_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISEÑO DE ACELERADORES HARDWARE PARA PROCESADO DE AUDIO EN ARQUITECTURAS SOC

Carlos Aznar Ruiz

Tutor: Germán Ramos Peinado

Cotutor: Rafael Gadea Girones

Trabajo Fin de Máster presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad Politécnica de Valencia, para la obtención del Título de Máster Universitario en Ingeniería de Telecomunicación

Curso 2015-16

Valencia, 6 de julio de 2016

Agradecimientos

A mis compañeros Javier, Vicente y Enrique, por ser una referencia y un apoyo sobre el que trabajar.

A mis compañeros de trabajo por sus consejos y sugerencias, somos un equipo.

A mi familia por su apoyo siempre en todo lo que hago.

A mí mismo, por haber llegado tan lejos, y por todo lo que aún tiene que venir.

Sólo podemos ser lo que somos, nada más y nada menos (Terry Godkind)

Resumen

Las aplicaciones de procesado de audio en tiempo real cada vez demandan más capacidad de cálculo, bien sea debido al aumento de su complejidad, o al aumento del número de canales de audio a procesar.

Tradicionalmente el procesado de audio ha sido (y sigue siendo) realizado sobre arquitecturas DSP específicas, las cuales pueden abordar una gran cantidad de aplicaciones, pero que ante una gran demanda de cálculo empiezan a tener dificultades. Para superar esta barrera se plantean varias posibilidades. Una opción es aumentar el número de núcleos en los DSP, con un enfoque "software", mientras que por otro lado existe un enfoque "hardware", diseñando el proceso de audio sobre arquitecturas FPGA, paralelizando en hardware el proceso.

Recientemente, los dos grandes fabricantes de FPGAs (*XILINX* y *ALTERA*) han incluido en su portfolio nuevos dispositivos mixtos que incorporan FPGAs junto a 2 procesadores *ARM Cortex-A9* de propósito general, todo ello en un único encapsulado. El objetivo de estos nuevos SoC es claro: particionar los diseños en software (ARM) y aceleradores hardware (FPGA). Con ello se busca también atraer a los diseñadores software hacia los dispositivos programables, ya que los flujos de diseño permiten un fácil transvase de software a hardware (incluso desde código C), resolviendo las herramientas todos los drivers para su uso y comunicación.

Este trabajo plantea el uso de estas nuevas arquitecturas mixtas, en particular sobre la *Zynq*-7020 de *XILINX* montada en la placa comercial *ZerdBoard*, en aplicaciones de procesado de audio.

El objetivo es el de plantear aceleradores hardware sobre la FPGA de diversos procesos de audio (filtrados, mezcla, procesadores de dinámica, efectos), en los que se realizará un estudio teórico de las necesidades de tamaños de palabra para cada proceso (algo muy crítico en audio, especialmente en baja frecuencia), estructuras de conformado de ruido, pipelining, reutilización de los recursos hardware, entrada-salida, etc.

Resum

Les aplicacions de processament d'àudio en temps real demanen cada vegada més capacitat de càlcul, bé siga a causa de l'augmentació de la seua complexitat, o bé a l'augmentació del nombre de canals d'àudio per a processar.

Tradicionalment el processament d'àudio ha sigut (i continua sent) realitzat sobre arquitectures DSP específiques, les quals poden abordar una gran quantitat d'aplicacions, però que davant una gran demanda de càlcul comencen a tenir dificultats. Per a poder superar esta barrera es plantegen diverses possibilitats. Una opció és augmentar el nombre de nuclis en els DSP, amb un enfocament "software", mentre que d'altra banda existeix un enfocament "hardware", dissenyant el procés d'àudio sobre arquitectures FPGA, fent el procés en paral·lel amb hardware.

Recentment, els dos més grans fabricants de FPGAs (*XILINX* i *ALTERA*) han inclòs en el seu "portfolio" nous dispositius mixts, els quals incorporen FPGAs al costat de dos processadors *ARM Cortex-A9* de propòsit general, tot això dins d' un únic encapsulat. L'objectiu d'estos dos nous SoC és clar: dividir els dissenys en software cap als dispositius programables, ja que els fluxos de disseny permeten un fàcil transvasament de software a hardware (fins i tot des de codi C), resolent les eines tots els *drivers* per al seu ús i comunicació.

Este treball planteja l'ús d'estes noves arquitectures mixtes, particularment sobre la *Zynq*-7020 de *XILINX* muntada en la placa comercial *ZerdBoard*, en aplicacions de processament d'àudio.

L'objectiu és plantejar acceleradors hardware sobre FPGA de diversos processos d'àudio (filtrats, mescla, processadors de dinàmica, efectes), en els quals es realitzarà un estudi teòric de les necessitats de grandàries de la paraula per cada procés (cosa molt crítica en àudio, especialment en baixa freqüència), estructures de conformat de soroll, *pipelining*, reutilització dels recursos hardware, entrada-eixida, etc.

Abstract

Real time audio processing applications are each time demanding more computing power, either due to the increased complexity or due to the increasing number of audio channels to be processed.

Traditionally the audio processing has been (and it still is) performed on specific DSP architectures, which can address many applications, however they start having difficulties facing a high calculation demand. To overcome this barrier several possibilities arise. One option is to increase the number of cores in the DSP, with a software approach, while another option is a hardware approach, designing the audio processing over FPGA architectures, hardware parallelizing the process.

Recently, the two major manufacturers of FPGAs (*XILINX* and *ALTERA*) have included in their portfolio new devices that incorporate FPGAs mixed with 2 processors *ARM Cortex-A9* of general purpose, all in a single package. The aim of these new SoC is clear: partition software designs (ARM) and hardware accelerators (FPGA). It also seeks to attract software designers to programmable devices, because design flows allow easy decanting of software to hardware, solving the tools all drivers for its use and communications.

This paper proposes the use of these new mixed architectures, particularly on the Zynq-7020 *XILINX* mounted ZerdBoard commercial plattform in audio processing applications.

The goal is to raise accelerator hardware on the FPGA of various audio processing (filtering, mixing, dynamics processors, effects), in which a theoretical study of the needs of word sizes for each process will be carried out (something very critical audio, especially in low frequency) noise shaping structures, pipelining, re-use of hardware resources, input-output, etc.

Índice de Capítulos

Índice de Ilustraciones

Índice de Tablas

Capítulo 1. Introducción

El procesado digital de señales de audio es un campo muy vivo actualmente y que cuenta ya con varias décadas de trabajo e investigación a sus espaldas.

Hasta hace pocos años todos los esfuerzos se concentraban en la implementación de distintos algoritmos en dispositivos tipo DSP (*Digital Signal Processor*), debido a que eran las mejores opciones en aquella época. Por suerte, la tecnología avanza y cambia el paradigma de desarrollo para estos sistemas. Hace poco tiempo, empezaron a aparecer en el mercado nuevos SoC (*System on Chip*) que implementan conjuntamente microcontroladores y/o microprocesadores y FPGA (*Field Programmable Gate Array*), añadiendo herramientas para el diseño y gestión de ambas partes en su conjunto.

El motivo por el cual es tan importante mejorar los equipos de desarrollo e implementación para el audio digital, es debido al prominente aumento del número de canales en la mayoría de aplicaciones, como procesadores y mesas de mezcla, y por otro lado, la necesidad de implementar nuevos algoritmos con unos requisitos tan extremos que las fórmulas tradicionales no podían satisfacerlos.

Haciendo hincapié en diferentes ejemplos de aplicación que puedan servirse de este tipo de desarrollos, cabe destacar:

- Sistemas de arrays de altavoces en columna.
- Sistemas de acústica variable.
- Transporte digital de audio.

Siendo la dificultad común de los 2 primeros casos el gran número de dispositivos que intervienen (altavoces, micrófonos, etc.) y para el tercer caso, la complejidad de los algoritmos implementados.

Hay que resaltar la diferenciación en temas de implantación de los diseños en cada una de las partes de estudio: mientras que los DSP trabajan con algoritmos en software, de normal en lenguajes de medio-alto nivel, las FPGA implementan diseños en hardware, lo cual conlleva un replanteamiento de algunas soluciones ya existentes, pero a su vez, abre un amplio abanico de nuevas oportunidades de mejora. El trabajar con un SoC que incluye ambas partes, permite gestionar el diseño del sistema como un conjunto de bloques, y dejar que cada parte se encargue del bloque que mejor pueda gestionar.

Centrándonos en un momento en las particularidades de cada uno, y analizándolas superficialmente, podemos hablar de que los DSP trabajan sobre una arquitectura hardware fija, en la cual destaca la unidad MAC (*Multiplier – Accumulator*), bloque muy específico de trabajo que consigue unas velocidades muy altas. Sin embargo, esto conlleva a que el proceso sea secuencial, o en el mejor de los casos, que el integrado lleve varias de estas unidades para poder paralelizar ciertos procesos, a costa de un incremento del coste económico. Un ejemplo simplificado puede apreciarse en la Ilustración 1.

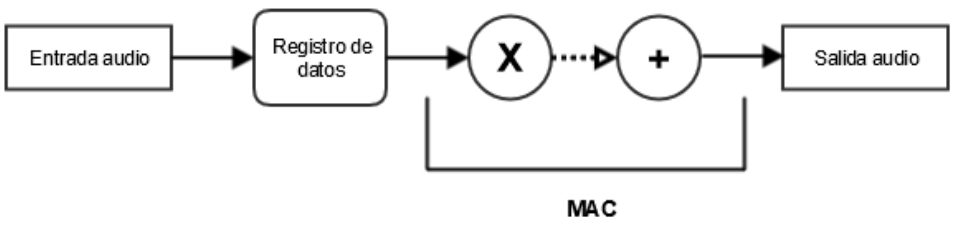

Ilustración 1 - Multiplier Accumulator Example

Por el contrario, en un diseño "estándar" de audio digital en una FPGA nos podemos encontrar una arquitectura completamente diferente, ya que ofrece la posiblidad de añadir tantos bloques hardware como sean necesarios para paralelizar todo lo posible o necesario el sistema, sin incurrir en un aumento del coste del producto. Cada bloque individualmente puede realizar las operaciones secuencialmente, tantas veces como se necesite. Como parte negativa, destacar que un mismo diseño hardware es dificilmente reutilizable para distintos proyectos, por lo que se suele diseñar con un pensamiento de diseño modular, para luego hacer la labor pesada de diseño en una capa superior que implemente todos los módulos en su conjunto. Se presenta un ejemplo en la Ilustración 2.

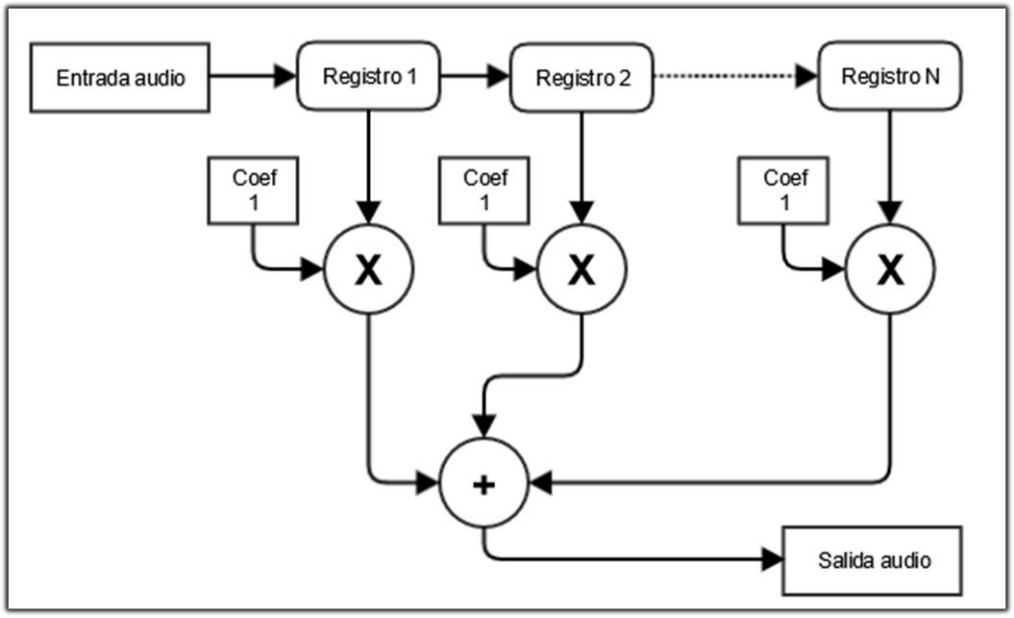

Ilustración 2 - Ejemplo Implementación FPGA

A lo largo de este documento se hablará de las diferentes aplicaciones que puede tener esta tecnología en el audio digital y el proceso de desarrollo que se plantea como solución para su aceleración. Se hará especial hincapié en las herramientas utilizadas y en todo el proceso de desarrollo que ha sido necesario para lograr que la placa de trabajo sea funcional con los requerimientos del proyecto.

Se detallará, en su sección correspondiente, el plan de trabajo seguido, la gestión del proyecto que se hizo en su inicio y su evolución, así como un análisis detallado de los resultados y unas consideraciones para posibles trabajos futuros en este tema.

A lo largo del documento, y por simplicidad, se denominará **PS** (*Processing System*) a la parte "software" del chip, y **PL** (*Programmable Logic*) a la parte referida a la FPGA.

Para finalizar esta sección, es necesario e importante señalar que los diseños que se llevan a cabo en este proyecto a nivel software en C/C++ se han realizado con una visión puesta en su implementación en hardware. En caso de que se intenten utilizar las herramientas para transladar código no pensado para ejecutarse en dispositivo de este tipo, el resultado puede ser, como poco, catastrófico.

Capítulo 2. Objetivos

A día de hoy se puede apreciar claramente como las principales casas de fabricación y comercialización de FPGAs llevan en su catálogo un amplio abanico de opciones, y entre todas esas resalta la inclusión de nuevos SoC que incluyen partes de microcontroladores y partes de FPGA. Con estos sistemas, el trabajo de co-diseño hardware-software se ve lanzado al primer plano de las líneas de trabajo a abordar para mantenerse a la vanguardia en cuanto a opciones de mercado. (1) (2)

Partiendo de este punto, se ha querido tomar como referencia el trabajo sobre una placa comercial de evaluación, partiendo desde cero y realizando todo el trabajo necesario para arrancar el sistema, ponerlo a punto con las condiciones necesarias para que cumpla la demanda requerida, y finalmente poder diseñar un módulo de filtrado de audio que será utilizado como referencia de experimentación para la evaluación de la aceleración de un sistema software mediante hardware. La placa seleccionada ha sido la ZedBoard (3) que incluye un SoC ZynQ de la serie 7000 (4), concretamente la 7020.

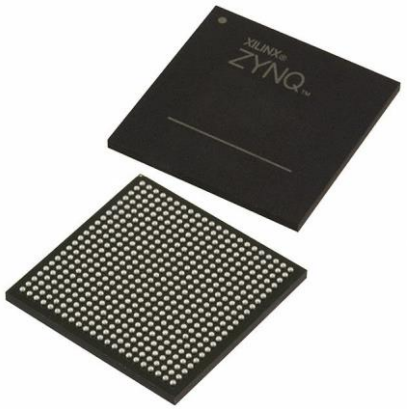

Ilustración 3 – Muestra de ZynQ comercial

En el caso del diseño del módulo de filtrado, el principal problema que presenta es la falta de herramientas, investigaciones y soluciones específicas (comparado a la literatura existente para fórmulas "clásicas" de implementación) de este tipo de algoritmos para sistemas de co-diseño como el propuesto para investigación.

Con todo esto expuesto, el principal objetivo del presente trabajo será la investigación y desarrollo de nuevas posibilidades y oportunidades a la hora de trabajar con esta nueva vertiente de SoCs, y en concreto con productos de la casa Xilinx, centrándonos en el aspecto de co-diseño hardware-software.

Es necesario resaltar que aunque se hable de acelerar un módulo de filtrado, no se tomará la velocidad como una máxima, sino que se intentarán utilizar los recursos óptimos, analizando cada caso de desarrollo en profundidad para determinar en punto de equilibrio entre velocidad de procesamiento y recursos necesarios para implementarlo. Este será el *trade-off* que gobernará el diseño del módulo a estudiar.

Además del desarrollo se harán consideraciones con respecto a las diferentes características y resultados de las distintas interpretaciones, reuniendo los resultados en tablas-resumen y evaluando las mejoras obtenidas respecto a implementaciones clásicas.

Capítulo 3. Conceptos básicos sobre audio digital

En este capítulo se hará una exposición de los conceptos necesarios e imprescindibles sobre temas de audio digital que vayan a ser utilizados a lo largo del proyecto, con el fin de que su entendimiento sea más sencillo.

3.1 Audio Digital

Se denomina audio digital a las ondas acústicas que han sido captadas, convertidas en señales eléctricas analógicas, haciendo uso de micrófonos, y posteriormente en señales digitales a través de un ADC (Analog-toDigital Converter). A la hora de querer volver a transformar esa señal digital en una onda acústica, es necesario el trayecto inverso, usando un DAC (Digital-to-Analog Converter) se reconstruye la señal eléctrica, con la mayor fidelidad posible, y luego esta pasará a una onda acústica por medio de un altavoz, generalmente.

La ventaja de hacer esta última conversión es que facilita y mejora las tareas de compresión, almacenamiento, procesamiento y reproducción, sobretodo este último punto tiene grandes diferencias con el tratamiento analógico, pues en este la creación sucesiva de copias de una grabación va deteriorando el resultado, hasta que puede llegar a ser irreconocible si se compara con el original.

En la década de 1970 fue cuando los estudios y el desarrollo de tecnologías en este campo cobró fuerza, pero no fue hasta los 90s cuando realmente los sistemas digitales sustituyeron a los analógicos, a pesar de que hoy en día hay corrientes y tendencias (los denominados "hipsters") que quieren volver al mundo analógico.

3.1.1 Frecuencia de muestreo

Según dice el conocídisimo teorema de Nyquist-Shannon "[…] *dada una frecuencia de muestro f_s, una resconstrucción perfecta de la señal es posible para un ancho de banda $B < f_s/2$*" (5) podemos conocer qué frecuencia óptima se debería de usar para audio, más aún sabiendo los límites de voz y oído humanos: siendo de entre 300 Hz y 3400 kHz el primero, y de entre 20 Hz y 20 kHz el segundo. (6)

De los datos anteriores se puede extraer la conclusión de que la frecuencia de muestreo deberá de ser al menos, mayor de 40 kHz. Ahora bien, esto sería en el caso ideal de que los efectos que queramos introducir con diferentes filtros fuesen perfectos, pero pensemos que nunca podrá darse tal caso, es por este caso y por compatibilidad con protocolos de vídeo, que hoy en día los 48 kHz es la frecuencia de muestro más común (7), tanto a nivel de usuario como profesionalmente.

3.2 FPGA

Una FPGA (*Field Programmable Gate Array*) es un dispositivo semiconductor basado en una matriz de células lógicas (CLBs) configurables, conectadas a través de interconexiones configurables. Esto se traduce a que un mismo hardware puede adaptarse a diferentes diseños en apenas unos segundos, permitiendo pruebas variadas reduciendo el coste en modelos de prueba o bien implementar complejos sistemas utilizando un mismo dispositivo tanto para pruebas como para el producto final.

Además de las matrices de células lógicas, cada vez las FPGA incorporan más tipos diferentes dispositivos, como pueden ser bloques de memorias dedicadas o incluso pequeñas células DSP.

Esta caracterísitica distingue a las FPGA de dispositivos ASIC (*Application Specific Integrated Circuits*) que se solían usar en el pasado para resolver problemas de procesado de audio. Siendo la ventaja de estos segundos, antiguamente, su mayor potencia a la hora de atajar un problema concreto. Por desgracia para los ASIC, el avance de la tecnología ha logrado que se puedan utilizar FPGA a más de 500 MHz, lo cual las coloca en posición de competir en cuanto a rendimiento, además de poder ofrecer otras muchas funcionalidades. (8)

Su programación viene con lenguajes de descripción de hardware (*HDL*), en los cuales dominan actualmente el sector *Verilog* y *VHDL*. En cuanto a empresas fabricantes de estos dispositivos, el mercado se encuentra dividido entre dos grandes compañías: *Altera Corporation* y *Xilinx Inc*.

3.3 Representación en coma fija

La coma fija delimita dentro de un rango de bits cierta cantidad para usarse como parte decimal y otra como parte entera. Si se tienen 24 bits y se dice que la coma fija está en el 23, nuestro sistema será capaz de representar un gran número de decimales, es decir, mucha resolución, pero muy poca diferencia entre el máximo y el mínimo número que se pueda representar, lo que es equivalente a poco rango dinámico. Al desplazar la coma fija de forma que la parte entera ocupe un mayor número de bits, disminuye la resolución pero ganamos rango dinámico.

3.4 Representación en coma flotante

En un sistema de coma flotante, la posición de la misma varía de forma automática, mantenimiendo constante un compromiso entre resolución y rango dinámico, en función del número que haya de ser representando. El movimiento de la coma se logra mediante el uso de exponentes, lo que ayuda a que el rango dinámico disponible sea bastante elevado. El problema de usar este tipo de representación en sistemas de audio es que al implementarse en hardware, todas las operaciones tienen que pasarse a ser realizadas en coma flotante, lo que conlleva un aumento del uso de recursos considerable.

Además de lo expuesto anteriormente, los sistemas de coma flotante tienen frecuencias de operación más bajas que las de los sistemas de coma fija, como normal general. Sin ermbargo, es cierto que en aplicaciones concretas, al no tener que hacer uso de control de rango dinámico, ni de otros métodos de control, pueden resultar beneficiciosos en términos de frecuencias de operación.

3.5 Filtros FIR

Se define como filtro FIR los que presentan una respuesta finita en el tiempo a un impulso. Lo que tarda la respuesta del filtro en volver a 0 después de la introducción de un pulso delta es de N+1 muestras. Su expresión más simplificada queda resumida en la Ilustración 4.

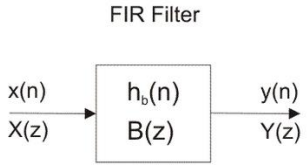

Ilustración 4 - Filtro FIR

Su principal desventaja es que requieren más potencia de procesamiento que un filtro IIR con una características similares en cuanto a rizado o selectividad, especialmente con frecuencias de operación bajas. Esto se puede solventar con sistemas de procesamiento especializado para acelerar este tipo de filtros (9), como podrá verse a lo largo del presente proyecto.

3.6 Filtros IIR

Los filtros IIR son filtros digitales de respuesta al impulso infinita. A diferencia de los FIR, tienen una rama de realimentación (una parte recursiva del filtro), por ello se les conoce como filtros digitales recursivos.

Un ejemplo sencillo puede verse en la Ilustración 5.

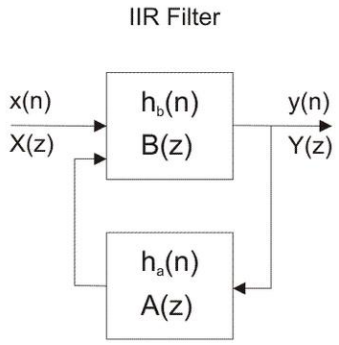

Ilustración 5 – Filtro IIR

Esta particularidad les aporta una respuesta en frecuencia mejor que la de los filtros FIR de un mismo orden, pero a cambio, su fase no es lineal.

Además, al tener una realimentación, se pueden generar inestabilidades si la ganancia es mayor que 1, o bien, si el filtro posee polos fuera de z = 1.

La función de transferencia de un filtro IIR viene definida por la divisón de 2 polinomios de variable compleja z^{-1}. El numerador define la localización de los ceros, mientras que el denominador define la localización de los polos.

3.7 Ruido en filtros

Se entiende por ruido aquellas deformaciones en la información de una señal respecto a la señal deseada que se desearía tener. El origen puede provenir bien por el proceso de adquisición (errores en los sensores), como en la transformación (ADC, DAC o cuantificación) o en la transmisión, e incluso en el almacenamiento, donde se suele dar al trabajar con registros o memorias de muchos bits. Además, también hay que tener en cuenta el ruido por redondeo en las operaciones. o más rara vez, en el almacenamiento. (10)

A lo largo de este proyecto se obviarán las posibles ruidos provenientes de los sensores y del almacenamiento. En el caso del ruido introducido por el ADC/DAC se estimará en el caso de que sea posible su cálculo con la información facilitada por el fabricante, pero en ningún caso será determinante para el resultado del proyecto. El ruido de almacenamiento de los datos es a fin de cuentas el ruido de cuantificación.

3.7.1 Ruido de cuantificación

Mientras que las señales analógicas tienen teóricamente una resolución infinita en nivel de señal y en tiempo, las digitales tienen que ser discretizadas en diferentes niveles cuantificables en diferentes "escalones" de amplitud. El número de "escalones" a utilizar por un sistema digital viene dado por los bits que se asignen a la señal, y se le conoce por niveles de cuantificación.

Hay que tener en cuenta que además del número de niveles de cuantificación, habrá una implicación directa en el error final de cuantificación según el nivel de señal que entre respecto al máximo al que el sistema esté ajustado, siendo el dato más relevante a extraer de esta información que el error máximo de cuantificación introducido será como máximo la mitad de lo representado por el bit menos significativo (LSB) del sistema. (11)

Lo anteriormente explicado se traduce en que para nuestro sistema, cuanto mayor sea el tamaño de palabra a usar para representar la señal digital, mayor será la diferencia entre el máximo de la señal respecto al error que se puede llegar a introducir, mejorando así la relación señal a ruido. Este punto también presenta desventajas para el sistema: si se diseña para que el ruido de cuantificación sea el menor posible, el máximo de la señal se alejará, normalmente, de los valores que se suelen representar, por lo que estaremos usando tan solo un subconjunto del total de bits disponibles.

No se llevará a cabo un cálculo del ruido de cuantificación, pero será tenido en cuenta a la hora de elegir el tamaño de palabra a emplear por el sistema, y los bits extras que se añadirán a las diferentes etapas del sistema.

3.8 Conceptos de Aceleración de Hardware

3.8.1 Latencia

Se conoce por latencia al número de ciclos de reloj que necesita una instrucción o juego de instrucciones en generar un resultado válido para una aplicación. En términos de sistema se utiliza para medir el tiempo que tarda un sistema desde que le llega un dato hasta que ese dato ha salido del sistema y está disponible para que otro lo pueda utilizar.

Es una métrica que se utiliza para medir rendimiento en cuanto a tiempos tanto en procesadores como en FPGAs. En ambos casos, los problemas derivados de la latencia se suelen resolver mediante técnicas de *pipelining*.

3.8.2 Pipelining (Segmentación)

El *pipelining* o segmentación es un método que tiene como objetivo aumentar el rendimiento de algunos sistemas electrónicos digitales. Se basa en la sincronización o en el registro de las operaciones para que la ruta crítica (el tramo con una carga o retardo computacional mayor entre dos registros de reloj) se reduzca.

La ruta crítica marca la frecuencia máxima de trabajo que puede alcanzar como conjunto un sistema. Con un ruta crítica mayor, la frecuencia de trabajo máxima será menor, y a la inversa.

Este método no puede ser todo ventajas, ya que al realizar la segmentación de los cálculos, se deben utilizar más recursos para registrar los datos intermedios, además de añadir un retardo a la línea.

De forma genérica para un sistema con pipeling que haya conseguido la frecuencia máxima de operación, tendrá un retardo mayor para llenar la línea principal (*pipe*),pero a partir de ahí, los resultados de cada comando saldrán en un número menor de ciclos, y en el caso óptimo, sin latencia añadida.

Normalmente se usa con un parámetro II (Initiation Interval) que marca el número de ciclos de reloj que existen entre una muestra de datos y la siguiente, con el fin de poder adaptar el sistema a cumplir con esa restricción.

3.8.3 Throughput

El *throughput* es otra métrica que se usa para determinar el rendimiento general de un sistema implementado. Representa el número de ciclos de reloj que tarda el procesamiento lógico en poder aceptar una nueva muestra de datos desde que aceptó la última. Va asociado inevitablemente a la latencia y al *pipelining.*

En un sistema que solo se ejecute en una dimensión (es decir, que no esté nada paralelizado) el *throughput* y la latencia tendrán el mismo valor.

3.8.4 Unroll

Al igual que el pipelining, es una técnica que se utiliza para la paralelización de instrucciones, pero en este caso compete solamente a los bucles for, los cuales están determinados a ejecutarse un número finito de veces (con bucles de dimensiones variables no se podrá aplicar).

Se basa en crear copias del cuerpo del bucle original y ajustar el número de iteraciones de cada una, permitiendo así que en cada ejecución del bucle existan un mayor número de instrucciones las cuales puedan ser atacadas para su paralelización en recursos, lo cual, por norma general aumenta el *throughput* de nuestro sistema.

Se puede ejecutar de manera parcial en el bucle o “desenrollándolo” por completo.

Capítulo 4. Metodología

4.1 Gestión del proyecto

En esta sección se abordará el tema de la gestión de proyectos, desde la concepción del proyecto, hasta el cierre del proyecto, pasando por todas las etapas medias: diseño de solución, propuesta, análisis de riesgos, planificación de tareas y del plan de validación, así como el desarrollo y seguimiento del mismo.

4.1.1 Fase de Concepto

Durante una reunión entre el tutor del TFM y el alumno a cargo de su realización se debatió sobre posibles proyectos a realizar en temas de co-diseño hardware-software.

Salió a relucir la posiblidad de utilizar la ZedBoard que tenía el tutor disponible para investigar qué resultados se podrían esperar en cuanto a nuevas herramientas nunca antes trabajadas por el departamento y sus posibles líneas futuras de trabajo.

Se plantearon ciertos requisitos a cumplir y se formalizó el trabajo, asignándoselo al alumno.

4.1.2 Fase de Planificación

En esta fase, lo primero que se establecerá serán los requisitos del proyecto y de ellos se formará un diseño técnico de la solución propuesta.

Se toman como requisitos del sistema los siguientes:

- Estudiar la creación y gestión de un proyecto mediante Vivado 2015 y sus posiblidades de desarrollo a nivel de co-diseño.
- Manejar la herramienta Vivado HLS 2015 para generar un bloque IP propio, acelerarlo e implantarlo en un sistema.
- Que la implementación del bloque IP sea sencilla y que el trabajo en la parte software sea lo más transparente posible.
- Realizar un análisis de resultados con el fin de comprender mejor el funcionamiento en conjunto de todas las herramientas.

Con todos estos puntos en mente, se decide acudir a un SoC de nueva generación, como es el ZynQ-7070 de Xilinx Inc., el cual se puede encontrar en la placa de evaluación ZedBoard, que será la seleccionada para este proyecto.

Se escoge esta placa de evaluación debido a que tiene gran respaldo por parte de Xilinx Inc. y debido a que ya se disponía de ella por parte del tutor del Trabajo Fin de Máster.

Como módulo a desarrollar sobre el cual realizar el trabajo de aceleración, se escoge que sea un filtro FIR, debido a su amplio abanico de posiblidades de diseño y a la cantidad ingente de ejemplos y soluciones propuestas de implementación tanto software como hardware.

4.1.3 Hitos del proyecto

En esta parte se analizarán los objetivos a conseguir durante el desarrollo del proyecto, siendo estos puntos clave tanto para conseguir un resultado satisfactorio, como para poder realizar un seguimiento en términos generales del avance del proyecto.

- Realización de ejemplo dummy sobre la placa de evaluación ZedBoard, que aborde un co-diseño hardware-software.
- Realización de un test de audio para la configuración deseada en la placa de evaluación ZedBoard.
- Diseño de un filtro FIR básico y validación mediante TestBench.
- Aceleración del diseño del filtro FIR hasta obtener unos resultados satisfactorios en términos del compriso velocidad/recursos.

- Documentación del proyecto.

4.1.4 Análisis de riesgos

Antes de comenzar la definición del proyecto y de sus tareas, es importante pararse y plantearse qué situaciones se podrían dar que entorpecieran el desarrollo o incluso que impidiesen su ejecución.

- Cuello de botella en el chip de audio, la velocidad de salida de audio podría ser insuficiente.
- Recursos escasos en la placa de evaluación.
- Inestabilidad del módulo generado mediante Vivado HLS.

Ante estos riesgos, se plantean soluciones que se podrían adoptar en el caso de que sucediesen, en el mismo orden que han sido listados los riesgos.

- Utilizar uno de los microcontroladores como back-up para la salida del audio.
- Los recursos que más posiblidades tienen de "agotarse" son las células DSP, las cuales pueden ser sustituidas por FF y LUTs si se diese el caso.
- Dado que no se tiene conocimiento previo de la herramienta, no se tiene información sobre qué tipo de resultados esperar de la misma, pero debido a que está en el mercado y ha sido desarrollada y publicada por una empresa con gran reputación en el sector, se puede confiar en que los resultados sean, como poco, admisibles.

4.1.5 Plan de validación

Se desarrollará un TestBench para realizar una comprobación del módulo del filtro FIR básico, asegurando que no hay problemas en cuanto a las operaciones con coma fija, ni saturaciones, y que existe una precisión suficiente para la aplicación deseada.

Este TestBench realizará una comparación entre los datos obtenidos al introducir una señal generada de manera propia en el módulo y los obtenidos en el caso ideal mediante Matlab 2014, al realizar la misma operación.

4.1.6 Planificación

Con la información presente en los puntos anteriores se propone la siguiente planificación a seguir a lo largo del proyecto, teniendo plena consciencia de que se va a tener que cambiar y adaptar a los problemas e imprevistos que surjan a lo largo del mismo:

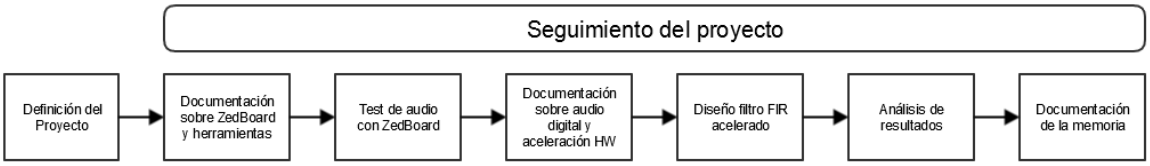

Ilustración 6 – Planificación del proyecto

Esto representa una planificación *grosso modo*, la división en tareas completa se encuentra en la sección 4.1. Ahí se puede encontrar tanto la planificación inicial como la definitiva, con el registro de horas y una gráfica comparativa del error cometido en la estimación de horas de dedicación a cada tarea.

Como datos relevantes de la planificación, se destacan los siguientes:

- **Fecha de inicio**: 2 de noviembre de 2015.
- **Trabajo planificado**: 83 días a jornada completa, haciendo un total de 664 horas.
- **Duración planificada**: 141 días. Dado que el alumno se encuentra en prácticas al comienzo del proyecto y la intención es que se mantenga a lo largo del mismo, se asigna una valor de producción del 45%, provocando este diferencia con el trabajo.
- **Fecha de fin**: 31 de marzo de 2016.

Ante la planificación anterior, se puede comprobar que se dispone aproximadamente de un mes de "colchón" frente a imprevistos y dificultades para llegar a la fecha de entrega de memoria del proyecto en óptimas condiciones.

Con toda la información recogida hasta este punto se da por definido el proyecto y se procede a comenzar su desarrollo y consecuente seguimiento.

4.1.7 Fase de Desarrollo y Seguimiento

A lo largo de la duración del proyecto se ha realizado un seguimiento de las tareas, comprobando su progreso y contranstándolo con la planificación. Las decisiones de replanificación no han sido necesarias, puesto que al solo contar con un recurso para trabajar, el trabajo que se podía hacer en paralelo era mínimo.

Se ha realizado una comparativa entre las fechas previstas para los hitos y las que se han logrado conseguir.

	Dummy Zedboard	Test audio Zedboard	Filtro FIR con Testbench	Aceleración filtro FIR	Documentación
Trabajo planificado	22 días	13 días	12 días	18 días	11 días
Trabajo real	24 días	16 días	15 días	20 días	12 días
Error	10%	23%	25%	11%	9%

Tabla 1 – Seguimiento hitos del proyecto

En las tablas Tabla 2 y Tabla 3, se puede encontrar la distribución planificada de tareas y el resultado real del proyecto, así como una comparativa entre el esfuerzo planificado y el real.

Haciendo una media del error en porcentaje de cada una de las tareas, obtenemos un error de planificación medio del proyecto del **15,6%**. Con la perspectiva de que todo el trabajo ha sido realizado por un alumno con experiencia mínima, así como la planificación, se puede decir, sin riesgo a equivocarse, que ha sido una planificación muy acertada.

A los datos expuestos en la Tabla 1 habría que sumar el trabajo empleado en la definición de proyecto y el empleado en el seguimiento del mismo.

4.1.8 Cierre del Proyecto

A pesar del mes de margen que existía entre la fecha a finalizar el proyecto planificada y la fecha de entrega del proyecto, se ha exprimido el tiempo hasta su último segundo, debido a una acumulación continuada de dificultades a la hora de encontrar documentación apropiada para las herramientas con las que se trabaja.

A pesar de eso, y con los resultados finales obtenidos, y valorando que son suficientes para dar por cumplidos los requisitos fundamentales del proyecto, se da cierre al proyecto mediante el presente documento.

Evidentemente, ha quedado mucho trabajo por realizar, tanto en temas de profundización en el co-diseño como en la experimentación con las posiblidades que ofrecen las diferentes herramientas que componen la Suite Vivado con la que se ha trabajado.

Es de considerar que el trabajo realizado a lo largo de este proyecto ha logrado desarrollar un procedimiento de trabajo en nuevas herramientas que no habian sido usadas en el departamento de electrónica. Además de que a raíz de este proyecto, pueden florecer con facilidad otros módulos implementados siguiendo la metodología establecida y a través de la investigación realizada.

4.2 Distribución en tareas

4.2.1 Distribución planificada

ID Tarea	Nombre de tarea	Trabajo planificado	Comienzo	Fin	Predecesoras	Recursos	Desripción
1	**Definición del Proyecto**	1 día	lun 02/11/15	mar 03/11/15		**CAR**	**Se establecen los requisitos y se genera la planificación.**
2	**Seguimiento del Proyecto**	6,2 días	mar 03/11/15	lun 09/05/16	**28FF;1**	**CAR[5%]**	**Tarea de seguimiento del proyecto, checkeo del progreso de los hitos y replanificación.**
3	**Documentación sobre Xilinx, ZynQ y ZedBoard**	**22 días**	**mar 03/11/15**	**mié 23/12/15**	**1**		**Se obtendrán los conocimientos básicos sobre estos temas.**
4	Documentación sobre la Suite Vivado	4 días	mar 03/11/15	lun 16/11/15	1	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante.
5	Tutoriales para generar y gestionar proyectos	4 días	lun 16/11/15	vie 27/11/15	4	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante.
6	Documentación sobre ZynQ	3 días	mar 03/11/15	jue 12/11/15	1	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante.
7	Documentación sobre ZedBoard	3 días	jue 12/11/15	vie 20/11/15	6	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante.
8	Tutoriales ZedBoard básicos	4 días	vie 27/11/15	jue 10/12/15	7;5	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante, además de otros recomendados por Internet.
9	Tutoriales ZedBoard avanzados	4 días	jue 10/12/15	mié 23/12/15	8	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante, además de otros recomendados por Internet.

10	**Puesta a punto ZedBoard con Test de Audio**	**13 días**	**mié 23/12/15**	**mié 20/01/16**	**3**		**Comprobar limitaciones del chip de audio y tener un códec funcional.**
11	Creación del proyecto con especificaciones	1 día	mié 23/12/15	mar 05/01/16	9	CAR [45%]	Tener un entorno de trabajo en Vivado estable y con lo necesario.
12	Documentación sobre chip de audio	3 días	mié 23/12/15	jue 31/12/15	9	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante.
13	Documentación sobre AXI	2 días	mié 23/12/15	mar 29/12/15	9	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante.
14	Tutoriales AXI Master 4 y Lite	2 días	mar 29/12/15	mar 05/01/16	13	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante.
15	Diseño solución test de audio	2 días	mar 05/01/16	lun 11/01/16	14;12;11	CAR [45%]	Diseño robusto para comprobar que el proyecto va a ser viable.
16	Pruebas del diseño	1 día	lun 11/01/16	mié 13/01/16	15	CAR [45%]	Probar los distintos componentes que lo integran.
17	Rectificaciones en el diseño	2 días	mié 13/01/16	mié 20/01/16	16	CAR [45%]	Rectificaciones necesarias para ajustar los resultados del diseño.
18	**Documentación sobre audio digital y aceleración HW con Vivado HLS**	**6 días**	**mié 20/01/16**	**jue 28/01/16**	**10**		**Se obtendrán los conocimientos básicos sobre estos temas.**
19	Revisión de bibliografía sobre audio digital	3 días	mié 20/01/16	jue 28/01/16	17	CAR [45%]	Se realizará búsqueda profunda por la bibliografía accesible mediante Internet.
20	Tutoriales Vivado HLS avanzados	3 días	mié 20/01/16	jue 28/01/16	17	CAR [45%]	Se utilizará la documentación proporcionada por el fabricante.
21	**Diseño filtro FIR acelerado**	**19 días**	**jue 28/01/16**	**mar 12/04/16**	**18**		**Se hará el diseño basándose en los requisitos y los objetivos requeridos del proyecto.**
22	Diseño FIR en Matlab	1 día	jue 28/01/16	mar 02/02/16	20	CAR [45%]	Algo básico. No es importante los resultados.
23	Diseño FIR en SW básico	3 días	mar 02/02/16	mié 10/02/16	22	CAR [45%]	Estructura básica y clara.

24	Testbench FIR	2 días	mié 10/02/16	mié 17/02/16	23	CAR [45%]	Diseño y ejecución de Testbench
25	Pruebas de aceleración	8 días	mié 17/02/16	vie 11/03/16	24	CAR [45%]	Diferentes implementaciones tanto HW como SW.
26	Implementación en un diseño con Vivado 2015	2 días	vie 11/03/16	mar 22/03/16	25	CAR [45%]	Generación de IP e inclusión en un diseño para la ZedBoard.
27	Desarrollo del diseño para hacerlo funcional en ZedBoard	3 días	mar 22/03/16	mar 12/04/16	26	CAR[45%]	Gestión de I2C y de los procesos de audio.
28	**Análisis de resultados**	5 días	mar 12/04/16	mié 27/04/16	21	**CAR[45%]**	**Establecer por qué aumenta la velocidad o los recursos e intentar administrar los datos de forma gráfica.**
29	**Documentación de la memoria del proyecto**	11 días	mié 27/04/16	mar 31/05/16	28	**CAR[45%]**	**Generar el presente documento para que sirva de referencia tanto del trabajo realizado por el alumno como para posibles futuros proyectos.**

Tabla 2 – Distribución en tareas planificada

4.2.2 Distribución real

ID Tarea	Nombre de tarea	Trabajo real	Comienzo real	Fin real	Error (días)	Error (%)
1	**Definición del Proyecto**	**1 día**	**lun 02/11/15**	**mar 03/11/15**	**0**	**0**
2	**Seguimiento del Proyecto**	**6,2 días**	**lun 23/11/15**	**vie 27/05/16**	**0**	**0**
3	**Documentación sobre Xilinx, ZynQ y ZedBoard**	**24 días**	**mar 03/11/15**	**mar 29/12/15**	**2**	**10%**
4	Documentación sobre la Suite Vivado	4 días	mar 03/11/15	lun 16/11/15	0	0
5	Tutoriales para generar y gestionar proyectos	4 días	lun 16/11/15	vie 27/11/15	0	0
6	Documentación sobre ZynQ	3 días	mar 03/11/15	jue 12/11/15	0	0
7	Documentación sobre ZedBoard	3 días	jue 12/11/15	vie 20/11/15	0	0

8	Tutoriales ZedBoard básicos	5 días	vie 27/11/15	lun 14/12/15	1	25%
9	Tutoriales ZedBoard avanzados	5 días	lun 14/12/15	mar 29/12/15	1	25%
10	**Puesta a punto ZedBoard con Test de Audio**	**16 días**	**mar 29/12/15**	**jue 04/02/16**	**3**	**23%**
11	Creación del proyecto con especificaciones	1 día	mar 29/12/15	lun 11/01/16	0	0
12	Documentación sobre chip de audio	3 días	mar 29/12/15	jue 07/01/16	0	0
13	Documentación sobre AXI	2 días	mar 29/12/15	mar 05/01/16	0	0
14	Tutoriales AXI Master 4 y Lite	3 días	mar 05/01/16	mié 13/01/16	1	50%
15	Diseño solución test de audio	2 días	mié 13/01/16	mié 20/01/16	0	0
16	Pruebas del diseño	2 días	mié 20/01/16	mar 26/01/16	1	100%
17	Rectificaciones en el diseño	3 días	mar 26/01/16	jue 04/02/16	1	50%
18	**Documentación sobre audio digital y aceleración HW con Vivado HLS**	**8 días**	**jue 04/02/16**	**vie 19/02/16**	**2**	**33%**
19	Revisión de bibliografía sobre audio digital	3 días	jue 04/02/16	vie 12/02/16	0	0
20	Tutoriales Vivado HLS avanzados	5 días	jue 04/02/16	vie 19/02/16	2	67%
21	**Diseño filtro FIR acelerado**	**22 días**	**vie 19/02/16**	**jue 12/05/16**	**3**	**16%**
22	Diseño FIR en Matlab	1 día	vie 19/02/16	mar 23/02/16	0	0
23	Diseño FIR en SW básico	4 días	mar 23/02/16	lun 07/03/16	1	33%
24	Testbench FIR	2 días	lun 07/03/16	vie 11/03/16	0	0
25	Pruebas de aceleración	10 días	vie 11/03/16	mié 27/04/16	2	25%
26	Implementación en un diseño con Vivado 2015	2 días	mié 27/04/16	mar 03/05/16	0	0
27	Desarrollo del diseño para hacerlo funcional en ZedBoard	3 días	mar 03/05/16	jue 12/05/16	0	0
28	**Análisis de resultados**	**5 días**	**jue 12/05/16**	**vie 27/05/16**	**0**	**0**
29	**Documentación de la memoria del proyecto**	**12 días**	**vie 27/05/16**	**mar 05/07/16**	**1**	**10%**

Tabla 3 – Distribución en tareas real a la finalización del proyecto

4.3 Diagrama temporal

Para esta parte se ha elegido representar tan solo las tareas más importantes, con el fin de clarificar en mayor medida la distribución a lo largo del tiempo. Por esta misma razón se ha utilizado una división en semanas y no en días, tal como se ha establecido en la planificación. El resultado se puede apreciar en la Tabla 4.

ID Tarea	1ª nov	2ª nov	3ª nov	4ª nov	1ª dic	2ª dic	3ª dic	4ª dic	1ª ene	2ª ene	3ª ene	4ª ene	1ª feb	2ª feb	3ª feb	4ª feb	1ª mar	2ª mar	3ª mar	4ª mar	1ª abr	2ª abr	3ª abr	4ª abr	1ª mayo	2ª mayo	3ª mayo	4ª mayo	1ª jun	2ª jun	3ª jun	4ª jun	1ª jul
2																																	
9																																	
17																																	
20																																	
27																																	
28																																	

Tabla 4 – Diagrama temporal de las tareas principales

Capítulo 5. Desarrollo y resultados

5.1 Herramientas necesarias

Antes de plantearse cualquier desarrollo con la placa ZedBoard, se creyó conveniente instalar todo programa que podría llegar a ser de utilidad de Xilinx. Así pues, y gracias a la licencia que se incluye en la caja de la ZedBoard, se obtuvieron los siguientes programas con la licencia de *Design Edition* común para todos ellos:

- Vivado 2015.4
- Vivado HLS 2015.4
- Xilinx SDK 2015.4
- System Generator 2015.4
- ISE Design Suite 14.7
- Xilinx PlanAhead 14.7

Los más importantes para el desarrollo están explicados en la sección correspondiente: **6.3**.

Con el software instalado, el siguiente paso es atacar al hardware que tenemos ante nosotros: la placa de desarrollo ZedBoard.

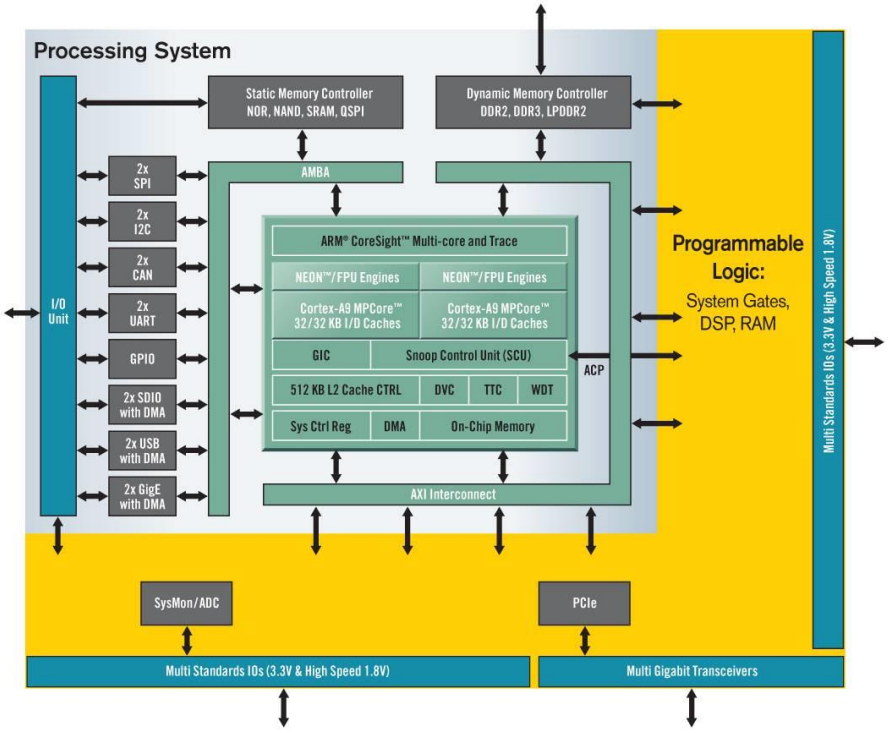

Ilustración 7 – Diagrama de bloques del ZynQ

5.2 Empezando con el hardware

El mayor desafío al que se encuentra cualquier desarrollador ante un proyecto de este estilo es el arranque del mismo, sobretodo si se desconocen tanto la plataforma de trabajo como el material a utilizar. Es por ello, que se escogió comenzar por una formación previa en temas de diseño con Xilinx, para poder comprender el funcionamiento de sus herramientas y las características destacables que estas pueden ofrecer.

A su vez, el estudio no acabó con las herramientas, si no que hubo de profundizar en la plataforma de trabajo, la placa de desarrollo ZedBoard, y el SoC principal que incorpora: ZynQ.

Una vez que se tenía conocimiento suficiente en estos campos, se dio inicio a la tarea de configuración del hardware:

- ¿Qué se necesita para que arranque?
- ¿Qué se necesita para programar la FPGA?
- ¿Qué se necesita para programar los microcontroladores?
- ¿Cómo pueden interactuar entre ellos?
- ¿Cómo se configura el chip de audio?

Estos han sido los puntos principales a los que se ha dedicado una gran parte del proyecto, pero en este apartado nos centraremos en las carácterísticas técnicas de cada uno, con el objetivo de que pueda servir de manual para cualquier futuro desarrollador.

5.2.1 Arranque del HW

La placa dispone de varios *jumpers* que efectúan la tarea de configuradores del hardware. Los que interesan para este proyecto y su configuración son los siguientes:

JP7	JP8	JP9	JP10	JP11
Boot_Mode[3]	Boot_Mode[0]	Boot_Mode[1]	Boot_Mode[2]	Boot_Mode[4]
Modo del JTAG	Selectores del modo de arranque del sistema (ver Tabla 6 – Modos de configuración de ZedBoard)			Selector de PLL

Tabla 5 – Configuración de los Jumpers

	Boot_Mode[4]	Boot_Mode[2]	Boot_Mode[1]	Boot_Mode[0]	Boot_Mode[3]
JTAG MODE					
Cascaded JTAG					0
Independent JTAG					1
BOOT DEVICES					
JTAG		0	0	0	
Quad - SPI		1	0	0	
SD Card		1	1	0	
PLL MODE					
PLL Used	0				
PLL Bypassed	1				

Tabla 6 – Modos de configuración de ZedBoard

La configuración que se usará para todo el proyecto es la misma: J7-11 conectados a GND (0).

Además, será necesario utilizar la entrada y salida de audio, así como las conexiones micro-usb para el JTAG y el *Debugger*.

Para poder utilizar correctamente las características del USB OTG (On the Go), será necesario instalar los drivers específicos del dispositivo USB: Microsoft Certified USB UART Driver. Además de instalar los drivers, será necesario realizar configuraciones adicionales al dispositivo desde el panel de Windows de administrador de dispositivos: "*Enable Port Persist*".

Una vez que esté alimentada a la corriente eléctrica, con el switch 8 se controlará el encendido/apagado de la placa.

En el caso de que se programe la FPGA, el LED 12 será indicativo de que la programación se ha realizado correctamente, cuando se ilumine de azul.

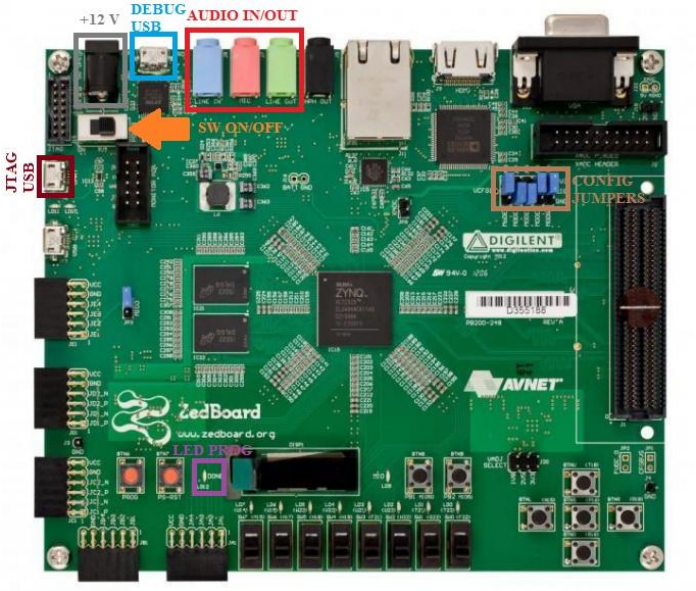

Ilustración 8 – Componentes más importantes de la ZedBoard

5.2.2 Programación de la FPGA y de los microcontroladores

Como se ha comentado antes, la programación tanto de la FPGA como de ambos microcontroladores se realizará mediante la conexión micro-usb del JTAG.

El software de Xilinx SDK será el encargado de gestionar la programación de las 2 partes.

Para la FPGA necesitará un BitStream generado o bien por Vivado HLS o bien por Vivado. En el caso de los microcontroladores, será un fichero binario que se podrá generar directamente desde la herramienta de Xilinx SDK, pudiendo elegir a cuál de los dos programárselo.

5.2.3 Interacción entre PS y PL – Protocolo AXI

La interacción principal se realiza de normal sobre un bus dedicado AXI de gran velocidad, el cual admite diferentes tipos de conexiones, para que se pueda ajustar a la aplicación deseada.

Existen dos formas de implementarlo en el PL: o bien documentarse sobre ellos y realizar manualmente los módulos de interconexión, o mejor aún, utilizar Vivado HLS para realizar las conexiones, aunque el resto del proyecto se haya realizado con Vivado.

En el PS se simplifica, y simplemente será una dirección de memoria en la que leer, como podría ser cualquier tipo de memoria conectada.

Esta interconexión es configurable, como se ha comentado antes, tanto en prestaciones como en tamaños de palabras, etc. El protocolo que se usa para esta conexión es el ***AXI***, cuyas características más destacables son:

- Tiene separadas las fases de control/direccionamiento de las de transferencia de datos.
- Gracias a sus bytes de validación en paralelo con los datos, permite la transferencia de datos no alineados.
- Puede transmiritir un *burst* de datos tan solo con que se transmita la dirección desada.
- Los canales de lectura y escritura están separados, lo que permite implementaciones poco costosas de DMA (*Direct Memory Access*).
- Se pueden usar varias direcciones de memoria como destino.
- Puede incorporar fácilmente registros para el ajuste de temporización.

Además del protocolo principal, existen tres variantes, cada una pensada y desarrollada para cumplir una función diferente. A continuación, se detallarán sus cualidades:

- **AXI4** — El básico y el de mayor rendimiento de todos, perfecto para mapeos de memoria, permitiendo una transferencia de hasta 256 ciclos por cada fase de direccionamiento.
- **AXI4-Lite** — Una versión más ligera que el anterior, el cual trabajo mejor en comunicaciones de mapeo de memoria con un solo canal de transacción. No soporta las transmisiones *burst* de datos, tan solo de un canal.
- **AXI4-Stream** — Es el único que no trabaja con un mapeado de memoria, sino con un canal de transacción dedicado. La comunicación es de tipo maestro-esclavo, pudiendo ser bidireccional, pero teniendo que ajustarse los extremos al tipo requerido. Dependiendo de las necesidades de cada proyecto, se usarán un protocolo u otro, pudiendo incluso combinar distintos para diferentes módulos.

Existen ciertos interfaces y ciertas interconexiones que son comunes a distintos tipos de conexiones AXI, y otras que son de uso general para que se pueda especializar cada una de ellas. En la **Ilustración 9** se pueden apreciar en conjunto las diferentes líneas de comunicación que utiliza el estándar AXI, y en la **Tabla 7** están recogidas las funciones de cada interfaz, explicadas con mayor claridad y profundidad justo debajo de esta.

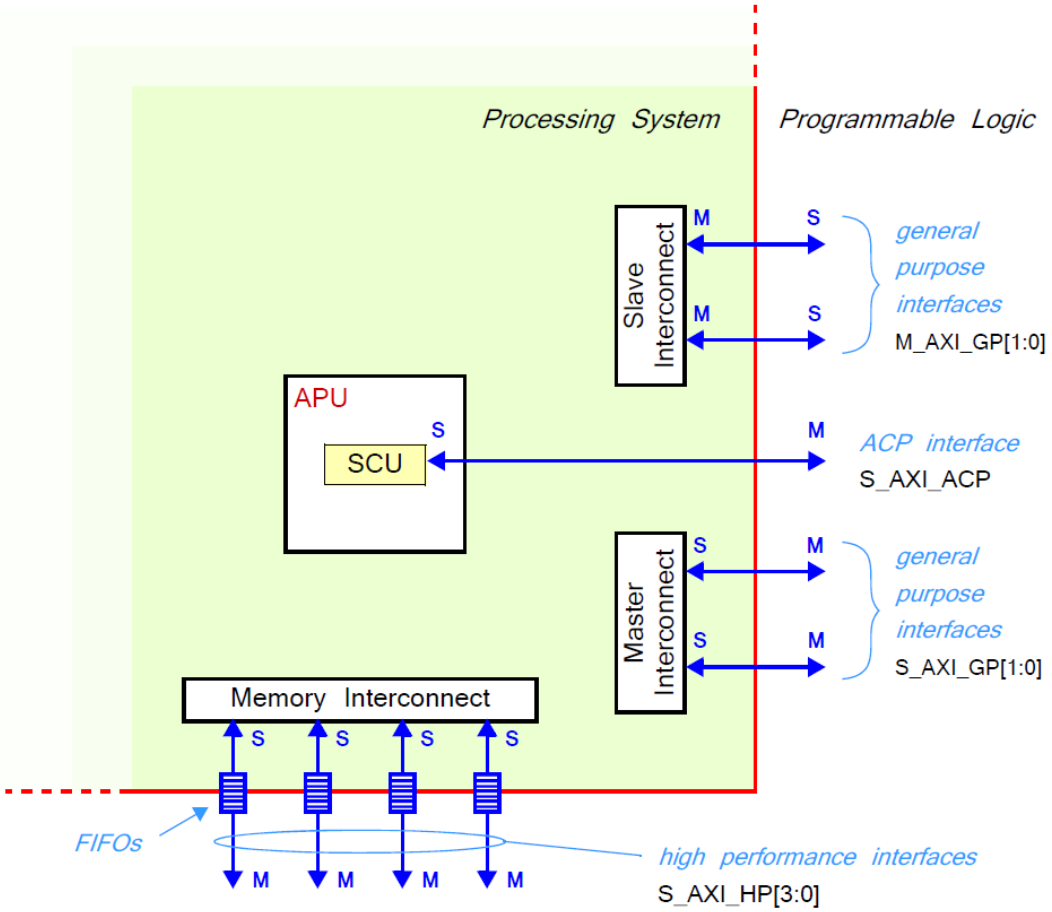

Ilustración 9 – Interconexiones e interfaces del protocolo AXI

Nombre	Descripción	Master	Slave
M_AXI_GP0	Propósito General	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	Propósito General	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	Gestiona la coherencia de la caché	PL	PS
S_AXI_HP0	Puertos de alto rendimiento con FIFOs de lectura/escritura.	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

Tabla 7 – Interfaces del protocolo AXI

- **GP AXI**: Un bus de 32 bits, el cual es perfecto para comunicaciones de baja o media tasa entre PL y PS. La interfaz es directa, y no necesita buffering.

- **ACP**: Un conexión asíncrona directa entre el PL y la APU del PS, con un tamaño de 64 bits. Se usa para conseguir coherencia entre la caché del APU y los elementos del PL.

- **HP**: Se utilizan para facilita el modo "burst" de comunicación y también, aquella comunicaciones que requieran una alta tasa de comunicación entre el PL y elementos de la memoria del PS. El tamaño puede ser de 32 bits o de 64 bits.

Además del estándar AXI, existe otro protocolo de comunicación importante para el desarrollo con ZynQ, el **EMIO**.

El EMIO o Extended MIO, es un protocolo que establece una comunicación directa entre el PS y un dispositivo externo, a través del PL, aunque también se puede utilizar para conectar el PS con un bloque IP que se haya añadido al PL.

Se implementa con un conjunto de conexiones básicas, por lo que no podría utilizarse para algunas comunicaciones MIO que exijan una gran tasa de transmisión. En conjunto, se habla de que se implementa con dos bancos de 32 bits.

Las conexiones que se realizan mediante este protocolo se suelen definir en el proyecto mediante constraints de los puertos.

5.2.4 Configuración del chip de audio

Como primer aproximamiento al integrado de estudio cabe estudiar sus características más detacadas de las cuales se pueda sacar provecho para el presente proyecto. Dichas características están comentadas en la sección 6.2.3, por lo que no es menester volver a comentarlas en este punto. Sin embargo, es de suma importancia conocer cómo se ha de llevar a cabo la configuración y el flujo de datos.

La configuración puede realizarse mediante los protocolos SPI e I^2C, a continuación se explicarán ambos métodos. Cabe destacar que ambos funcionarán basándose en el mismo método: se irán escribiendo bytes en los registros del ADAU1761 que se desee para su correcta configuración. También es necesario destacar, que en ambos casos el chip de audio actuará como Slave y el SoC ZynQ como Master.

En el caso del SPI la interfaz constará de 4 hilos: uno dedicado a ser el reloj del sistema, generado por el Master (CCLK), otro para los datos enviados desde el Master al Slave (CDATA) y para los recibidos del Slave (COUT) y un último que sería para elegir uno de los diversos Slave (CLATCHn) pero que en este caso no será de utilidad.

Para escribir un byte en un registro determinado, será necesario el envío de 4 bytes como mínimo:

- 0000000 + R/W. R = 0, W = 1. Indica que vamos a leer, por lo que quedaría 00000001.
- Parte alta del registro de 16 bits donde se quiera escribir.
- Parte baja del registro.
- Byte a escribir.

A partir de este momento el Master puede seguir enviando bytes y se irán escribiendo en los registros continuos.

Como se puede apreciar en la Ilustración 10, el inicio de la comunicación lo marca la detección en un flanco de subida del CCLK del estado "0" en la línea CLATCHn.

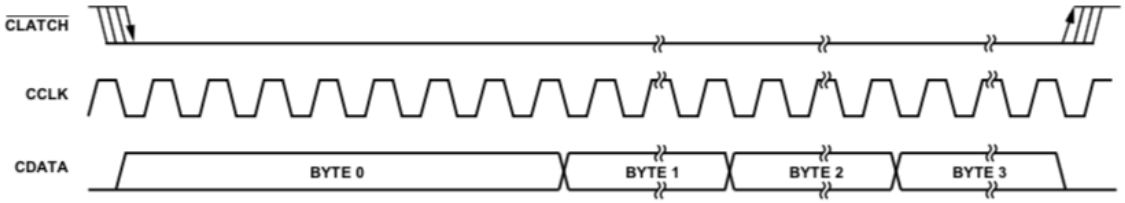

Ilustración 10 – Escritura SPI

En el caso de que se quiera leer un dato, el flujo de trabajo será idéntico hasta mandar el tercer byte, momento en el que el Master esperará a que el Slave mande por COUT el byte a leer.

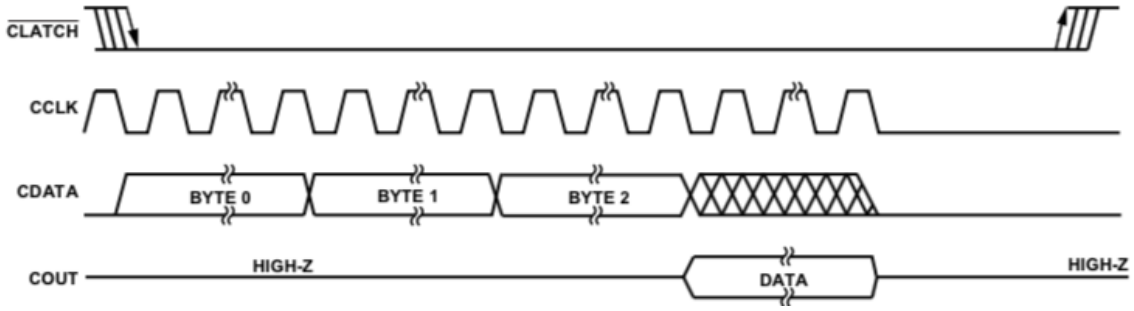

Ilustración 11 – Lectura SPI

El caso del I^2C varía ligeramente, ya que usará un protocolo para la configuración, el I^2C estándar, y una modificación del mismo para la lectura/escritura de muestras de audio, el I^2S.

Comenzamos explicando el I^2C: constará de 2 líneas de comunicación, SDA (línea de datos) y SCL (reloj de la comunicación). El SCL lo gestionará únicamente el Master, pero el SDA será compartiendo por Master y Slave.

Cada dispositivo Slave I^2C tiene una dirección que lo identifica para que el Master pueda establecer comunicaciones directamente con él. Por defecto, el ADAU1761 tiene como dirección la 0111000, en este caso es de 7 bits, pero el I^2C también soporta direcciones de 10 bits.

La comunicación comienza cuando el Master envía la condición de Start (SDA se pone a nivel alto y luego vuelve a bajar, mientras que el SCL se queda a nivel alto).

Después de esto, el Master envía los 7 bits de dirección del Slave por SDA añadiendo un bit para indiciar si la comunicación es de escritura (0) o lectura (1). Si el Slave lo ha leído e indentifica la dirección como la suya, pondrá la línea SDA a nivel bajo en el noveno ciclo, siendo esto interpretado como un ACK.

Ahora que el Master sabe que el Slave está activo y atendiéndole, comienza a mandar la parte alta del registro donde quiere escribir primero, esperando al ACK y luego repetirá para la parte baja del registro.

Después de enviar la dirección del registro, tan solo queda enviar el valor que quiera que se escriba en dicha dirección, esperando también un ACK de confirmación.

A partir de este punto, el Master puede seguir enviando datos de escritura, los cuales el Slave irá escribiendo en los registros consecutivos, hasta el momento en que el Master envíe una condición de Stop (sube la línea SDA y mantiene subida la del SCL).

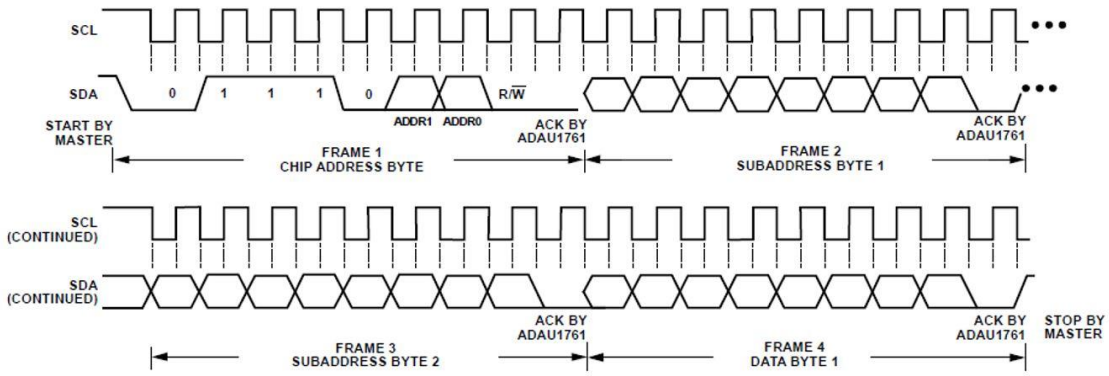

Ilustración 12 – Escritura I^2C

Para la extracción o introducción de datos en el chip de audio, tan solo se puede utilizar el protocolo modificado del I^2C, el **I^2S** (Inter-IC Sound). Esta modificación funciona de idéntica manera al I^2C, con la peculiaridad de que cuenta con 2 líneas de datos (SDATA_ADC y SDATA_DAC) y con 2 líneas de reloj (BCLK y LRCLK).

Haciendo analogía a lo explicado antes, BCLK funcionará como SCL, y LRCLK servirá para discernir si las muestras son del canal derecho (nivel alto) o del canal izquierdo (nivel bajo).

En cuanto a las líneas de datos, cada una representa la comunicación con el ADC o con el DAC por parte del Master, siendo de uso único cada uno de ellos.

Los registros más importantes a configurar serán los que tengan que ver con:

- Configuración del PLL para ajustar el reloj de trabajo.
- La frecuencia de muestreo deseada.
- La activación de los ADC/DAC. Mono o estéreo.
- El tamaño de palabra a usar.
- El modo de representación que se desea: coma fija o coma flotante.

5.3 Elección modelo de representación.

Una vez tenemos el hardware configurado y listo para usar, con las herramientas de desarrollo instaladas, el siguiente paso es decidir el sistema de representación binario a utilizar: coma fija o coma flotante.

Las principales características de cada método ya han sido presentadas en los puntos **3.3** y **3.4**, por lo que solo queda analizar cuál conviene más para este proyecto.

Debido a que los sistemas con coma flotante presentan frecuencias de funcionamiento menores frente a los de coma fija, y a que estos primeros presentan en su mayoría de casos problemas derivados de la cuantificación en proyectos de audio digital, se ha decidido optar por realizar todas las implementaciones con coma fija.

5.4 Configuración definitiva del ADAU1761

Una vez conocidos los métodos de actuación con el chip, se debe de tomar la decisión de qué parámetros relativos al audio utilizar en el proyecto. Para mayor simplicidad, se ha decidido utilizar los mismos para todo el proyecto, pudiendo tener así una comparación más efectiva de las características de cada sistema desarrollado.

En lo relativo a la frecuencia del reloj con la que trabajará el chip de audio, se ha decidido que sea de 10 MHz, proporcionada por el ZynQ, lo cual habrá de ser debidamente configurado en el diseño hardware. En cuanto a la frecuencia propia de funcionamiento del I2C Master, se ha establecido en 400 kHz.

Para la frecuencia de muestreo se ha decidido que su valor sea de 48 kHz, asegurando así la calidad del audio y debido a que es uno de los valores más comunes hoy en día en el sector. El tamaño de palabra elegido es de 24 bits y como se ha comentado anteriormente, será con representación de coma fija.

A partir de este punto se introducirá la secuencia de escritura en los registros necesaria para configurar el sistema para que funcione como ha sido descrito arriba. Los bits que no sean nombrados de los registros se tomarán como reservados o que no impactan en el proyecto.

R0_CLOCK_CONTROL = 0x0E

Bit 3 a 1: El reloj del sistema se coge de un PLL, a partir de una señal que entrará por MCLK.

Bits 2-1 a 1: Frecuencia de la señal de reloj. Al estar con PLL externo se queda en 1024 x fs fijo. De aquí él extrae el valor de fs para el sistema, según la frecuencia que tenga a la salida del PLL.

Bit 0 a 0: Desactiva el reloj interno. Es necesario para modificar su valor más adelante.

Si la frecuencia de muestreo va a ser 48kHz, y queda configurado como 1024 x fs, podemos deducir la frecuencia de salida que deberá tener el PLL y así extraer sus parámetros de configuración:

$$\boldsymbol{PLLout = 1024 * 48\,kHz = 49.152\,MHz} \quad \textbf{(1)}$$

Del catálogo del integrado de audio, se puede extraer la fórmula para el PLL, que dado la frecuencia de trabajo del reloj, estará en modo fraccional.

$$\boldsymbol{\frac{PLLout}{MCLK} = R + \frac{N}{M}} \quad \textbf{(2)}$$

Siendo R, N y M parámetros configurables a través del registro R1, con la característica de que R debe estar comprendido entre 2 y 8.

Dado el grado de libertad para ajustar estos 3 parámetros, se ha decidido fijar R a 4 primero, y de la relación N/M restante asignar 572 a N y 625 a M, siendo soportable, ya que ambos tienen 16 bits configurables.

El resultado a escribir en el registro R1 será el siguiente:

R1_PLL_CONTROL = 0x02 0x71 0x02 0x3C 0x21 0x01

Byte 1: MSB de M

Byte 2: LSB de M

Byte 3: MSB de N

Byte 4: LSB de N

Byte 5: Bits 6-3 R. Bits 1-0 PLL configurado sin divisor.

Byte 6: Bit 1 PLL lock. Bit 0 PLL enable.

Ahora que el PLL ha sido configurado correctamente, es necesario volver al R0 y habilitar el último bit, el cual estando a nivel bajo permite la configuración del mismo.

R0_CLOCK_CONTROL = 0x0F

R19_ADC_CONTROL = 0x13

Bit 4 a 1: No invierte la polaridad de los datos del micrófono.

Bits 1-0: Controlan la habilitación de los ADC. Estando a 11 se habilitan ambos.

R29_PLAYBACK_HEADPHONE_LEFT_VOLUME_CONTROL = 0xFF

Bits 7-2: Controlan el volumen. Estando a 111111 corresponde a + 6 dB de ganancia.

Bit 1 a 1: Habilita la salida.

Bit 0 a 1: Aplica el nivel de volumen seleccionado.

R30_PLAYBACK_HEADPHONE_RIGHT_VOLUME_CONTROL = 0xFF

Igual al R29 pero para el canal derecho.

R31_PLAYBACK_LINE_OUTPUT_LEFT_VOLUME_CONTROL = 0xFE

Igual que R29. Estos registros están pensados para usarse si no se van a usar auriculares, y en su lugar se usarán altavoces.

R32_PLAYBACK_LINE_OUTPUT_RIGHT_VOLUME_CONTROL = 0xFE

Igual que R31 pero para el canal derecho.

R35_PLAYBACK_POWER_MANAGEMENT = 0x03

Bit 1 a 1: Habilita la salida del canal derecho.

Bit 0 a 1: Habilita la salida del canal izquierdo.

R36_DAC_CONTROL_0 = 0x03

Bits 1-0: Controlan la habilitación de los DAC. Estando a 11 se habilitan ambos.

R58_SERIAL_INPUT_ROUTE_CONTROL = 0x01

Bits 3-0: Configuran en qué orden se recibirán las muestras de datos y por qué canales. Estando en 0001 serán L0,R0 hacia L,R de los DAC.

R59_SERIAL_OUTPUT_ROUTE_CONTROL = 0x01

Bits 3-0: Configuran en qué orden se enviarán las muestras de datos y por qué canales. Estando en 0001 serán L,R de los ADC hacia los canales L0,R0.

R65_CLOCK_ENABLE_0 = 0x7F

Bit 6 a 1: Habilita el control del Slew Rate digital.

Bit 5 a 1: Habilita la opción para el ALC

Bit 4 a 1: Habilita la opción para el control automático del *jitter*.

Bit 3 a 1: Habilita un control del reloj serie, para que no se desfasen los datos de salida.

Bit 2 a 1: Otra opción para el control automático del *jitter*.

Bit 1 a 1: Como el bit 3, pero para los datos de entrada.

Bit 0 a 1: Controla las opciones de los bits 1 y 3.

R65_CLOCK_ENABLE_1 = 0x03

Bit 1 a 1: Habilita la generación del reloj digital 1.

Bit 0 a 1: Habilita la generación del reloj digital 0.

Tal y como se puede apreciar, no se ha hecho referencia ni al tipo de representación de los datos ni al tamaño de palabra de los mismos. Esto es debido a que las opciones elegidas en la fase de planificación resultaron ser las que tenía por defecto el integrado.

5.5 Prueba de test: aplicando conocimientos adquiridos

En este punto, además de abarcar el diseño de un proyecto simple a modo de ejemplo y prueba de todos los conocimientos mencionandos anteriormente, tiene como segundo objetivo el de introducir el entorno de trabajo con sus características más relevantes.

La prueba a realizar consistirá en una implementación básica de captura, mezclado y reproducción de audio. Se tendrá una señal de audio de entrada que se generará desde un PC mediante Matlab 2014, y para que la señal reproducida al final, mediante unos altavoces, sea diferente, se ha decidido incluir un módulo **NCO** (*Numerically Controlled Oscillator*) de código abierto, con el fin de generar una seál sinusoidal que pueda ser añadida a la de entrada. La señal generada dependerá de los switches activos en cada momento, por lo que habrá que añadir este módulo al proyecto. Y por último, para tener una referencia visual, no solo para este proyecto, sino para cualquier futuro, se decide incluir también un módulo de LEDs que marque los switches activos.

El control del audio se basará en un bloque IP estándar de código abierto, el cual viene proveido por la documentación de ayuda de Xilinx respecto a la ZedBoard.

5.5.1 Desarrollo de la parte PL

Se empezará por crear un nuevo proyecto en Vivado 2015. Para cualquier proyecto que se quiera llevar a cabo, el primer paso será incluir en el proyecto el módulo del sistema ZynQ, y adaptarlo a nuestras necesidades.

Para que el módulo funcione correctamente se debe de conectar apropiadamente los pines **DDR** y **FIXED_IO** que gobiernan la parte de conexiones "al exterior" del módulo, controlando la RAM y los pines IO respectivamente. Si al crear el proyecto con Vivado se selecciona la placa de desarrollo ZedBoard, este paso se puede automatizar con un botón. En caso contrario habría que declarar dichos pines como externos y conectarlos manualmente a sus respectivas conexiones en la placa de evaluación.

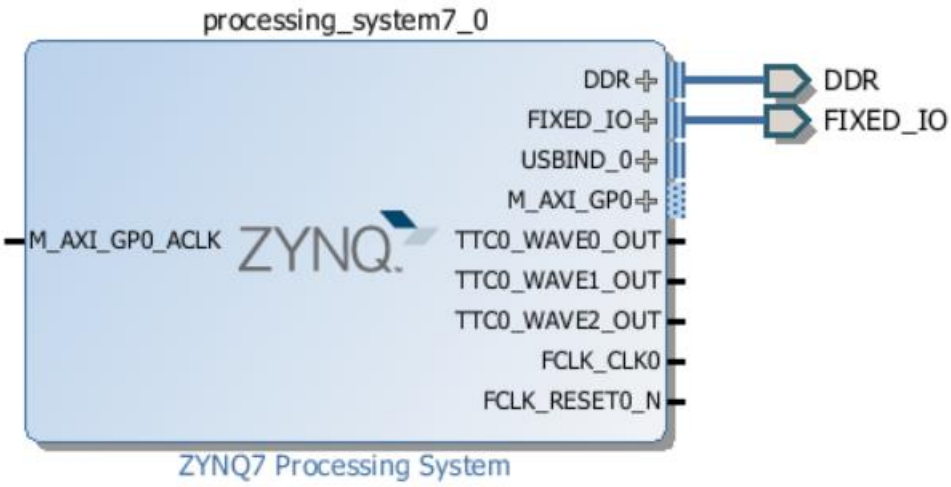

Ilustración 13 – Diagrama ZynQ básico

En este momento añadiremos el bloque IP referente al NCO Open Source (12). Se conectará mediante la opción automática, lo cual añadirá también el bloque referente a la conexión AXI, además de un bloque de control del reste para el ZynQ y para la comunicación AXI.

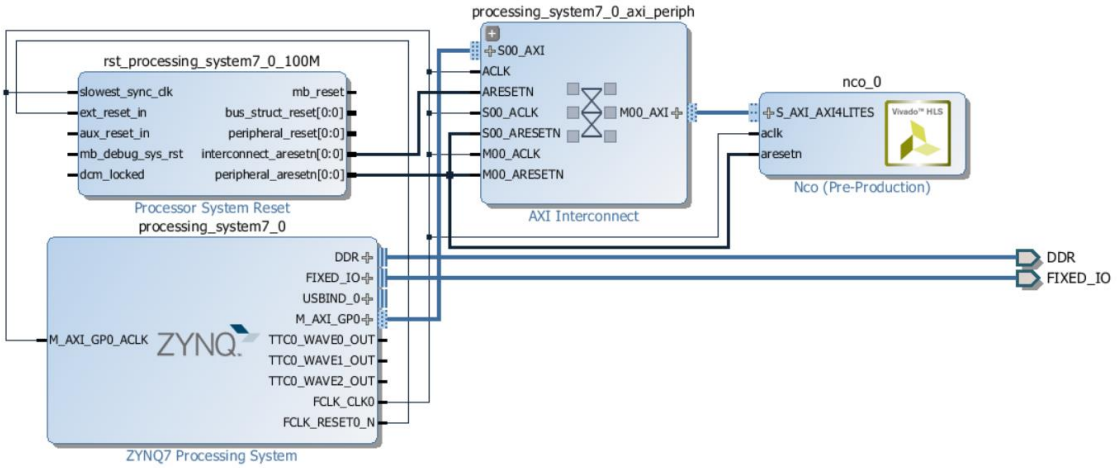

Ilustración 14 – Diagrama del Test de Audio con NCO

A continuación, se añadirá el bloque IP zed_audio_ctrl (13). Este bloque, de código abierto, implementa lo necesario para establecer la conexión I2C con el chip de audio, además de una comunicación AXI que se utilizará entre el ZynQ y el microcontrolador ARM para por ella circulen los datos de audio, tanto de salida como de entrada.

Utilizaremos la conexión automática por defecto, lo cual nos dejará con 4 pines sueltos todavía, los cuales serán las conexiones al exterior y se deberán marcar como externos.

Para que funcione correctamente faltaría por añadir un segundo reloj de 10 MHz, el cual se generará desde las opciones de ZynQ/PL Fabric y se conectará a la entrada de reloj del módulo.

Es momento de configurar las conexiones relativas al I2C, donde habrá que elegir si se quiere tener el chip de audio conectado mediante multiplexación (*MIOs*) o bien como una conexión extendida (*EMIOs*). En este caso se seleccionará el caso de *EMIO*, por dos razones, porque realmente sólo trabajaremos con un dispositivo externo al ZynQ, y porque no existe ninguna limitación que pueda implicar el uso de *MIO*.

Ahora dispondremos de dos nuevas salidas (**IIC_O** y **FCLK_CLK1**) las cuales, nuevamente, se pondrán como external.

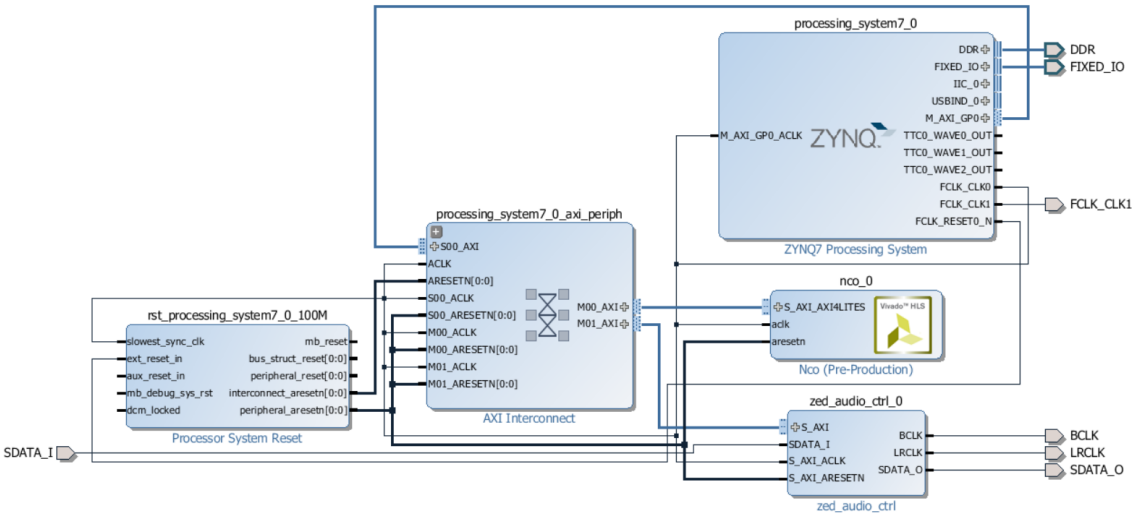

Ilustración 15 – Diagrama Test de Audio con control de audio

Llegados a este punto, tan solo falta por añadir los bloques GPIO correspondientes a los LED y a los switches. Con un solo bloque GPIO se pueden asignar diferentes grupos de entrada, dándole una dirección específica a cada uno, para que a la hora de crear el programa para el microcontrolador, se pueda discenir cuando se actúa sobre uno o sobre el otro. La salida de ambos grupos de GPIO también será external.

Antes de finalizar, crearemos una constante ADDRESS que conectaremos a un puerto de salida, con el objetivo de tener en una variable controlada el cambio entre la dirección de escritura y lectura del bus I2C del ADAU1761.

El resultado final es el mostrado en la Ilustración 16. Antes de continuar, es importante utilizar la herramienta de *Validación de Diseños* de Vivado 2015 para comprobar si se ha cometido algún error significante al realizar alguna conexión. A su vez, también se deberían de comprobar las direcciones asignadas a memoria de los diferentes módulos.

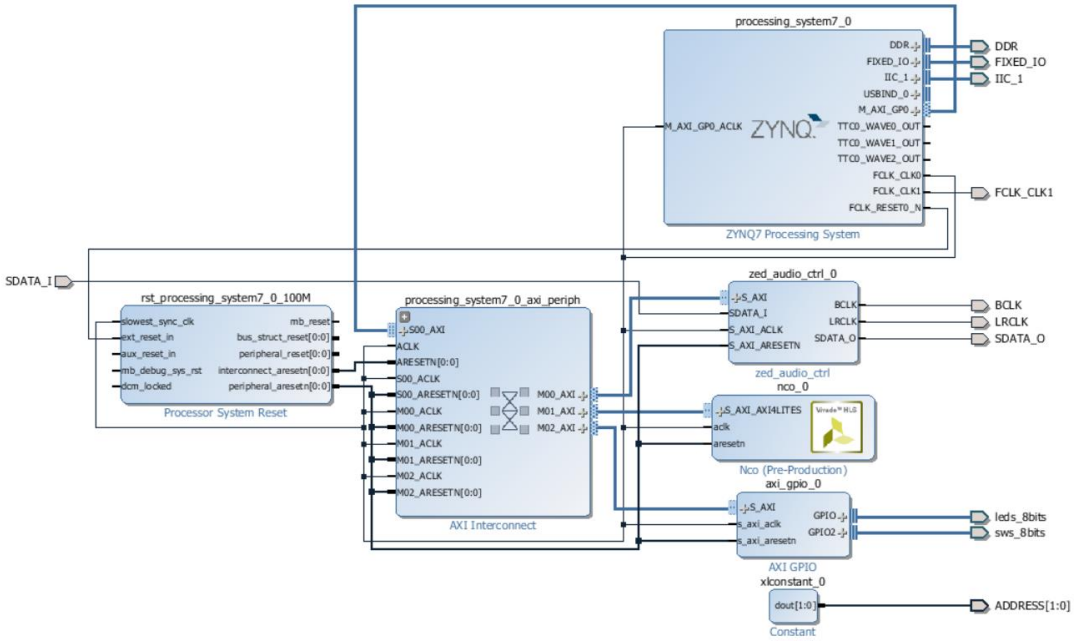

Ilustración 16 – Diagrama final PL Test de Audio

Ahora es momento de generar el HDL Wrapper, fichero con el cual el programa se prepara para transladar esta información a un futuro BitStream. Antes de generar dicho BitStream, que será cargado en la FPGA, es necesario añadir al proyecto un fichero de constraints que delimitará las conexiones de todos los pines utilizados.

Después de crear el BitStream, el último paso que se dará en Vivado 2015, es de exportar el diseño implementado, el cual por defecto, va a una subcarpeta del proyecto actual. Esto es importante porque si abrimos el Vivado SDK para trabajar en la misma carpeta del proyecto, accederá directamente a los ficheros referentes al diseño PL que acabamos de generar.

5.5.2 Desarrollo de la parte PS

Es el momento de abrir el programa de Vivado SDK para poder trabajar en la parte referente a los microcontroladores. La mejor opción para esto es lanzarlo desde la opción presente en Vivado 2015, debido a que ya tendremos el entorno configurado para trabajar con el proyecto actual, y automáticamente se generará una carpeta referente al SDK en la ubicación en la que reside el proyecto.

Crearemos un proyecto en blanco, incluyendo in fichero audio.c y otro audio.h. Toda la información y definiciones referentes al sistema físico estarán accesibles en la parte de BSP, por si se tuviera que consultar algo. En la mayoría de los casos, toda la información que será de nuestro interés será la referente a las direcciones de memoria de los diversos componentes añadidos al sistema, las cuales se pueden encontrar en el fichero **xparemeters.h**.

Como se puede ver en la Ilustración 17, el programa comienza con la inicialización de la comunicación I2C, teniendo que configurar la dirección del periférico con el que se va a trabajar, la frecuencia del reloj a utilizar y los puertos a utilizar. Seguidamente, es necesario tener configurado los registros relativos al PLL del ADAU1761, como se explicó en el punto 5.2.4, donde también se comentó la configuración necesaria para el correcto funcionamiento de los puertos de entrada y salida, que será lo siguiente a configurar.

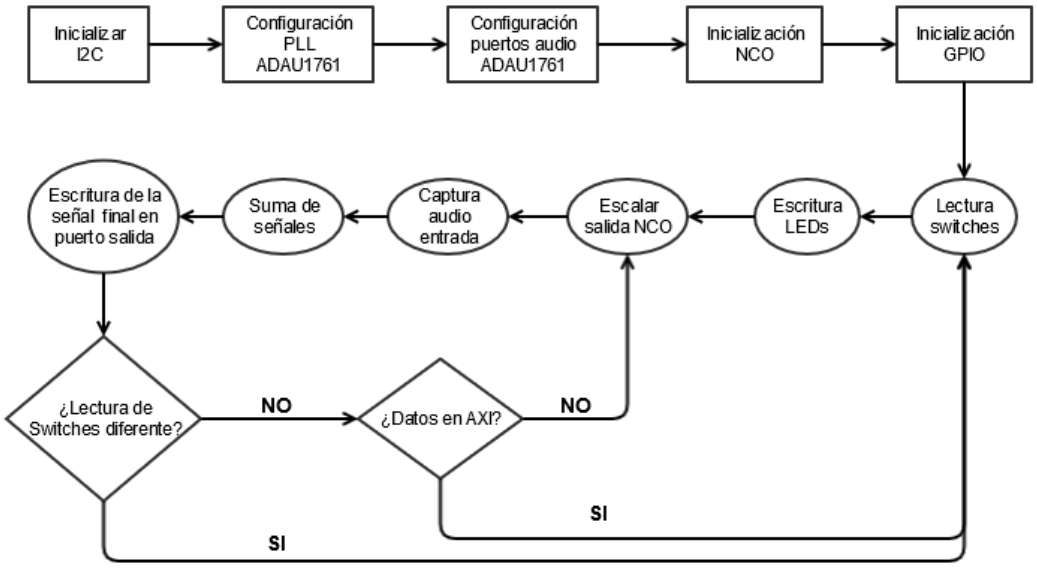

Ilustración 17 – Diagrama de flujo del test de audio

Una vez que finaliza la configuración relativa a la comunicación I2C y al chip ADAU1761, es el momento de inicializar los módulos restantes: el NCO y los GPIO para los switches y los LEDs. En el caso de los puertos GPIO, basta con asignar cada grupo a las direcciones asignadas en el PL y darles un valor por defecto. Como los switches son de lectura, el valor por defecto debe ser 0xFF, y los LEDs se dejan apagados a 0x00.

Al terminar todo el proceso básico de configuración e inicialización, es momento de atacar al bucle principal del programa, el cual realizará las siguientes tareas secuencialmente en su primera iteración:

- Lectura de los switches.
- Escritura de los LEDs según la lectura de switches.
- Escalado de la futura salida del NCO según la lectura de switches.
- Generación de la muestra procedente del NCO.
- Lectura del NCO.
- Lectura de los datos de entrada de audio.
- Generación de la muestra de salida, sumando la salida del NCO a la entrada captura de audio.
- Escritura de la señal de salida por el canal I2C.

A partir de este punto se realizan diferentes comprobaciones:

- Si la lectura de los switches ha finalizado, sale del bucle y vuelve a iniciar la función.
- Si llegan datos por el canal UART (AXI), finaliza el bucle y vuelve a iniciar la función.

Con esto es imposible que el programa termine, puesto que siempre se está volviendo a la función principal. El único problema puede ser que se quede enganchada la ejecución en algún punto, y si ello legase a pasar, se utilizaría la herramienta de *Debug* para depurarlo y solucionar el error.

Con el código escrito y compilado es momento de probarlo en la placa, para lo cual se utiliza un cable doble-jack para conectar la salida de audio de un PC a la entrada de audio de la placa ZedBoard. A su vez, se conectan unos altavoces a la salida de audio de la placa ZedBoard, y se conecta la misma a la corriente eléctrica.

Para poder cargar el programa generado, tanto la parte PS como la PL, es necesario conectar un cable USB-micro al puerto JTAG de la placa, y el otro extremo USB al PC. Primero se programará la FPGA con el BitStream generado en la sección 5.5.1 y después se realizará la programación del microcontrolador que fuese elegido al generar el proyecto.

Con toda la instalación montada y la programación realizada, se procede a probar el diseño, con todas las posibles configuraciones de switches posibles, introduciendo por el canal de audio primero un tono puro generado con Matlab 2014, y después una muestra de una canción, prestando atención a las variaciones producidas al superponer la señal de entrada con la generada por el NCO.

Todo el código de desarrollo al que se ha hecho referencia en esta parte está presente en el documento de Anexos.

5.6 Primer diseño: Filtro FIR acelerado

Este punto será utilizado tanto para hablar sobre el diseño en sí, como para profundizar en el flujo de trabajo que se ha seguido, y que a grandes rasgos marca el camino general para cualquier desarrollo de este tipo que se quiera abordar.

5.6.1 Flujo de trabajo con ZynQ

Al trabajar con un SoC que integra dos partes diferencias que cubren los campos de software y de hardware, es importante replantearse el esquema de trabajo que se podría plantear para cualquier otro proyecto de electrónica digital. En este proyecto se seguirá el flujo de trabajo que se muestra en la Ilustración 18.

Ilustración 18 – Diagrama de flujo del esquema de trabajo con ZynQ

Quizás la parte más importante a la hora de plantear el desarrollo de un sistema de estas características sea el particionado entre hardware y software. Las herramientas proporcionadas por Xilinx tienen una complejidad muy alta y ante pequeñas variaciones los resultados pueden ser abrumadores, tanto en el lado positivo como en el negativo, por lo que por norma general, se volverá recursivamente a esta fase para retocar partes del sistema.

Normalmente, las partes de desarrollo del hardware y software se deberían plantear como tareas a desarrollar en paralelo, con el fin de que haya simbiosis entre ambas tareas.

Siguiendo el esquema anterior, se realizaron las tareas de establecimiento de los requisitos del sistema y de sus especificaciones técnicas asociadas, tal y como se ha comentado en el apartado **4.1**.

La siguiente tarea será un diseño de un filtro FIR básico mediante el programa de Vivado HLS, para posteriormente plantear diferentes formas de aceleración y evaluar cuál es la definitiva a implementar. Para la básica se realizará un TestBench que pueda asegurar su correcto funcionamiento.

A partir de ese momento, se generará un bloque IP del filtro FIR acelerado, el cual se implementará en un diseño mediante Vivado 2015, añadiendo las partes necesarias para las entradas/salidas de audio y su control mediante la programación de un microcontrolador.

Es importante recalcar que los diseños se han realizado teniendo que en cuenta que se va a utilizar la señal básica de 100 MHz que dispone el SoC ZynQ, sin tener en cuenta que posteriormente para otros diseños se pudiese aumentar mediante los PLL y llegar a frecuencias muy superiores.

5.6.2 Diseño de FIR básico con HLS

Para comenzar el diseño se ha tomado como referencia un filtro general FIR con un número de coeficientes grandes, debido a que el objetivo es que el diseño final sea capaz de gestionar un gran número de coeficientes en un solo ciclo de reloj.

Se propone realizar un diseño inicial de 200 coeficientes, dejando el diseño abierto a que se puedan encadenar sucesivos filtros de estas características de forma "modular" con relativa facilidad, para así afrontar requerimientos de escalado en cuanto a coeficientes.

Antes de atacar a temas de diseño "*físico*" del filtro, es importante hacer el diseño de coeficientes, es decir, qué tipo de respuesta se pretende que tenga el filtro FIR. Para este proyecto, no es un objetivo conseguir respuestas mejores en cuanto a ganancia ni en cuanto a rizado, o cualquier otro parámetro, por lo que usará la herramienta FDA Tool de Matlab 2014 para hacer un diseño básico de un filtro paso banda de 200 coeficientes, los cuales se encuentran en el Anexo. Los resultados se pueden ver en la Ilustración 19 y sus características en la Tabla 8.

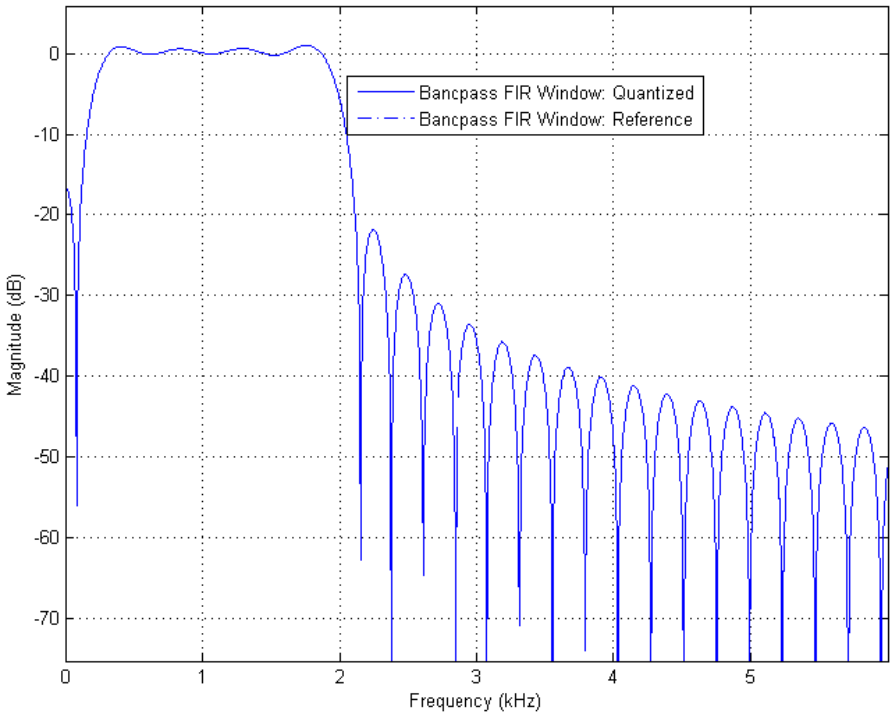

Ilustración 19 – Respuesta en frecuencia del filtro FIR diseñado

Parámetros	Valores
fs (frecuencia de muestreo)	48000 Hz
N (Orden)	200
Fc1 (primera frecuencia de corte)	200 Hz
Fc2 (segunda frecuencia de corte)	2000 Hz
Tipo	Vector Ventana
Algoritmo	Káiser
Parámetro de Ventana	0.5

Tabla 8 – Características del filtro FIR diseñado

La implementación del filtro se plantea de base como un filtro FIR clásico, que siga una estructura como la que se pueda apreciar en la Ilustración 20. Como se puede apreciar, tiene los elementos clásicos: una línea de retardos (un buffer), diversos multiplicadores y sumadores (operaciones básicas).

La aproximación se hará desde un enfoque puramente software, no hardware. Esto es debido, por una parte, a que el programa de Vivado HLS 2015 lo que mejor interpreta es el código en C/C++, y es dicho programa quien después de analizarlo con las directrices que marquemos para su aceleración lo transladará a VHDL/Verilog, y por otra parte, este estudio de posiblidades de la herramienta es uno de los objetivos del trabajo, junto a su evaluación como posible atracción de programadores "clásicos" de lenguajes como pueden ser C/C++ al entorno de desarrollo de una FPGA.

Por tanto, en este momento los retardos serán una acumulación en buffer y no Flip-Flops, y no se hará una implementación teniendo en cuenta el rutado entre los diferentes componentes del SoC ZynQ.

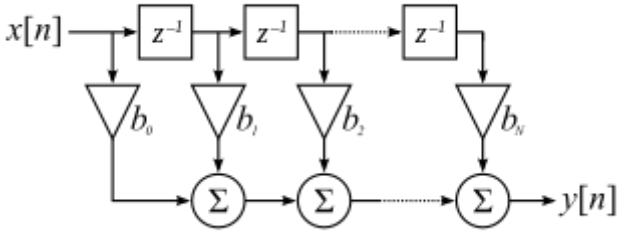

Ilustración 20 – Filtro FIR básico

En la Ilustración 21 se puede ver un diagrama representativo de la implementación que se ha llevado a cabo para el FIR básico, cuyo código se encuentra en el documento de Anexos, y resumido en su funcionalidad principal, debajo de la mencionada ilustración. Se puede apreciar como se ha delimitado la entrada y la salida a 24 bits, tal como se comentó en la sección 5.4, correspondiente a la configuración del ADAU1761, así como las operaciones, que se han realizado todas teniendo en cuenta que se trabaja con una representación en punto fijo, con 1 bit asignado a la parte entera y 23 a la parte decimal. Esto tendrá consecuencias a la hora de tratar los datos, que se verá más adelante.

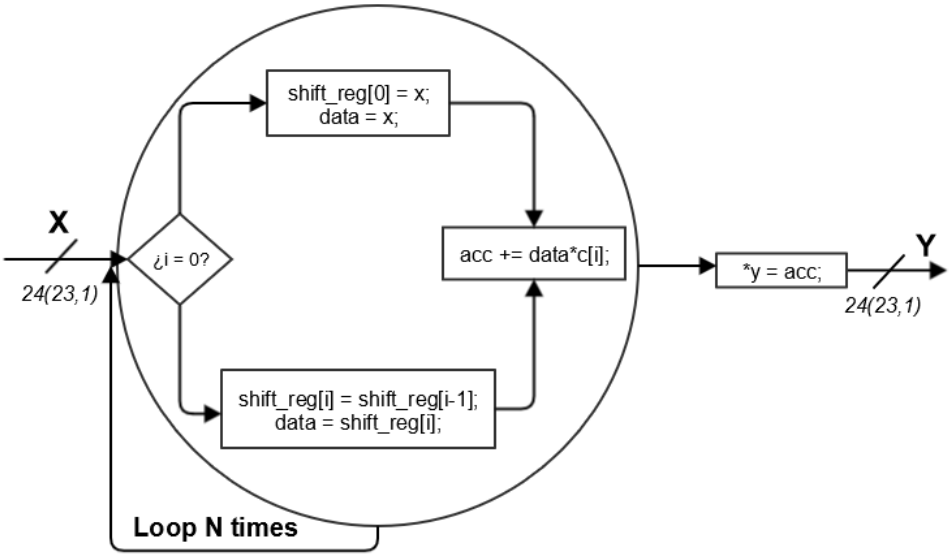

Ilustración 21 – Diagrama de flujo del filtro FIR en SW

```
Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
        if (i==0) {
                shift_reg[0]=x;
                data = x;
         } else {
                shift_reg[i]=shift_reg[i-1];
                data = shift_reg[i];
         }
        acc+=data*c[i];
}
 *y=acc;
```

Después de comprobar que el proyecto compila sin problemas, es hora de pasar a temas más importantes: ¿qué hacer con el *overflow*? Se entiende por *overflow* aquellos datos que "sobresalen" de los límites del sistema, por ejemplo: si tenemos una muestra de audio retardado,

por ejemplo, de 0.8 sobre 1, y se multiplica por 0.7 y al resultado se le suma otra entrante de 0.9, el resultado sería de 1,46, es decir, se estaría perdiendo información en el mejor de los casos, ya que el sistema lo saturaría a 1, o bien, estaría "dando la vuelta" y llegando a -0,54, lo cual en temas de audio puede destrozar completamente la señal.

Para evitar este molesto problema, se debe de tener en cuenta en las etapas de diseño un sobredimensionamiento de las etapas intermedias del filtro FIR. Para la variable que gobierne todo el sumatorio del filtro, se usarán 8 bits de más, que luego serán restringidos a los 24, antes de extraerse del sistema. Esto se hace porque la acumulación en el sumatorio hacia el final del filtro puede llegar a ser significativamente mayor que en cualquier otro lugar.

Con el filtro ya diseñado en su modo más básico, es un buen momento para realizar la verificación mediante el TestBench y después la primera síntesis del bloque funcional.

Por suerte, la herramienta de Vivado HLS 2015 permite añadir un fichero de TestBench en C++ al proyecto, encargándose el programa de toda la gestión.

Para el TestBench en sí, se generará una señal artificial de 600 muestras, con valores que irán escalando en 0.1, desde 0 hasta 1 y luego hasta -1, para dar la vuelta sucesivamente. Dicha señal se utilizará para realizar el análisis, introduciéndola en el filtro y guardando la salida que produzca. Mediante Matlab también se introducirá esa señal en el filtro diseñado, guardando su salida en un fichero, contra el que se comprobará el fichero generado desde el TestBench. Al fichero generado desde Matlab, por ser el de referencia ideal, se le conoce como *Golden Model*. El resultado final será "PASS" si concuerdan los resultados y "FAIL" si no concuerdan. Los resultados están restringidos a 6 decimales, para que la computación no sea excesivamente larga y costosa.

Se lanza el TestBench y en ese momento el programa realizará la síntesis de nuestro diseño en Verilog o VHDL, según se escoja, para luego simularlo con la rutina de TestBench en C++. Se comprueba que el resultado es "PASS", tanto para la co-simulación con Verilog como con VHDL, sin diferencias aparentes entre ambos.

Los resultados del TestBench son los siguientes:

		Latencia			**Throughput**			**Timing (ns)**
RTL	**Status**	**min**	**avg**	**max**	**min**	**avg**	**max**	-
VHDL	**PASS**	1401	1401	1402	1401	1401	1402	6,08
Verilog	**PASS**	1401	1401	1402	1401	1401	1402	6,08

Tabla 9 – Resultados TestBench diseño básico

Y a continuación los resultados de la primera síntesis:

Name	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
DSP	-	-	-	-
Expression	-	-	0	65
Instance	-	2	0	0
Memory	2	-	0	0
Multiplexer	-	-	-	110
Register	-	-	180	-
Total	**2**	**2**	**180**	**175**

Utilization(%)	**~0**	**~0**	**~0**	**~0**

Tabla 10 - Resultados síntesis diseño básico en recursos

Esta vez se ha incluido la tabla completa de recursos utilizados, para todas las siguientes iteraciones de síntesis, tan solo se incluirá el total y el porcentaje de utilización de cada uno, así como el resumen de latencia, que en este caso, viene indicado en el TestBench.

En la tabla superior se puede apreciar los valores de latencia y de *throughput*, explicados en la sección 3.8. Además, se incluye el valor del *timing*, que representa el tiempo mínimo que tarda en realizar un ciclo, lo cual determina la frecuencia máxima a la que podría funcionar. Dado que el reloj principal con el que trabaja el ZynQ es de 100 MHz, sin PLLs, nuestro objetivo será quedarnos por debajo de los 10 ns.

En la tabla inferior, la referente a los recursos, existen varias columnas que representan los distintos componentes que componen, de manera genérica una FPGA del estilo de la ZynQ. Primero tenemos los bloques RAM de 18kB, después las células DSP48E, explicadas en la sección 6.2.4, el número de Flip-Flops, y por último, las células LUT (Look-Up-Table). Estas últimas son células que actúan como tablas dinámicas donde se graban resultados de operaciones complejas que se repiten en gran medida a lo largo de un procesado, con el objetivo de que no se deban realizar las operaciones y baste con consultar la tabla.

Por último, y antes de pasar a la sección de aceleración, se deberá añadir la parte de interfaz entre el ADAU1761 y el ZynQ. Para ello se usará un interfaz AXI Lite, el cual se configurará utilizando las directrices de diseño que permite añadir Vivado HLS 2015,

Antes de añadir las directrices, es necesario crear una nueva *solution*, las cuales actúan como proyectos independientes de configuración en cuanto a directrices y reglas de síntesis, permitiendo una mayor flexibilidad e intentos en un mismo proyecto sin tener que realizar un control de versiones por parte del usuario.

Dichas directrices se podrán añadir de dos formas: bien introduciéndolas en el código fuente con el prefijo "*#pragma*" o bien en un fichero dedicado *directives.tcl*, que se encuentra en la ruta raíz de cada *solution*, siendo por tanto restringidas las directrices que se incluyan en dicho fichero a la *solution* con la que se esté trabajando.

Se asignará las interfaces a los puertos de entrada y salida.

```
set_directive_interface -mode s_axilite "fir" y
set_directive_interface -mode s_axilite "fir" x
```

Además de esto, también es necesario añadir la directriz a la entidad superior de la función para que el programa pueda entender que todo el bloque IP funcionará bajo este interfaz. Por temas de sencillez y ahorro de recursos, se puede añadir otra directriz para que no se implementen las líneas de control, las cuales en su mayoría de ocasiones son innecesarias.

```
set_directive_interface -mode s_axilite "fir"
set_directive_interface -mode ap_ctrl_none "fir"
```

Con todo esto, se sintetizará el código en una implementación hardware. Para ello hay que elegir el lenguaje de salida, Verilog o VHDL, y debido a que Xilinx recomienda VHDL, ya que es el lenguaje para el que están optimizadas estas herramientas, se decide trabajar con ese, aunque en el TestBench no se hayan apreciado diferencias.

Latencia		**Intervalos**		**Timing (ns)**
min	**max**	**min**	**max**	**-**

1401	1401	1402	1402	6,08

Tabla 11 – Resultados de temporización del diseño con interfaz

Name	BRAM_18K	DSP48E	FF	LUT
Total	2	2	300	311
Utilization (%)	**~0**	**~0**	**~0**	**~0**

Tabla 12 - Resultados de recursos del diseño con interfaz

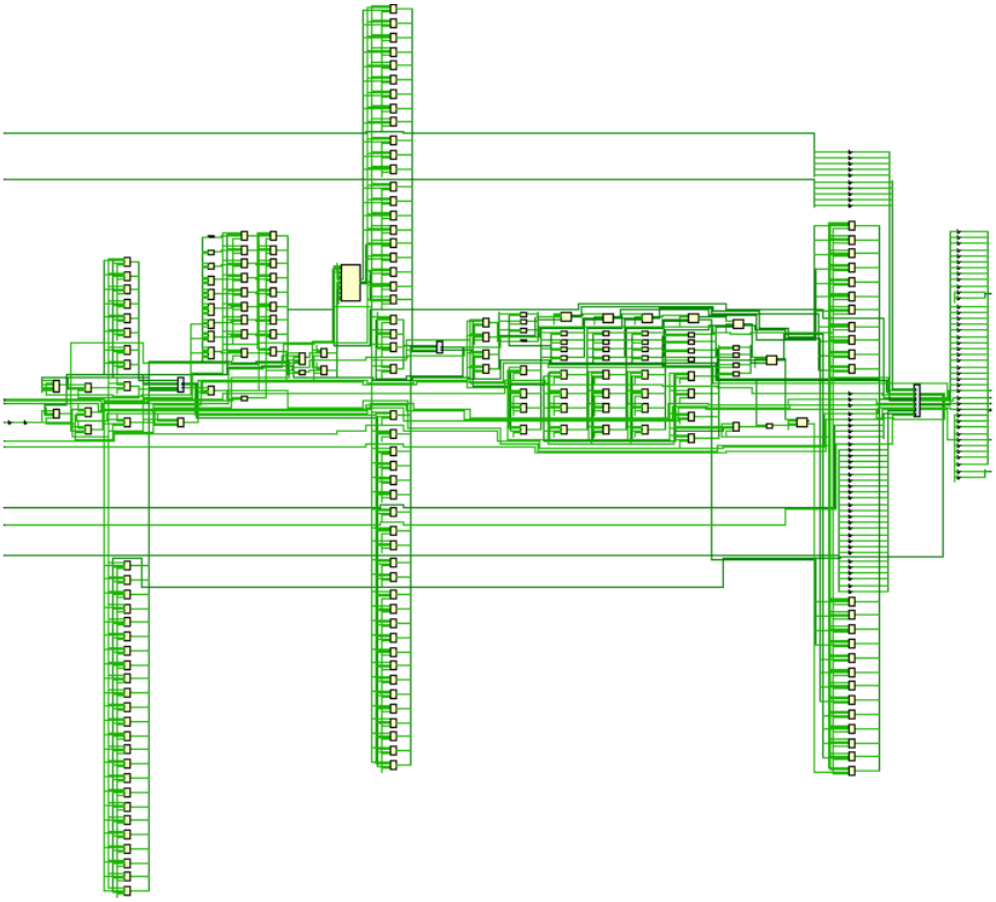

Ilustración 22 – Esquemático de la síntesis del filtro FIR básico con la interfaz AXI Lite

5.6.3 Diseño de filtro FIR acelerado

Una vez que el modelo básico ha sido diseñado, sintetizado, testeado e implementado en la placa de desarrollo, es momento de atacar a las capacidades de aceleración del mismo. La aceleración se centrará en la paralelización (*pipelining*) del bucle *for* principal del programa, además de intentar "*desenrrollar*" al máximo dicho bucle. Aparte de estos dos métodos, se tiene planteado intentar dividir el búffer principal para que sean registros individuales, con el fin de que cada uno de ellos sea accesible a la vez.

Se comenzará el desarrollo de esta parte mediante las pruebas de *unroll* del bucle principal del programa, mediante una directriz específica, que en el caso de no especificar datos concretos de un factor que determine la divisón según la cual se realizará el *unroll*. En este caso, se dejará sin especificar ya que el objetivo es que se llegue a la máxima paralelización.

```
set_directive_unroll "fir/Shift_Accum_Loop"
```

Con esta nueva implementación se realizará la síntesis para comparar los datos con los obtenidos en el apartado anterior.

Latencia		Intervalos		Timing (ns)
min	**max**	**min**	**max**	**-**
199	199	200	200	10,4

Tabla 13 – Resultados de temporización del diseño acelerado 1

Name	BRAM_18K	DSP48E	FF	LUT
Total	2	201	7833	3569
Utilization (%)	**~0**	**91**	**7**	**6**

Tabla 14 – Resultados de recursos del diseño acelerado 1

Se puede ver en la Tabla 13 como la mejora ha sido sustancial en los ciclos de reloj de funcionamiento del sistema, a costa de un gran incremento en el número de células DSP que se han usado, y ligeramente el número de Flip-Flops y LUTs utilizados.

Es importante destacar que las directrices se pueden aplicar a las diferentes entidades del programa según las etiquetas que admite definir C++, lo cual puede suponer una gran ayuda si se quiere atacar a un diseño mucho más complejo, para poder tener una referencia directa de qué hace cada uno.

Es curioso analizar el límite actual de latencia, el cual se sitúa justo en los 199 ciclos que tiene que realizar el bucle principal, por lo que se puede discernir, claramente, que lo más probable es que sea debido al acceso que tiene que realizar al búffer donde se almacenan los datos que van entrando desde el puerto de audio.

Para conseguir este objetivo, se deberá permitir el acceso múltiple al búffer de datos, por lo que se usará la directriz de *Array Partition*, la cual consiste en dividir un búffer grande en registros individuales, los cuales permiten que se pueda acceder a todos ellos para escribir a la vez.

```
set_directive_array_partition -type complete -dim 1 "fir" shift_reg
```

Con el tipo *complete* le indicamos al programa que particione todo el búffer además de indicarle que lo haga en su primera (y única) dimensión.

Los resultados obtenidos con este añadido son los siguientes:

Latencia		Intervalos		Timing (ns)
min	**max**	**min**	**max**	**-**
100	100	101	101	10,4

Tabla 15 - Resultados de temporización del diseño acelerado 2

Name	BRAM_18K	DSP48E	FF	LUT
Total	2	201	7902	969
Utilization (%)	**~0**	**91**	**7**	**1**

Tabla 16 - Resultados de recursos del diseño acelerado 2

Es importante analizar los datos y percartarse de que ante un aumento casi mínimo del número de Flip-Flops, el número total de LUTs se ha reducido considerablemente, y la velocidad ha incremente casi el doble pero todavía sigue siendo demasaido alta.

La pregunta ahora es: ¿por qué sigue siendo la latencia de 100 ciclos de reloj? Bien, para conocer la respuesta la opción más accesible es utilizar una herramienta presente en Vivado HLS, el *Analyzer*, que permite ver qué sucede en cada ciclo de reloj, y así discernir dónde está el *cuello de botella* de nuestro sistema.

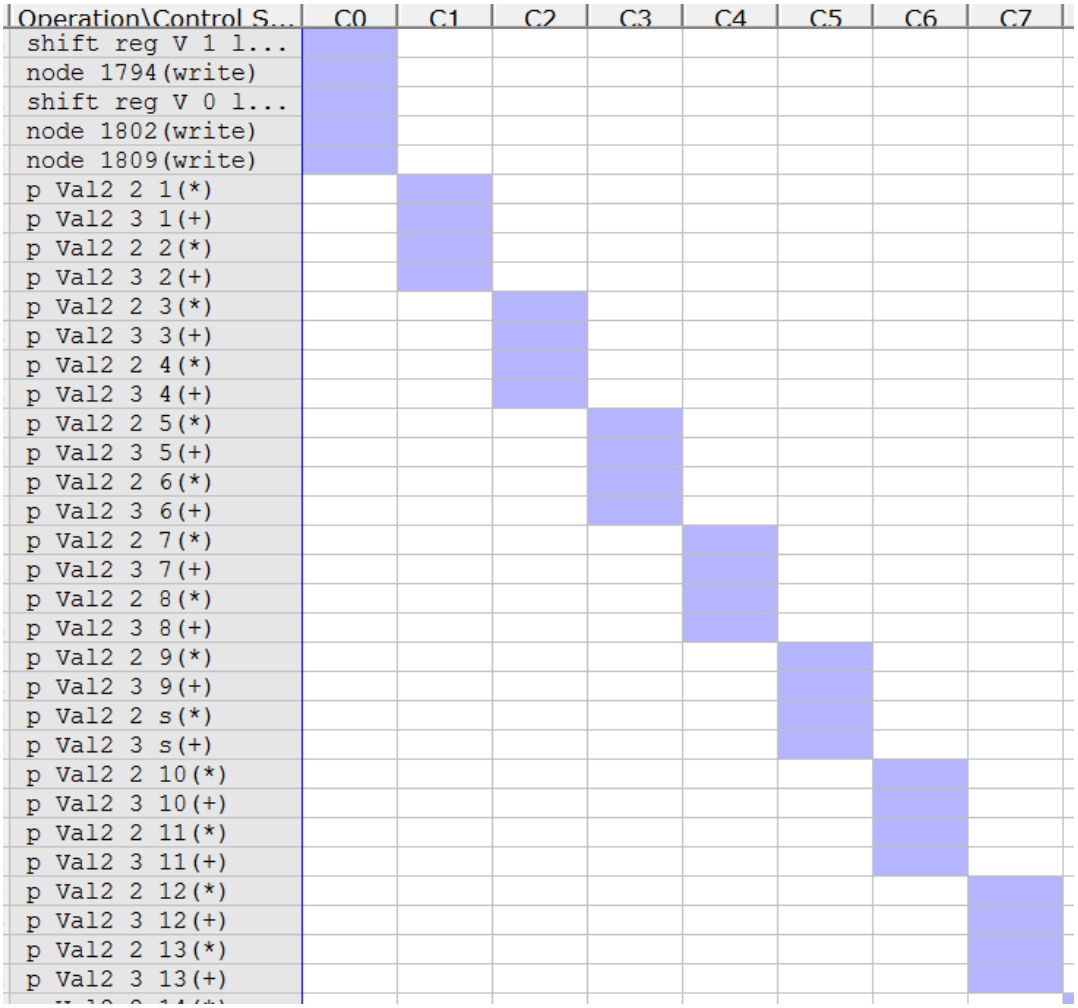

Ilustración 23 – Captura de la herramienta Analyzer de Vivado HLS

En la imagen superior se puede apreciar como el sistema queda ralentizado por tener que esperar a que se complete una operación de multiplicación, de suma, otra multiplicación y otra suma respectivamente. La herramienta te permite ir a la parte del código que genera estas instrucciones, la cual es la siguiente:

```
acc+= data*c[i];
```

Por lo tanto, es el acumulador quien está ralentizado todo el sistema, así pues, intentaremos usar directrices para que Vivado HLS comprenda que este es el punto crítico el cual queremos mejorar.

Comenzaremos aplicando una directriz de PIPELINE:

```
set_directive_pipeline -II 1 "fir"
```

Con esto se marca que queremos que toda la función sea paralelizado mediante *pipelining*, y además, decimos al programa que el II (Initiation Interval) será de 1 ciclo, es decir, que cada ciclo de trabajo habrá una nueva muestra de dato disponible, con el objetivo de que el programa intente optimizar todos los recursos y las distintas etapas para adaptarse a ese intervalo de trabajo.

Los resultados quedan reflejados en las siguientes tablas:

Latencia		Intervalos		Timing (ns)
min	**max**	**min**	**max**	**-**
100	100	1	1	9,4

Tabla 17 – Resultados de temporización del diseño acelerado 3

Name	BRAM_18K	DSP48E	FF	LUT
Total	2	201	12604	10393
Utilization (%)	**~0**	**91**	**11**	**19**

Tabla 18 – Resultados de recursos del diseño acelerado 3

Por desgracia, los resultados no son tan esperanzadores como podrían ser, sigue existiendo el mismo cuello de botella en cuanto a latencia, por lo que de nuevo utilizaremos el *Analyzer*.

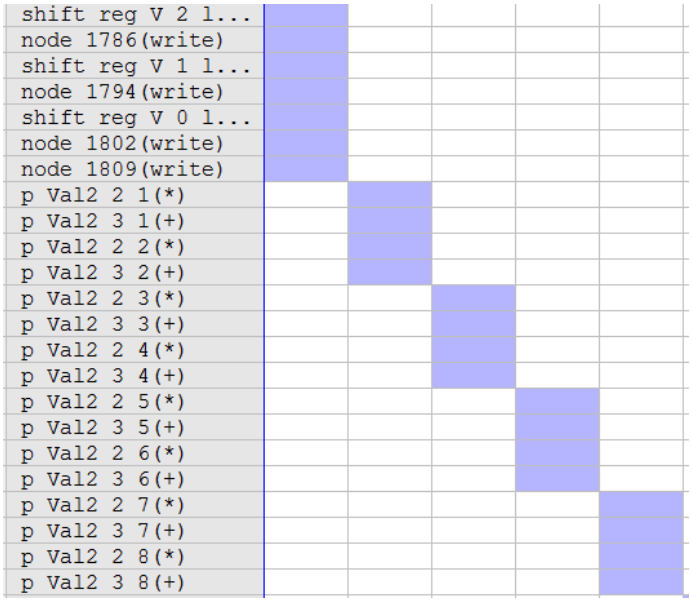

Ilustración 24 – Captura Analyzer diseño Acelerado 3

Se puede apreciar que tenemos la misma situación que con el diseño anterior. Por lo que se prueban todas las posibles directrices que afectan a la aceleración de un sistema, pero ninguna suerte efecto positivo. Esto ya no es un problema que se pueda solucionar añadiendo algunas directrices, es un problema de que el sintetizador no está comprendiendo bien qué hace nuestro código y por tanto no sabe cómo optimizarlo.

La solución que se adopta es la de dejar que las habilidades artísticas del co-diseño hardware-software salgan a relucir y llevar a cabo un nuevo modelo de implementación de filtro FIR sobre el que trabajar.

5.6.4 Nuevo camino: reconfiguración del bucle principal

Para mejorar la eficiencia del diseño se plantea la solución de generar un búffer de registros individuales para actuar como acumulador, con el fin de que la escritura en los mismos se pueda realizar en paralelo, mejorando las prestaciones del sistema en ese sentido.

La nueva implementación será la que queda reflejada en el diagrama de flujo de la Ilustración 25, estando su código fuente en el fichero de Anexos, y su función principal justo debajo de la imagen antes citada. Nótese que existen dos bucles anidados, con contadores *i* y *j*, los cuales tienen al

final el mismo resultado que el anterior global, pero en este se añade un tercer bucle para unificar los datos de los diferentes registros del acumulador en búffer.

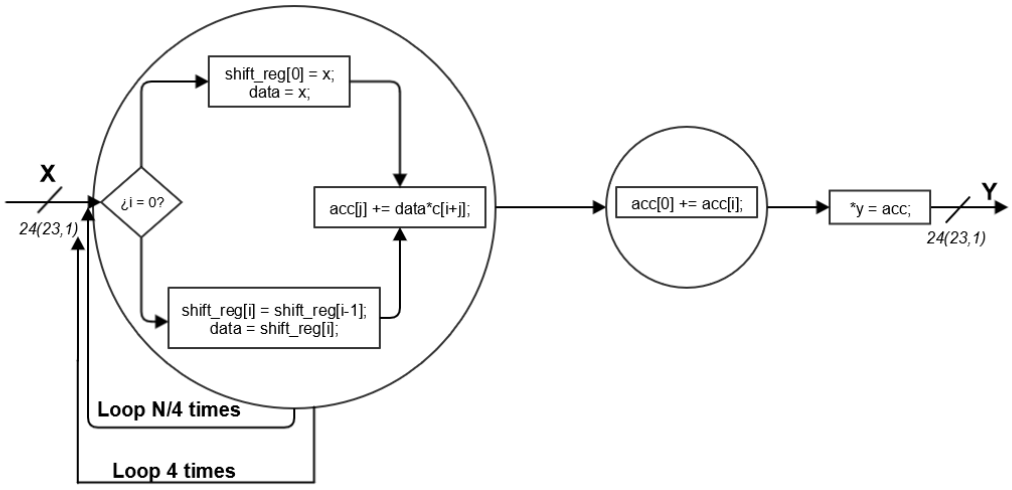

Ilustración 25 – Diagrama de flujo del filtro FIR en SW mejorado

```
Shift_Accum_Loop: for (i=(N-1)/4;i>=0;i-=4) {
                Sub_Accum_Loop: for (int j = 0; j < 4; j++){
                        if (i==0) {
                                shift_reg[0]=x;
                                data = x;
                        } else {
                                shift_reg[i]=shift_reg[i-1];
                                data = shift_reg[i];
                        }
                acc[j]+=data*c[i+j];;
                }
}
Final_Accum_Loop: for (i = 1; i < 4; i++){
        acc[0] += acc[i];
}
*y=acc[0];
```

Una vez tenemos el código, es necesario realizar la síntesis mediante Vivado HLS para comprobar si conlleva una mejora o no, respecto a la implementación básica con interfaz anterior.

Latencia		Intervalos		Timing (ns)
min	**max**	**min**	**max**	**-**
344	344	345	345	7,35

Tabla 19 – Resultados de temporización del filtro FIR mejorado 1

Name	BRAM_18K	DSP48E	FF	LUT
Total	2	2	484	826
Utilization (%)	**~0**	**~0**	**~0**	**1**

Tabla 20 - Resultados en recursos del filtro FIR mejorado 1

Evidentemente los resultados son suficientemente satisfactorios como para seguir mejorando por esta línea. Dado que el diseño está sin paralelizar, lo primero que se probará es a introducir una directriz de *pipelining* para toda la función.

```
set_directive_pipeline -II 1 "fir_improved"
```

Y su correspondiente resultado:

Latencia		Intervalos		Timing (ns)
min	**max**	**min**	**max**	**-**
15	15	13	13	10,08

Tabla 21 – Resultados de temporización del filtro FIR mejorado 2

Name	BRAM_18K	DSP48E	FF	LUT
Total	2	51	1476	513
Utilization (%)	**~0**	**23**	**1**	**~0**

Tabla 22 – Resultados en recursos del filtro FIR mejorado 2

De nuevo la mejora es más que evidente, aunque es en el siguiente paso donde de nuevo nos encontramos ante un cuello de botella, muy parecido al del anterior apartado, por lo que se decide tomar la decisión de probar lo mismo: si diviendo entre 4 el acumulador ha mejorado, entre 8 podría mejorar aún más.

Así pues, se vuelve a generar un código en el cual se cambian los bucles para adaptarse a este incremento y se dota al búffer del acumulador de 4 unidades más. Se realiza la síntesis y se obteinen los resultados de las Tabla 23 y Tabla 24.

Latencia		Intervalos		Timing (ns)
min	**max**	**min**	**max**	**-**
6	6	4	4	7,82

Tabla 23 – Resultados de temporización del filtro FIR mejorado 3

Name	BRAM_18K	DSP48E	FF	LUT
Total	2	31	730	356
Utilization (%)	**~0**	**14**	**~0**	**~0**

Tabla 24 – Resultados en recursos del filtro FIR mejorado 3

Aún sigue habiendo margen de mejora, aunque cada vez menos, ya que evidentemente, el límite no estará en 1 ciclo de reloj, debido a que se necesita realizar el acceso a un interfaz para lectura al inicio e ir escribiendo en los acumuladores en dos momentos diferentes. El acceso a la intefaz AXI Lite para lectura ocupa 2 ciclos de reloj, y como mínimo deberá haber 1 ciclo de reloj de

procesamiento para guardar los datos en los registros de los acumuladores intermedios y otro ciclo para juntarlos todos, y por último, otro ciclo para escribir los datos en la interfaz AXI Lite. Por lo tanto el mínimo teórico se sitúa en los 5 ciclos (en los resultados corresponde con 4, porque cuenta el 0 como el inicio).

Si la primera vez ha funcionado, y la segunda también, ¿por qué no intentarlo una tercera? Esta vez, probaremos a realizar la división entre 10, intetando así ganar esos 2 ciclos que nos separan del límite teórico.

Se modifica el fichero, se crea una nueva solución para realizar la síntesis y se extraen los resultados:

Latencia		Intervalos		Timing (ns)
min	**max**	**min**	**max**	**-**
4	4	2	2	9,96

Tabla 25 – Resultados de temporización del filtro FIR mejorado 4

Name	BRAM_18K	DSP48E	FF	LUT
Total	2	19	422	314
Utilization (%)	**~0**	**8**	**~0**	**~0**

Tabla 26 – Resultados en recursos del filtro FIR mejorado 4

Objetivo conseguido. Se ha logrado acelerar hasta el máximo que se podía esperar con esta implementación.

Por último se prueba cambiar el interfaz por AXI4 Master, para comprobar si afecta en algo a la velocidad en este diseño, se cambian las directrices del AXI Lite por las siguientes y se sintetiza el diseño:

```
set_directive_interface -mode m_axi "fir" x
set_directive_interface -mode m_axi "fir" y
```

Se comprueba que no hay diferencias de tiempos y sí un incremento de recursos, por lo tanto se deja el AXI Lite, el cual consume menos recursos.

5.6.5 Resultados y comparativa

En este apartado se incluyen tablas que comprenden los resultados de las síntesis de los diferentes diseños comentados con anterioridad.

Implementación	Latencia		Throughput		Timing (ns)
-	**min**	**max**	**min**	**max**	**-**
Básico	1401	1402	1401	1402	6,08
Interfaz	1401	1402	1401	1402	6,08
Acelerado 1	199	199	200	200	10,40
Acelerado 2	100	100	101	101	10,40
Acelerado 3	100	100	1	1	9,40

Mejorado 1	344	344	345	345	7,35
Mejorado 2	15	15	13	13	10,08
Mejorado 3	6	6	4	4	7,82
Definitivo	**4**	**4**	**2**	**2**	**9,96**

Tabla 27 – Comparativa temporización de las diferentes implementaciones

Implementación	BRAM_18K	DSP48E	FF	LUT
Básico	2	2	180	175
Inferfaz	2	2	300	311
Acelerado 1	2	201	7833	3569
Acelerado 2	2	201	7902	969
Acelerado 3	2	201	12604	10393
Mejorado 1	2	2	484	826
Mejorado 2	2	51	1476	513
Mejorado 3	2	31	730	356
Definitivo	**2**	**19**	**422**	**314**

Tabla 28 – Comparativa recursos de las diferentes implementaciones

A raíz de los resultados obtenidos, se puede asegurar que los objetivos del proyecto se han cumplido en mayor manera de lo esperado. Sin duda son unos resultados interesantes que deberían de ser estudiados en profundidad para comprender cómo cambios tan pequeños en una directriz o en la reconfiguración de un bucle, pueden tener un impacto tan grande a la hora de que Vivado HLS interprete el diseño a implementar.

Es curioso analizar como no siempre una mejora en los ciclos de reloj necesarios para el sistema conlleva un aumento de recursos, esto es debido a que las células DSP que lleva incorporadas el SoC ZynQ son ciertamente potentes, pero Vivado HLS no tiene porqué utilizarlas en su totalidad, y eso es algo que no se puede controlar directamente, por ello, un remodelo del código o nuevas directrices pueden conllevar tan grandes cambios.

En términos de recursos, se ha conseguido que el diseño sea lo suficientemente reducido, dado el gran número de coeficientes utilizados, como para que sea plausible plantearse tener un sistema de gran alcance como una mesa de mezclas implementada mediante un SoC del estilo del ZynQ.

Caso especial sería el de las implementaciones "Mejorado 2", "Acelerado 1" y "Acelerado 2", estas 3 tienen valores de *timing* superiores a lo esperado de 10 ns, lo cual supondría que de ser las que finalmente se quisiera implementar, habría que realizar un ajuste fino a esos valores, siendo en el primer caso presumiblemente sencillo, y en los otros dos posiblemente más complejo lograrlo.

Con el fin de que se puedan apreciar los resultados de manera más gráfica y visual, se incluyen las siguientes imágenes, con la misma información que las tablas Tabla 27 y Tabla 28.

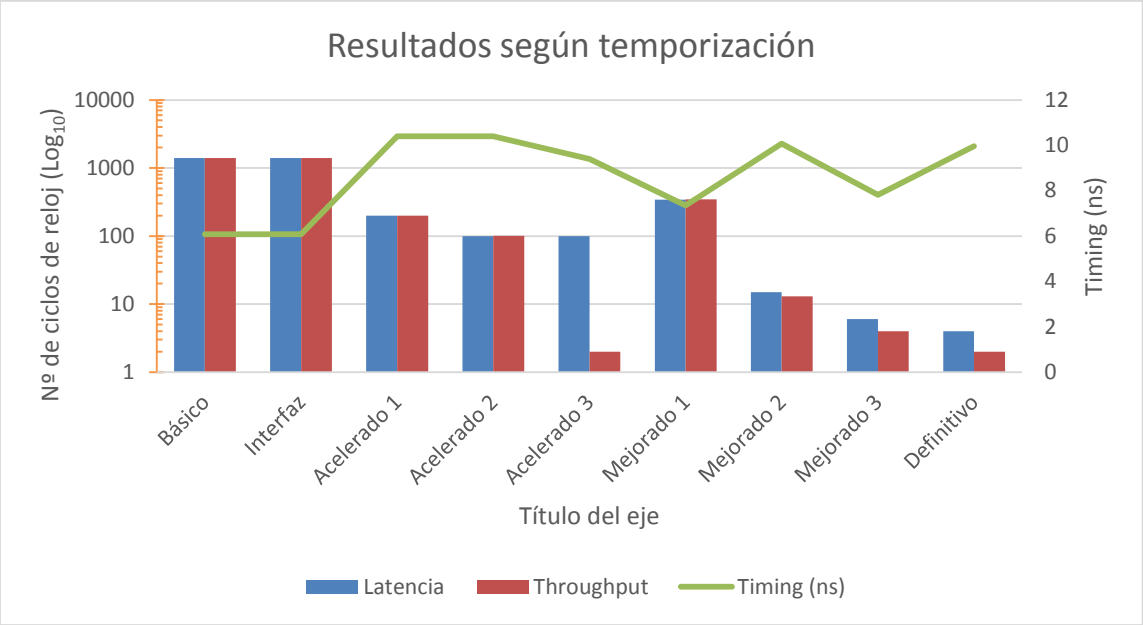

Ilustración 26 – Gráfico resultados temporización

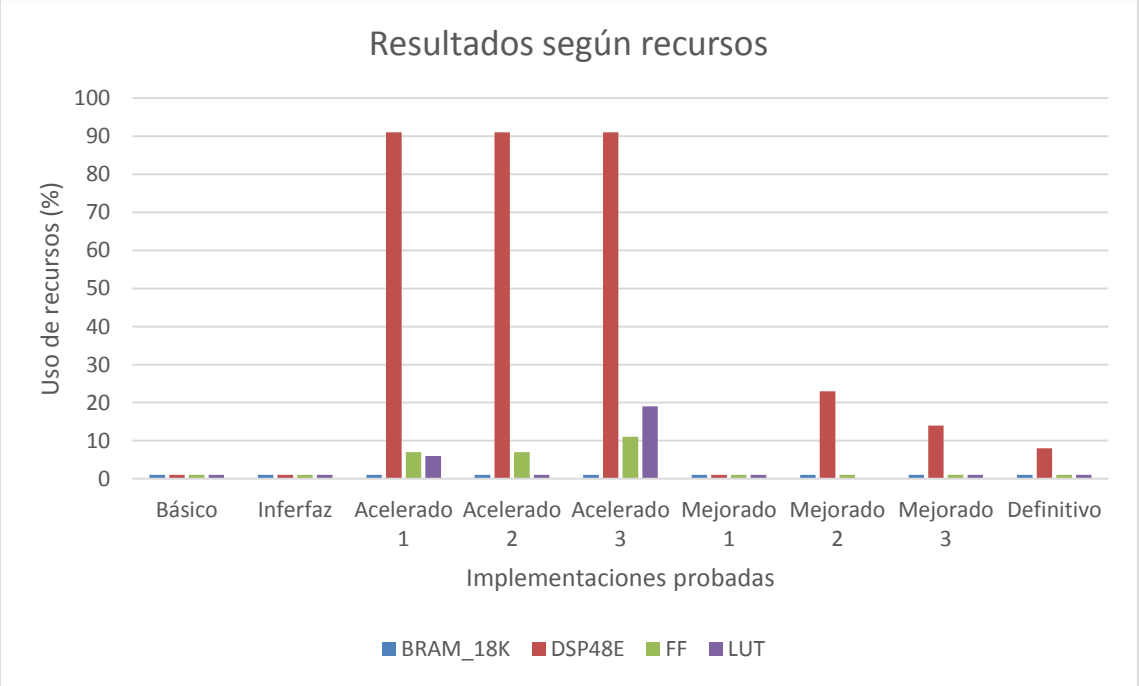

Ilustración 27 – Gráfico resultados recursos

5.7 Implementación final en ZynQ

Se parte del proyecto generado para el Test de Audio en la sección 5.5, pero sustituyendo el bloque IP del NCO por el filtro FIR generado mediante Vivado HLS. Además, se han eliminado la entrada GPIO de los switches y los LEDs.

Dado que el driver generado para el control del ADAU1761 divide las muestras de audio en canal izquierdo y canal derecho, se ha realizado el diseñado con un filtro individual para cada canal, introduciendo en ellos las muestras de audio de entrada y obteniendo las de salida.

No es necesario realizar ningún cambio a nivel del sistema en Vivado 2015, más allá de lo comentado anteriormente. En cambio, en la parte software desde el Vivado SDK será necesario generar una función de inicialización de los módulos FIR e incluir las funciones de entrada y salida de datos de los mismos, las cuales se encontrarán incluidas en las librerías del proyecto, provenientes del bloque IP adjuntado al Vivado 2015.

Capítulo 6. Pliego de condiciones

6.1 Esquemas

6.1.1 Test de audio

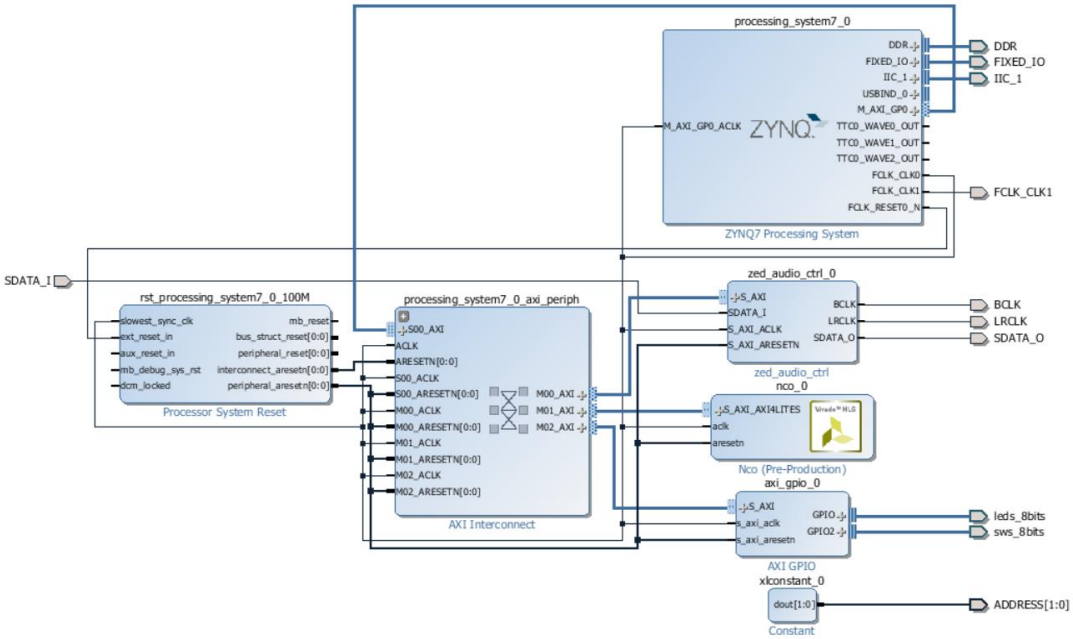

Ilustración 28 – Esquema Test de audio completo

6.1.2 Filtro FIR básico

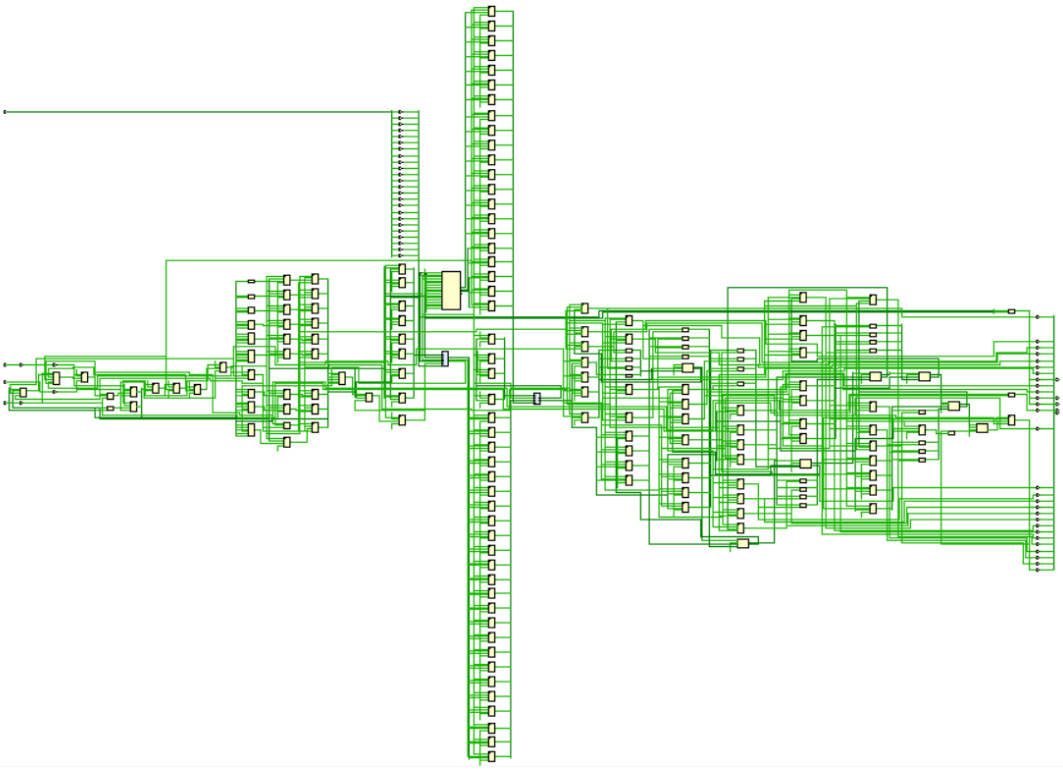

Ilustración 29 – Esquemático filtro FIR básico

6.1.3 Filtro FIR acelerado: nueva idea, implementación mejorada

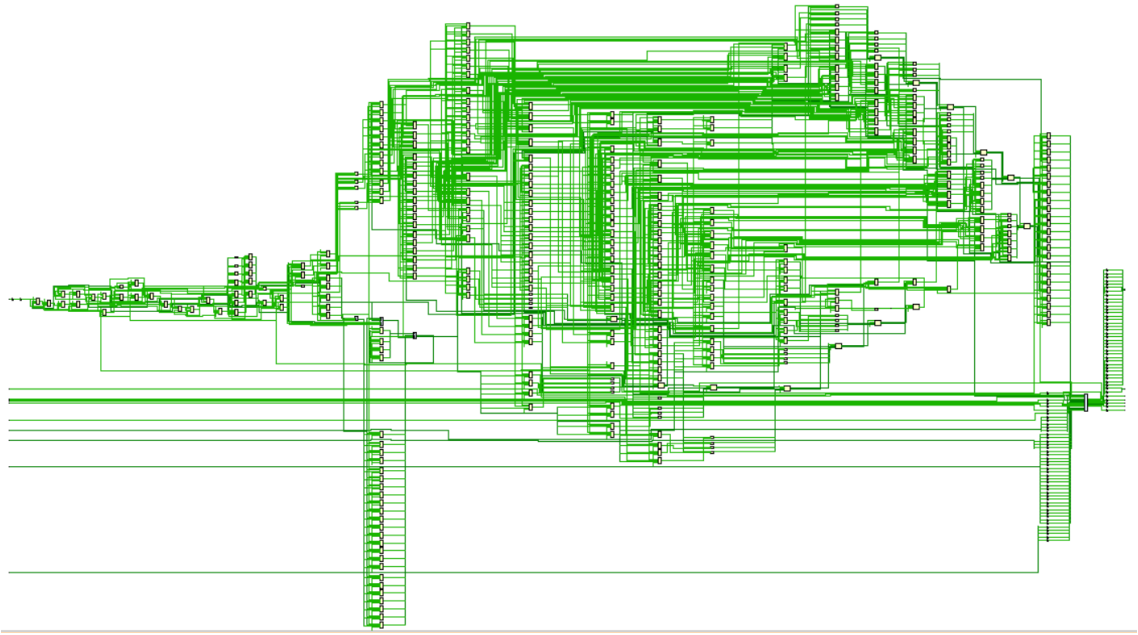

Ilustración 30 – Esquemático del filtro FIR con una implementación más eficiente

6.1.4 Filtro FIR acelerado definitivo

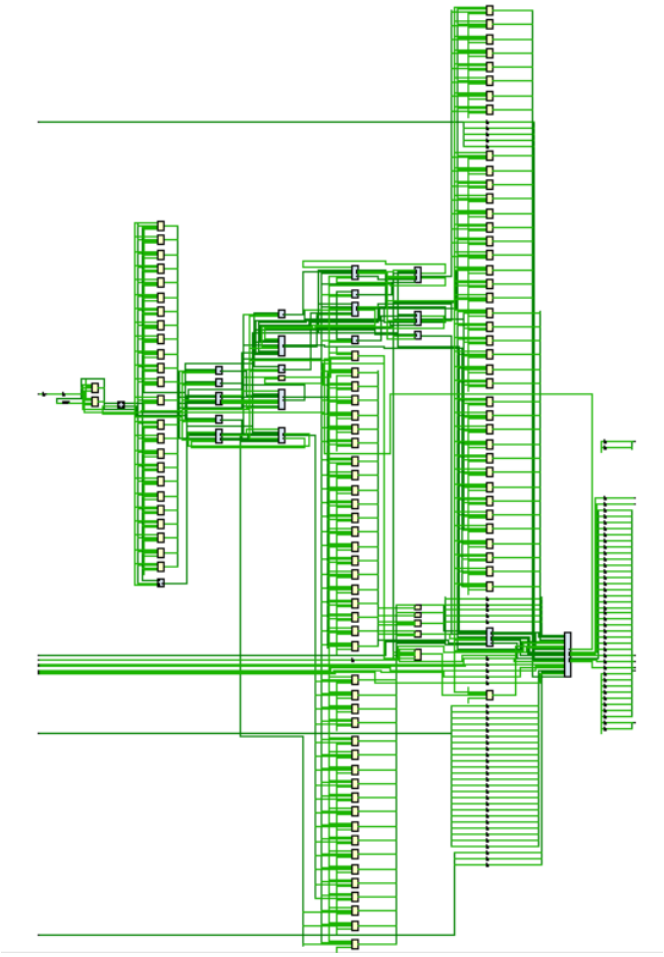

Ilustración 31 – Esquemático del filtro FIR definitivo

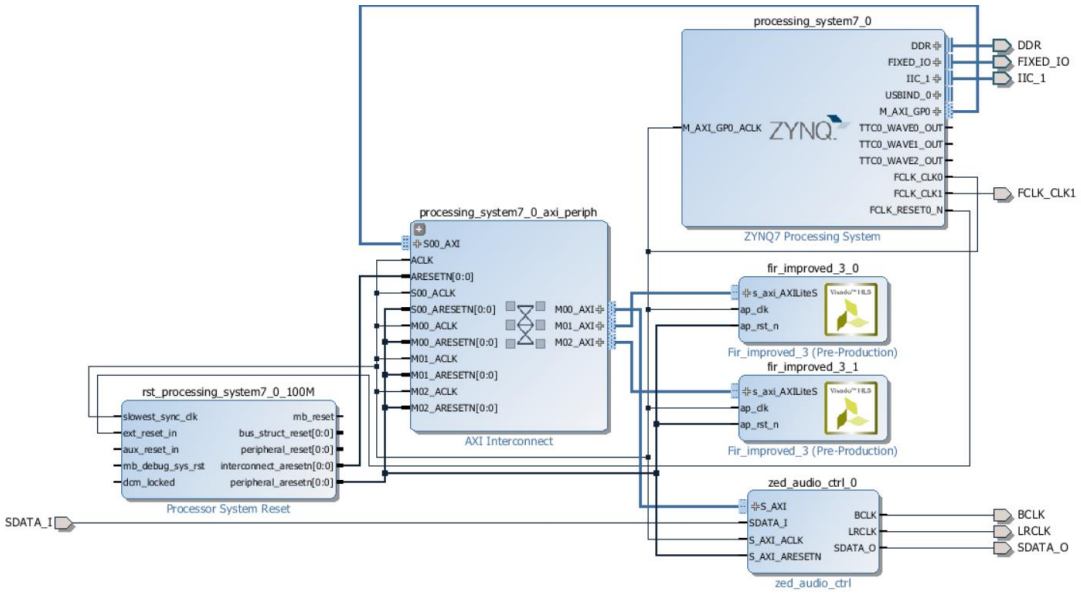

Ilustración 32 – Esquemático de la implementación final del sistema

6.2 Material

6.2.1 ZedBoard

La pieza fundamental para el desarrollo de este proyecto es la placa de desarrollo ZedBoard.

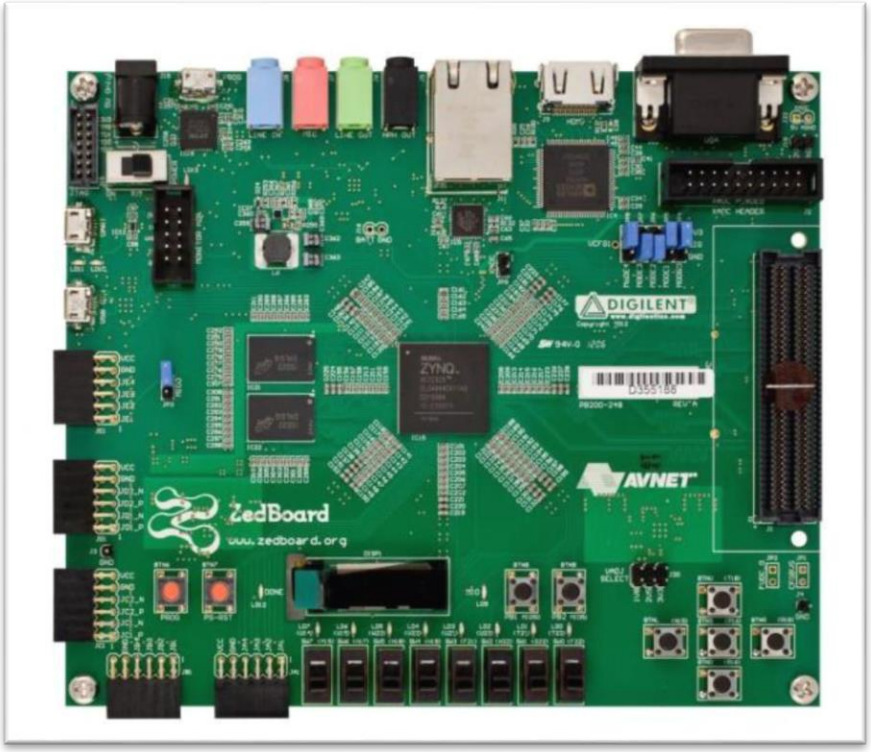

Ilustración 33 - Vista superior de ZedBoard

Sus especificaciones técnicas son las siguientes, recogidas desde la página web del distribuidor. (3)

- Processor
  - Zynq™-7020 AP SoC XC7Z020-CLG484-1
- Memory
  - 512 MB DDR3
  - 256 Mb Quad-SPI Flash
- Communication
  - Onboard USB-JTAG Programming
  - 10/100/1000 Ethernet
  - USB OTG 2.0 and USB-UART
- Expansion connectors
  - FMC-LPC connector (68 single-ended or 34 differential I/Os)
  - 5 Pmod™ compatible headers (2x6)
  - Agile Mixed Signaling (AMS) header
- Clocking
  - 33.33333 MHz clock source for PS
  - 100 MHz oscillator for PL
- Display
  - HDMI output supporting 1080p60 with 16-bit, YCbCr, 4:2:2 mode color
  - VGA output (12-bit resolution color)
  - 128x32 OLED display
- Configuration and Debug
  - Onboard USB-JTAG interface
  - Xilinx Platform Cable JTAG connector
- General Purpose I/O
  - 8 user LEDs
  - 7 Push buttons
  - 8 DIP switches

La parte más importante, el núcleo de esta plataforma de desarrollo es el SoC **ZynQ**, el cual se detallará a continuación.

6.2.2 ZynQ

Este dispositivo incluye en el mismo chip un procesador ARM Cortex-A9 de deoble núcleo (revisión r3p0, basado en ARMv7-A) y una FPGA Artix, los cuales están comunicados mediante el estándar AXI. Este tipo de chips está orientado a utilizar la FPGA como un acelerador hardware de la parte del procesador ARM.

Recordemos, que **PS** hace referencia a la parte "software" del SoC, y **PL** a la parte referida a la FPGA.

La parte PS se encarga de ejecutar software, y dado el incremento continuo en la capacidad de proceso de los dispositivos ARM, este dispositivo está orientado al uso de sistemas operativos para la parte software. Además de los dos ARM, cuenta con diferentes recursos como una APU (*Application Processing Unit*), interfaces de periféricos, memoria caché, interfaces de memoria, interconexiones y circuitos generadores de reloj.

Las carecterísticas más importantes de la parte PL se pueden apreciar en la siguiente tabla, sacada de la página web de Xilinx Inc. (4)

Características	ZynQ-7020
Células Lógicas	85000
Bloques RAM (Mb)	4.9
Bloques DSP	220
Pines I/O máximos	100

Tabla 29 - Características de ZynQ

Ante la cantidad de herramientas y de trabajos disponibles sobre esta plataforma, se puede plantear el desarrollo del trabajo desde la perspectiva de HLS (*High Level Synthesis*) en lugar desde un planteamineto RTL (*Register Transfer Level*).

La placa de desarrollo incluye, además, entradas y salidas de audio, gestionadas por el chip **ADAU1761** de Analog Devices. Se entiende que el funcionamiento de este integrado es fundamental para el correcto desarrollo del presente proyecto, por lo que se le dedicará una entrada explica en exclusiva.

6.2.3 ADAU1761

El chip de audio (codificador-decodificador) se puede configurar a través de los buses I2C (B-I2C) o SPI, indistintamente, pero los datos solo se pueden manejar mediante I2C (C-I2C), comunicándose con los ADC o con los DAC, ya que cuenta con 2 de cada uno de 24 bits de palabra, con un rango de muestreo de 8 kHz a 96 kHz, llegando hasta un máximo posible de 98 dB de SNR. Ambos pueden trabajar tanto con un overshampling de 64x como 128x, configurable desde los registros del chip.

Para su correcto funcionamiento requiere de una señal de reloj propia de 10 MHz que vaya al PLL, la cual se puede configurar directamente desde la parte PS del SoC ZynQ.

El esquema del chip se puede apreciar en la Ilustración 34.

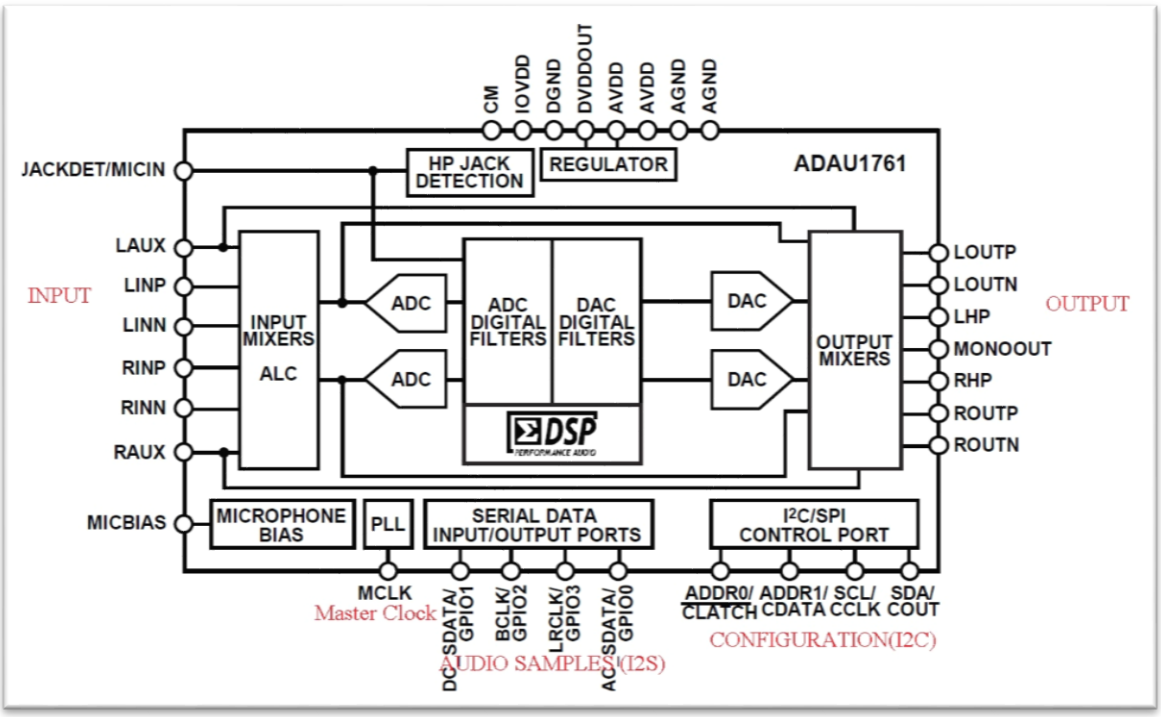

Ilustración 34 - Esquemático chip ADAU1761

Para la configuración del mismo se dispone de 67 registros, accesibles desde el puerto de control por I2C o SPI, y mapeados desde la dirección 0x4000 hasta la 0x40FA. Los registros tienen un tamaño de palabra de 8 bits, aunque algunos tienen bits reservados internamente.

Tiene su propia RAM interna, dividida en RAM de parámetros, RAM de programa y RAM de datos.

Los interfaces que tiene disponibles el chip ADAU1761 en la placa ZedBoard son los siguientes:

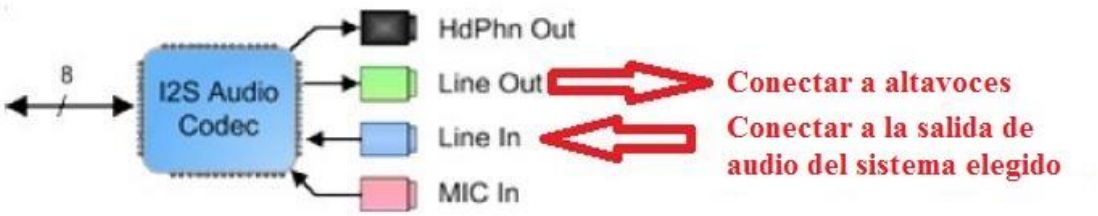

Ilustración 35 – Interfaces audio ADAU1761

Y el conector que se usará a lo largo del proyecto, será uno estándar de 2 canales estéreo:

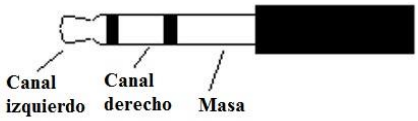

Ilustración 36 – Conector audio estéreo

Para finalizar la información necesaria de conocer sobre el ZynQ, es propicio conocer detalladamente la célula DSP que incluye, el modelo **DSP48E1**.

6.2.4 DSP48E1

Como se ha comentado en el párrafo anterior, esta es la célula DSP que se incluye en el SoC ZynQ, el cual posee en su interior 220 de estas células, las cuales están pensadas para ser de uso dedicado, full-custom, de bajo consumo y cuyo resultado combina perfectamente una alta velocidad de procesamiento con un tamaño reducido, a la vez que mantiene la cualidad de hacer el sistema flexible a diferentes aplicaciones.

Su uso es idóneo para otras aplicaciones aparte del audio digital, como pueden ser registros de desplazamiento dinámicos, multiplexadores con un bus de entrada/salida amplio y registros de entrada/salida mapeados en memoria. (14)

Su funcionalidad básica puede apreciarse en la Ilustración 37.

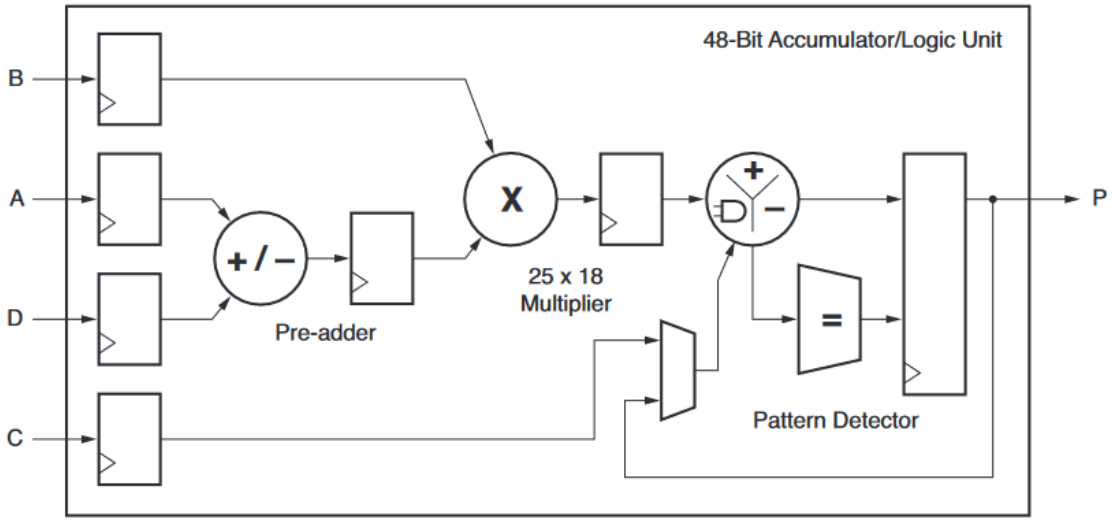

Ilustración 37 – Funcionalidad básica del DSP48E1

Un buen resumen de lo que se incluye, sobretodo lo útil para el proyecto actual sería el siguiente:

- Multiplicador de complemento a 2 de 25x18 bits. Incluye bypass dinámico.
- Acumulador de 48 bits. Puede usarse como un contador síncrono.
- Pre-sumador colocado antes del multiplicador. Útil para optimizar aplicaciones de filtrado simétrico, reduciendo los requerimientos de la célula DSP.
- Unidad aritmética SIMD (Single-Instruction-Multiple-Data). Puede funcionar como sumador, restador o acumulador, en dos modos dual de 24 bits o cuádruple de 12 bits.
- Unidad lógica que puede generar cualquier de las 10 funciones lógicas diferentes de 2 operandos.
- Detector de patrón. Convergente o redondeador simétrico. Puede llegar a implementaciones de hasta 96 bits en conjunto con la unidad lógica.
- Tiene buses opcionales para implementaciones en cascada o *pipelining*.

6.3 Software de diseño

6.3.1 Vivado HLS (High-Level Synthesis)

Vivado HLS es una herramienta de síntesis HDL basada en descripción de algoritmos en lenguaje C y C++, que facilita la exportación de los módulos o sistemas generados para integrarlos en la FPGA.

Es especialmente útil para explorar diferentes implementaciones sobre la FPGA, en cuanto a optimización de recursos. Permite que una vez la estructura del algoritmo haya sido definida, ensayar diferentes implementaciones desde el código fuente, sin necesidad de implementar el diseño en la FPGA, lo cual ahorra tiempos muy largos de compilación. Por lo tanto es posible ensayar diferentes soluciones para un algoritmo dado, de forma que se optimicen los requerimientos de velocidad de proceso, capacidad de cálculo y los recursos disponibles en la FPGA.

6.3.2 Xilinx SDK (Software Development Kit)

Herramineta básica para el desarrollo y la implementación del código fuente en los ARM.

Es un entorno de desarrollo de software embebido para los procesadores embebidos de Xilinx. Se basa en la plataforma de código abierto Eclipse, siendo sus principales características las siguientes:

- Entorno con soporte de edición y compilación de C/C++.
- Gestión de proyectos.

- Configuración del "build" específico de cada aplicación, así como la generación automática del "Makefile".
- Visualización de errores.
- Entorno integrado para debugueo a nivel de código fuente con diferenciación según el sistema embebido objetivo.
- Control de versiones del código fuente.
- Soporta los juegos de instrudciones de 16 y 32 bits de ARM, así como los de 8 bits de "Java byte codes".

Además de las características anteriores, que provienen de la implementación de Eclipse, el SDK de Xilinx ofrece además las siguientes herramientas para el desarrollo de software para sistemas embebidos:

- *Xilinx System Debugger* (XSDB): Interfaz basada en línea de comandos para los sistemas Xilinx. Permite diversas caracterísiticas de debugueo a bajo nivel.
- FPGA programmer: Permite programa la FPGA de Xilinx con un fichero de bitstream seleccionado.
- Flash programmer: Se usa para grabar bitstreams y aplicaciones software en dispositivos externos Flash.
- Simple bootloader generator: Permite arrancar directamente un sistema embebido con una Flash conectada en paralelo.
- Linker script generator: Gestiona de forma sencilla en mapeo de imágenes de aplicaciones en el espacio de memoria del hardware.

6.3.3 Vivado 2015 Design Suite

Vivado permite a los desarrolladores sintetizar sus diseños, realizar análisis temporales, examinar diagramas RTL, simular las reacciones de un diseño ante diferentes estímulos, y poder configurar el sistema objetivo de diseño.

Es un entorno de desarrollo integrado o IDE (*Integrated Design Environment*) orientado a las diferentes FPGAs de Xilinx, y está ligado a las diferentes arquitecturas de dichos chips. No acepta trabajar con FPGAs de otras marcas.

Incluye herramientas de diseño de sistema a nivel electrónico, orientadas a la síntesis y verificación de algoritmos basados en C, IP externas y/o propias, o bien diseños propios.

6.4 Presupuesto

Se ha realizado una estimación del presupuesto que sería computable a la realización del presente proyecto en una empresa privada que estuviese interesada en una investigación en este ámbito.

El presupuesto se ha dividido en varios apartados, pero de todos ellos, el que mayor peso representa en el cómputo global es el coste asociado a la mano de obra del estudiante que ha desarrollado el Trabajo Fin de Máster, según la planificación real, que se puede ver en Tabla 3.

	Nº Horas	**Coste / h**	**Total**
Mano de obra	752	18 €	**15.040 €**

Tabla 30 – Coste mano de obra

A continuación se tendrá en cuenta la amortización de los bienes utilizados a lo largo del proyecto que correspondan.

Amortizables	Coste	Años amortizables	Tiempo de uso (meses)	Coste amortizable
Mesa de escritorio	150 €	15	6	5 €
Ordenador de trabajo	900 €	5	6	90 €
Silla de escritorio	60 €	8	6	4 €
Total				**99 €**

Tabla 31 – Presupuesto de bienes amortizables

Por último, se tendrá también en cuenta el material que se ha tenido que adquirir directamente para el proyecto.

Material	Coste
Placa de evaluación ZedBoard (Incluye licencia del software)	400 €
Cable doble-Jack	10 €
Total	**410 €**

Tabla 32 – Presupuesto material

Con los datos expuestos en las tablas anteriores, se ha realizado un cómputo global, añadiendo para el mismo un 10% extra de coste para gastos indirectos (luz, agua, etc…) y un 5% para gastos imprevistos.

Presupuesto TFM	Coste
Mano de obra	15.040 €
Amortizables	99 €
Material adquirido	410 €
Acumulado	**15.549 €**
Gastos indirectos	1.555 €
Gastos imprevistos	777 €
Total	**16.326 €**

Tabla 33 – Presupuesto total TFM

Capítulo 7. Conclusiones y propuesta de trabajo futuro

7.1 Conclusiones

A lo largo del presente texto se ha expuesto la idea original del proyecto, el proceso de desarrollo a la que la sometió, su evolución y el resultado final que se consiguió. Cada una de las partes en las que se dividió el trabajo fue resuelta satisfactoriamente, pudiendo incluso agregar nuevas ideas para mejorar los resultados obtenidos.

Este proyecto se enfocó como un estudio de las posiblidades que ofrece un sistema novedoso como es el ZynQ, con el fin de que la metodológia utilizada y aprendida pueda ser extrapolada a cualquier tipo de proyecto que necesite de la aceleración de hardware o el co-diseño hardware-software en sistema de este tipo.

Cuando se comenzó el desarrollo no se tenían conocimientos referentes al conjunto de programas de Xilinx Inc., tampoco se había trabajado nunca con alguno de sus sistemas, y apenas se tenían conocimientos sobre la aceleración de hardware. Dado el desarrollo y los resultados, se puede concluir que el proyecto ha sido satisfactorio tanto por dichos resultados como por el proceso de aprendizaje y el conocimiento adquirido. Sin duda, una de la haiblidades que más favorecidas ha salido de todo el tiempo invertido en el proyecto es la de gestión de proyectos, así como la búsqueda y consolidación de la información necesaria para cubrir una necesidad específica.

7.1.1 Propuesta de implementación

Una posible aplicación para todo este estudio sería utilizar el SoC ZynQ, con todo el conocimiento adquirido a lo largo del presente proyecto, para implementar una mesa de mezclas a nivel comercial, debido a su gran potencial como FPGA y además con la ayuda de los dos microcontroladores.

El tema de los microcontroladores podría ser empleado dejando a uno corriendo un RTOS como FreeRTOS, ligero y fácil de usar, el cual llevaría tareas gráficas conectado a un PC para tener una interfaz asequible para el usuario. El otro microcontrolador se encargaría del control de la mesa de mezclas, tal como se ha visto en el presente proyecto hacer con bloques funcionales sencillos.

Sería intersante desarrollar esta propuesta y compararla con modelos comerciales ya presentes en el mercado.

7.1.2 Utilizar SDSoC de Xilinx Inc para profundizar en el apartado software

En el presente proyecto se ha abordado el tema del co-diseño hardware-software con el objetivo de la aceleración de sistemas de audio, pero centrándose en la parte hardware. Sería interesante poder comprobar qué puede llegar a ofrecer una herramienta que permitiese optimizar los algoritmos y el uso de instrucciones a nivel de software y emplear ambos conocimientos en conjunto para mejorar aún más futuros proyectos que se puedan plantear.

Capítulo 8. Bibliografía

1. **Altera Corporation.** Catálogo SoC. [En línea] [Citado el: 5 de julio de 2016.] https://www.altera.com/products/soc/portfolio.html.

2. **Xilinx Inc.** Catálogo SoC. [En línea] [Citado el: 5 de julio de 2016.] http://www.xilinx.com/products/silicon-devices/soc.html.

3. **AVNET.** Especificaciones ZedBoard. [En línea] [Citado el: 5 de julio de 2016.] http://zedboard.org/sites/default/files/product_briefs/PB-AES-Z7EV-7Z020_G-v12.pdf.

4. **Xilinx Inc.** Tabla de producto ZynQ. [En línea] [Citado el: 5 de julio de 2016.] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable.

5. **Wikipedia.** Nyquist–Shannon sampling theorem. [En línea] [Citado el: 5 de julio de 2016.] https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem.

6. **Elert, Glenn.** Frequency Range of Human Hearing. [En línea] [Citado el: 5 de julio de 2016.] http://hypertextbook.com/facts/2003/ChrisDAmbrose.shtml.

7. **Wikipedia.** Sampling (signal processing) - Audio. [En línea] [Citado el: 5 de julio de 2016.] https://en.wikipedia.org/wiki/Sampling_%28signal_processing%29#Audio.

8. **Xilinx Inc.** Field Programmable Gate Array (FPGA). [En línea] [Citado el: 5 de julio de 2016.] http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm.

9. **Wikipedia.** FIR Filters. [En línea] [Citado el: 5 de julio de 2016.] https://en.wikipedia.org/wiki/Finite_impulse_response.

10. —. Digital filter. [En línea] 5 de julio de 2016. https://en.wikipedia.org/wiki/Digital_filter.

11. **ULCM.** RUIDO DE CUANTIZACIÓN Y DITHER. [En línea] [Citado el: 5 de julio de 2016.] http://www.info-ab.uclm.es/labelec/Solar/Otros/Audio/html/ruido_cuant.html.

12. **The ZynQ Book.** Repositorio Github bloque IP NCO. [En línea] [Citado el: 5 de 7 de 2016.] https://github.com/RayHightower/zynq-book/tree/master/adventures_with_ip_integrator/ip/xilinx_com_hls_nco_1_0.

13. —. Repositorio Github bloque Zed Audio Control IP. [En línea] [Citado el: 5 de 7 de 2016.] https://github.com/RayHightower/zynq-book/tree/master/adventures_with_ip_integrator/ip/zed_audio_ctrl.

14. **Xilinx Inc.** User Guide DSP48E1. [En línea] [Citado el: 5 de julio de 2016.] http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.

15. **Crockett, Louise H, y otros.** *Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC.* Junio 2014.

16. **Analog Devices Inc.** Datasheet ADAU1761. [En línea] [Citado el: 5 de julio de 2016.] http://www.analog.com/media/en/technical-documentation/data-sheets/ADAU1761.pdf.

17. **MikroElektronika.** IIR Filters. [En línea] [Citado el: 5 de julio de 2016.] http://learn.mikroe.com/ebooks/digitalfilterdesign/chapter/introduction-iir-filter/.