de bloques del TOP AUDIO SYSTEM

HDL Wrapper

El siguiente paso en nuestra implementación del sistema final una vez creado el diseño de bloques, será generar el código correspondiente del mismo. Al haber creado un "Block Design" Vivado™ nos brinda la oportunidad de generar un código VHDL correspondiente a este diseño, de modo que si pulsamos botón derecho en nuestro diseño de bloques en el panel de archivos y seleccionamos "**Create HDL Wrapper**" generamos este código en el fichero "***top_audio_system_wrapper.vhd***".

El "Wrapper" de nuestro sistema contiene la declaración del componente de sistema global (*top_audio_system*), donde los puertos entrada/salida son los puertos que hicimos externos. En este fichero también se realiza un "port map" en el que se conecta nuestra entidad (*top_audio_system_wrapper*) con nuestro componente (*top_audio_system*), los cuales comparten los mismos puertos.

El componente *top_audio_system* contiene todas las conexiones realizadas en el diseño de bloques, pudiendo verlas en un fichero (solo de lectura) ubicado debajo de nuestro diseño de bloques en cuanto a nivel de arquitectura.

```
entity top_audio_system_wrapper is
  port (
    EN_combs : in STD_LOGIC;
    EN_effects : in STD_LOGIC;
    ap_clk_0 : in STD_LOGIC;
    ap_rst_n_0 : in STD_LOGIC;
    m_axis_tready_0 : in STD_LOGIC_VECTOR ( 10 downto 0 );
    x_drc_V_0_tdata : in STD_LOGIC_VECTOR ( 15 downto 0 );
    x_drc_V_0_tready : out STD_LOGIC;
    x_drc_V_0_tvalid : in STD_LOGIC;
    y_comb_V_0_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 );
    y_comb_V_0_tready : in STD_LOGIC;
    y_comb_V_0_tvalid : out STD_LOGIC;
    y_comb_V_1_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 );
    y_comb_V_1_tready : in STD_LOGIC;
    y_comb_V_1_tvalid : out STD_LOGIC;
    y_comb_V_2_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 );
    y_comb_V_2_tready : in STD_LOGIC;
    y_comb_V_2_tvalid : out STD_LOGIC;
    y_echo_V_0_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 );
    y_echo_V_0_tvalid : out STD_LOGIC;
    y_reverb_V_0_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 );
    y_reverb_V_0_tvalid : out STD_LOGIC;
    y_reverb_delay_V_TVALID_0 : in STD_LOGIC;
    y_tremolo_V_0_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 );
    y_tremolo_V_0_tvalid : out STD_LOGIC;
    y_wahwah_V_0_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 );
    y_wahwah_V_0_tvalid : out STD_LOGIC
  );
end top_audio_system_wrapper;
```

Figura 157: entity top_audio_system_wrapper

Testbench

A continuación, se creará el banco de pruebas que ponga en funcionamiento, a nuestra manera, a la entidad creada *top_audio_system_wrapper*. El cual simulará el diseño de bloques completo generando datasets de salida a partir de un único dataset de entrada dado. Nuestro testbench será llamado: "***TOP_AUDIO_SYSTEM_tb.vhd***".

El objetivo de nuestro testbench es obtener los datasets de salida binarios, los cuales serán el resultado final del sistema. Resultados que después comprobaremos vía MATLAB®, transformando los datos binarios en datos decimales.

Para ello, debemos elaborar un testbench que ponga en marcha el sistema. Crearemos dos procesos, donde el primero se encargará de:

- Creación de la señal de reloj (***ap_clk_0***) con una frecuencia de 22050 Hz (**45,35 us**). Debe de ser igual que la frecuencia de muestreo al querer generar audios con esta velocidad de muestreo.

El segundo proceso o **proceso principal** se encargará de:

- Abir dataset de entrada (modo lectura): "***dataset_in_bin.txt***" y los 7 datasets de salida (modo escritura) en caso de que las señales Enable correspondientes a cada uno lo autoricen: "***dataset_out_echo_bin.txt***", "***dataset_out_reverb_bin.txt***", "***dataset_out_wahwah_bin.txt***", "***dataset_out_tremolo_bin.txt***", "***dataset_out_comb0_bin.txt***", "***dataset_out_comb1_bin.txt***" y "***dataset_out_comb2_bin.txt***".

- Desactivar la señal de reset (***ap_rst_n_0***) tras dos ciclos de reloj (señal de reset activa si está a nivel bajo).

```
process
begin
  wait for TbPeriod/2;
  ap_clk_0 <= not ap_clk_0;
end process;

process
    variable dataIn_Line: line;
    variable dataOutEcho_Line: line;
    variable dataOutWahwah_Line: line;
    variable dataOutReverb_Line: line;
    variable dataOutTremolo_Line: line;
    variable dataOutComb0_Line: line;
    variable dataOutComb1_Line: line;
    variable dataOutComb2_Line: line;
    variable dataIn: std_logic_vector(15 downto 0);

    begin
        file_open(dataset_in, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_in_bin.txt", read_mode);

if EN_effects = '1' then
        file_open(dataset_out_echo, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_echo_bin.txt", write_mode);
        file_open(dataset_out_wahwah, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_wahwah_bin.txt", write_mode);
        file_open(dataset_out_reverb, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_reverb_bin.txt", write_mode);
    file_open(dataset_out_tremolo, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_tremolo_bin.txt", write_mode);
end if;
if EN_combs = '1' then
    file_open(dataset_out_comb0, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_comb0_bin.txt", write_mode);
    file_open(dataset_out_comb1, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_comb1_bin.txt", write_mode);
    file_open(dataset_out_comb2, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_comb2_bin.txt", write_mode);
end if;
        ap_rst_n_0 <= '0';
        wait for 2*TbPeriod;
        ap_rst_n_0 <= '1';
```

Figura 158: testbench top audio system parte 1

- Crear un bucle while que finalice al acabar de leer todo el dataset de entrada.

- Leer e introducir un dato de entrada al sistema en caso de cumplirse el "**apretón de manos**" cada periodo de la señal de reloj.

- Escribir los datos de salida en los datasets de salida en caso de cumplirse el

"**apretón de manos**" (regido por las señales de Enable) cada periodo de la señal de reloj.

- Cerrar los ficheros debidos al acabar el bucle while.

```
        while not endfile(dataset_in) loop
            if x_drc_V_0_tready = '1' and x_drc_V_0_tvalid = '1' then
                readline(dataset_in, dataIn_Line);
                read(dataIn_Line, dataIn);
                x_drc_V_0_tdata <= dataIn;
            end if;
            wait until ap_clk_0' event;
if EN_effects = '1' then
            if y_echo_V_0_tvalid = '1' then
                    write(dataOutEcho_Line, y_echo_V_0_tdata);
                    writeline(dataset_out_echo, dataOutEcho_Line);
            end if;
            if y_wahwah_V_0_tvalid = '1' then
                    write(dataOutWahwah_Line, y_wahwah_V_0_tdata);
                    writeline(dataset_out_wahwah, dataOutWahwah_Line);
            end if;
            if y_reverb_V_0_tvalid = '1' then
                    write(dataOutReverb_Line, y_reverb_V_0_tdata);
                    writeline(dataset_out_reverb, dataOutReverb_Line);
            end if;
            if y_tremolo_V_0_tvalid = '1' then
                    write(dataOutTremolo_Line, y_tremolo_V_0_tdata);
                    writeline(dataset_out_tremolo, dataOutTremolo_Line);
            end if;
end if;
if EN_combs = '1' then
            if y_comb_V_0_tvalid = '1' and y_comb_V_0_tready = '1' then
                    write(dataOutComb0_Line, y_comb_V_0_tdata);
                    writeline(dataset_out_comb0, dataOutComb0_Line);
            end if;
            if y_comb_V_1_tvalid = '1' and y_comb_V_1_tready = '1' then
                    write(dataOutComb1_Line, y_comb_V_1_tdata);
                    writeline(dataset_out_comb1, dataOutComb1_Line);
            end if;
            if y_comb_V_2_tvalid = '1' and y_comb_V_2_tready = '1' then
                    write(dataOutComb2_Line, y_comb_V_2_tdata);
                    writeline(dataset_out_comb2, dataOutComb2_Line);
            end if;
end if;
            wait until ap_clk_0' event;
        end loop;
```

Figura 159: testbench top audio system parte 2

```
                file_close(dataset_in);
           if EN_effects = '1' then
                file_close(dataset_out_echo);
                file_close(dataset_out_wahwah);
                file_close(dataset_out_reverb);
                file_close(dataset_out_tremolo);
           end if;
           if EN_combs = '1' then
                file_close(dataset_out_comb0);
                file_close(dataset_out_comb1);
                file_close(dataset_out_comb2);
           end if;
               wait;
           end process;
      end testbench;
```

Figura 160: testbench top audio system parte 3

El "**apretón de manos**" (visto en apartado 2.4) en el protocolo AXI4 se da cuando las señales TREADY y TVALID de una interfaz están a nivel alto a la vez:

- Validando el dato generado en una señal maestra.
- Aceptando un dato de entrada en una interfaz esclava.

- Es por ello que únicamente cuando TREADY y TVALID sean '1' en la interfaz esclava del DRCompressor se leerá un dato, indicando que el bloque está **listo** para recibirlo.

- Así como, únicamente cuando TREADY y TVALID sean '1' en la interfaz maestra de los bloques de salida se escribirá un dato, indicando que el bloque **valida** el dato generado.

Las señales externas que sean entradas TREADY o TVALID se forzarán a '1' desde el inicio, tanto en interfaces maestras como esclavas, con la excepción de que alguna de las señales Enable ponga a '0' las señales TREADY de las interfaces maestras que controle dicho Enable. De este modo dependemos de la señal de salida de dicha interfaz (TREADY o TVALID) para aceptar o validar datos.

El siguiente paso será simular nuestro banco de pruebas y ver los resultados, a partir de ahí sintetizaremos el diseño y comprobaremos su implementación para dispositivos reales.

4.4 Resultados sistema final

Una vez visto todo el desarrollo teórico/práctico del sistema global, pasaremos a obtener resultados del mismo mediante las simulaciones que realizaremos en Vivado™. Todas las simulaciones se basarán en el sistema visto en el apartado 3.5, donde además se elaboraba el testbench que pondrá en práctica al diseño creado.

4.4.1 Simulación de comportamiento

En primer lugar, realizaremos la "**Behavioral Simulation**" o simulación de comportamiento, la cual se trata de una simulación, previa a síntesis, que muestra el comportamiento del sistema tal y como hemos descrito en el testbench.

Como recordatorio, se vuelven a destacar las interfaces AXI del sistema que vamos a ver en las simulaciones y que podemos ver en el apartado 4.1 y 4.3:

x_drc_V_0: interfaz esclava AXI4-Stream del bloque DRCompressor. Su señal *x_drc_V_0_tdata[15:0]* introducirá al sistema los datos de entrada secuencialmente.

y_efectoX_V_0: interfaz maestra AXI4-Stream de cualquier bloque efecto individual (sustituir "efectoX" por cualquier nombre de efecto). Su señal *y_efectoX_V_0_tdata[15:0]* generará los datos de salida finales de cada efecto.

y_combX_V_0: interfaz maestra AXI4-Stream de cualquier combinación de efectos del sistema (sustituir "combX" por comb0, 1 y2). Su señal *y_combX_V_0_tdata[15:0]* generará los datos de salida finales de cada combinación.

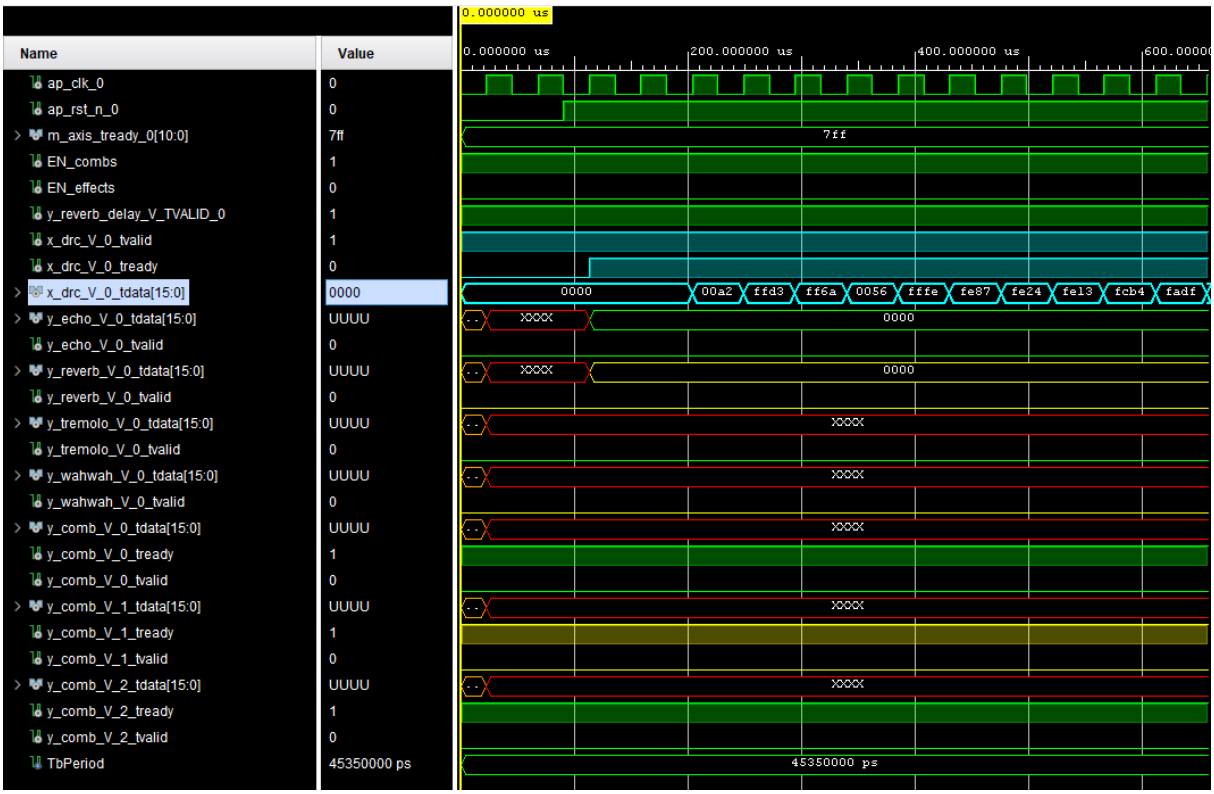


Figura 161: captura 1 simulación top audio system

Viendo las capturas de simulación, podemos corroborar como, cuando la señal de salida TREADY del bloque DRCompressor es igual a '1', se van introduciendo datos al sistema por el bus TDATA en cada ciclo de reloj,

Los datos de salida son generados cuando la señal TVALID de la interfaz maestra de cada salida se activa a nivel alto ('1').

Además, vemos como los datos de salida no existen ("XXXX") hasta que cesa la **latencia** que dispone cada camino, la cual equivale al tiempo que tarda el sistema en generar un dato de salida dado un dato de entrada. Siendo la latencia para cada camino la suma de la latencia de todos los bloques:

$$t_{total} = t_{DRCompressor} + t_{FIR} + t_{Broadcaster} + t_{Efecto\,X}$$

- Salida Echo: 14+267+0+3 = **284** ciclos de latencia = **12,88 ms**
- Salida Reverb: 14+267+0+3 = **284** ciclos de latencia = **12,88 ms**
- Salida Wah-Wah: 14+267+0+36 = **317** ciclos de latencia = **14,37 ms**
- Salida Trémolo: 14+267+0+20 = **301** ciclos de latencia = **13,65 ms**
- Salida Combinación 0: 14+267+0+36+20 = **337** ciclos de latencia = **15,28 ms**
- Salida Combinación 1: 14+267+0+3+36 = **320** ciclos de latencia = **14,51 ms**

- Salida Combinación 2: 14+267+0+3+20 = **304** ciclos de latencia = **13,78 ms**

En primer lugar se muestra una simulación con *EN_effects ='1'* y *EN_comb='0'*, habilitando el camino de los efectos individuales y anulando el de los efectos combinados.

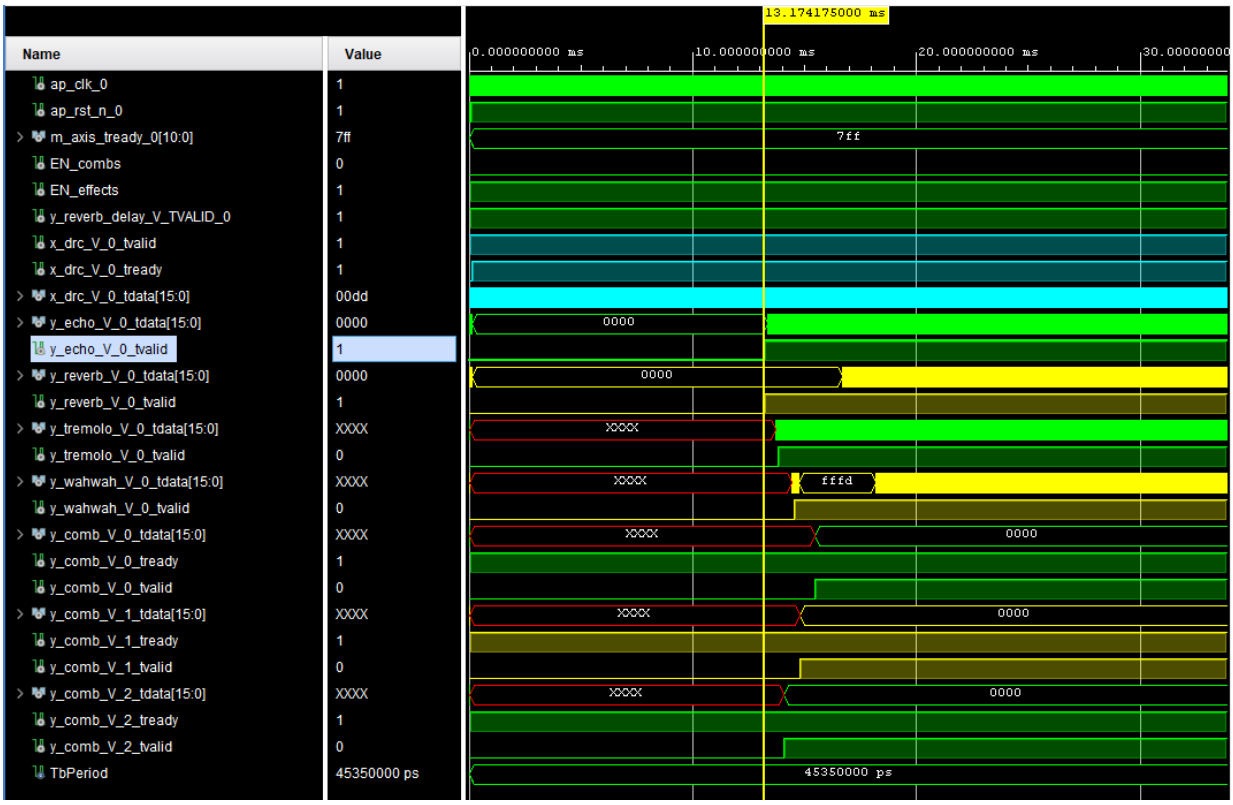


Figura 162: captura 2 simulación top audio system con efectos individuales

Desde esta captura vemos la como las salidas de los efectos individuales (separadas por colores) empiezan a obtener valores cuando se cumple la latencia, momento en el que **TVALID = '1'**. Y al estar las señales **TREADY** a nivel alto también, se crear "apretón de manos" del protocolo, donde la interfaz maestra indica a la esclava que está lista para recibir datos y que además, los que está recibiendo, **por cada ciclo de reloj**, son validos. En la siguiente captura podemos verlo más en detalle.

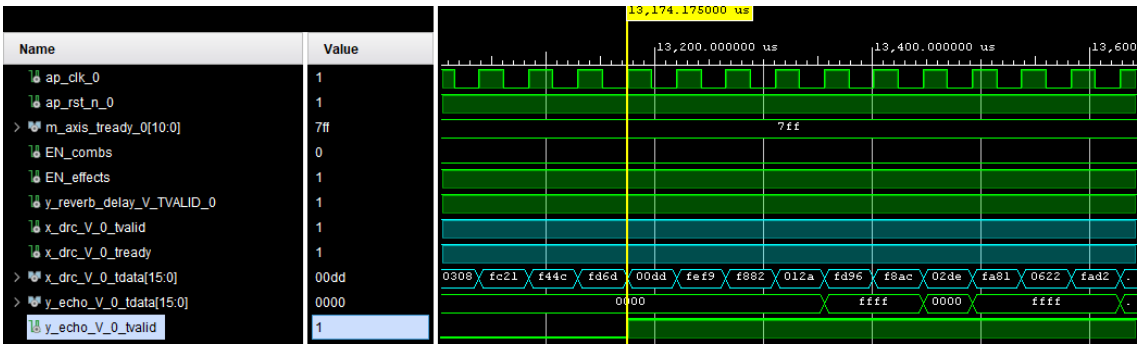


Figura 163: captura 3 simulación top audio system con efectos individuales

En segundo lugar se muestra una simulación con *EN_effects ='0'* y *EN_comb='1'*, anulando el camino de los efectos individuales y habilitando el de los efectos combinados. Donde vemos como a su debido momento las salidas de los efectos combinados van obteniendo valores con la latencia descrita previamente.

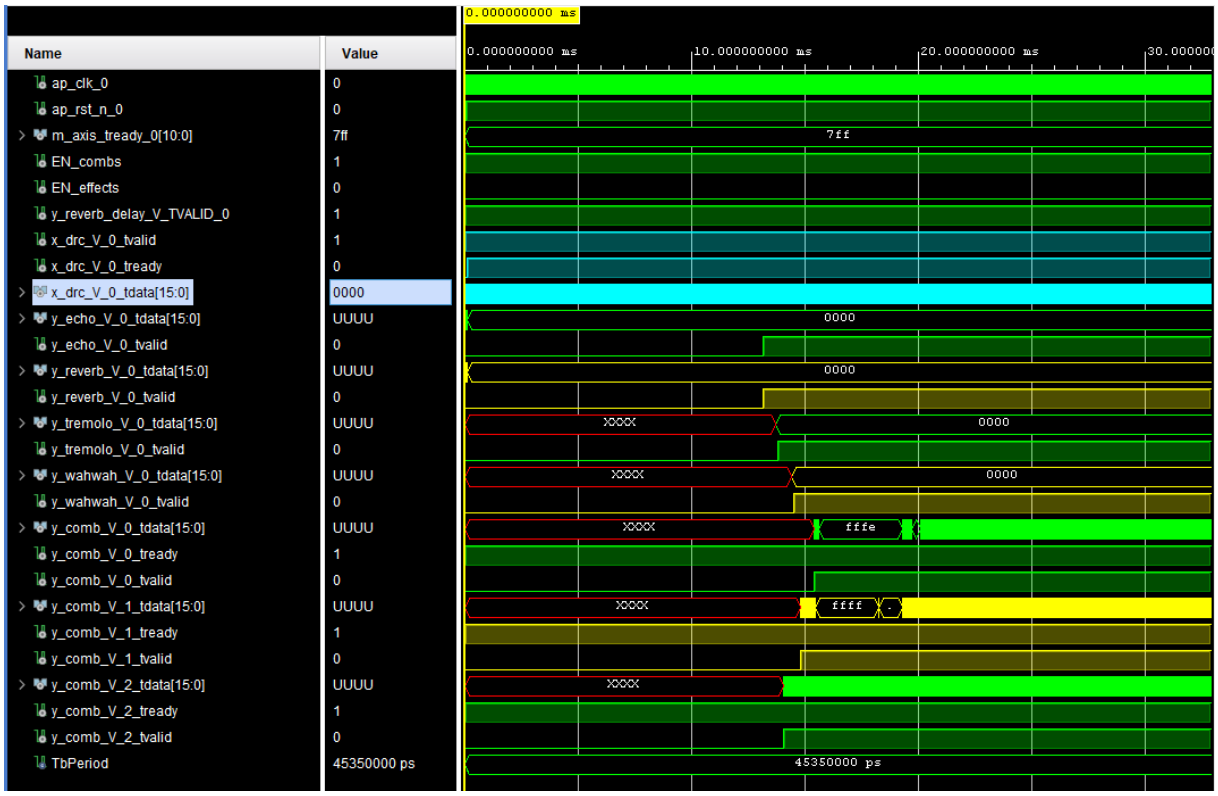


Figura 164: captura 4 simulación top audio system con efectos combinados

El simulador de Vivado™ nos da la opción de seleccionar bloques internos del sistema para ver el valor de sus señales, pudiendo comprobar el transcurso de las señales bloque a bloque en caso de error en los resultados, en este caso no se entrará en esos detalles.

Una vez finalizada la simulación, la cual debe durar el mismo tiempo que dura el audio, aproximadamente **6 segundos**, pasaremos a exportar los 7 datasets obtenidos y leerlos en MATLAB®, donde representaremos los mismos y compararemos con los obtenidos en ese mismo software, calculando el error entre ambos.

Dataset binario a Dataset decimal dada nuestra codificación

Para pasar nuestro fichero binario a fichero de datos decimales necesitaremos realizar una conversión, para ello haremos lo siguiente:

- Lectura fichero binario.
- Paso dato de binario a decimal con *bin2dec*.
- Comprobación de si el número es negativo o positivo, si el bit de mayor peso es '1' (32768) será negativo si no, positivo.
- Si número positivo dividimos número entre 2^q donde "*q*" es el número de bits dado para la parte decimal, 14 en nuestro caso.

- Si número negativo pasamos de complemento a2 a número real y dividimos entre -2^q.

```
while ischar(y_line)
    y_dec(j) = bin2dec(y_line);
    if(y_dec(j) > 32767)            %número negativo
        y_dec(j) = 2^16 - y_dec(j);
        y(j) = -y_dec(j)/(2^14);
    else                            %número positivo
        y(j) = y_dec(j)/(2^14);
    end
    j=j+1;
    y_line = fgetl(fp_bin2);
end
```

Figura 165: pasar de binario a decimal según nuestra codificación

Una vez dispongamos de nuestros datasets en decimal pasaremos a obtener resultados gráficos y sonoros **finales**, donde los resultados son:

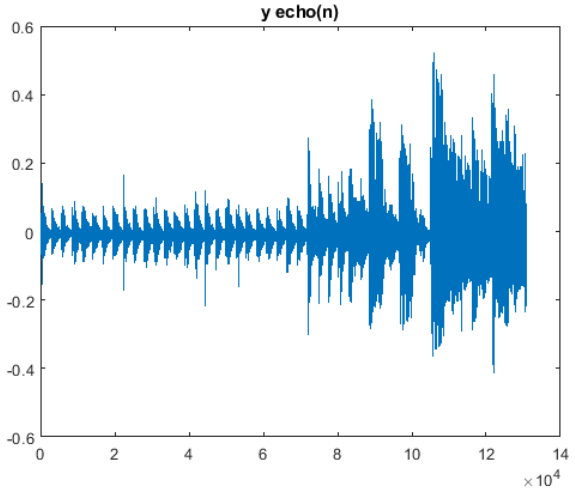


Figura 169: señal de salida Echo del top audio system

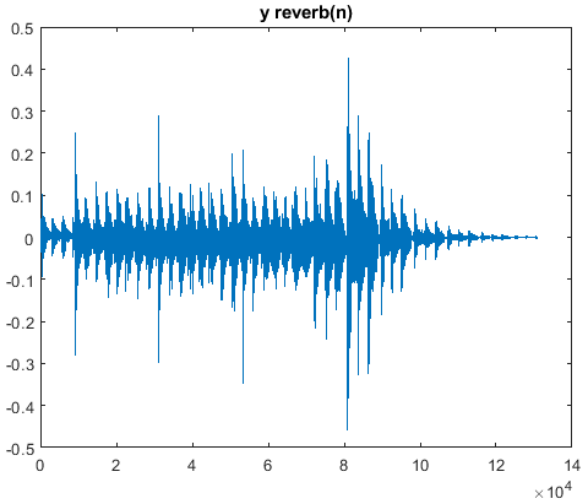


Figura 168: señal de salida Reverb del top audio system

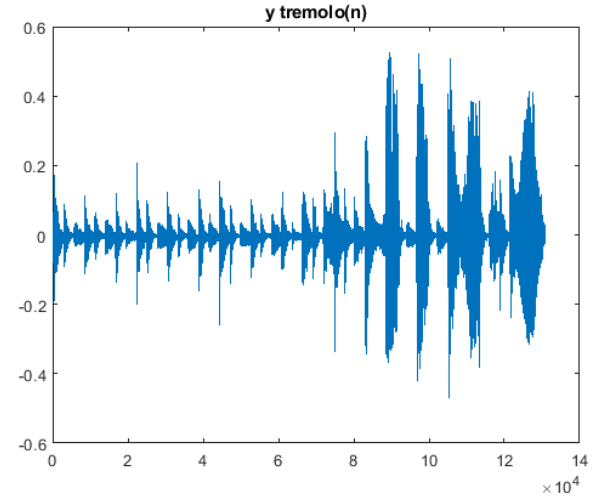


Figura 167: señal de salida Trémolo del top audio system

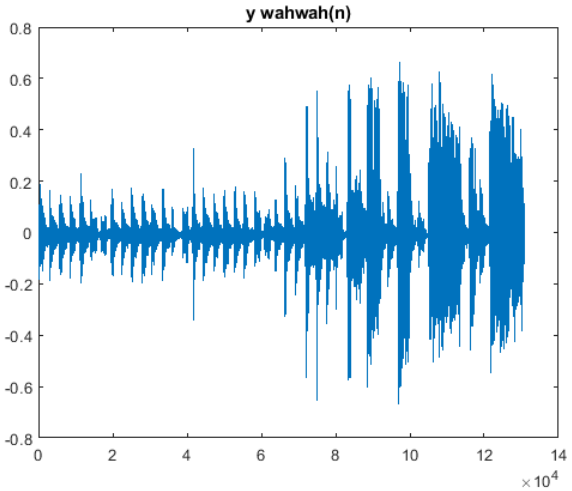


Figura 166: señal de salida Wah-Wah del top audio system

Se observan gráficos similares a los obtenidos en el apartado 3.4.7, así como en las expresiones sonoras al reproducir los audios, por lo que calcularemos el **error**

cuadrático medio entre los datasets obtenidos en Vivado™ (reales), de los efectos individuales, en este apartado y los obtenidos en MATLAB® (ideales) como vimos en el apartado 3.1.3, dando lugar a:

- Para **Echo** se obtiene un error entre ambos datasets de salida del **6,24%**.
- Para **Reverb** se obtiene un error entre ambos datasets de salida del **2,68%**.
- Para **Wah-Wah** se obtiene un error entre ambos datasets de salida del **10,32%**.
- Para **Tremolo** se obtiene un error entre ambos datasets de salida del **6,55%**.

Errores más elevados en esta ocasión debido a la modificación que sufrió el bloque DRCompressor (error que arrastramos todo el camino) visto anteriormente y al hecho de trabajar con un sistema no ideal con una precisión limitada donde se realiza una conversión de tipo de numeración previa y posterior al procesado del sistema.

Sin embargo, estos errores **no son perceptibles** a la escucha cuando reproducimos los audios.

Finalmente las representaciones gráficas obtenidas por los 3 datasets de las salidas de los efectos combinados son los siguientes:

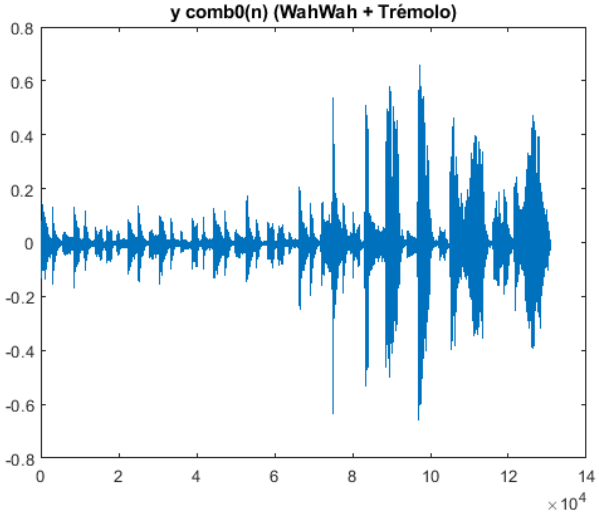


Figura 171: señal de salida comb0 del top audio system

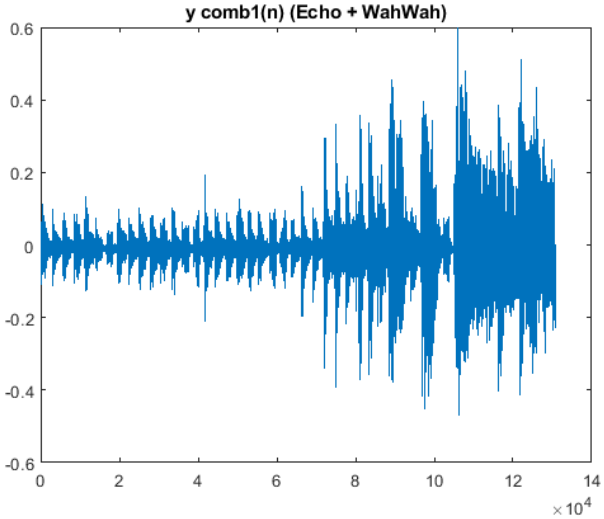


Figura 170: señal de salida comb1 del top audio system

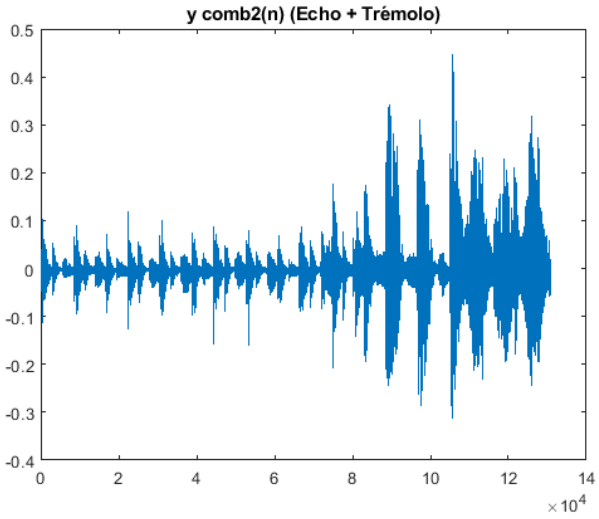


Figura 172: señal de salida comb2 del top audio system

Dando lugar a lo esperado, una mezcla entre ambos efectos en cada caso, resultando en una respuesta sonora que describe perfectamente esta combinación.

4.4.2 Simulaciones post-síntesis

A continuación, se realizará la síntesis, viendo su informe donde se muestre el uso de recursos hardware empleados del dispositivo y pudiendo realizar las simulaciones **post-síntesis** funcional y temporal. Las cuales nos permitirán comprobar el funcionamiento del sistema sintetizado, viendo si cumple o no el comportamiento esperado. Siendo la síntesis el proceso que convierte nuestro código VHDL en un circuito real.

La diferencia entre una simulación **funcional** y una **temporal** es que la funcional se basa únicamente en el funcionamiento del diseño, dejando de lado los tiempos que haya en el mismo como retardos o duración de señales (componentes ideales), sin embargo, la simulación temporal es estricta en cuanto a los tiempos de nuestro sistema, teniendo en cuenta todos los retardos o restricciones temporales.

Al ejecutar la síntesis del sistema obtenemos el siguiente informe donde vemos que la utilización de los puertos entrada/salida es del 55% del total de nuestro dispositivo:

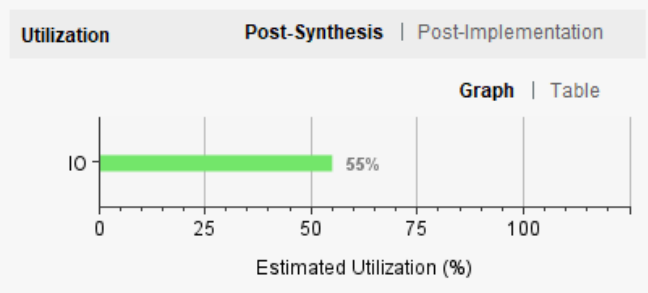


Figura 173: gráfico utilización de recursos estimada post síntesis

Como se ha contado antes, una vez realizada la síntesis del diseño se pasará a realizar las simulaciones temporal y funcional con la totalidad de salidas habilitadas.

En primer lugar se realiza la simulación **funcional**, que podemos ver en la siguiente figura, se obtienen resultados iguales a los de la simulación de comportamiento, a diferencia de que todas las señales de datos de salida se inicializan a cero (“0000”) en lugar de iniciar con datos erróneos/desconocidos. Vemos también a lo largo de la simulación que las señale TVALID de las interfaces maestras de los bloques de salida se mantienen a nivel alto durante toda la duración de la simulación y de la generación de datos.

Además, en estas simulaciones post síntesis y post implementación se ha decidido habilitar tanto las salidas individuales como las salidas de efectos combinados para comprobar la totalidad de los datos de salida posibles.

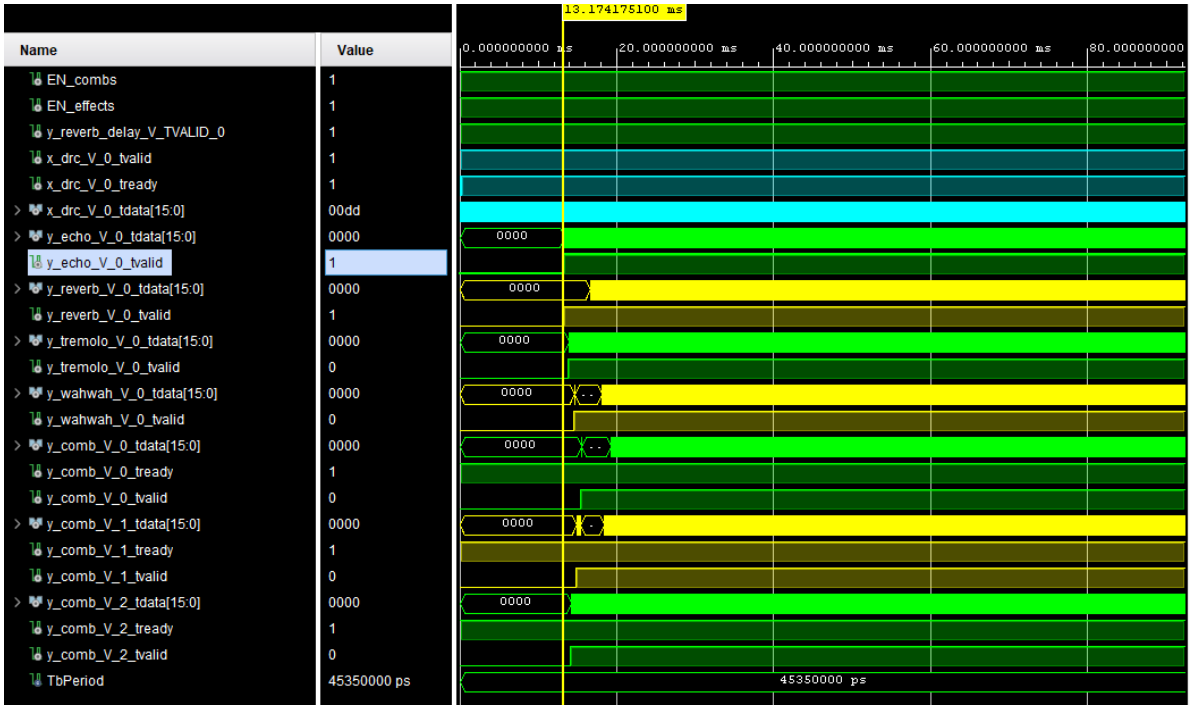


Figura 174: simulación funcional post-síntesis (captura 1)

En la siguiente captura vemos, en detalle, el estado inicial de la simulación funcional post síntesis, viendo como los datos de salida están inicializados a 0 y como, mediante el puerto de entrada ***x_drc***, los datos de entrada empiezan a entrar secuencialmente al sistema.

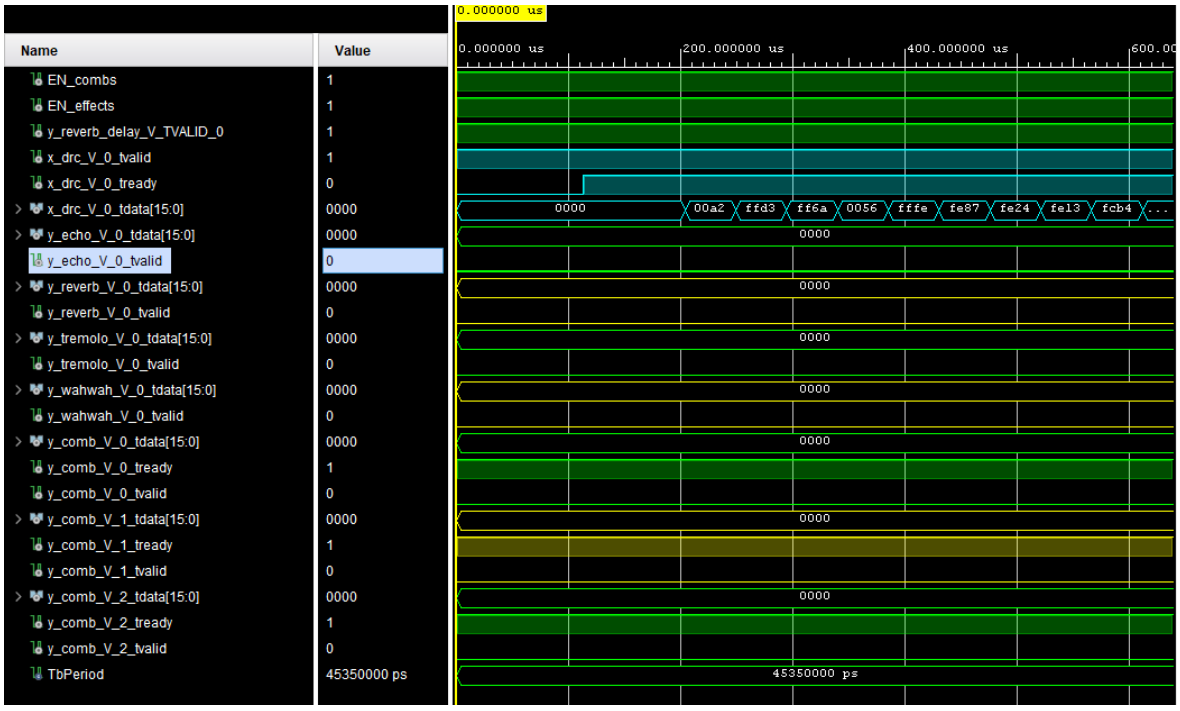


Figura 175: simulación funcional post-síntesis (captura 2)

Los resultados o datos numéricos obtenidos por esta simulación funcional son similares a los conseguidos en el anterior apartado por lo que no se volverán a mostrar.

En segundo lugar se realiza la simulación **temporal**, que podemos ver en las siguientes figuras, se obtienen resultados iguales, en cuanto a los datos obtenidos, a los de la simulación de comportamiento y funcional, pero con valores diferentes en cuanto a tiempos se refiere como veremos a continuación.

La simulación temporal nos da información acerca de los retardos intrínsecos del propio dispositivo, por lo que vamos a poder observar esos pequeños retrasos en momentos donde señales cambian de valor.

En la siguiente captura no se ve nada distinto a las simulaciones funcionales o a les de comportamiento, pero en las siguientes capturas haremos zoom en esos momentos clave que se comentaban antes. Además, la señal de reloj dispone de su frecuencia impuesta por nosotros mismos de 22050 Hz o 45,35 us.

Figura 176: simulación temporal post-síntesis (captura 1)

Como se comentaba antes, en la siguiente figura vemos un zoom en el momento en el que se obtienen datos a la salida del efecto Echo. En las simulaciones de comportamiento o funcionales veríamos como estos datos se obtienen justo cuando en la señal de reloj se realiza un flanco a nivel alto, pero en este caso podemos ver un pequeño delay de **12,15 ns** gracias a los dos cursores temporales añadidos.

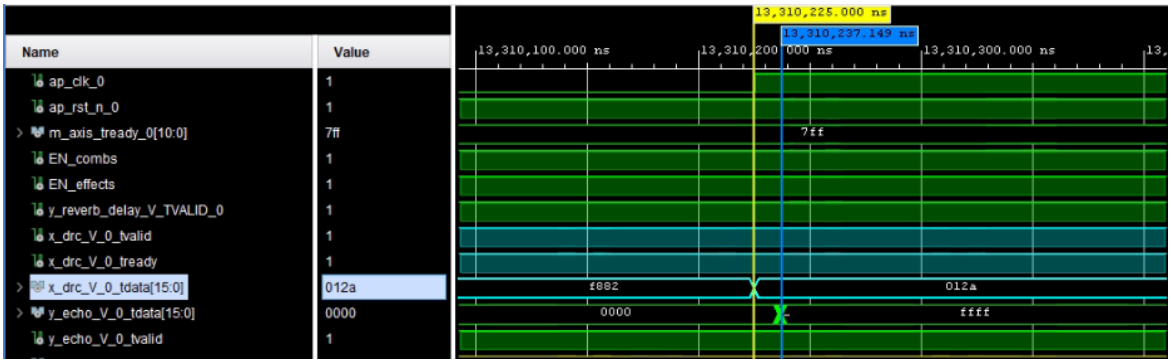


Figura 177: retardo en la salida Echo simulación temporal post-síntesis

Otro ejemplo es el siguiente, donde vemos el retardo, con respecto a la señal de reloj y su flanco a nivel alto, de la activación de la señal TVALID de la salida de la combinación de efectos 1. Se observa un retardo de **13,53 ns**.

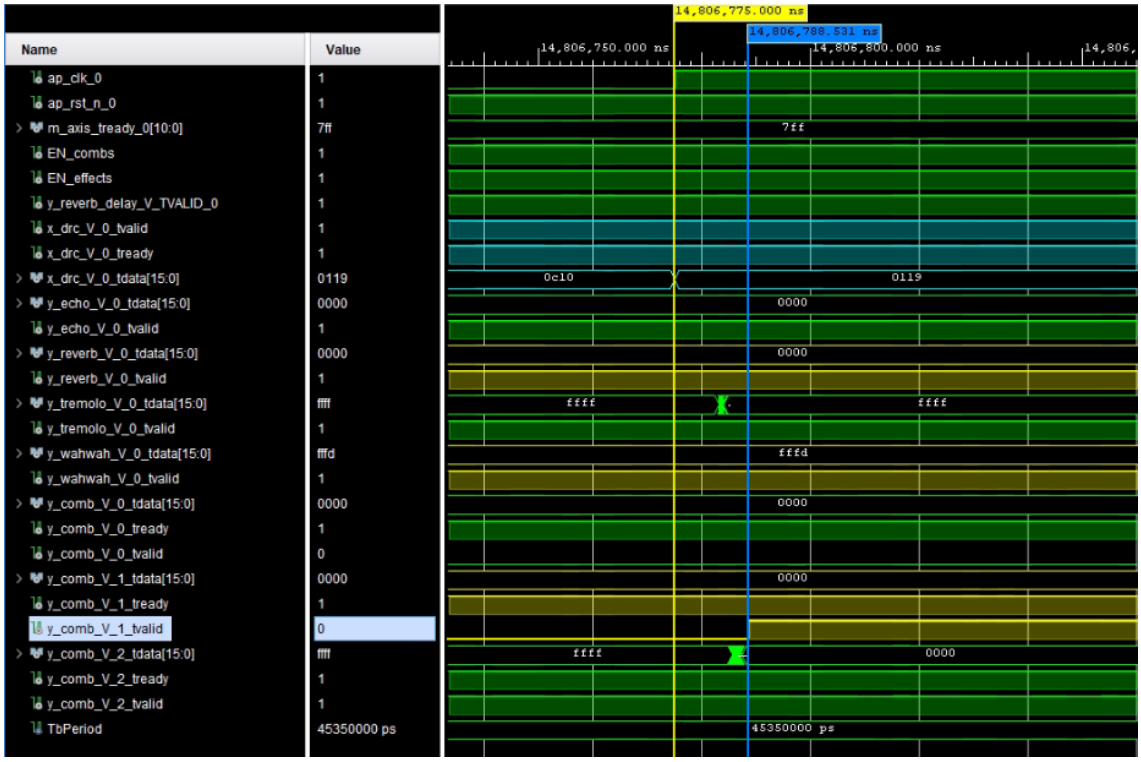


Figura 178: retardo en señal simulación temporal post-síntesis

Al obtener las mismas resultados números en las salidas en ambos modos de simulación no volveremos a comprobar los resultados en MATLAB®.

4.4.3 Simulaciones post-implementación

El siguiente paso a realizar la síntesis del sistema es la implementación del mismo. Este proceso consiste en implementar el "circuito real" o circuito lógico, obtenido en la síntesis, y crear las conexiones físicas entre los recursos hardware del dispositivo en cuestión.

El proceso de implementación prepara el diseño para realizar el "bitstrem" y poder ser volcado a la propia FPGA. Una vez hecha la implementación, podremos ejecutar la simulación temporal y funcional post-implementación.

Cuando finaliza el proceso de implementación obtenemos un informe que nos indica el uso estimado de la totalidad de recursos hardware del dispositivo, donde vemos que la mayor utilización es de los BRAM con un 82%.

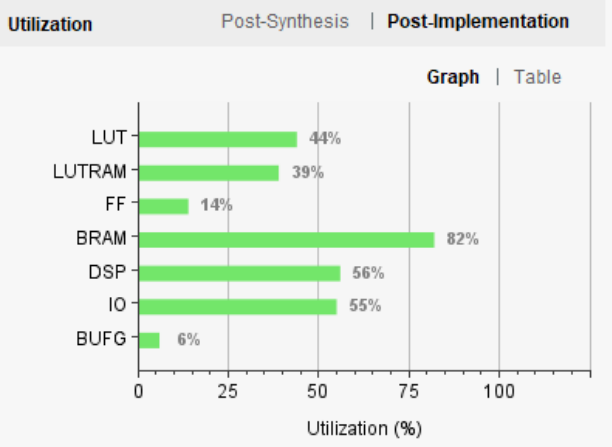


Figura 179: gráfico utilización recursos hardware post implementación

En primer lugar, se realiza la simulación **funcional**, que podemos ver en la siguiente figura (de una captura de la misma), se obtienen resultados iguales a los de la simulación de funcional post síntesis, resultados esperados y positivos que indican la correcta funcionalidad del sistema.

Además, todo la simulación funciona de la misma manera que lo hacia la simulación de comportamiento, a excepción de obtener los datos de salida inicializados a "0000" como vimos en la simulación funcional post-síntesis.

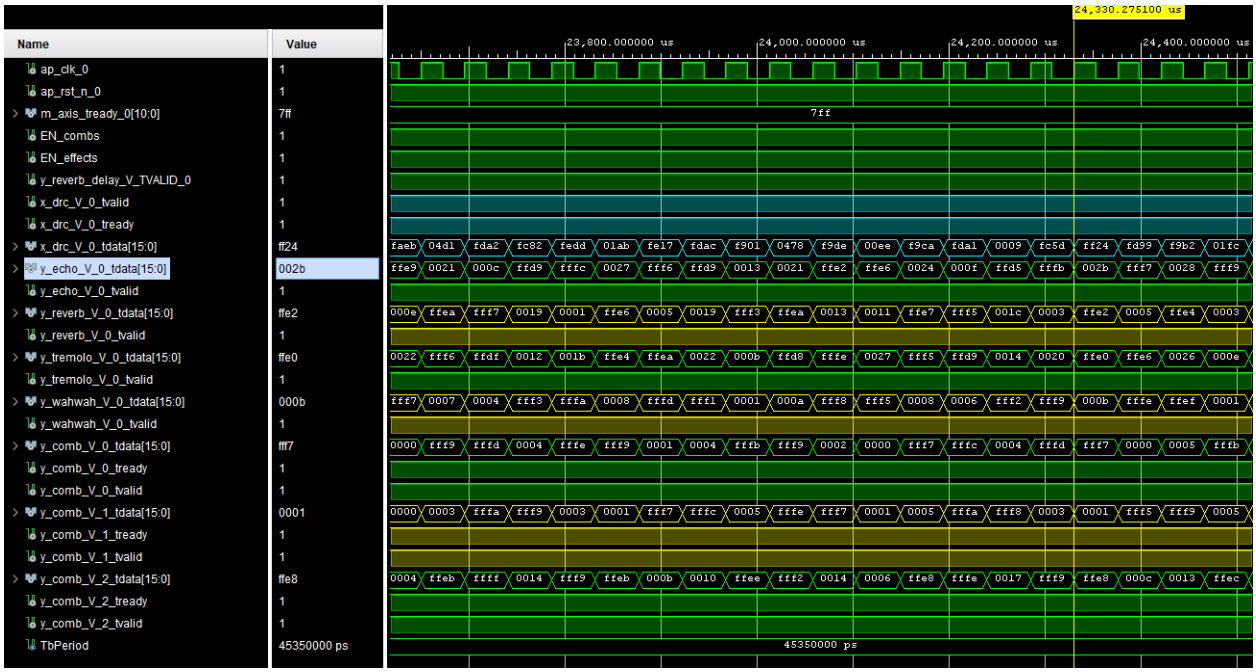


Figura 180: simulación funcional post implementación (captura 1)

En la siguiente captura vemos el momento en el que la señal TVALID de la salida del efecto Wah-Wah se pone a nivel alto, habilitando la validez de los datos generados y mostrándolos a la salida.

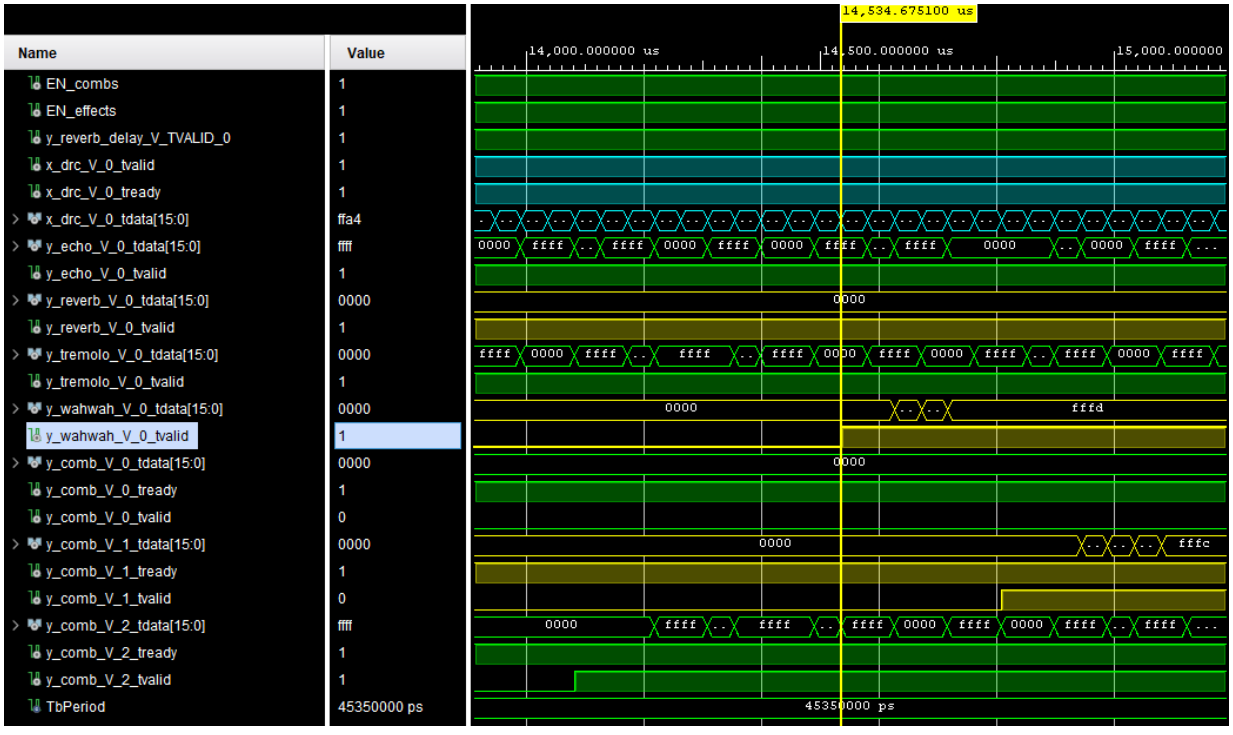


Figura 181: simulación funcional post implementación (captura 2)

Por último, se realiza la simulación **temporal**, que podemos ver en las siguientes figuras, muy similar a la realizada en la post-síntesis, siendo esta simulación lo más cercano que podremos ver en cuanto a nivel de realidad si descargásemos el programa en placa en cuanto a retardos se refiere.

En esta primera captura se observa, al igual que en la simulación temporal post-síntesis, un comportamiento similar al de otras simulaciones vistas anteriormente, pero en las figuras posteriores haremos zoom en momentos clave para comprobar esos pequeños delays generados al no tener componentes temporales ideales.

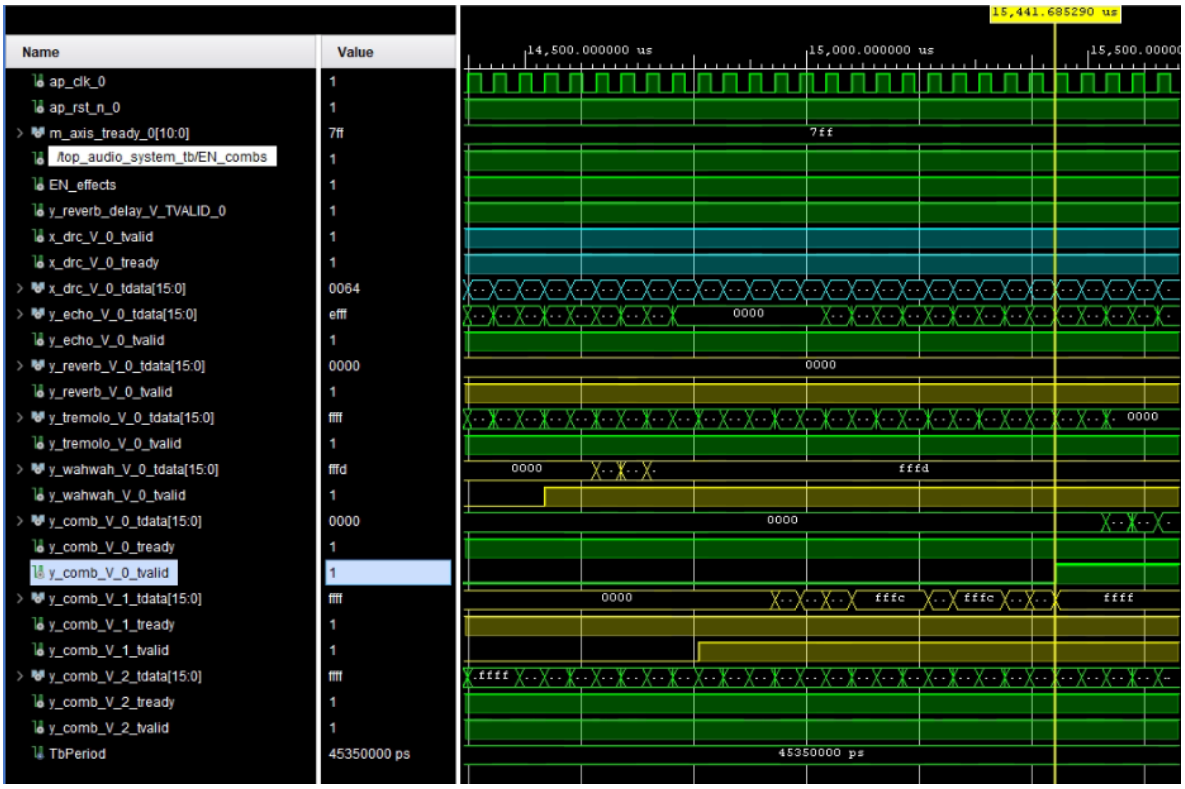


Figura 182: simulación temporal post-implementación (captura 1)

En la siguiente figura vemos un ejemplo de retardo de la señal TVALID de la salida del efecto Trémolo, donde, con respecto al flanco a nivel alto de la señal de reloj, tenemos un delay de **9,32 ns**.

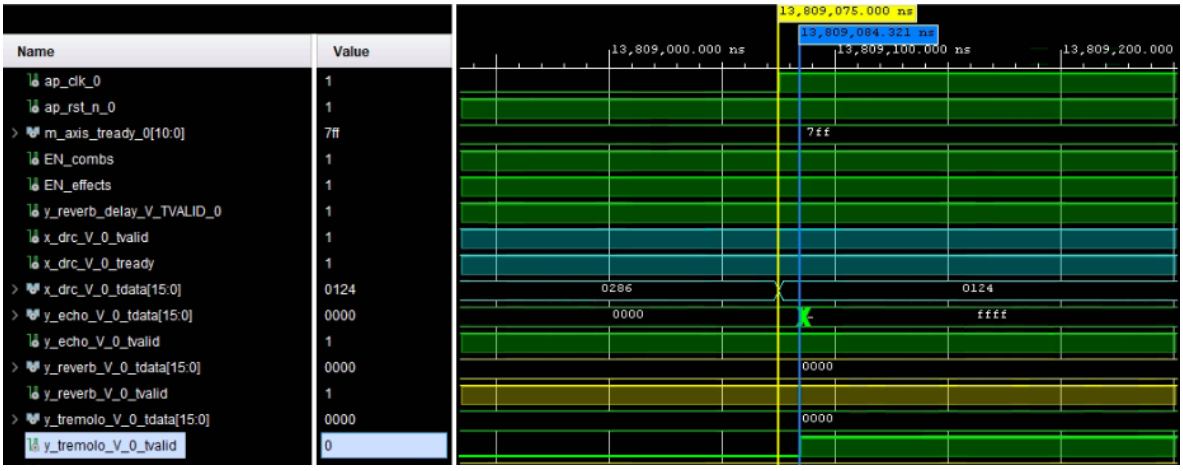


Figura 183: retardo en señal simulación temporal post-implementación

Por último, se visualiza una captura donde, a los 26,28 ms, vemos el retardo obtenido en la aparición del siguiente dato de la señal de salida del efecto Reverb con respecto a la señal de reloj y su flanco de subida, siendo este retardo de **9,22 ns**.

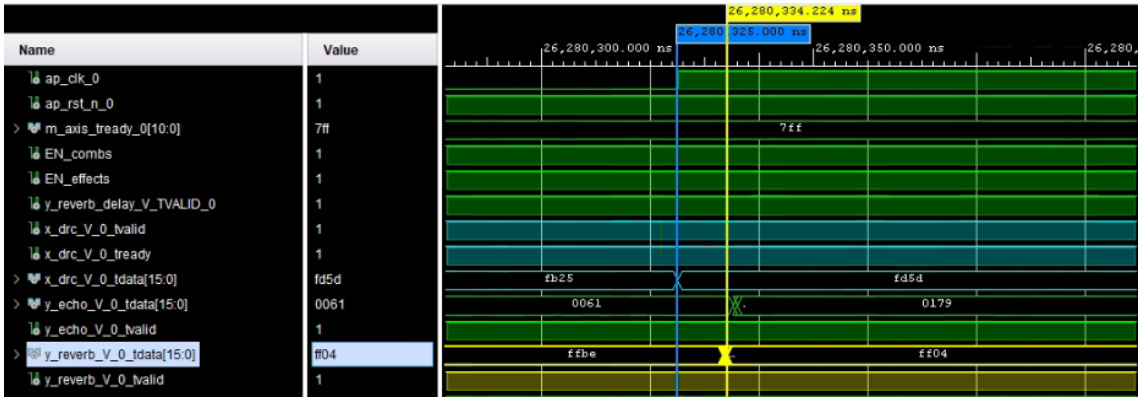


Figura 184: retardo en la salida Reverb simulación temporal post-implementación

Estas comprobaciones vistas en las distintas figuras nos han servido para ver que los tiempos en nuestro dispositivo no son ni ideales ni inmediatos y que afectan a nuestras señales directamente, aunque los mismos sean del orden de nano segundos, teniendo cada señal su propio retardo particular.

Comentar también que en este caso los resultados numéricos obtenidos en las diferentes 7 salida han sido los mismos que hemos obtenido en todas las demás simulaciones y que ya pudimos ver en el apartado 4.4.1 por lo que no se volverán a mostrar.

5. Conclusiones y trabajos futuros

El presente trabajo ha tratado sobre el desarrollo de un sistema de procesado digital de audio implementado en la tecnología FPGA el cual se apoye en el empleo de la síntesis de alto nivel (HLS). Esto ha conllevado a hacer un estudio previo sobre la tecnología FPGA, la síntesis de alto nivel, los sistemas de procesado de audio y sus algoritmos.

Nuestro objetivo principal fue desarrollar este sistema de procesado de audio, por lo que entre las diferentes opciones tecnológicas a nuestro alcance se vio que una FPGA era la opción idónea para ello, debido a su gran eficiencia en aplicaciones de procesado de datos, su alto grado de paralelismo, su reprogramabilidad o su flexibilidad con respecto a un procesador común.

El hacer uso de la síntesis de alto nivel para nuestra aplicación, deriva en que elaboremos un sistema de bloques IP los cuales se comuniquen entre sí mediante un protocolo de comunicaciones digitales, habiendo previamente desarrollado cada bloque según unos pasos que nos propusimos para realizar el proceso de la mejor manera posible.

Además, el hecho de que se eligiese al audio como protagonista, en el procesado de nuestros datos, conllevó a realizar una selección de algoritmos que serían transformados en bloques IP basados en lenguaje hardware o VHDL, conformando un diseño de bloques el cual tendría como objetivo tratar el audio para generar modificaciones audibles en él así como mejorar ciertos aspectos de dicho fichero, ya que los datos o muestras de audio a procesar provenían de un audio pregrabado de tipo ".wav" del cual obteníamos dichas muestras en un fichero ".txt" dándoles el nombre de dataset.

Todos los algoritmos seleccionados han sido estudiados, desarrollados e implementados, de modo que mediante MATLAB® desarrollaríamos cada uno de ellos como modelos ideales, para posteriormente hacer lo mismo en Vivado HLS™ y finalmente volver a MATLAB® para validar los resultados obtenidos. Siendo el objetivo final repetir este proceso con cada uno de los bloques o algoritmos y crear nuestro sistema en Vivado™, el cual también dispondría de bloques nativos del propio software.

Cabe destacar, que lo que hace que nuestra aplicación o sistema encaje a la perfección con lo que una FPGA puede ofrecer, es el hecho de disponer de un paralelismo a la hora de procesar señales en tiempo real y que seamos capaces de reprogramar el sistema. Esto se representa de una manera muy visual en el banco de efectos, con los efectos individuales y los combinados.

Este proceso de creación de bloques IP necesitó de conceptos a implementar como la codificación en coma fija y la precisión de nuestros datos, el uso de directivas de optimización que mejoraran distintos parámetros del bloque o la validación de los mismos bloques mediante el estudio de los errores generados entre el modelo ideal y el real.

Se creó un diseño final, que albergaba todo nuestro diseño previo, basado en el

protocolo AXI4-Stream, el cual encajaba perfectamente con la idea de funcionamiento del sistema, secuencial y unidireccional, de modo que se estudió y se implementó en el diseño para finalmente simular el mismo y obtener los resultados finales a modo de datasets de salida binarios transformados posteriormente en audios.

Con todo ello, pudimos comprobar unos resultados positivos a pesar de tener cierto margen de mejora a la hora de determinar los errores finales de cada dataset de salida, aunque dichos errores no fueran perceptibles al oído. Además, el sistema pasó el proceso de implementación, dejándolo listo para una posible descarga en placa.

Por lo que finalmente, se obtuvo una aplicación para el procesado de audio totalmente implementable en una FPGA de familia Artix-7, obteniendo todo el conocimiento estudiado en este trabajo como otro resultado positivo. Además, se ha desarrollado una aplicación parara FPGAs de una forma diferente a la habitual al haber hecho uso de la síntesis de alto nivel, comprobando, al mirar el proyecto con perspectiva, su gran utilidad y su valor añadido tan positivo que ha sido capaz de aportar al proyecto.

En **trabajos futuros** este sistema podría ser mejorado o ampliado de varias formas:

- Implementación de nuevos algoritmos como un compresor de ficheros de audio sin pérdidas, a pesar que este algoritmo se tratase como una única aplicación debido a su envergadura.

- Implementación de un algoritmo de detección de tono musical o de modificación de tono, los cuales usan algoritmos de mayor complejidad.

- Mejora del efecto “Reverb” mediante la creación de un “Schroeder Reverb”, el cual también podría tratarse de una única aplicación o sistema debido a los altos recursos de memoria que demanda el mismo.

- En relación con el anterior punto, podríamos tratar audios de mayor tamaño o implementar algoritmos con mayor requerimiento de memoria mediante el uso de memorias externas físicas, que el usuario manejaría.

- Descarga de sistema en placa. Sistema de adquisición de datos para capturar audio desde un micrófono (Conversión Analógico-Digital), procesarlo mediante el sistema creado y reproducirlo en un altavoz en tiempo real (Conversión Digital-Analógico).

6. Bibliografía

[1] "Unidad 33 – La segunda revolución del sonido, del rock a la electrónica". HISTORIA DE LA MÚSICA. [Online]. Available: https://bustena.wordpress.com/historia-de-la-musica-online/el-siglo-xx-y-la-era-del-sonido/unidad-33-2/ [Accessed: Jul. 4, 2022].

[2] M. Carrington. "Pequeñas historias del sonido (II): más sobre el compresor". Hispasonic. 14, May. 2013. [Online]. Available: https://www.hispasonic.com/reportajes/pequenas-historias-sonido-ii/38141 [Accessed: Jul. 4, 2022].

[3] "What is an FPGA? Field Programmable Gate Array". Xilinx. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html [Accessed: Jul. 10, 2022].

[4] "Field-programmable gate array - Wikipedia, la enciclopedia libre". Wikipedia, la enciclopedia libre. 23, May. 2004. [Online]. Available: https://es.wikipedia.org/wiki/Field-programmable_gate_array [Accessed: Jul. 10, 2022].

[5] "CLB(Configurable Logic Block) Wiki - FPGAkey". FPGAkey - Best Resource For Online FPGA. [Online]. Available: https://www.fpgakey.com/wiki/details/51 [Accessed: Jul. 11, 2022].

[6] S. Anuar and M. Nazrin "Field Programmable Gate Array (FPGA): From Conventional to Modern Architectures" in *Digital and Analogue Electronics Circuits and Systems* (pp. 53) Chapter: 5, Jan. 2015.

[7] R. Kaibou, M. Salah and A. Maali, "FPGA HW/SW Codesign Approach for Real-time Image Processing Using HLS" in 1st International Conference on Communitations, Control Systems and Signal Processing. May. 2020.

[8] "Search Results". Xilinx - Adaptable. Intelligent | together we advance_. 20, Aug. 2018. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/search.html?searchQuery=pragma+HLS+pipeline [Accessed: Jul. 20, 2022].

[9] "pragma HLS pipeline". Xilinx - Adaptable. Intelligent | together we advance_. 20, Aug. 2018. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/fde1504034360078.html?hl=pragma,hls,pipeline [Accessed: Jul. 20, 2022].

[10] "pragma HLS dataflow". Xilinx - Adaptable. Intelligent | together we advance_. 20, Aug. 2018. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/sxx1504034358866.html?hl=pragma,hls,pipeline [Accessed: Jul. 20, 2022].

[11] ARM: *AMBA® AXI™ and ACE™ Protocol Specificationk*, Oct. 28, 2011.

[12] "Xilinx AXI Stream tutorial – Part 1". FPGA Site. 15, Jul. 2017. [Online]. Available: http://fpgasite.blogspot.com/2017/07/xilinx-axi-stream-tutorial-part-1.html [Accessed: Aug. 3, 2022].

[13] Ignacio Bravo Muñoz, *Representación en Coma Fija*, Apr. 2022.

[14] Xilinx: *Data Types, Vivado HLS 2013.3 Version*, 2013.

[15] L. Tan and J. Jiang. "Signal Sampling and Quantization" in *Digital Signal Processing (Third Edition)* Chapter 2. ScienceDirect. 2019. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/shannon-sampling-theorem [Accessed: Aug. 12, 2022].

[16] "WAVE PCM soundfile format". [Online]. Available: http://soundfile.sapp.org/doc/WaveFormat/ [Accessed: Setp. 1, 2022].

[17] Udo Zölzer, Ed., *DAFX: Digital Audio Effects, Second Edition*, United Kingdom: John Wiley & Sons, 2011.

[18] Udo Zölzer, Ed., *Digital Audio Signal Processing, Second Edition*, United Kingdom: John Wiley & Sons, 2011.

[19] "Compresor (sonido) - Wikipedia, la enciclopedia libre". Wikipedia, la enciclopedia libre. 13, Jul. 2022. [Online]. Available: https://es.wikipedia.org/wiki/Compresor_(sonido) [Accessed: Jul. 27, 2022].

[20] "¿Qué es un Compresor de Audio? (Una manera efectiva de comprenderlo)". Ser Productor de Música. [Online]. Available: https://serproductordemusica.com/que-es-un-compresor-de-audio-una-manera-efectiva-de-comprenderlo/ [Accessed: Jul. 27, 2022].

[21] "FIR (Finite Impulse Response) - Wikipedia, la enciclopedia libre". Wikipedia, la enciclopedia libre. 29, Jul. 2019. [Online]. Available: https://es.wikipedia.org/wiki/FIR_(Finite_Impulse_Response) [Accessed: Aug. 11, 2022].

[22] "IIR - Wikipedia, la enciclopedia libre". Wikipedia, la enciclopedia libre. 29, Jul. 2019. [Online]. Available: https://es.wikipedia.org/wiki/IIR#:~:text=IIR%20es%20una%20sigla%20en,decir%2C%20nunca%20vuelve%20al%20reposo. [Accessed: Aug. 11, 2022].

[23] M. Olmo "Reverberación". [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbasees/Acoustic/reverb.html [Accessed: Aug. 21, 2022].

[24] Universidad Miguel Hernández. 9, Dic. 2010 [Online]. Available: https://sites.google.com/site/umhcopia/ejercicio-grupo-4?tmpl=%2Fsystem%2Fapp%2Ftemplates%2Fprint%2F&showPrintDialog=1 [Accessed: Aug. 23, 2022].

[25] F. Andriollo, F. Spitale and L. G. Castellanos “Procesadores de efectos” Universidad Tecnológica Nacional, Argentina, Jun. 2011.

[26] "Tremolo". Hack Audio. 2020. [Online]. Available: https://www.hackaudio.com/digital-signal-processing/combining-signals/tremolo/ [Accessed: Aug. 25, 2022].

7. Pliego de condiciones

En este capítulo se definen las condiciones o requisitos hardware y software necesarios para poder realizar este proyecto al completo, desde todo el desarrollo práctico del sistema hasta la elaboración de esta memoria, son las siguientes.

7.1 Condiciones hardware

Ordenador Personal:

- Procesador: Intel® Core i5-7500 CPU (3,4 GHz).
- Memoria RAM: 32 GB.

7.2 Condiciones software

Condición general:

- Sistema operativo: Windows® 10 de 64 bits.

Condición para el desarrollo de la práctica:

- Vivado™ HLS 2017.4.
- Vivado™ ML Edition 2022.1.
- MATLAB® 2022a.

Condición para el desarrollo de la memoria:

- Microsoft® Office Word 2013.

8. Manual de usuario

En el presente capítulo se desarrollará un manual de usuario que tendrá como objetivo explicar, de la manera más sencilla y gráfica, como manejar el sistema TOP AUDIO SYSTEM, ejecutarlo y obtener los resultados como hemos visto en el desarrollo del proyecto.

Para ello deberemos de tener en cuenta el capítulo anterior donde se especifican las condiciones hardware y software necesarios para ello.

Los pasos a seguir para el completar el objetivo propuesto en este capítulo son los siguientes:

- **Apertura del proyecto final: TOP_AUDIO_SYSTEM**

  - Abrir Vivado™ 2022.1 clicando doblemente sobre el acceso directo.

  - Abrir el proyecto TOP AUDIO SYSTEM clicando (en barra de navegación) *File -> Open Proyect* -> seleccionar fichero "***TOP_AUDIO_SYSTEM.xpr***" en el directorio del proyecto.

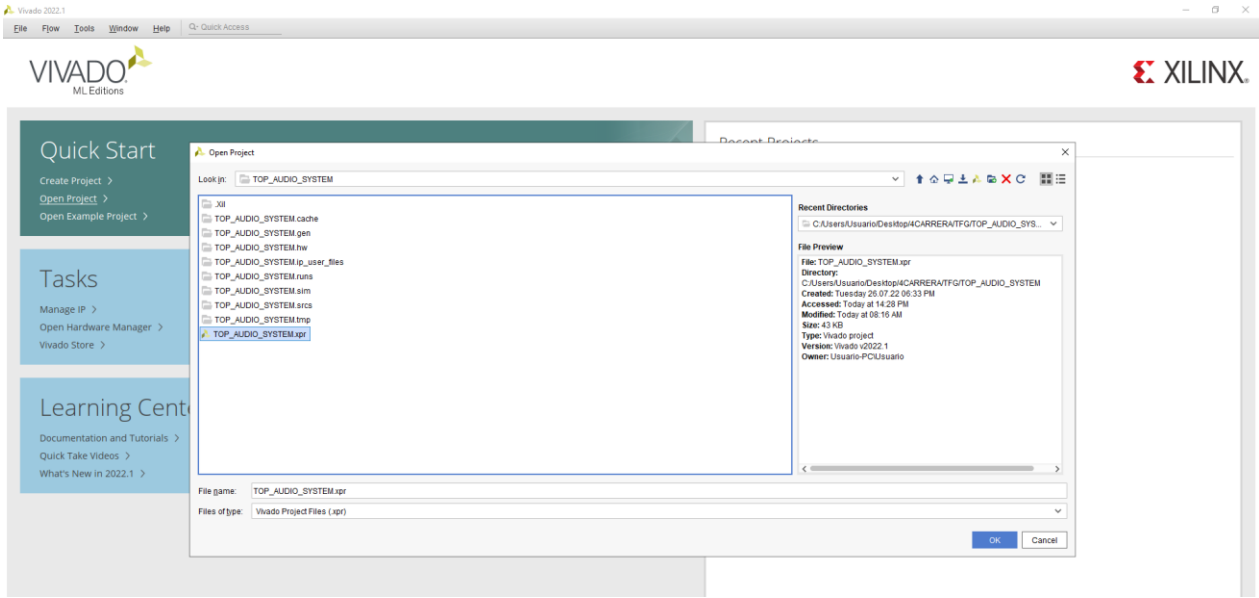


Figura 185: apertura proyecto TOP AUDIO SYSTEM

- **Importación de los bloques IP-core generados durante el trabajo**

  - Seleccionamos (en barra de navegación) -> *Tools -> Settings.*

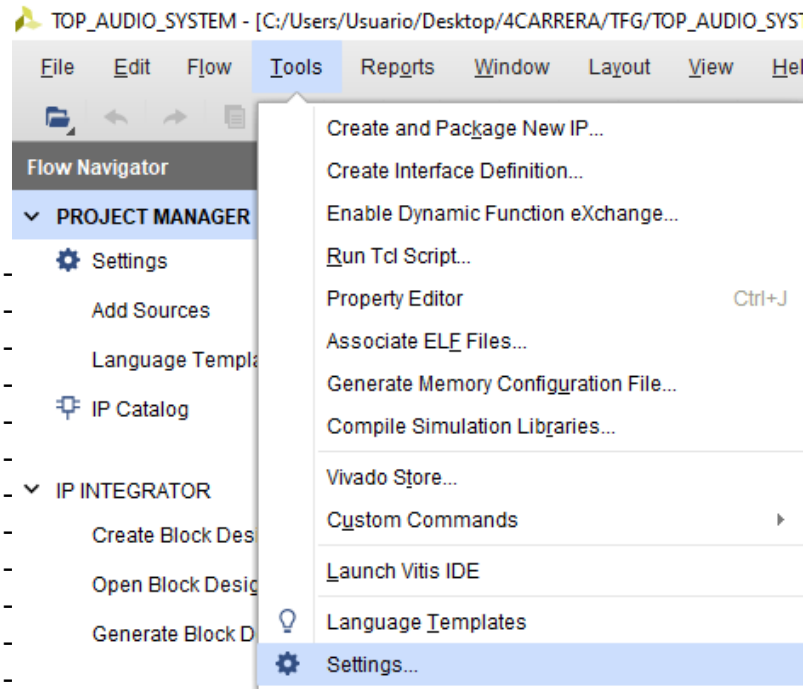


Figura 186: selección de "Settings"

- Una vez en el desplegable ***Proyect Setting***, seleccionamos: *IP -> Repository*.

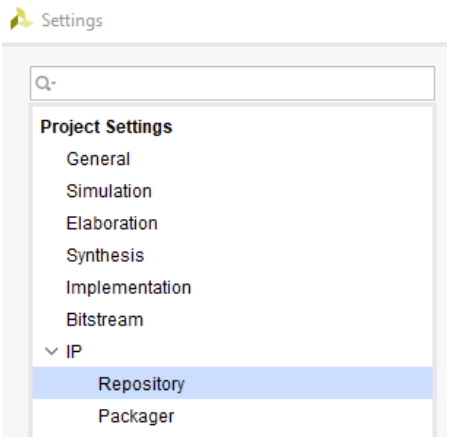


Figura 187: Selección del repositorio de IPs

- Seleccionamos el icono "+" de la figura y añadiremos los directorios llamados "***ip***" de los 5 bloques creados. Los directorios llamados "***ip***" se encuentran dentro del directorio global de cada bloque: *solution1 ->impl -> **ip***.

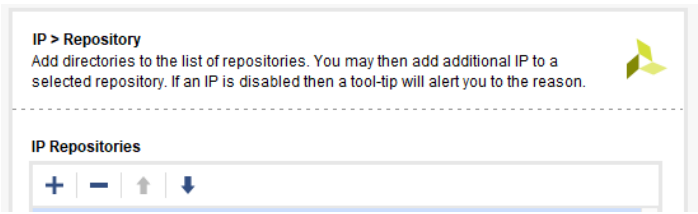


Figura 188: añadir directorio IP

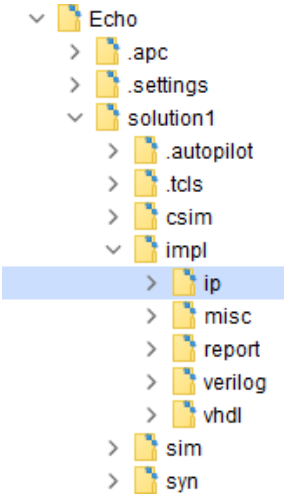


Figura 189: ejemplo ubicación directorio IP

➢ Modificaciones de ubicación de datasets en testbench

- Abrimos nuestro fichero testbench ubicado en el panel *Sources*, en la carpeta *Simulation Sources -> sim_1 ->* ***top_audio_system_tb***.

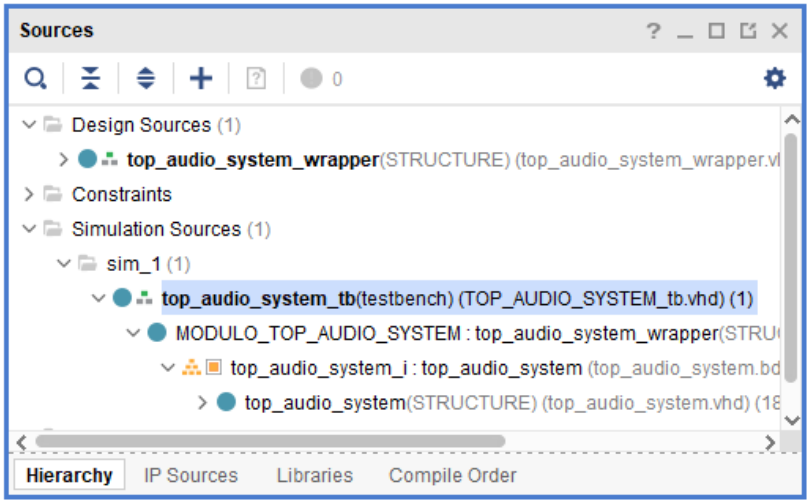


Figura 190: ubicación testbench

- Modificamos las líneas 171, 174, 175, 176, 177, 180, 181, 182 con las ubicaciones de dichos ficheros en el ordenador que estemos empleando.

```
        file_open(dataset_in, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_in_bin.txt", read_mode);

if EN_effects = '1' then
        file_open(dataset_out_echo, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_echo_bin.txt", write_mode);
        file_open(dataset_out_wahwah, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_wahwah_bin.txt", write_mode);
        file_open(dataset_out_reverb, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_reverb_bin.txt", write_mode);
    file_open(dataset_out_tremolo, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_tremolo_bin.txt", write_mode);
end if;
if EN_combs = '1' then
    file_open(dataset_out_comb0, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_comb0_bin.txt", write_mode);
    file_open(dataset_out_comb1, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_comb1_bin.txt", write_mode);
    file_open(dataset_out_comb2, "C:\Users\Usuario\Desktop\4CARRERA\TFG\TOP_AUDIO_SYSTEM\dataset_out_comb2_bin.txt", write_mode);
```

Figura 191: modificación de ubicaciones dataset.

➢ Simulación de comportamiento del sistema

Una vez realizado los pasos anteriores ya podemos empezar con las simulaciones del sistema.

- A la izquierda de la pantalla nos encontramos con el *Flow Navigator*, donde seleccionaremos *Run Simulation* dentro de *SIMULATION*. Se desplegará una ventana donde seleccionaremos *Run Behavioral Simulation* y comenzará la simulación.

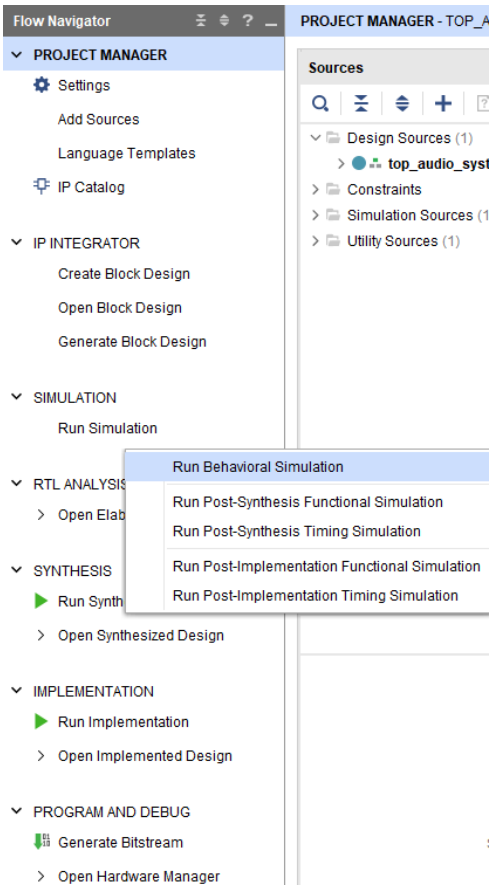


Figura 192: lanzar simulación de comportamiento

Una vez el proceso de simulación haya finalizado nos aparecerá en pantalla la ventana de señales. Para ver la simulación y obtener los resultados (vistos en anteriores capítulos del trabajo) seleccionaremos 6,2 segundos en el tiempo de salto de simulación. De este modo será tiempo de sobra para generar los ficheros de muestras que representarán los 5,94 segundos de audio (tiempo de reproducción del audio de entrada).

- Seleccionamos los 6.2 s y lanzamos la simulación con el botón de ejecución finita de tiempo.

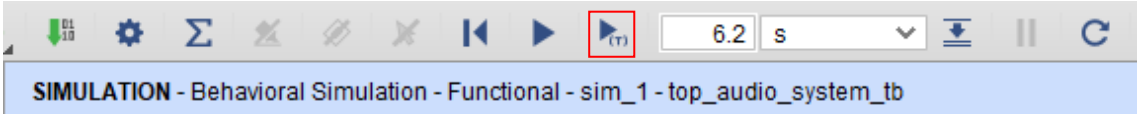


Figura 193: ejecución simulación de comportamiento

- Al finalizar la simulación podemos comprobar cómo los ficheros de salida (ubicados según el usuario) correspondientes disponen de todas las muestras generadas.

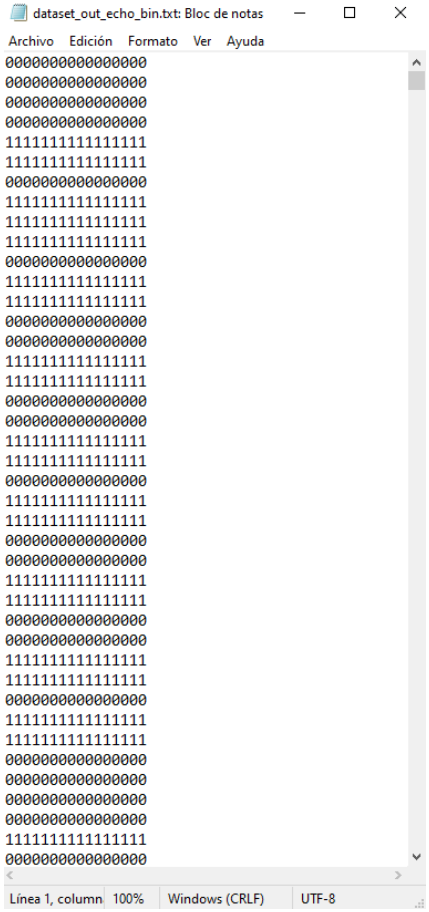


Figura 194: dataset binario de salida formato txt

➢ Síntesis y simulaciones post-síntesis

El siguiente paso es lanzar la síntesis y posteriormente, sus simulaciones.

- A la izquierda de la pantalla nos encontramos con el *Flow Navigator*, donde seleccionaremos *Run Synthesis* dentro de *SYNTHESIS*.

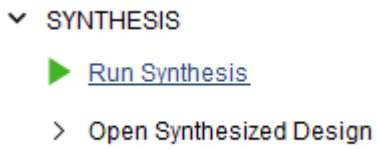


Figura 195: lanzar síntesis del sistema

- Al completar la síntesis nos aparece en la ventana *Project Summary* los detalles de la misma, así como el gráfico de utilización de puertos entrada/salida visto en el capítulo 4.

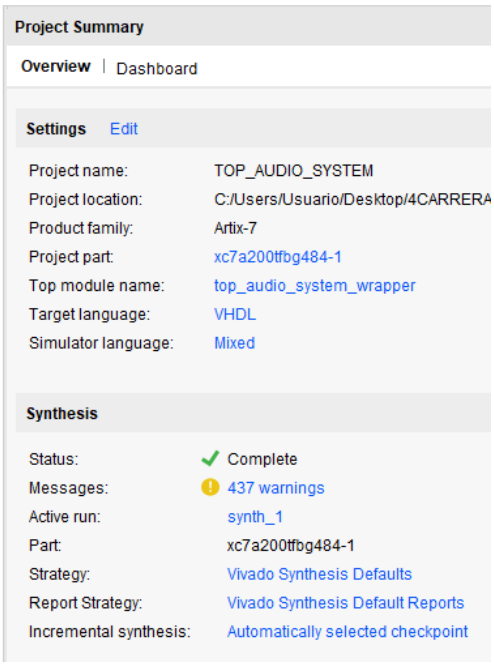


Figura 196: estado de la síntesis

- Seguidamente, lanzaremos las simulaciones temporales y funcionales, *Flow Navigator -> SIMULATION -> Run Simulation -> Run Post-Synthesis Functional Simulation/ Run Post-Synthesis Timing Simulation*.

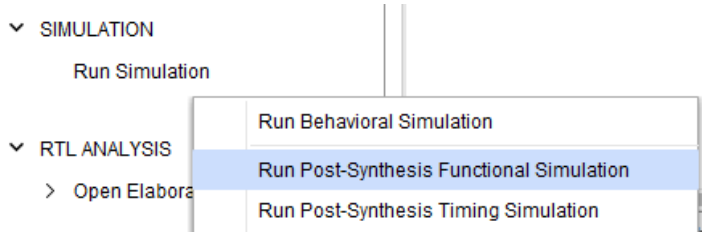


Figura 197: lanzar simulaciones post síntesis

Una vez el proceso de simulación haya finalizado nos aparecerá en pantalla la ventana de señales, donde seguiremos los mismos pasos que cuando lanzamos la simulación de comportamiento para generar los resultados y comprobar la simulación.

➢ Implementación y simulaciones post-implementación

El siguiente paso es lanzar la implementación y posteriormente, sus simulaciones.

- A la izquierda de la pantalla nos encontramos con el *Flow Navigator*, donde seleccionaremos *Run Implementation* dentro de *IMPLEMENTATION*.

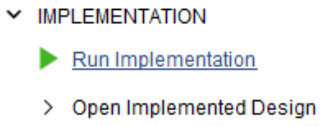


Figura 198: lanzar implementación del sistema

- Al completar la implementación nos aparece en la ventana *Project Summary* los detalles de la misma, así como el gráfico de utilización de recursos hardware del sistema visto en el capítulo 4.

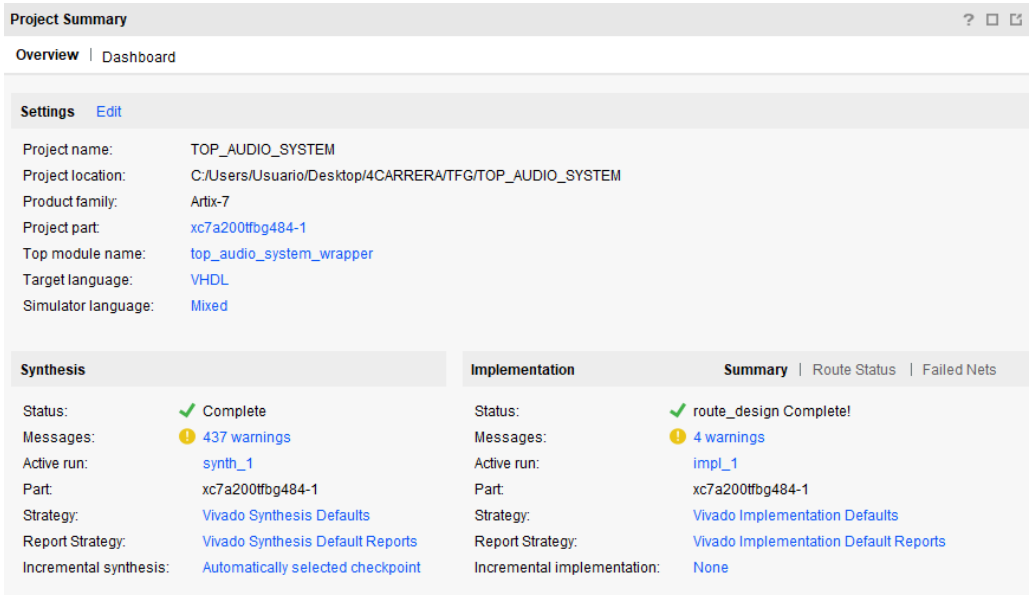


Figura 199: estado de la implementación

- Seguidamente, lanzaremos las simulaciones temporales y funcionales, *Flow Navigator -> SIMULATION -> Run Simulation -> Run Post-Implementation Functional Simulation/ Run Post- Implementation Timing Simulation.*

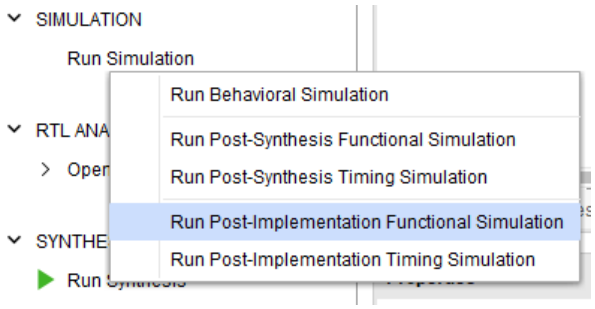


Figura 200: lanzar simulaciones post implementación

Una vez el proceso de simulación haya finalizado nos aparecerá en pantalla la ventana de señales, donde seguiremos los mismos pasos que cuando lanzamos la simulación de comportamiento.

➢ Comprobación de resultados obtenidos vía MATLAB®

Una vez hechos los pasos anteriores pasaremos a comprobar los resultados obtenidos por las simulaciones del sistema, estos resultados ya fueron mostrados en el capítulo 4.

- Abrir MATLAB® 2022a clicando doblemente sobre el acceso directo.

- Abrir el script llamado "***analisisDatasets.m***"

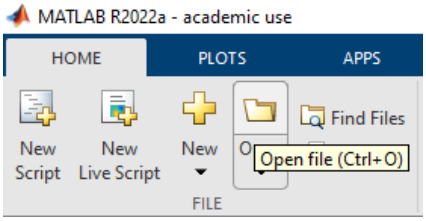


Figura 201: apertura script

- Abrir el script llamado "***analisisDatasets.m***". Script en el cual se analizan, los datasets de unidades reales que se fueron obteniendo durante el transcurso del trabajo por los bloques generados y los datasets binarios. La estructura del script es la siguiente:

    1. Lectura ficheros decimales provenientes de las salidas de los bloques de Vivado™ HLS.
    2. Cálculo error cuadrático medio de *y_drc* de HLS comparada con la obtenida en MATLAB®.
    3. Cálculo error cuadrático medio de la salida del efecto que queramos de HLS comparada con la obtenida en MATLAB®.
    4. **Lectura ficheros binarios.**
    5. Cálculo error cuadrático medio de la salida del efecto que queramos del sistema final de Vivado™ comparada con la obtenida en MATLAB®.

- En nuestro caso, solo nos interesa el punto 4. Comentaremos todas las líneas del script menos las líneas entre la 70 y la 118.

```
68  %%%%%%%%%%%% LECTURA FICHEROS BINARIOS %%%%%%%%%%%%
69  %%%%%%%%%%%%%%%%%%%%%%%% AUDIO IN %%%%%%%%%%%%%%%%%%%%%%%%
70  fp_bin = fopen('dataset_in_bin.txt', 'rb');
71  x_line = fgetl(fp_bin);
72  i=1;
73
74  x_bin = zeros(1, length(y_drc));
75  x_dec = zeros(1, length(y_drc));
76  x_in = zeros(1, length(y_drc));
77  while ischar(x_line)
78      x_dec(i) = bin2dec(x_line);
79      if(x_dec(i) > 32767)        %número negativo
80          x_dec(i) = 2^16 - x_dec(i);
81          x_in(i) = -x_dec(i)/(2^14);
82      else                        %número positivo
83          x_in(i) = x_dec(i)/(2^14);
84      end
85      i=i+1;
86      x_line = fgetl(fp_bin);
87  end
88  fclose(fp_bin);
89  figure
90  plot(x_in)
91  title("x in (n)")
92  sound(x_in, 22050)
```

Figura 202: código para leer y analizar dataset binario de entrada

- En primer lugar, comprobaremos que el audio de entrada "contenido" en el fichero "*dataset_in_bin.txt*" sea correcto, para ello lanzaremos la simulación clicando al icono "*Run*".

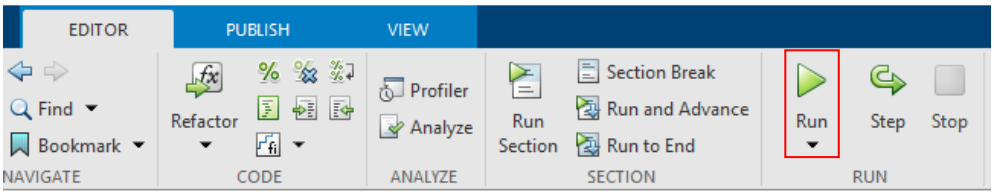


Figura 203: ejecutamos script

- Al finalizar la simulación escuchamos el audio de entrada y vemos su representación gráfica de onda ya conocida en el trabajo.

- Comentamos de la línea 89 a la 92 al no querer volver a ver los resultados de la señal de entrada.

- En la siguiente parte de código ejecutaremos el mismo algoritmo pero esta vez abriendo el fichero de salida binario que nosotros queramos, pudiendo elegir entre los 7 posibles a generar en el sistema. Únicamente cambiaremos el nombre del archivo para comprobar cada salida del sistema.

```
94  %%%%%%%%%%%%%%%%%%%%%% SALIDAS DEL TOP AUDIO SYSTEM %%%%%%%%%%%%%%%%%%%%%%
95  fp_bin2 = fopen('dataset_out_echo_bin.txt', 'rb'); %cambiar fichero segun queramos
96  y_line = fgetl(fp_bin2);
97  j=1;
98
99  y_bin = zeros(1, length(y_drc));
100 y_dec = zeros(1, length(y_drc));
101 y = zeros(1, length(y_drc));
102 while ischar(y_line)
103     y_dec(j) = bin2dec(y_line);
104     if(y_dec(j) > 32767)            %número negativo
105         y_dec(j) = 2^16 - y_dec(j);
106         y(j) = -y_dec(j)/(2^14);
107     else                            %número positivo
108         y(j) = y_dec(j)/(2^14);
109     end
110     j=j+1;
111     y_line = fgetl(fp_bin2);
112 end
113 fclose(fp_bin2);
114
115 figure
116 plot(y)
117 title("y Echo(n)")
118  sound(y, 22050)
119 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figura 204: código para leer y analizar datasets binarios de salida

- Lanzaremos la simulación clicando al icono "*Run*".

- Al finalizar la simulación escuchamos el audio de salida y vemos su representación gráfica de onda ya conocida en el trabajo.

- Repetir para cada fichero de salida del sistema.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá