ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería de Electricidad y Computación

“Diseño y prototipado de un sistema digital de filtración de voz humana en VHDL para la mejora de la comunicación verbal bidireccional interindividual y colectiva en personas que presentan discapacidad auditiva.”

PROYECTO INTEGRADOR

Previo la obtención del Título de:

Ingeniero en Electrónica y Automatización

Presentado por:
Alexis Xavier Mora Roca
Angel Eduardo Román Veloz

GUAYAQUIL - ECUADOR
Año: 2021

DEDICATORIA

Con todo el aprecio del mundo, dedico este proyecto a mis padres, Antonio Mora Alcívar y Gabriela Roca Rosales, por haber sido pilares en mi vida y siempre haberme apoyado a pesar de lo críticas que han llegado a ser las circunstancias. Dedico también este trabajo a mi abuelita, María Teresa Rosales, si no fuera por ella, jamás sería la persona en la que me he convertido actualmente. Finalmente dedico este pequeño gran resultado de perseverancia a mi pareja, Angie Quijije Díaz, por haber soportado mi mal humor en los días más pesados que confronté realizando este proyecto. Ustedes son y serán siempre lo más importante en mi vida.

Alexis Mora Roca

DEDICATORIA

Fervientemente dedico este proyecto en gran manera a mis padres, pilares importantes en mi vida. Recuerdo todo el trayecto recorrido en esta maravillosa carrera en donde con mucho sacrificio se ha llegado hasta este momento. Ellos son el motivo por el cual he podido seguir adelante. Dedico el presente trabajo a mis hermanos, mis compañeros de batalla. Dedico los resultados de mi esfuerzo a mis abuelos. Y de manera general a todos quienes en su momento aportaron con un granito de arena en mi formación académica.

Angel Román Veloz

AGRADECIMIENTOS

Al ing. Víctor Asanza por su paciencia, comprensión y soporte. Al ing. Wilton Ágila por sus consejos y amabilidad. A mi compañero, Ángel Román, por siempre estar dispuesto a dar una mano. Sin ustedes, este proyecto sería un cúmulo de información sin sentido.

Alexis Mora Roca

AGRADECIMIENTOS

A nuestros tutores Ing. Asanza e Ing. Agila por el tiempo invertido en aclarar las dudas presentadas a lo largo de este proyecto. A mi compañero y amigo Alexis Mora por su esfuerzo y dedicación sobresaliente.

Angel Román Veloz

DECLARACIÓN EXPRESA

“Los derechos de titularidad y explotación, nos corresponde conforme al reglamento de propiedad intelectual de la institución; *Alexis Xavier Mora Roca y Angel Eduardo Román Veloz,* damos nuestro consentimiento para que la ESPOL realice la comunicación pública de la obra por cualquier medio con el fin de promover la consulta, difusión y uso público de la producción intelectual"

Autor 1
Alexis Xavier Mora Roca

Autor 2
Angel Eduardo Román Veloz

EVALUADORES

Firmado electrónicamente por:
VICTOR MANUEL
ASANZA ARMIJOS

Ing. Wilton Agila

PROFESOR DE LA MATERIA

Ing. Víctor Asanza

PROFESOR TUTOR

RESUMEN

El presente proyecto tiene como finalidad asentar las bases para el diseño de filtros del tipo de respuesta al impulso infinito en lenguaje de descripción de hardware VHDL, para ser implementados en sistemas de amplificación de sonidos y mejorar la comunicación verbal bidireccional, interindividual y colectiva en personas que presentan discapacidad auditiva, debido a que en el mercado actual, el costo promedio de un dispositivo electrónico que realiza esta función y es recomendado por médicos o audiólogos fácilmente puede superar los seiscientos dólares americanos.

Para llevar a cabo la implementación de un filtro digital pasa-banda, se utilizó una tarjeta de desarrollo FPGA Mojo V3, y el módulo I2S2 de la marca Digilent que contiene un ADC y un DAC que intercambian información mediante comunicación serial mediante protocolo I^2S. Para diseñar el filtro se utilizó MATLAB y el código VHDL se escribió y compiló en el software ISE.

Las características del filtro escogidas como óptimas para la presentación de este proyecto incluyen el corte de baja frecuencia en 100 Hz, el corte en alta de 3 KHz, y un orden de filtro equivalente a dieciséis. Valores determinados como tales después de haberse realizado pruebas en diferentes fuentes de audio.

Se logró demostrar que, en diálogos y monólogos, se distingue de mejor manera al locutor debido a la reducción de ruido fuera del rango de frecuencias que corresponden al espectro de la voz humana. Además, al haberse trabajado con hardware, se demostró una alta escalabilidad para futuras nuevas implementaciones sobre este proyecto.

Palabras Clave: Discapacidad auditiva, Filtro IIR, VHDL, FPGA, I2S.

ABSTRACT

This project aims to be a starting point for infinite impulse response digital filters into hardware description language VHDL to be implemented in sound amplification systems and improve bidirectional, interindividual and collective verbal communication to people who presents hearing loss, because of the average price of existing electronic devices in the actual market which performs the previously mentioned function and is recommended by medics or audiologists, may easily exceed six hundred US dollars.

A Mojo V3 FPGA development board and the I2S2 module from Digilent which contains an ADC and DAC that exchange information via serial communication through I^2S specification, were used to achieve the implementation of a digital band-pass filter. MATLAB was used for filter designing and ISE software for VHDL code writing and compiling.

Chosen characteristics for optimal filter includes a low cutoff frequency of 100 Hz, high cutoff of 3 KHz, and sixteen as filter order. These values were determined as is after some tests within different audio sources.

It was proven that, with dialogues and monologues sources, speaker was distinguished better because of the reduction of noise that exists outside the human voice frequency range. Moreover, the fact of working with hardware showed a high scalability for new future implementations on the top of this project.

Keywords: *Hearing loss, IIR Filter, VHDL, FPGA, I2S.*

ÍNDICE GENERAL

ABREVIATURAS

ESPOL: Escuela Superior Politécnica del Litoral.

ADC: Convertidor analógico a digital, del inglés *Analog to Digital Converter.*

CAR: Control activo de ruido, del inglés Active Noise Control (*ANC).*

DAA: Dispositivo Auditivo Activo.

DAC: Convertidor digital analógico, del inglés *Digital to Analog Converter.*

DSP: Procesador digital de señal, del inglés *Digital Signal Processor.*

FPAA: Arreglos analógicos programables, del inglés *Field Programmable Analog Array.*

FPGA: Arreglos de puertas lógicas programables, del inglés *Field Programmable Gate Array.*

FIR: Referente a un filtro, respuesta al impulso finita, del inglés *Finite Impulse Response.*

I2S: Referente a un estándar de comunicación serial para audio, Chip interno integrado de sonido, del inglés Integrated Interchip Sound

IIR: Referente a un filtro, respuesta al impulso infinita, del inglés *Infinite Impulse Response.*

LMS: Algoritmo adaptativo del mínimo error cuadrático, del inglés *Least Mean Square.*

RAM: Memoria de trabajo de un sistema basado en microprocesador, del inglés *Random Access Memory.*

SNR: Relación señal a ruido, del inglés *Signal to Noise Ratio*

TI: Texas Instruments.

USB: Estándar de conexionado y protocolo de comunicación de datos, del inglés *Universal Series Bus.*

SIMBOLOGÍA

dB: Decibelio, unidad logarítmica empleada para medir la proporción entre dos magnitudes, décima parte del belio.

Hz: Hertz, unidad de frecuencia empleada para medir la cantidad de ciclos de un evento determinado que ocurren en un segundo.

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1

1. INTRODUCCIÓN

El presente proyecto intenta asentar las bases para futuras implementaciones, ofreciendo un prototipo como una alternativa para la problemática de falta de ayuda en los procesos de comunicación en personas que presentan discapacidad auditiva.

1.1 Descripción del problema

Las personas que presentan una discapacidad tienden a aislarse de manera voluntaria del resto de la sociedad debido a sus limitaciones, la comunicación verbal bidireccional interindividual y colectiva son la más comúnmente usadas en la interacción social. Sin embargo, esta interacción social se ve afectada en los canales de comunicación en los individuos que presentan el caso particular de discapacidad auditiva. Con la tecnología actual es posible obtener mejoras en determinados canales de comunicación, y aunque existe el lenguaje de señas como alternativa de interacción, se tiene la limitante social del número de personas que conocen y usan este lenguaje.

En la actualidad, la deficiencia en la capacidad para escuchar presenta soluciones a base de la amplificación del sonido ambiental que recolectan a través de uno o varios micrófonos, sin embargo, aquellas soluciones que presentan un filtrado de voz, mantienen un costo excesivo para el usuario final, no obstante, el presupuesto que este debe tener para adquirir un dispositivo auxiliar regularmente económico sigue sin ser lo suficientemente accesible. El costo del producto Sony WH1000XM3 son auriculares inalámbricos Noise Cancelling (Bluetooth, compatible con Alexa y Google Assistant, 30h de batería, óptimo para trabajar en casa. Su adquisición representa más del 50% del salario básico en el país (Sony, 2021)

Esto se debe no sólo a la calidad de la materia prima utilizada en la fabricación de estos dispositivos, pues los sistemas de control de amplificación y de filtración son más determinantes a la hora de establecer un precio, ya que el diseño de estos sistemas es en su gran mayoría de servicio privado. Los dispositivos más accesibles son aquellos que no presentan una filtración de ruido y mucho menos

un sistema de detección de voz que la separe del sonido ambiental para ser amplificada de manera independiente.

Según (Catalán Urra, Arenas, & Gerges, 2017), es recomendable la evaluación de al menos cinco DAA para tener un resultado más representativo para categorizar todos los dispositivos. También se recomienda realizar evaluaciones de comunicación, tales como la prueba de inteligibilidad y otros. La existencia de ambientes con niveles de ruido elevados es un hecho concreto en nuestra sociedad tecnológicamente en desarrollo. Las sensaciones auditivas provocadas por estas fuentes de ruido en las personas pueden ser molestas, incómodas, tener efectos perjudiciales para la salud, interferir en las comunicaciones o simplemente arruinar un rato de entretenimiento.

Un dispositivo amplificador sin un filtrado adecuado del ruido puede llegar a provocar malestar en el usuario final debido al incremento de intensidad de sonidos indeseables al momento de mantener una conversación con otra persona, e incluso se corre el riesgo de deteriorar aún más el nivel de capacidad auditiva que posee su usuario.

1.2 Justificación del problema

En Ecuador, el Consejo Nacional para la Igualdad de Discapacidades (CONADIS) registra que en el mes de septiembre del presente año 65515 personas padecen de discapacidad auditiva, de las cuales 43407 mantienen una deficiencia de entre un 30% a un 49% de la escucha con procesos degenerativos, haciendo imprescindible tener soluciones óptimas y accesibles que provean una calidad de vida satisfactoria, sin dificultades para establecer nexos comunicativos en la sociedad.

A pesar de la previa existencia en el mercado actual de opciones para solucionar o aligerar el problema, no hay un modelo del que se pueda partir para entender o diseñar de manera intuitiva un sistema de control para el filtrado en la amplificación de voz. Sin embargo, existe documentación de los procesos implicados en la separación de sonidos mediante procesos digitales.

Según (Mondragón Estupiñan, Moreno García, Hernández Cely, & Becerra Vargas, 2013), la relación S/N (señal/ruido) de los transductores también influye en el rendimiento del sistema de control, a medida que la distorsión acústica corrompe la señal de entrada.

De hecho, la propuesta indicada en este proyecto se basa en el diseño de hardware de una matriz de compuertas lógicas programables en campo, o FPGA por sus siglas en inglés, para el diseño y prototipado de un dispositivo para mejorar el proceso de comunicación verbal a través de medios digitales.

1.3 Objetivos

1.3.1 Objetivo General

- Diseñar y prototipar un sistema digital de filtración de voz humana mediante configuración de hardware en lenguaje VHDL para el mejoramiento de los canales de comunicación verbal bidireccional interindividual y colectiva en personas que presenten discapacidad auditiva.

1.3.2 Objetivos Específicos

- Diseñar un filtro pasa-banda digital para lograr la atenuación de frecuencias fuera del rango de la voz humana.
- Diseñar la configuración de hardware de una FPGA para adaptar un filtro pasa-banda digital a un medio físico mediante lenguaje de descripción de hardware VHDL.
- Demostrar el funcionamiento de un filtro digital pasa-banda que inhibe frecuencias ajenas a las de la voz humana adaptado a una FPGA mediante un prototipo físico.
- Proponer alternativas viables para la producción en masa de un sistema digital que filtre las frecuencias de voz humana, demostrando la escalabilidad de trabajar con hardware para la manipulación de sonido.

1.4 Marco teórico

1.4.1 Antecedentes Investigativos

El proyecto realizado por el autor Diego Alejandro Guzmán Arellano, con el tema “Guante Electrónico para Traducir de Lenguaje de Señas a Caracteres con Voz Artificial y Conexión Inalámbrica a Dispositivos Móviles para Personas con Discapacidad Auditiva y de Lenguaje en la Universidad Técnica de Ambato”, se basa en la implementación de un sistema embebido con 3 protocolos de comunicación. Su aplicación desarrollada juntamente con el guante electrónico contó con una interfaz de manejo sencillo facilitando la comunicación entre emisor y receptor en ambas direcciones. (Guzmán, 2017)

El trabajo desarrollado por Rubén Aldair Pico Rodríguez, con el tema “Diseño de un prototipo de dispositivo móvil para personas con problemas auditivos” utiliza transductores estéreo y mono para reproducir música por medio de vibraciones orientada a personas con discapacidad auditiva, enfocándose en la percepción a bajas frecuencias, con un 70% de público satisfecho con las pruebas realizadas.

En el trabajo realizado por Daniel Alexis Catalán Urra con el tema “Determinación de la atenuación en dispositivos auditivos tipo orejera aplicados en la protección, comunicación, y entretenimiento con control activo de ruido” se expone un análisis detallado para la determinación del desempeño en la reducción del ruido. (Catalán Urra, Arenas, & Gerges, 2017)

El proyecto elaborado por José Diaz Benítez y Fernando Mora Bermúdez, con el tema “Dispositivo vibratorio portátil de percepción musical para personas con discapacidad auditiva” se concluye que los vibradores electrónicos poseen buenas características en cuanto a tamaño, respuesta en frecuencia y amplitud de la vibración. (Diaz & Mora, 2012)

1.4.2 Fundamentación teórica

El sentido del oído

El sentido del oído inicia su funcionar desde antes del nacimiento, y el lenguaje empieza su desarrollo en los primeros años de vida rápidamente. En conjunto, ambos oídos trabajan para diferenciar una diversidad de sonidos, incluyendo los fuertes y los suaves, a distinguir el lugar dónde proviene un sonido, y a reconocer el habla de uno o más individuos en diferentes ambientes.

El sonido entra al oído y recorre tres áreas importantes antes de que la señal sea procesada por el cerebro, estas son oído externo, oído medio, y oído interno. Luego de esto, se da la identificación de los sonidos y posteriormente el entendimiento de las palabras. (Organización Mundial de la Salud, Manual básico del cuidado del oído y la audición, 2020)

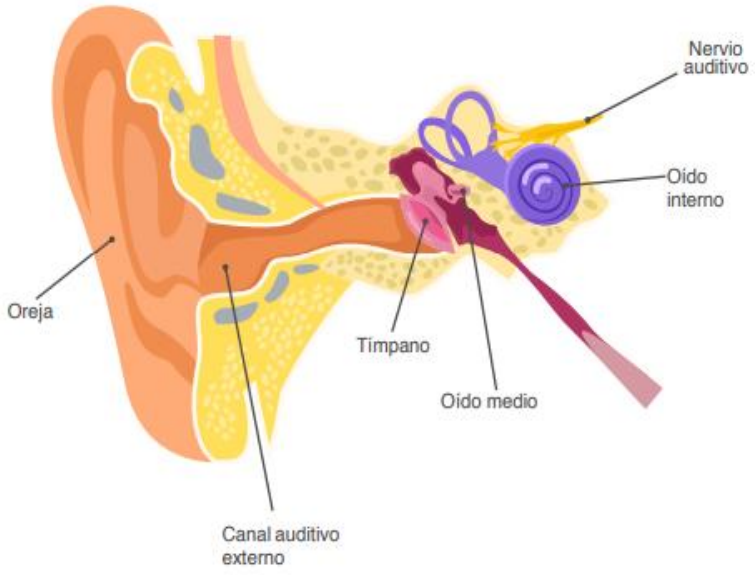

Figura 1.1 Estructura del oído humano (Organización Mundial de la Salud, Manual básico del cuidado del oído y la audición, 2020)

Discapacidad auditiva

Se considera a una persona con pérdida de audición cuando este no es capaz de oír tan bien como alguien cuyo sentido de audición es normal, es decir, donde el umbral auditivo en ambos oídos es igual o superior a 20 dB. La pérdida de audición puede ser leve, moderada, grave o profunda. Puede afectar tanto a uno o ambos oídos y empezar una serie de dificultades para

receptar una conversación o fuertes sonidos. (Organización Mundial de la Salud, Sordera y pérdida de audición, 2021)

Para la clasificación de la pérdida auditiva, se tiene el siguiente rango de umbral auditivo:

- Normal: 0-20 dB
- Hipoacusia leve: 20-40 dB
- Hipoacusia moderada: 40-60 dB
- Hipoacusia severa: 60-80 dB
- Hipoacusia profunda: 80 dB o más

1.4.3 Causas y clasificación de la pérdida auditiva

Aunque el individuo puede verse expuesto ante los factores indicados a continuación en diferentes etapas de su vida, será más susceptible a los efectos de estos durante determinados periodos críticos. (Organización Mundial de la Salud, Sordera y pérdida de audición, 2021)

Periodo prenatal y perinatal

- Factores genéticos
- Infecciones intrauterinas
- Bajo peso al nacer
- Asfixia perinatal
- Hiperbilirrubinemia

Infancia y adolescencia

- Presencia de líquido en el oído
- Otitis crónica
- Meningitis

Edad adulta y edad avanzada

- Degeneración neurosensorial relacionada con la edad
- Pérdida de audición neurosensorial repentina
- Otosclerosis

Factores a lo largo de la vida por alguno de los siguientes casos

- Medicamentos ototóxicos
- Ruido/sonido fuerte
- Carencia nutricional
- Tapón de cerumen
- Productos químicos ototóxicos en el ámbito laboral
- Infecciones virales y otras afecciones del oído

1.4.4 Soluciones para la sordera existentes en el mercado actual

El método más habitual para combatir la sordera es la adquisición de un dispositivo electrónico que colecta información sonora mediante un micrófono, se procesa, se amplifica y se expulsa de manera amplificada a través de una bocina incrustada en la zona auricular del paciente con problemas de audición. Lo recomendado, para una persona con esta discapacidad, es decidir el uso o no de este dispositivo auricular bajo la supervisión de un médico o audiólogo que llevaría a cabo una prueba para medir el rango de pérdida auditiva y concluir sobre la necesidad de un audífono.

En el mercado existen audífonos analógicos que cumplen con la función de aumentar el volumen de tonalidades que al usuario se le dificulte percibir, sin embargo, su producción es mucho menor en la actualidad gracias a la sencillez que los audífonos digitales presentan al momento de personalizar las frecuencias que serían amplificadas para mejorar la escucha del usuario. (Bupasalud, 2021)

No obstante, el precio de unos audífonos medicados, supera fácilmente los 600 USD, ya que estos pueden incorporar sistemas de control de reducción de ruido ambiental, control manual y automático de volumen, entre otros. Esta es la razón por la que las personas que padecen de esta deficiencia optan por adquirir audífonos de baja calidad y sin certificaciones, que únicamente amplifican lo que capta el micrófono receptor, incluyendo los ruidos de baja y alta frecuencia, resultando en un mayor agravamiento en el canal auditivo. (Audioguía, 2022)

CAPÍTULO 2

2. METODOLOGÍA

La metodología presentada a continuación inicia con la limitación del alcance del proyecto, en donde los parámetros auditivos del equipo a desarrollar se encuentran en el rango de umbral auditivo desde 20 [dB] a 40 [dB], prototipo el cual estará orientado a ayudar a las personas que presenten deficiencia auditiva en el rango seleccionado, llamado hipoacusia leve. Los cálculos y diseño electrónico se muestran a lo largo de este capítulo.

2.1 Introducción

Hoy en día, existen dos métodos que son utilizados para evaluar la atenuación de DAA utilizando individuos reales:

- Método REAT, “Real-Ear-Attenuation-Threshold” (Atenuación en El Umbral del Oído Real), estandarizada en la norma ANSI/ASA S12.6.
- Método MIRE, “Microphone-in-Real-Ear”, (Micrófono en el Oído Real), estandarizada en la norma ANSI/ASA S12.42.

Se considera a la metodología REAT como la mejor opción para iniciar la evaluación de atenuación a los PA, dado que depende de la respuesta subjetiva del individuo, permitiendo así detectar los cambios en el umbral auditivo de éste. La metodología MIRE depende de la instrumentación para realizar la evaluación de los PA, convirtiéndola en una metodología objetiva, pero de menor precisión que la mencionada anteriormente. (Catalán Urra, Arenas, & Gerges, 2017)

2.2 Método propuesto por los autores

Se decidió utilizar la tarjeta electrónica de desarrollo Mojo V3 basada en una FPGA Spartan 6 XC6SLX9 de Xilinx, la cual almacena su programación en la memoria no volátil de un microcontrolador ATmega32U4 de Altera que, a su vez, mediante una conexión USB, actúa como puente entre el software de programación y la FPGA.

La recepción y transmisión de audio se efectúa respectivamente mediante un convertidor analógico a digital Cirrus CS5343 y otro digital a analógico Cirrus

CS4344; los datos digitales siguen el protocolo de comunicación serial I^2S y tienen una resolución de 24 bits. Ambos convertidores se encuentran unificados en un módulo nombrado I2S2 y provisto por la empresa Digilent.

El sistema de control digital utilizado para la llevar a cabo la filtración de frecuencias dentro del rango de la voz humana está propuesto para ser adaptado a lenguaje de descripción de hardware mediante VHDL. El módulo I2S2 incluye un puerto de entrada y uno de salida para audio mediante conectores de 3.5 [mm]; para realizar pruebas en un prototipo se conectó, como fuente de sonido, la señal proveniente de un ordenador reproduciendo contenido desde plataformas de internet, y para percibir la diferencia en tiempo real, la señal de salida del filtro se conectó a un auricular.

2.3 Principios técnicos

2.3.1 Tarjeta Mojo v3 FPGA Development Board

La tarjeta de desarrollo FPGA Mojo v3, de la familia Spartan6 utiliza la lógica optimizada Spartan 6 Lx9 y el ATmega32U4 de alto rendimiento. El ATmega32U4 viene con un cargador de arranque USB (DFU) que le permitirá instalar futuras actualizaciones del firmware sin tener que comprar un programador. Una vez que la placa está encendida, el ATmega configura la FPGA desde la memoria flash. Luego de que la FPGA se ha configurado correctamente, el ATmega entra en modo esclavo. Esto permite que sus diseños de FPGA se comuniquen con el microcontrolador; que le da acceso al puerto serie y las entradas analógicas. (SparkFun, 2020)

Características:

- Spartan 6 XC6SLX9 FPGA
- 84 pines de entradas y salidas digitales
- 8 entradas analógicas
- 8 LEDs de uso general
- 1 LED para mostrar cuando la FPGA está configurada correctamente
- Regulación de voltaje a bordo que puede manejar 4.8-12V

- ATmega32U4 utilizado para configurar la FPGA, comunicaciones USB y leer los pines analógicos
- Memoria flash incorporada para almacenar el archivo de configuración FPGA.

2.3.2 Módulo Pmod I2S2

El Digilent Pmod I2S2 (Revisión A) incluye un conversor A/D (Analógico/Digital) de audio multibit Cirrus CS5343 y un conversor D/A (Digital/Analógico) estéreo Cirrus CS4344, cada uno conectado a uno de los dos conectores de audio. Estos circuitos permiten que una placa del sistema reciba y transmita señales de audio estéreo a través del protocolo I2S. El Pmod I2S2 permite adquirir una resolución de 24 bits por canal a frecuencias de muestreo de entrada de hasta 108 kHz y frecuencias de muestreo de salida de hasta 200 kHz. (Diligent, 2020)

La interfaz Digilent Pmod se utiliza para conectar módulos periféricos de baja frecuencia y bajo número de pines de E/S a placas de controlador host. Hay versiones de seis y doce pines de la interfaz definida, que abarcan los protocolos UART, I2C, SPI, I2S, GPIO y H-bridge. La versión de seis pines proporciona cuatro pines de señal de E/S digital, un pin de tierra y un pin de alimentación. La versión de doce pines proporciona ocho pines de señal de E/S, dos pines de tierra y dos pines de alimentación. Las señales de la versión de doce pines están dispuestas de modo que proporciona dos de las interfaces de seis pines apiladas. (Diligent, 2017)

En general, los módulos Pmod se pueden enchufar directamente en los conectores de la placa del controlador de host, llamados puertos de host, o puede ser conectado a la placa del controlador a través de cables de seis o doce pines. Se pueden conectar dos módulos periféricos de seis pines a un solo puerto host de doce pines a través de un cable divisor de doce pines a doble de seis pines. De manera similar, un solo módulo periférico de doce pines se puede conectar a dos puertos host de seis pines a través del mismo divisor de doce pines a doble de seis pines.

Dadas las propiedades indicadas en las hojas de datos de los circuitos integrados Cirrus CS5343 y CS4344, incluidos en el módulo I2S2 a utilizar, provisto por la marca Digilent, se tienen en uso seis pines por cada chip, dos de alimentación DC y cuatro de comunicación serial mediante el protocolo I^2S tal como se muestran en la Figura 2.1 y la Tabla 2.1.

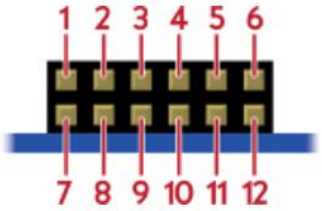

Figura 2.1 Clasificación de pines según proveedor (Digilent)

Tabla 2.1: Nombre de pines según Digilent y según la especificación I2S

CS4344 D/A (Pines 1-6)			CS5343 A/D (Pines 7-12)		
Pin	**Digilent**	**I2S**	**Pin**	**Digilent**	**I2S**
1	D/A MCLK	-	7	A/D MCLK	-
2	D/A LRCK	WS	8	A/D LRCK	WS
3	D/A SCLK	SCK	9	A/D SCLK	SCK
4	D/A SDIN	SD	10	A/D SDOUT	SD
5	GND	-	11	GND	-
6	VCC	-	12	VCC	-

Nota: Las entradas marcadas con un guion (-) no están identificadas dentro de la especificación I^2S, sin embargo, son necesarias para el funcionamiento de los circuitos integrados del módulo.

2.3.3 ISE de Xilinx

El software ISE (Entorno de síntesis integrado) de Xilinx controla todos los aspectos de flujo de diseño. A través de la interfaz de Project Navigator se

permite acceder a todas las herramientas de entrada de diseño e implementación de diseño. También es permitido acceder a los archivos y documentos asociados al proyecto. (Xilinx, 2011)

Vitis™ Model Composer es una herramienta para el dseño de modelos que permite una exploración rápida dentro del entorno MathWorks MATLAB y Simulink y guiando el camino para la producción en dispositivos Xilinx mediante la generación automática de código. Se permite diseñar sus algoritmos DSP e iterar a través de ellos utilizando bloques optimizados para el rendimiento de alto nivel y validar la corrección funcional a través de simulaciones a nivel de sistema. (Xilinx, 2021)

Vitis Model Composer transforma el diseño en una implementación de calidad de producción a través de optimizaciones de manera automática. Esta herramienta proporciona una biblioteca con más de 200 bloques HDL, HLS y AI Engine para la implementación de algoritmos y proceso de diseño en dispositivos Xilinx. También permite importar código HLS, HDL y AI Engine personalizado como bloques en la herramienta. (Xilinx, 2021)

2.3.4 MathWorks Matlab y Simulink

MATLAB es el laboratorio informático de cálculos numéricos y programación, este programa es utilizado por millones de científicos e ingenieros para crear modelos, desarrollar algoritmos y analizar datos. Simulink es un entorno de diagramas de bloque que se utiliza para diseñar sistemas con modelos multidominio, simular antes de implementar en hardware y desplegar sin necesidad de escribir código. (Matlab, 2021)

El software Communications Toolbox™ permite trabajar con una variedad de funciones, bloques y objetos que ayudan en el proceso de diseño e implementaciones de filtros. Otras capacidades de filtrado se encuentran en Signal Processing Toolbox™ y DSP System Toolbox™.

2.4 Filtros digitales

En el procesamiento digital de señales, los tipos de filtros más utilizados son el de respuesta finita al impulso o respuesta infinita al impulso (FIR o IIR por sus siglas en inglés). El filtro FIR se basa en un arreglo de elementos retrasados conectados en serie. Se almacena un elemento retrasado uno tras de otro, y, en cada instante de muestreo, el valor de cada bloque de retraso se multiplica por un coeficiente de filtro. Cada elemento de retraso necesita su propio multiplicador. Finalmente, la salida de todas las multiplicaciones se superpone para resultar en la señal de salida del filtro. (Winder, 2002)

El orden *n* de un filtro FIR corresponde al número de retrasos que este efectúa para expulsar la salida definitiva del mismo, sin embargo, se debe cumplir que existan *n*+1 multiplicadores, número que se define como “tap” en inglés. Por ejemplo, un filtro FIR de 7 taps corresponde a un orden 6 en retrasos y tiene la forma de la Figura 2.2 . (Winder, 2002)

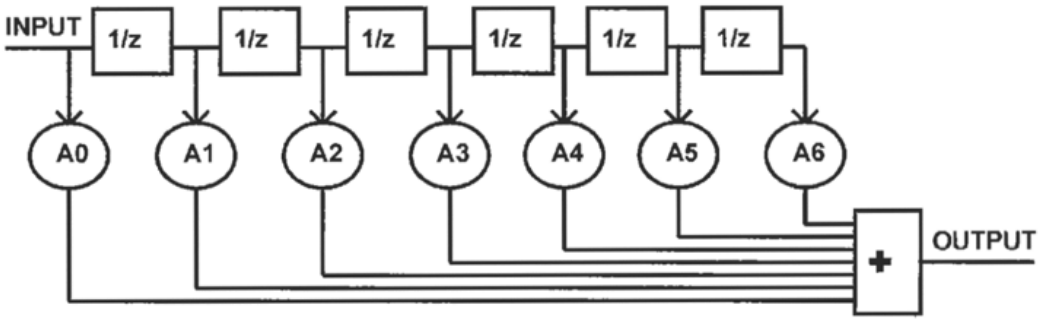

Figura 2.2 Filtro FIR de 7 taps

Es común que los coeficientes multiplicadores sean simétricos, lo que permite reducir la configuración del hardware, eliminando multiplicadores, aunque añadiendo sumadores. En este caso, como se ilustra en la Figura 2.3, se adicionan los elementos iniciales hasta los finales de manera lineal y esta suma se multiplica por los coeficientes del filtro. De la misma manera, se superpone la salida de las multiplicaciones para obtener la señal de salida del filtro. (Winder, 2002)

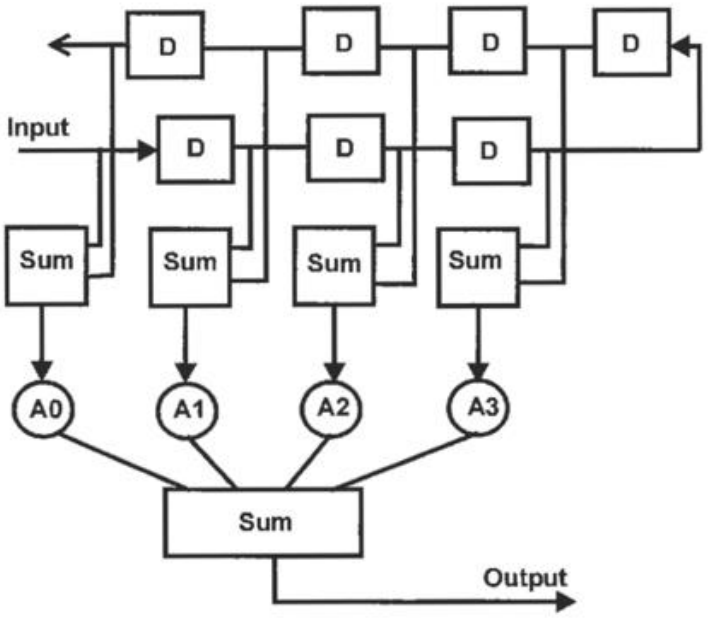

Figura 2.3 Filtro FIR de 7 taps reducido.

Los filtros IIR son más eficientes que los filtros FIR porque para una respuesta de frecuencia dada, requieren menos elementos de retraso, de suma y de multiplicadores. No obstante, su desventaja se da en que su respuesta de frecuencia es no linear, esto quiere decir que no todas las frecuencias experimentan el mismo retraso. Por lo que, los impulsos que contengan componentes con un amplio rango de frecuencias serán distorsionados cuando pasen a través de un filtro IIR. (Winder, 2002)

La mayoría de los filtros IIIR son diseñados a partir del modelado de filtros analógicos, de los cuales los más familiares son los de Butterworth, Chebyshev, Cauer (Elíptica), Chebyshev Inverso y Bessel. La respuesta de frecuencia lineal analógica puede ser convertida a su equivalente digital mediante el método del impulso invariante, escalón invariante o la transformación bilineal, siendo esta última la única capaz de proveer una función de transferencia de propósito general que puede ser usada para respuestas de pasa bajo, pasa alto, pasa banda y rechaza banda. (Winder, 2002)

El filtro IIR básico está basado en la estructura bicuadrada, como se muestra en la Figura 2.4.

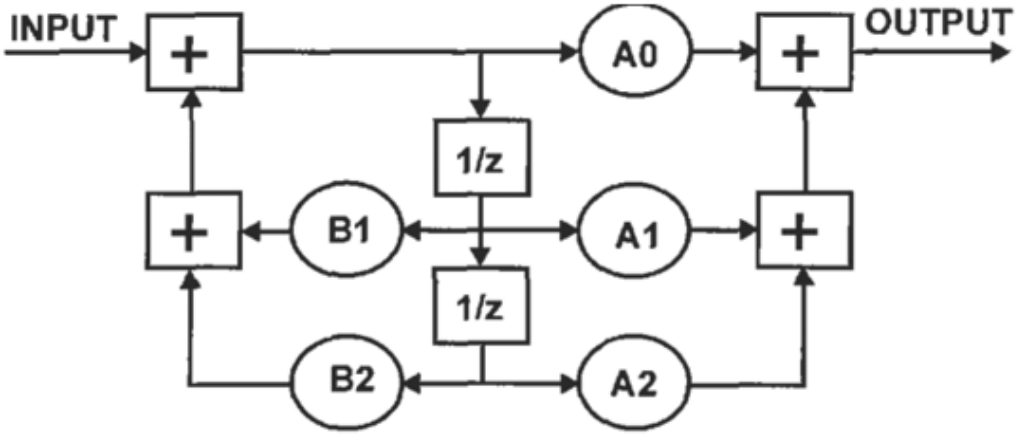

Figura 2.4 Filtro IIR con estructura bicuadrada.

Los filtros digitales IIR implican valores de la señal de salida calculados previamente, así como valores de la señal de entrada en el cálculo de la salida actual. Dado que la salida se "realimenta" para combinarla con la entrada, estos sistemas son ejemplos de la clase general de sistemas de retroalimentación. Desde un punto de vista computacional, dado que las muestras de salida se calculan en términos de valores de salida calculados previamente, el término filtro recursivo también se usa para estos filtros. (Winder, 2002)

Los filtros IIR de primer y segundo orden son útiles y brindan ejemplos simples, pero, en muchos casos, se usan filtros IIR de orden superior porque pueden realizar respuestas de frecuencia con bandas de paso y bandas de exclusión más planas y regiones de transición más nítidas. Las funciones *butter, chebyl, cheby2* y *ellip* en Signal Processing Toolbox de MATLAB se pueden utilizar para diseñar filtros con características selectivas de frecuencia prescritas.

2.5 Especificación de bus I2S

Se están introduciendo muchos sistemas de audio digital en el mercado de audio de consumo, incluidos discos compactos, cintas de audio digital, procesadores de sonido digital y sonido de TV digital. Las señales de audio digital en estos sistemas están siendo procesadas por una serie de circuitos integrados (V)LSI, como:

- Convertidores A/D y D/A;
- Procesadores de señales digitales;
- Filtros digitales;
- Interfaces de entradas/salidas digitales.

El bus solo tiene que manejar datos de audio, mientras que otras señales, como el control y la sub-codificación, se transfieren por separado. Para minimizar el número de pines necesarios y simplificar el cableado, se utiliza un bus serie de 3 líneas que consta de una línea para dos canales de datos multiplexados en el tiempo, una línea de selección de palabra y una línea de reloj. (Sparkfun, 2012)

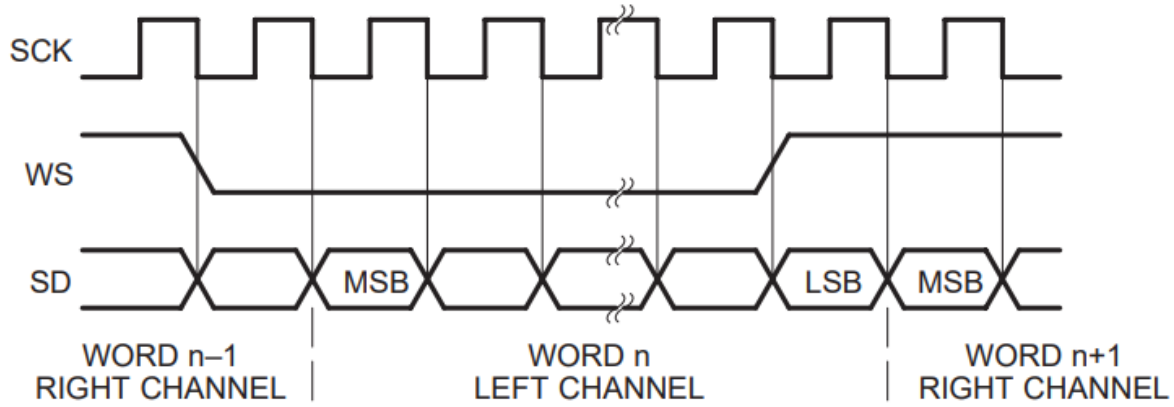

Figura 2.5 Señales de SCK, WS, SD

El bus mostrado tiene tres líneas:

- Reloj serie continuo (SCK);
- Selección de palabras (WS);
- Datos en serie (SD);

y el dispositivo que genera SCK y WS es el maestro

Los datos en serie se transmiten en operación complemento a dos con el MSB primero. El MSB se transmite primero porque el transmisor y el receptor pueden tener diferentes longitudes de palabra. No es necesario que el transmisor conozca cuántos bits puede manejar el receptor, ni el receptor necesita saber cuántos bits se transmiten. (Sparkfun, 2012)

La línea de selección de palabras es un indicador del canal que se está transmitiendo:

- WS = 0; canal 1 (izquierda)
- WS = 1; canal 2 (derecha)

WS puede cambiar ya sea en el borde anterior o posterior del reloj serial, pero no necesita ser simétrico. En el esclavo, esta señal se engancha en el borde de ataque de la señal del reloj. La línea WS cambia un período de reloj antes de que se transmita el MSB. Esto permite que el transmisor esclavo obtenga la temporización síncrona de los datos en serie que se configurarán para la transmisión. Además, permite que el receptor almacene la palabra anterior y borre la entrada para la siguiente palabra. (Sparkfun, 2012)

En el formato I2S, a cualquier dispositivo se le permite actuar como maestro del sistema proporcionando las señales de reloj necesarias. Un esclavo generalmente derivará su señal de reloj interno de una entrada de reloj externo. Esto significa que, teniendo en cuenta los retrasos de propagación entre el reloj maestro y las señales de datos y/o selección de palabras, que el retraso total es simplemente la suma de: (Sparkfun, 2012)

- El retraso entre el reloj externo (maestro) y el esclavo reloj interno
- El retraso entre el reloj interno y los datos y/o señales de selección de palabras.

2.6 Selección del filtro de voz humana

En un estudio de simulación, Thakur R. utilizó diferentes tipos de filtros analógicos, que posteriormente fueron digitalizados como filtros IIR debido a que resultaba más importante la respuesta de frecuencia ante la necesidad de preservar características de fase lineal, para lo cual hubiera sido conveniente utilizar filtros FIR. (Thakur, Pandey, & Gupta, 2018)

Para la respuesta de frecuencia, se utilizó un filtro pasa banda con una frecuencia de corte bajo en 100 Hz y una frecuencia de corte alto en 3 KHz y una frecuencia de muestreo de 10 KHz. Los filtros analógicos de los que se partieron fueron de Butterworth, Chebyshev tipo I, Chebyshev tipo II y el filtro elíptico.

Tabla 2.2 Amplitud de señal de voz para filtros digitales IIR según Thakur R. dada la simulación provista en su investigación

Orden del filtro	Butterworth	Chebyshev tipo I	Chebyshev tipo II	Filtro Elíptico
2	0.015	0.0002	0.010	0.0025
5	0.040	0.0005	0.020	0.0150
10	0.050	0.0050	0.030	0.0200
15	0.030	0.0030	0.005	0.0075

Tal como se puede observar en la **Error! Reference source not found.**, la técnica del filtro de Butterworth entregó mejor amplitud de señal de voz en cualquiera de los órdenes utilizados para el diseño del filtro analógico. Además, Thakur R. concluyó que el mismo filtro de Butterworth entregó el menor ruido a comparación del resto, demostrando mejor calidad de filtración en la voz humana. (Thakur, Pandey, & Gupta, 2018)

En otra investigación propuesta por Meiniar W., Afrida F., Irmasari A., Mukti A. y Astharini D., se intentó atenuar las frecuencias de la voz humana de las frecuencias de una canción, para lo cual utilizaron el diseño de Butterworth para elaborar diferentes filtros rechaza banda, cada uno con diferentes frecuencias de corte, para poder simularlos en un entorno de MATLAB y concluir qué frecuencias eran las óptimas para su caso de estudio. (Meiniar, Afrida, & Mukti, 2017)

Para una canción con una cantante femenina, Meiniar W. y sus compañeros concluyeron que un filtro rechaza banda con una frecuencia de corte bajo en 200 Hz y una frecuencia de corte alto en 5.5 KHz fue la mejor opción que atenuaba la voz de la cantante y que mantenía el tono de la canción lo más cercano al original.

Debido a los antecedentes recolectados, se decidió utilizar el diseño de filtro digital de Butterworth otorgado por la herramienta de MATLAB para realizar un filtro pasa banda para las frecuencias de la voz humana. Esta digitalización se eligió debido a que se priorizó la eficiencia, rapidez y bajo consumo de hardware que disponen los filtros IIR en comparación a los FIR, tomando en cuenta que se está utilizando una FPGA de bajos recursos. (Meiniar, Afrida, & Mukti, 2017)

CAPÍTULO 3

3. RESULTADOS Y ANÁLISIS

En este capítulo se presentan y analizan los resultados obtenidos con la implementación en físico del sistema de filtrado de voz en lenguaje VHDL.

3.1 Implementación del módulo I2S2 de Digilent en la tarjeta de desarrollo FPGA Mojo V3

El circuito integrado del módulo I2S2 que realiza la operación de convertidor analógico a digital tiene la limitación de una tasa de muestreo máxima de 108 [KHz] por canal de audio, sin embargo, para mantener la frecuencia de reloj de la tarjeta de desarrollo FPGA Mojo V3, que es de 50 [MHz], se decidió generar un divisor de tal manera que la frecuencia de la señal de reloj maestro (o Master Clock) que recibiría el módulo I2S2 fuera la mitad de la frecuencia del reloj de la FPGA. Además, se usó la velocidad 256x para realizar el muestreo de datos de manera serial, correspondiendo este número a la relación entre ciclos de reloj maestro y ciclos de reloj del selector de canal; en la hoja de datos del integrado Cirrus CS5343 este dato se muestra como "*MCLK/LRCK Ratio*". Dada esta velocidad, se logra obtener una frecuencia de muestreo tal que:

$$f_{MCLK} = \frac{f_{FPGA}}{2} = \frac{50\ [MHz]}{2} = 25\ [MHz]$$

$$f_{sampling} = \frac{f_{MCLK}}{MCLK/LRCK\ Ratio} = \frac{25\ [MHz]}{256} = 97656.25\ [Hz]$$

Tomando en cuenta la especificación de comunicación serial de sonido I^2S, y la recomendación por parte de la hoja de datos del integrado antes mencionado, la cantidad de ciclos del reloj de señal por ciclos del selector de canal, especificado como "*SCLK/LRCK Ratio*" debe ser de 64, la primera mitad para el canal izquierdo y la segunda para el canal derecho. El comportamiento de esta trama se puede observar en la Figura 3.1.

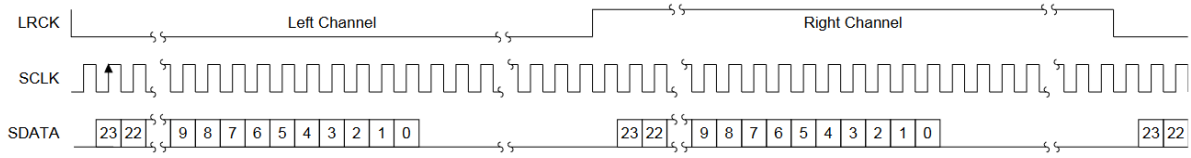

Figura 3.1 Interfaz serial de audio I²S del integrado CS5343.

Por otro lado, el circuito integrado que realiza la operación de convertidor digital a analógico mantiene características superiores a su contraparte, sin embargo, debido a las limitaciones del ADC, se utilizó el mismo valor de frecuencia para el reloj maestro, y la misma velocidad de muestreo.

3.2 Diseño del filtro pasa-banda para frecuencias de voz humana

Se escribió un script para MATLAB que genera los coeficientes para el filtro pasa-banda del tipo Butterworth dadas las siguientes condiciones iniciales:

- Orden del filtro: 16.
- Frecuencia de corte baja: 100 (Hz).
- Frecuencia de corte alta: 3000 (Hz).
- Frecuencia de muestreo: 97656.25 (Hz).

Nota: La frecuencia de muestreo se calcula dentro del script según lo explicando en la sección 3.1 de este mismo capítulo.

El software de MATLAB trabaja de manera predeterminada con un tipo de datos de doble precisión (64 bits) que cumple con la especificación IEEE 754, sin embargo, para reducir el consumo de recursos de la FPGA, los coeficientes del filtro que genera la función "*designfilt*" fueron convertidos a un tipo de datos de precisión simple (32 bits) que cumple con la misma especificación.

La función utilizada para el diseño del filtro genera un arreglo de coeficientes por etapas de segundo orden, por lo que, para el orden de filtro elegido, se generaron ocho etapas. El sistema completo adaptado a lenguaje de descripción de hardware primero se diseñó en SIMULINK, tal como se muestra en la Figura 3.2, para realizar simulaciones y analizar su respuesta de magnitud, mostrada en la Figura 3.3, mediante la herramienta de análisis lineal provista por este mismo software.

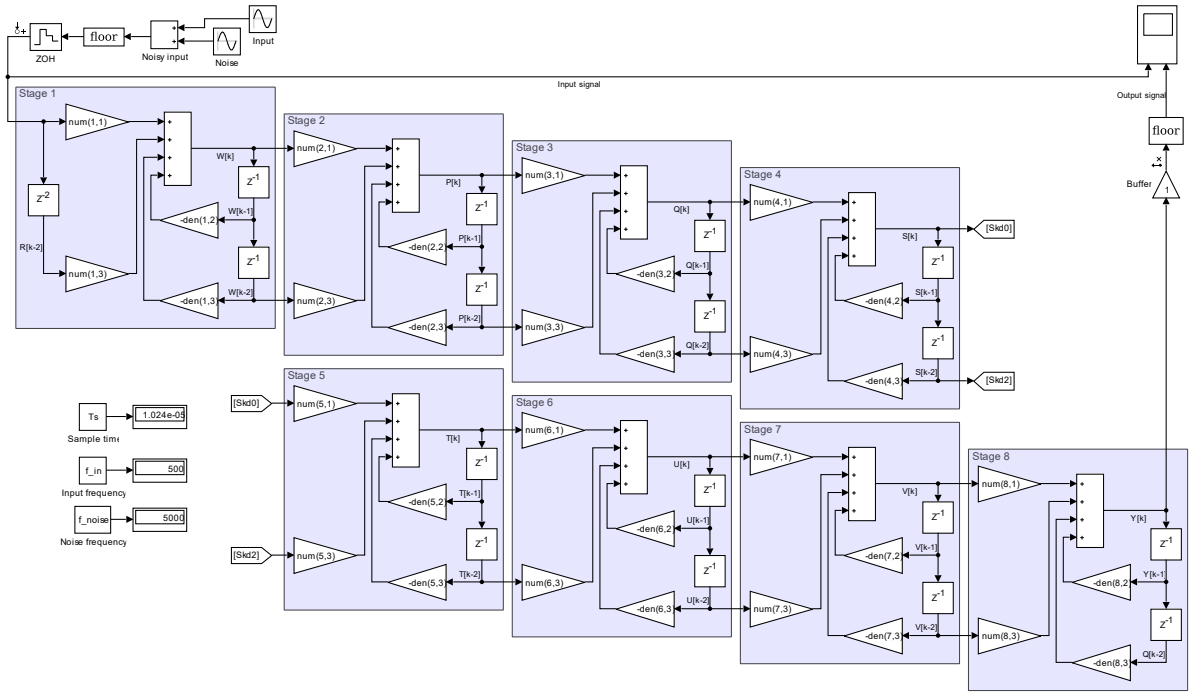

Figura 3.2 Modelo del filtro digital por etapas de segundo orden en SIMULINK.

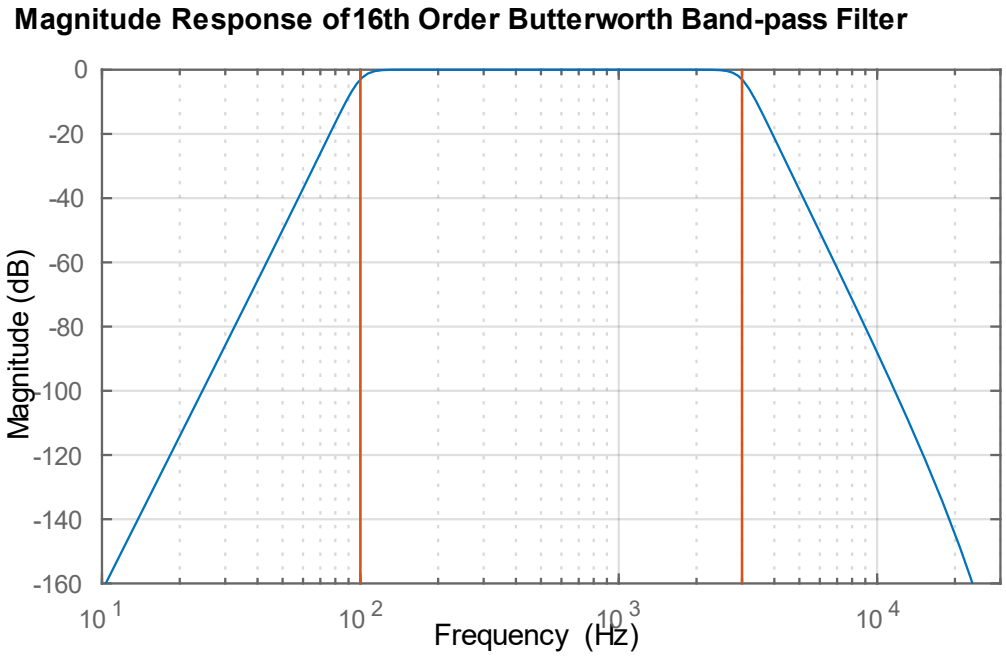

Figura 3.3 Respuesta de magnitud del filtro pasa-banda de orden 16.

En las frecuencias de corte existe una reducción de magnitud en 3 [dB] dado que este es el comportamiento de un filtro de Butterworth, y se observan las pendientes creciente y decreciente de aproximadamente 160 [dB/década] antes y después de las frecuencias de corte respectivamente.

Para demostrar el funcionamiento del filtro se realizó una simulación con una entrada senoidal de 500 [Hz] y un ruido externo de 5 [KHz]. Las gráficas de entrada y de salida del filtro después de la simulación se muestran en la Figura 3.4.

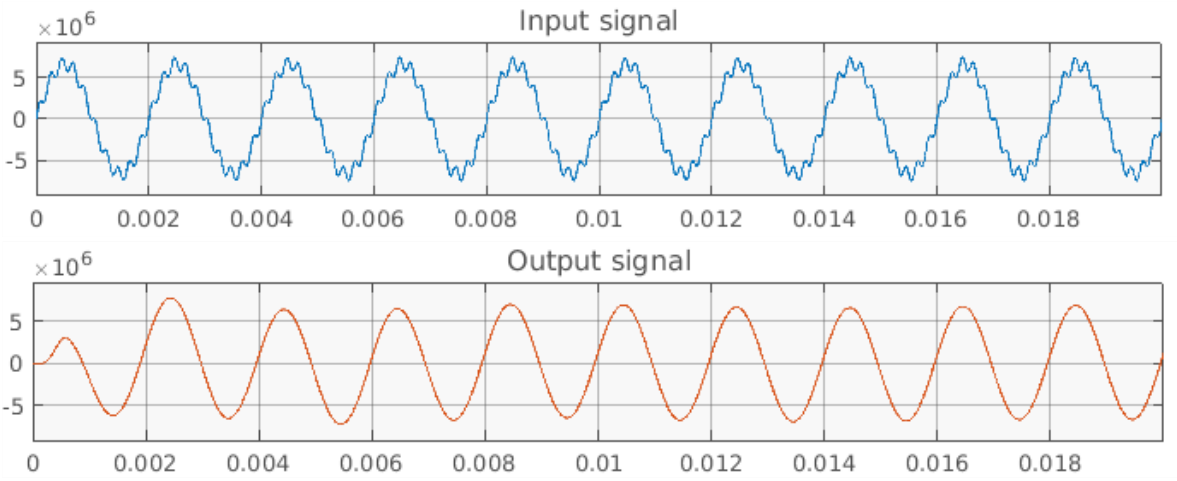

Figura 3.4 Simulación del modelo del filtro pasa-banda en SIMULINK.

3.3 Adaptación del filtro pasa-banda en VHDL

Para adaptar un sistema digital que opera con números codificados bajo la especificación IEEE 754 en VHDL, es necesario definir este tipo de datos o un algoritmo para resolver operaciones con entradas que mantengan el formato indicado en dicha especificación. Para lograr esto, el software ISE de Xilinx incluye la librería "*IEEE_PROPOSED*" que contiene los paquetes necesarios para manejar tipos de datos arreglados o "*fixed*" y con punto/coma flotante. Este último es el que sigue la especificación IEEE 754 para precisión simple con el tipo de dato "*float32*".

Inicialmente se intentó adaptar un filtro pasa-banda de orden 4 directamente en la FPGA únicamente mediante lógica combinatorial, sin embargo, debido a las limitaciones físicas del chip de gama media-baja seleccionado para implementar el prototipo del proyecto, el software ISE sugería que era necesario utilizar aproximadamente el 300% de recursos que ofrecía dicha FPGA.

Para solucionar este inconveniente, se eligió diseñar un módulo con nombre "*SingleAddProduct*" que recibiera tres entradas numéricas con el formato de precisión simple y que ejecutara la siguiente operación:

$$R = N_1 + (N_2 \cdot N_3)$$

Este módulo se ejecuta de manera recursiva hasta alcanzar el valor de salida del sistema de segundo orden correspondiente a cada etapa del filtro. La recursión se efectuó mediante la implementación de una máquina secuencial sincrónica y su comportamiento se puede observar de manera resumida en la Figura 3.5. Al tratarse de dos canales de audio, la máquina secuencial realiza el mismo algoritmo, pero en secciones diferentes dependiendo de si se activó una señal de inicio para el canal izquierdo o derecho.

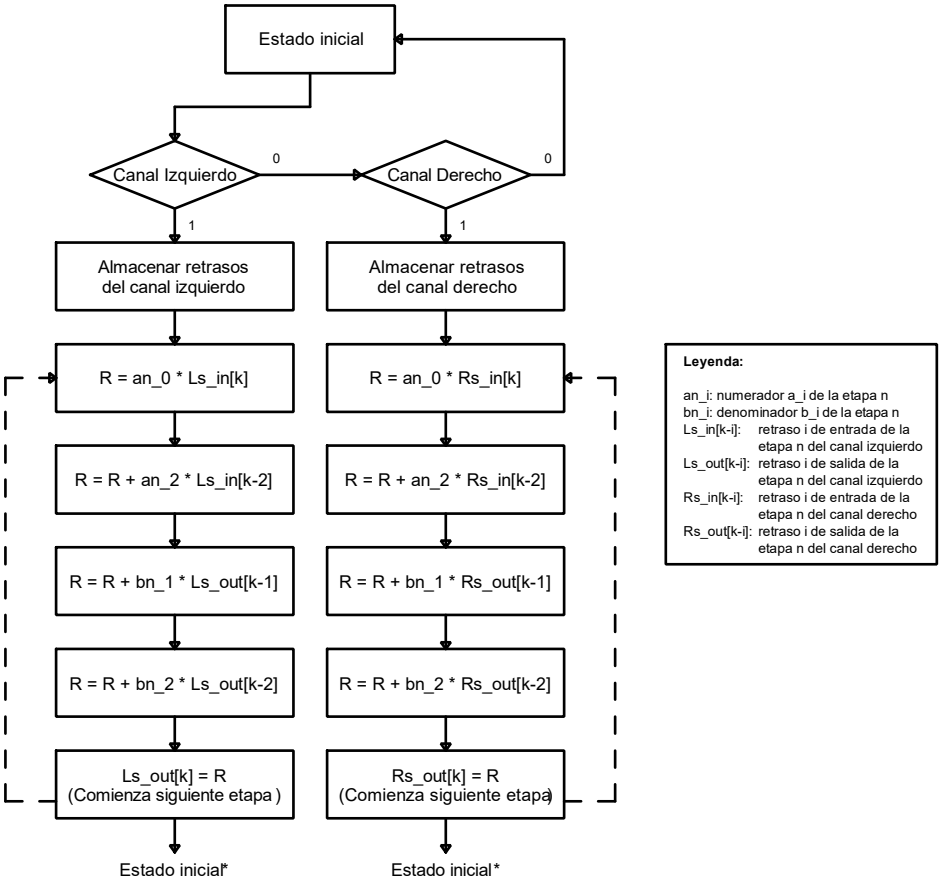

Figura 3.5 Diagrama de bloques simplificado del filtro pasa-banda digital implementado.

3.4 Integración del módulo I2S2 con el filtro pasa-banda en VHDL

Las codificaciones de los datos manejados por la especificación I^2S y el estándar IEEE 754 son diferentes, el módulo I2S2 entrega y recibe un número entero en la codificación de complemento a la base dos, mientras que el filtro pasa-banda diseñado requiere una entrada y una salida codificadas como punto flotante. El paquete "*float_pkg*", y sus respectivas dependencias, incluido en la librería "*IEEE_PROPOSED*" antes mencionada, contiene las funciones necesarias para transformar estos dos tipos de datos.

Se utilizaron las funciones de conversión provistas por dicho paquete, tomando en cuenta que la codificación en complemento a dos es análoga al tipo de dato "*sfixed*", que corresponde a un número con parte fraccionaria con codificación arreglada y con signo, sin embargo, la especificación I^2S entrega simplemente números enteros, por lo que se especificó que el dato a convertir no incluye esta parte fraccionaria. Para simplificar su uso se crearon los módulos "*Fixed2Float*" y "*Float2Fixed*" para convertir los tipos datos respectivamente antes de ingresar y después de salir del filtro pasa-banda.

Para unir todos los módulos antes mencionados y emparejar los pines de entrada y salida, se creó un módulo controlador con nombre "*TopDriver*", sin embargo, al realizar la implementación del proyecto dentro del software ISE, éste alertó que la configuración deseada no funcionaría de manera adecuada debido a que el módulo del filtro pasa-banda necesitaría cerca de 70 nanosegundos para completar las operaciones, esto debido al uso de un gran número de compuertas lógicas que utilizadas al momento de realizar las operaciones de suma y multiplicación.

Considerando que la frecuencia de reloj de la tarjeta de desarrollo Mojo V3 es de 50 [MHz] equivalente a 20 nanosegundos por ciclo, se dividió esta frecuencia para cuatro, de tal manera que cuadriplicamos el tiempo por ciclo de reloj, logrando tener una frecuencia de 12.5 [MHz] para el módulo del filtro pasa-banda equivalente a 80 nanosegundos por ciclo, corrigiendo la advertencia que indicaba el software ISE al momento de implementar el diseño del proyecto.

3.5 Implementación del proyecto en un prototipo físico

Para cargar la configuración de hardware deseada en la tarjeta de desarrollo FPGA Mojo V3, es necesario generar un archivo binario desde el software ISE, además de ejecutar los procesos de síntesis e implementación de diseño.

Al momento de ejecutar el proceso de implementación de diseño se obtuvo un resumen de consumo de recursos de la FPGA. Los apartados más importantes se ven reflejados en la Tabla 3.1.

Tabla 3.1 Consumo relevante de bloques lógicos del chip Xilinx Spartan 6 XC6SLX9 al implementar el proyecto.

Uso de bloques lógicos	Usado	Disponible	Proporción
Número de registros	1984	11440	17%
Número de LUTs	3191	5720	55%
Número de bloques (slices) ocupados	1249	1430	87%
Número de pares LUT-FF con Flip Flop no utilizado	2078	3989	52%
Número de pares LUT-FF con LUT no utilizado	798	3989	20%
Número de pares LUT-FF completamente utilizados	1139	3989	27%

Para llevar a cabo una prueba en tiempo real del presente proyecto, se implementó el prototipo en la tarjeta de desarrollo FPGA Mojo V3 cargando el binario desde un ordenador, mediante el software Mojo Loader provisto por Embedded Micro. En la Figura 3.6, Figura 3.7 y Figura 3.8, se puede observar respectivamente el binario totalmente cargado a la FPGA, un diagrama de conexiones entre la FPGA y el módulo I2S2, y una foto de la implementación final del prototipo.

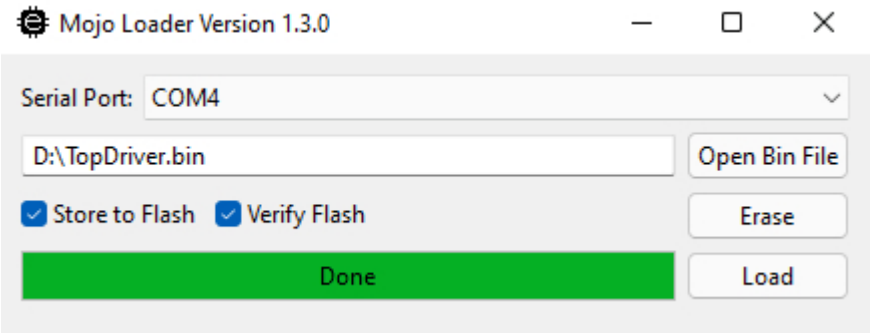

Figura 3.6 Software Mojo Loader después de cargar la configuración del proyecto a la tarjeta de desarrollo FPGA Mojo V3.

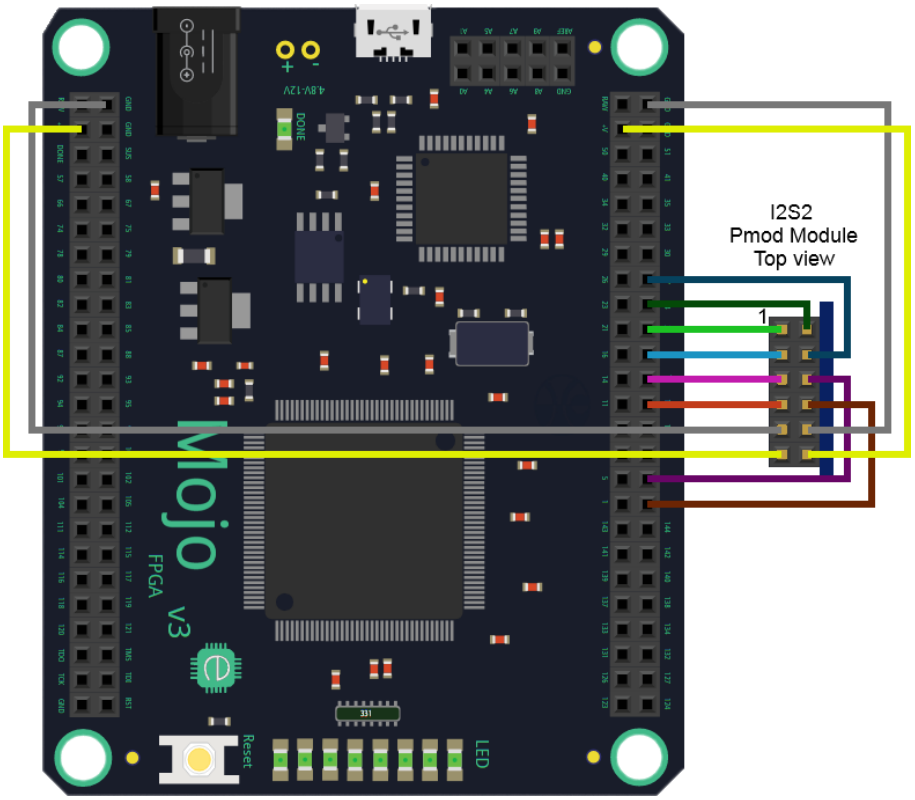

Figura 3.7 Diagrama de conexiones entre la tarjeta de desarrollo FPGA Mojo V3 y el módulo ADC/DAC de audio I2S2.

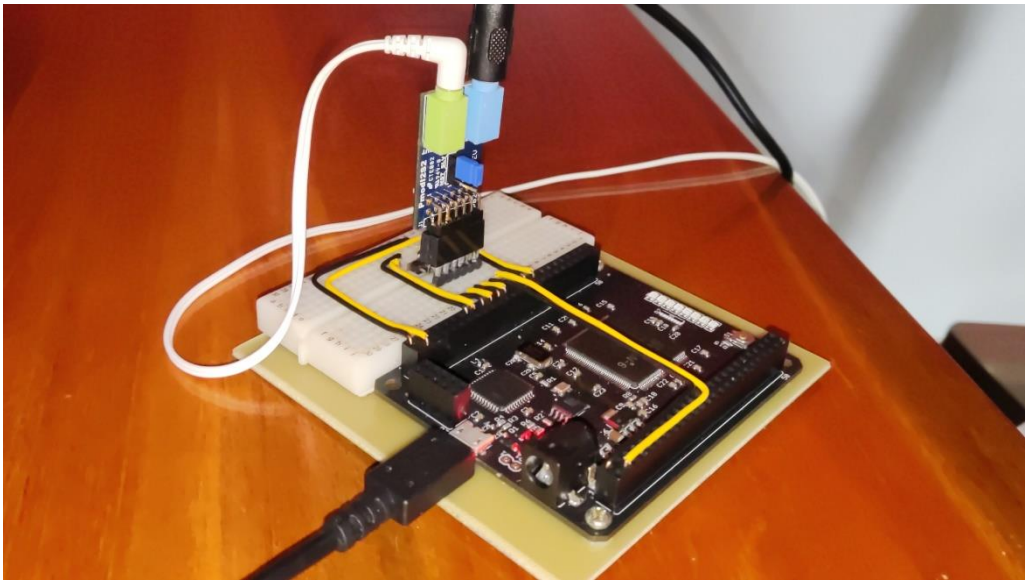

Figura 3.8 Implementación del prototipo final.

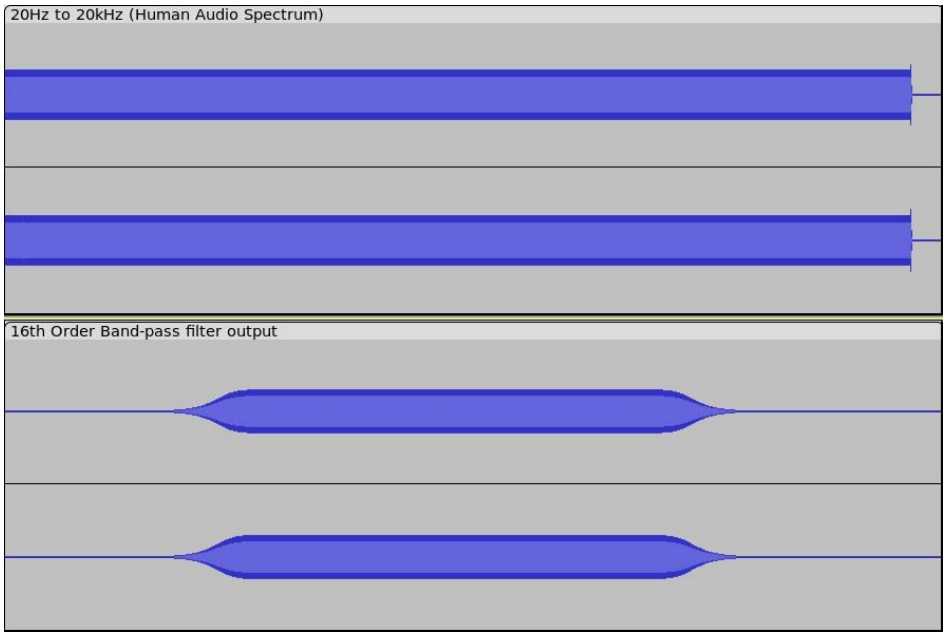

Figura 3.9 Respuesta de magnitud del filtro pasa-banda implementado dentro del rango de frecuencias audibles para el ser humano.

En la Figura 3.9 es observable cómo, dentro del rango de frecuencias del espectro audible de un ser humano (figura en la parte superior), en la salida del filtro pasa-banda (figura en la parte inferior) existe una atenuación fuera del rango que comprenden dos extremos de frecuencias diferentes. Esto es debido a la implementación física del filtro pasa-banda.

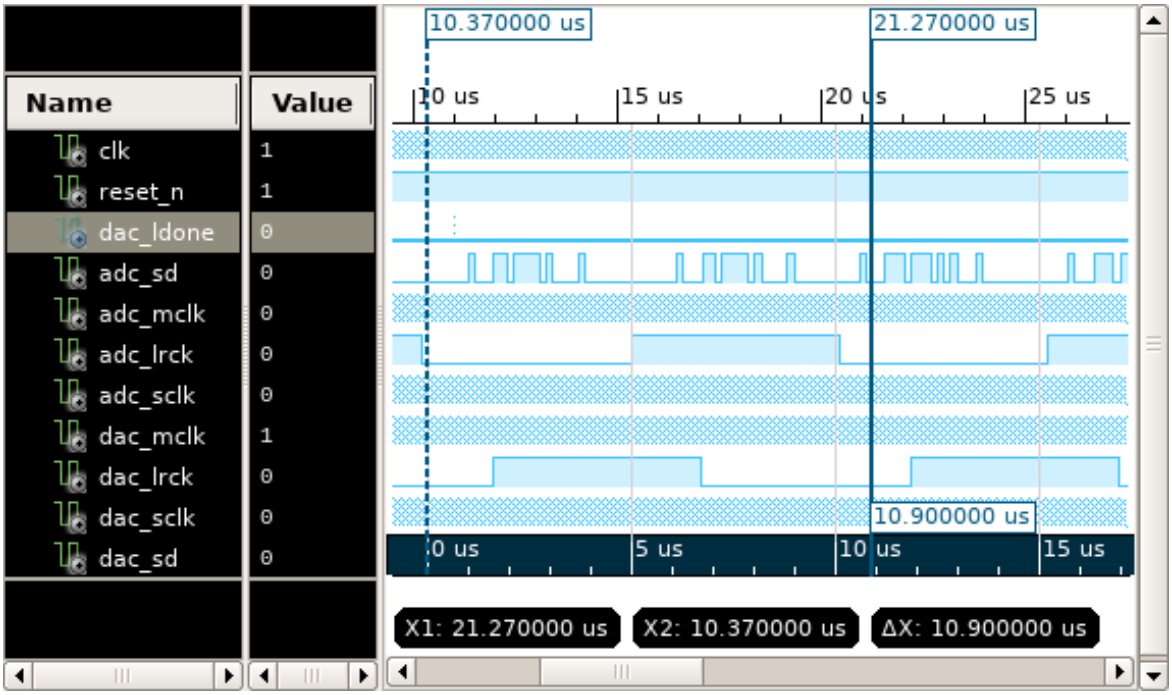

Figura 3.10 Tiempo de respuesta del filtro pasa-banda diseñado.

La Figura 3.10 indica el tiempo de respuesta de 10.9 microsegundos, siendo aproximado al de muestreo, equivalente a 10.24 microsegundos, es decir, el retraso entre la entrada y la salida del filtro pasa-banda, es de aproximadamente un tiempo de muestreo, intervalo imperceptible al oído humano.

3.6 Viabilidad económica y futuro alcance del proyecto

El proyecto diseñado es una base para el diseño de filtros digitales IIR de bajos recursos y de alta precisión, debido al uso de la especificación IEEE 754 para la operación de números de punto flotante de manera recursiva en el filtro pasa-banda, siendo que este puede llegar ser mejorado de varias maneras, por ejemplo:

- Optimizando el algoritmo del filtro pasa-banda.
- Aumentando el orden del filtro pasa-banda.
- Implementando un sistema de control digital adicional, por ejemplo, un sistema de cancelación activa de ruido externo.

Además, el diseño realizado no se limita únicamente a la tarjeta de desarrollo Mojo V3 que se utilizó para el prototipo, si no para cualquier otro integrado FPGA que tenga cualidades similares o superiores. Esto implica que el circuito puede incluso

ser reducido en tamaño si se usase microelectrónica y puede resultar económica su implementación de manera masiva si se contase con las herramientas o proveedores adecuados. Para demostrar lo expuesto anteriormente, se realizó un análisis de costos de los materiales más relevantes para fabricar un producto final, observable en la Tabla 3.2, considerando el uso de componentes relativamente pequeños si se consiguieran al por mayor

Tabla 3.2 Análisis de costos para una implementación física final

Componente	Tamaño	Precio unitario (+100 unidades)
FPGA XILINX Spartan 6 XC6SLX9-2CSG225C	13x13 [mm^2]	$8.3763
Microcontroller ATMEGA32U4-MU	7x7 [mm^2]	$6.4666
Clock Oscillator ASFLMB-50.000MHZ-LY-T	3.2x5 [mm^2]	$1.2209
8MHz Crystal Resonator S32080001010500	2x3 [mm^2]	$0.1778
ADC I^2S Cirrus Logic CS5343-CZZR	3x5 [mm^2]	$6.9228
DAC I^2S Cirrus Logic CS4344-CZZR	3x5 [mm^2]	$2.1928
3.3V Regulator ST LD39100PU33RY	3x3 [mm^2]	$1.4751

Los elementos pasivos o conectores se excluyen de la tabla no. 4 debido a sus precios considerablemente económicos al ser adquiridos al por mayor. Los precios de fabricación pueden variar dependiendo de si se contase con las herramientas adecuadas para realizar la mano de obra o si se fabrica mediante un agente externo.

CAPÍTULO 4

4. CONCLUSIONES Y RECOMENDACIONES

Se presentan a continuación las conclusiones de los análisis de resultados obtenidos en el capítulo anterior.

Conclusiones

El filtro pasa-banda digital que mejor se acomodó a las frecuencias de voz humana, atenuando los ruidos ajenos a este rango, se encuentra entre las frecuencias de corte bajo en 100 [Hz] y de corte en algo en 3 [KHz]. Además, considerando un menor consumo de recursos, un orden de 16 fue elegido como óptimo para el filtro pasa-banda.

- Un filtro pasa-banda de alta precisión y velocidad puede adaptarse en una FPGA de bajos recursos al reducir su comportamiento combinatorial, ejecutando los cálculos de manera recursiva con el comportamiento de una máquina secuencial sincrónica.
- Utilizando el prototipo implementado, se logró demostrar que la voz humana se percibe con mayor claridad cuando la fuente de audio es un diálogo o un monólogo. No obstante, debido a la capacidad de ciertos instrumentos musicales de alcanzar frecuencias similares a la de la voz humana, estos no llegan a ser atenuados en su totalidad cuando la fuente de audio es del tipo musical.
- Trabajar con hardware, aunque relativamente más complicado, es más versátil, pues permite reducir el tamaño de una implementación física en caso de usar microelectrónica, ya que un mismo componente puede estar fabricado en varios tamaños, además de que su arquitectura combinatorial le permite realizar más tareas en paralelo, permitiendo la adición de módulos extra en caso de querer modificar o mejorar un proyecto.
- Los componentes más relevantes para la implementación de un producto final son relativamente económicos en el mercado mayorista, incluso siendo de un tamaño muy reducido para una mejor portabilidad, por lo que su producción en masa sería muy accesible de contarse con la mano de obra necesaria.

Recomendaciones

- Se recomienda no unir las etapas del filtro pasa-banda en un único sistema, es decir, multiplicar las funciones de transferencia de cada etapa entre sí para obtener una única función, pues, al tratarse de un filtro de tipo IIR, la sensibilidad de los coeficientes resultantes aumenta; incluso la precisión doble (64 bits) puede llegar a tener inconvenientes con este tipo de sistemas y la curva de respuesta de magnitud puede verse distorsionada.
- Se recomienda no utilizar la función "*bode*" de MATLAB para obtener la respuesta de magnitud de un filtro IIR, pues esta recibe únicamente una función de transferencia y para lograr esto, se tendrían que multiplicar las funciones de cada sistema entre sí; para obtener esta gráfica se sugiere armar el modelo del sistema por etapas en SIMULINK y utilizar la herramienta de Análisis Lineal disponible en este mismo entorno de trabajo.
- Se recomienda tomar en cuenta las advertencias que indican las herramientas de implementación de diseño del software correspondiente a la FPGA a configurar; a pesar de no considerarse como errores, éstas mantienen información que podría ser esencial para explicar el comportamiento no deseado de la configuración diseñada.
- Para un futuro proyecto, se sugiere la adición de un sistema de cancelación activa de ruido externo para contrarrestar los efectos no aislantes al utilizar auriculares; además de la optimización del filtro ya sea rediseñando el comportamiento de este o aumentando su orden.

BIBLIOGRAFÍA

Referencias

Audioguía. (2022). audioguia.info. Recuperado el 20 de enero de 2022, de https://www.audioguia.info/audifonos/precios-audifonos/

Bupasalud. (2021). Bupasalud.com. Recuperado el 13 de noviembre de 2021, de https://www.bupasalud.com/salud/audifonos-sordos

Catalán Urra, D. A., Arenas, J. P., & Gerges, S. N. (2017). Determinación de la atenuación en dispositivos auditivos tipo orejera aplicados en la protección, comunicación, y entretenimiento con control activo de ruido. Síntesis Tecnológica, 4(2), 124. doi:https://doi.org/10.4206/sint.tecnol.2011.v4n2-07

Diaz, J., & Mora, F. (2012). Dispositivo vibratorio portatil de percepción musical. Recuperado el 11 de noviembre de 2021, de http://bibliotecadigital.usb.edu.co/bitstream/10819/5628/1/Dispositivo_vibratorio_portatil_diaz_2012.pdf

Diligent. (2017). Recuperado el 18 de diciembre de 2021, de https://digilent.com/reference/_media/reference/pmod/pmod-interface-specification-1_2_0.pdf

Diligent. (2020). Recuperado el 15 de diciembre de 2021, de https://digilent.com/reference/pmod/pmodi2s2/start

Guzmán, D. (2017). Guante Electrónico para Traducir de Lenguaje de Señas a Caracteres con Voz Artificial. Recuperado el 14 de noviembre de 2021, de https://repositorio.uta.edu.ec/bitstream/123456789/25193/1/Tesis_t1222ec.pdf

Matlab. (2021). Recuperado el 11 de diciembre de 2021, de https://la.mathworks.com/products/matlab.html?s_tid=hp_ff_p_matlab

Meiniar, W., Afrida, F., & Mukti, A. (2017). Human voice filtering with band-stop filter design in MATLAB. International Conference on Broadband Communication, Wireless Sensors and Powering (BCWSP), (págs. 1-4). doi:10.1109/BCWSP.2017.8272563.

Mondragón Estupiñan, J. E., Moreno García, F. E., Hernández Cely, M. M., & Becerra Vargas, J. A. (2013). Diseño e implementación de un sistema de control activo de ruido (CAR) desarrollado en tecnología FPAA. Revista Facultad de Ingeniería, 21(33), 43. Obtenido de https://revistas.uptc.edu.co/index.php/ingenieria/article/view/2117/2080

Organización Mundial de la Salud. (2020). Manual básico del cuidado del oído y la audición. Recuperado el 10 de noviembre de 2021, de https://www.who.int/es/publications/i/item/basic-ear-and-hearing-care-resource

Organización Mundial de la Salud. (2 de marzo de 2021). Sordera y pérdida de audición. Recuperado el 10 de noviembre de 2021, de https://www.who.int/es/news-room/fact-sheets/detail/deafness-and-hearing-loss

Pretexsa. (26 de octubre de 2021). Usos de la tecnología de asistencia. Recuperado el 10 de noviembre de 2021, de http://www.pretexsa.com/RXvO8vpV.html

Sony. (2021). Recuperado el 10 de noviembre de 2021, de https://www.amazon.es/Sony-WH-1000XM3B-Auriculares-inal%C3%A1mbricos-Compatibile/dp/B07GDR2LYK/ref=psdc_934056031_t3_B078VGQCZ4

Sparkfun. (2012). Recuperado el 13 de diciembre de 2021, de https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf

SparkFun, E. (2020). Mojo v3 FPGA Development Board. 6333 Dry Creek Parkway, Niwot, Colorado 80503. Recuperado el 29 de noviembre de 2021, de https://www.sparkfun.com/products/retired/11953

Thakur, R., Pandey, M., & Gupta, N. (2018). Filtering of Noise in Audio/Voice Signal. 3rd International Conference on Contemporary Computing and Informatics (IC3I), (págs. 119-123). doi:10.1109/IC3I44769.2018.9007299

Winder, S. (2002). Analog and Digital Filter Design (Segunda ed.). Newnes. Recuperado el 29 de noviembre de 2021, de http://dea.unsj.edu.ar/sredes/Biblioauxi/WINDER,%20S.%20(2002).%20Analog%20and%20Digital%20Filter%20Design%20(2nd%20ed.).pdf

Xilinx. (2011). Recuperado el 14 de diciembre de 2021, de https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ise_tutorial_ug695.pdf

Xilinx. (2021). Recuperado el 17 de diciembre de 2021, de https://www.xilinx.com/products/design-tools/vitis/vitis-model-composer.html

Xilinx. (2021). XUP PYNQ-Z1. Recuperado el 1 de diciembre de 2021, de https://www.xilinx.com/support/university/boards-portfolio/xup-boards/XUPPYNQ.html

V. Asanza, R. Estrada, J. Miranda, L. Rivas and D. Torres, "Performance Comparison of Database Server based on SoC FPGA and ARM Processor," 2021 IEEE Latin-American Conference on Communications (LATINCOM), 2021, pp. 1-6, doi: 10.1109/LATINCOM53176.2021.9647742.

V. Asanza, R. E. Pico, D. Torres, S. Santillan and J. Cadena, "FPGA Based Meteorological Monitoring Station," 2021 IEEE Sensors Applications Symposium (SAS), 2021, pp. 1-6, doi: 10.1109/SAS51076.2021.9530151.

APÉNDICES

Apéndice A

Script para el diseño de un filtro en MATLAB

Nombre del archivo: FilterDesign.m

Descripción: El programa a continuación genera los números binarios a ser introducidos en el código VHDL para la configuración del filtro pasa-banda. Las condiciones iniciales que requiere dependen de la frecuencia de muestreo, que se calcula a través de los datos preferidos para el ADC, las frecuencias de corte alta y baja y el orden del filtro.

```
close all
clear all
clc

%% Input for Cirrus CS5343 A/D I2S Slave Mode Approximation

% High dual speed data extracted from datasheet:
%      https://statics.cirrus.com/pubs/proDatasheet/CS5343-
44_F5.pdf

f_CLK = 50e6; % [Hz] Clock Speed of Mojo V3 Board

f_MCLK = f_CLK/2; % [Hz] Reduced clock to half speed
% This was made intentionally because of CS5343 limitations

MCLK_LRCK_ratio = 256; % Oversampling: the lowest, the
fastest
SCLK_LRCK_ratio = 64;  % the highest, the fastest

MCLK_SCLK_ratio = MCLK_LRCK_ratio / SCLK_LRCK_ratio; % To
obtain f_SCLK

f_SCLK = f_MCLK / MCLK_SCLK_ratio; % [Hz] Input Sample Rate

fs = f_SCLK / 64; % [Hz] Sampling frequency [Formula provided
in Datasheet]

fprintf('Mojo V3 Board clock frequency: %.2f MHz\n',
round(f_CLK/1e6, 2))
fprintf('Master Clock for Cirrus CS5343: %.2f MHz\n',
round(f_MCLK/1e6, 2))
fprintf('Audio sampling frequency: %.2f KHz\n',
round(fs/1e3, 2))
fprintf('Signal clocks per each master clock: %d\n\n',
MCLK_SCLK_ratio)
```

```
%% Butterworth filter design

% Cutoff frequencies
fL = 100; % Lower
fH = 3e3; % Higher

wL = 2*pi*fL; % Angular high cutoff frequency
wH = 2*pi*fH; % Angular low cutoff frequency

% ws = 2*pi*fs; Angular step frequency

fn = fs/2; % Nyquist frequency;
Ts = 1/fs; % Audio sample time

order = 16; % Single (low or high) filter order

% Design bandpass filter of order 4 with low/high frequency
at fL and fH
BPFd = designfilt('bandpassiir', ...
    'FilterOrder', order, ...
    'HalfPowerFrequency1', fL, ...
    'HalfPowerFrequency2', fH, ...
    'SampleRate', fs);
BPF = single(BPFd);        % Cast filter to single precision
number coding
coeff = BPF.Coefficients;  % Load coefficients

coeffdim = size(coeff);  % Dimensions of coefficients array
[rows columns]
% Each row of coefficient array is a stage of filter

num = coeff(1:coeffdim(1),1:(coeffdim(2)/2));
den = coeff(1:coeffdim(1),(coeffdim(2)/2 + 1):coeffdim(2));

%% Start displaying vectors of numerators and denominators

for i = 1:coeffdim(1)
    fprintf('Stage N° %d\n', i)
    fprintf('Num vector: [')
    for j = 1:coeffdim(2)/2
        fprintf('%f', coeff(i, j))
        if j ~= coeffdim(2)/2
            fprintf(' ')
        else
            fprintf(']\n')
        end
    end
    fprintf('Den vector: [')
    for j = (coeffdim(2)/2 + 1):coeffdim(2)
```

```
        fprintf('%f', coeff(i, j))
        if j ~= coeffdim(2)
            fprintf(' ')
        else
            fprintf(']\n\n')
        end
    end
end

%% Display binary values for constants in single precision
format IEEE 754

fprintf('Binary numbers with IEEE 754 single precision
format will be\n')
fprintf('displayed. These are coefficients of first and
second stage of\n')
fprintf('the BPF filter. Denominator coefficients have been
negated to\n')
fprintf('match the addition with the numerator
multiplications.\n\n')

fprintf('This is because each stage has the system
format:\n')
fprintf('y[k] = a_0*r[k] + a_2*r[k-2] - b_0*y[k-1] - b_1*y[k-
2]\n\n')
fprintf('So the final system with negated b coefficients
would be:\n')
fprintf('y[k] = a_0*r[k] + a_2*r[k-2] + (-b_0)*y[k-1] + (-
b_1)*y[k-2]\n\n')

q = quantizer('single');

for i = 1:coeffdim(1)
    fprintf('Stage N° %d binary coefficient values\n', i)
    fprintf('')
    for j = 1:coeffdim(2)/2
        fprintf('+a%d_%d: %s\n', i, j-1, num2bin(q, coeff(i,
j)))
    end
    fprintf('')
    for j = (coeffdim(2)/2 + 1):coeffdim(2)
        fprintf('-b%d_%d: %s\n', i, j-1-coeffdim(2)/2, ...
            num2bin(q, -coeff(i, j)))
    end
    fprintf('\n')
end

%% Final transfer function

H = tf(single(1), single(1), Ts);
```

```
for i = 1:coeffdim(1)
    ai = coeff(i, 1:coeffdim(2)/2);
    bi = coeff(i, (coeffdim(2)/2 + 1):coeffdim(2));
    Hi = tf(ai, bi, Ts);
    H = H * Hi;
end

clear ai bi Hi

display(H)

%% Parameters for filter simulation (using SIMULINK)

% I2S two's complement output approximation

% Input

data_width = 24;

% Algorithm

resolution = data_width - 1; % first bit corresponds to sign

max_amplitude = 2^resolution - 1;
min_amplitude = -2^resolution;

offset = (max_amplitude + min_amplitude) / 2;
peak = max_amplitude + offset; % or offset - min_amplitude

% Define amplitudes and frequencies for input and noise
signal

A_in = floor(peak * 0.8);
A_noise = floor(peak * 0.1);

f_in = 500; % [Hz] Input frequency
f_noise = 5e3; % [Hz] Noise frequency
```

Apéndice B

Módulo para el receptor de audio mediante comunicación serial I^2S.

Nombre del archivo: I2Sreceiver.vhd

Descripción: El siguiente código describe la configuración dentro de una FPGA para realizar la lectura de audio de manera serial mediante la especificación I^2S.

```
----------------------------------------------------------------------
-- Escuela Superior Politécnica del Litoral (ESPOL)
-- Facultad de Ingeniería en Electrónica y Computación (FIEC)
-- Final undergraduate project: VHDL Human Voice Filtering
--
-- Degree to obtain: Electronics and Automation Engineer
--
-- Made by: Ángel Román Velóz & Alexis Mora Roca
-- Contact: (+593) 98 817 2232 & (+593) 99 587 0443
--
-- Parent Module:   TopDriver.vhd
-- Module Name: I2Sreceiver.vhd
--
-- Description: Receive audio signal via serial with I2S
protocol.
-- Comment:         Received signal will be in two's
complement format.
--                  This module was written to work with
Cirrus CS5343
--                  A/D I2S in slave mode, extra MCLK needed.
--                  Further information about Cirrus CS5343
A/D:
--
    https://statics.cirrus.com/pubs/proDatasheet/CS5343-
44_F5.pdf
--
-- Dependencies:    IEEE standard library for standard
behaviour.
----------------------------------------------------------------------
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity I2Sreceiver is
    generic(
        MCLK_SCLK_ratio : integer := 4;   -- MCLK clock times
per SCLK clock
        SCLK_LRCK_ratio : integer := 64;  -- SCLK clock times
per LRCK clock
```

```
        data_width          : integer := 24   -- Data width
(24 bits two's complement)
        );
    port(
        Reset_N : in std_logic;  -- Low logic asynchronous
reset
        MCLK        : in std_logic;  -- Master clock (I2S)
        LRCK        : out std_logic := '0';  -- Left/Right
clock or Word clock (I2S)
        SCLK        : out std_logic := '0';  -- Signal clock
(I2S)
        SDRX        : in std_logic;  -- Serial data input
(I2S) (Output from I2S module)
        -- Left and right data with specified size with two's
complement format
        L_DATA  : out std_logic_vector(data_width-1 downto
0) := (others => '0');
        R_DATA  : out std_logic_vector(data_width-1 downto
0) := (others => '0');
        Ldone       : out std_logic := '0';    -- Output
notification for left data already loaded
        Rdone       : out std_logic := '0'    -- Output
notification for right data already loaded
        );
end I2Sreceiver;

-- Port is written to fit I2S2 module pinout from Digilent.
--                   Further                   information:
https://digilent.com/reference/pmod/pmodi2s2/start

-- Even when 24 bits are being loaded, is a good practice to
send 32 signal clocks per
-- left/right clock as the first will be omitted and after
reading 24 bits the rest of
-- signal clock reading will be omitted as well.

architecture Logic of I2Sreceiver is

    -- Two internal signals to preserver LRCK and SCLK as
outputs
    signal LRCK_int : std_logic := '0';
    signal SCLK_int : std_logic := '0';
    -- Internal data signal to catch serial information
before being sent to output
    signal DATA_int :  std_logic_vector(data_width-1 downto
0) := (others => '0');
    -- Internat counters to divide master clock into signal
and left/right clock
    -- Size: ceil(log2(SCLK_LRCK_ratio))
```

```
    signal LRCK_cnt : unsigned(5 downto 0) := (others =>
'0');
    -- Size: ceil(log2(MCLK_SCLK_ratio/2))
    signal SCLK_cnt : unsigned(0 downto 0) := (others =>
'0');

begin

    process(MCLK,Reset_N)      --  Sensitivity  list  for
synchronous sequential logic
    begin
        if Reset_N = '0' then   -- Asynchronous low logic
reset
            LRCK_int <= '0';  -- Clear internal LRCK
            SCLK_int <= '0';  -- Clear internal SCLK
            L_DATA <= (others => '0');  -- Clear left data
output
            R_DATA <= (others => '0');  -- Clear right data
output
            DATA_int <= (others => '0');  -- Clear internal
data signal
            LRCK_cnt  <=  (others  =>  '0');   --  Clear  LRCK
counter
            SCLK_cnt  <=  (others  =>  '0');   --  Clear  SCLK
counter
            Ldone <= '0';  -- Clear left channel notification
output
            Rdone  <=  '0';     --  Clear  right  channel
notification output
        elsif MCLK'event and MCLK = '1' then  -- Synchronous
behaviour
            Ldone <= '0';  -- Keep left output notification
to low until data is ready
            Rdone <= '0';  -- Keep right output notification
to low until data is ready
            --  when  SCLK_cnt  is  less  than  a  half  of
MCLK_SCLK_ratio
            if SCLK_cnt < ((MCLK_SCLK_ratio/2) - 1) then
                SCLK_cnt  <=  SCLK_cnt  +  1;    --  Count-up
SCLK_cnt
            else   -- If SCLK_cnt = less than a half of
MCLK_SCLK_ratio
                SCLK_cnt  <=  (others  =>  '0');    --  Clear
SCLK_cnt
                SCLK_int <= not SCLK_int;     -- Toggle SCLK
                --    when    LRCK_cnt    is    less    than
SCLK_LRCK_ratio
                -- It shouldn't be less than a half because
this will be performed
```

```
                -- when SCLK rise or fall, so it's performed
twice.
                if LRCK_cnt < (SCLK_LRCK_ratio - 1) then
                    LRCK_cnt <= LRCK_cnt + 1;  -- Count-up
LRCK_cnt
                    -- SCLK = 0 means that previous signal
clock was low, but now is high
                    --  LRCK_cnt  must  be  greater  than  1
because first SCLK is ignored (I2S specification)
                    if SCLK_int = '0' and LRCK_cnt > 1 and
LRCK_cnt < ((data_width*2)+2) then
                        --  Left  shift  internal  data  and
append serial input
                        DATA_int  <=  DATA_int(data_width-2
downto 0) & SDRX;
                    -- If LRCK_cnt finished with last serial
input
                    elsif  SCLK_int  =  '0'  and  LRCK_cnt  =
((data_width*2)+2) then
                        if  LRCK_int  =  '1'  then  --  Right
channel
                           -- Assign R_DATA value and notify
output is ready
                            R_DATA <= DATA_int;
                            Rdone <= '1';
                        else  --  LRCK_int  =  '0'  --  Left
channel
                           -- Assign L_DATA value and notify
output is ready
                            L_DATA <= DATA_int;
                            Ldone <= '1';
                        end if;
                        DATA_int  <=  (others  =>  '0');   --
Clear internal data signal
                    end if;
                else     --   If   LRCK_cnt   =   less   than
SCLK_LRCK_ratio
                    LRCK_cnt <= (others => '0');  -- Clear
LRCK_cnt
                    LRCK_int <= not LRCK_int;     -- Toggle
LRCK
                end if;
            end if;
        end if;

    end process;

    LRCK <= LRCK_int;  -- Assign LRCK
    SCLK <= SCLK_int;  -- Assign SCLK
end Logic;
```

Apéndice C

Módulo para el transmisor de audio mediante comunicación serial I^2S.

Nombre del archivo: I2Stransmitter.vhd

Descripción: El siguiente código describe la configuración que necesita la FPGA para enviar información de audio de manera serial mediante la especificación I^2S.

```
------------------------------------------------------------
----------------------
-- Escuela Superior Politécnica del Litoral (ESPOL)
-- Facultad de Ingeniería en Electrónica y Computación (FIEC)
-- Final undergraduate project: VHDL Human Voice Filtering
--
-- Degree to obtain: Electronics and Automation Engineer
--
-- Made by: Ángel Román Velóz & Alexis Mora Roca
-- Contact: (+593) 98 817 2232 & (+593) 99 587 0443
--
-- Parent Module:   TopDriver.vhd
-- Module Name: I2Sreceiver.vhd
--
-- Description: Receive audio signal via serial with I2S
protocol.
-- Comment:         Received signal will be in two's
complement format.
--                  This module was written to work with
Cirrus CS5343
--                  A/D I2S in slave mode, extra MCLK needed.
--                  Further information about Cirrus CS5343
A/D:
--
    https://statics.cirrus.com/pubs/proDatasheet/CS5343-
44_F5.pdf
--
-- Dependencies:    IEEE standard library for standard
behaviour.
------------------------------------------------------------
----------------------
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity I2Sreceiver is
    generic(
        MCLK_SCLK_ratio : integer := 4;   -- MCLK clock times
per SCLK clock
```

```
        SCLK_LRCK_ratio : integer := 64;  -- SCLK clock times
per LRCK clock
        data_width          : integer := 24   -- Data width
(24 bits two's complement)
        );
    port(
        Reset_N : in std_logic;  -- Low logic asynchronous
reset
        MCLK        : in std_logic;  -- Master clock (I2S)
        LRCK        : out std_logic := '0';  -- Left/Right
clock or Word clock (I2S)
        SCLK        : out std_logic := '0';  -- Signal clock
(I2S)
        SDRX        : in std_logic;  -- Serial data input
(I2S) (Output from I2S module)
        -- Left and right data with specified size with two's
complement format
        L_DATA  : out std_logic_vector(data_width-1 downto
0) := (others => '0');
        R_DATA  : out std_logic_vector(data_width-1 downto
0) := (others => '0');
        Ldone       : out std_logic := '0';    -- Output
notification for left data already loaded
        Rdone       : out std_logic := '0'    -- Output
notification for right data already loaded
        );
end I2Sreceiver;

-- Port is written to fit I2S2 module pinout from Digilent.
--                   Further                   information:
https://digilent.com/reference/pmod/pmodi2s2/start

-- Even when 24 bits are being loaded, is a good practice to
send 32 signal clocks per
-- left/right clock as the first will be omitted and after
reading 24 bits the rest of
-- signal clock reading will be omitted as well.

architecture Logic of I2Sreceiver is

    -- Two internal signals to preserver LRCK and SCLK as
outputs
    signal LRCK_int : std_logic := '0';
    signal SCLK_int : std_logic := '0';
    -- Internal data signal to catch serial information
before being sent to output
    signal DATA_int :  std_logic_vector(data_width-1 downto
0) := (others => '0');
    -- Internat counters to divide master clock into signal
and left/right clock
```

```
    -- Size: ceil(log2(SCLK_LRCK_ratio))
    signal LRCK_cnt : unsigned(5 downto 0) := (others =>
'0');
    -- Size: ceil(log2(MCLK_SCLK_ratio/2))
    signal SCLK_cnt : unsigned(0 downto 0) := (others =>
'0');

begin

    process(MCLK,Reset_N)      --  Sensitivity  list  for
synchronous sequential logic
    begin
        if Reset_N = '0' then   -- Asynchronous low logic
reset
            LRCK_int <= '0';  -- Clear internal LRCK
            SCLK_int <= '0';  -- Clear internal SCLK
            L_DATA <= (others => '0');  -- Clear left data
output
            R_DATA <= (others => '0');  -- Clear right data
output
            DATA_int <= (others => '0');  -- Clear internal
data signal
            LRCK_cnt  <=  (others  =>  '0');   --  Clear  LRCK
counter
            SCLK_cnt  <=  (others  =>  '0');   --  Clear  SCLK
counter
            Ldone <= '0';  -- Clear left channel notification
output
            Rdone   <=   '0';     --   Clear   right   channel
notification output
        elsif MCLK'event and MCLK = '1' then  -- Synchronous
behaviour
            Ldone <= '0';  -- Keep left output notification
to low until data is ready
            Rdone <= '0';  -- Keep right output notification
to low until data is ready
            --  when  SCLK_cnt  is  less  than  a  half  of
MCLK_SCLK_ratio
            if SCLK_cnt < ((MCLK_SCLK_ratio/2) - 1) then
                SCLK_cnt  <=  SCLK_cnt  +  1;    --  Count-up
SCLK_cnt
            else   -- If SCLK_cnt = less than a half of
MCLK_SCLK_ratio
                SCLK_cnt  <=  (others  =>  '0');    --  Clear
SCLK_cnt
                SCLK_int <= not SCLK_int;     -- Toggle SCLK
                --    when    LRCK_cnt    is    less    than
SCLK_LRCK_ratio
                -- It shouldn't be less than a half because
this will be performed
```

```
                -- when SCLK rise or fall, so it's performed
twice.
                if LRCK_cnt < (SCLK_LRCK_ratio - 1) then
                    LRCK_cnt <= LRCK_cnt + 1;  -- Count-up
LRCK_cnt
                    -- SCLK = 0 means that previous signal
clock was low, but now is high
                    --  LRCK_cnt  must  be  greater  than  1
because first SCLK is ignored (I2S specification)
                    if SCLK_int = '0' and LRCK_cnt > 1 and
LRCK_cnt < ((data_width*2)+2) then
                        --  Left  shift  internal  data  and
append serial input
                        DATA_int  <=  DATA_int(data_width-2
downto 0) & SDRX;
                    -- If LRCK_cnt finished with last serial
input
                    elsif  SCLK_int  =  '0'  and  LRCK_cnt  =
((data_width*2)+2) then
                        if  LRCK_int  =  '1'  then  --  Right
channel
                           -- Assign R_DATA value and notify
output is ready
                            R_DATA <= DATA_int;
                            Rdone <= '1';
                        else  --  LRCK_int  =  '0'  --  Left
channel
                           -- Assign L_DATA value and notify
output is ready
                            L_DATA <= DATA_int;
                            Ldone <= '1';
                        end if;
                        DATA_int  <=  (others  =>  '0');   --
Clear internal data signal
                    end if;
                else     --   If   LRCK_cnt   =   less   than
SCLK_LRCK_ratio
                    LRCK_cnt <= (others => '0');  -- Clear
LRCK_cnt
                    LRCK_int <= not LRCK_int;     -- Toggle
LRCK
                end if;
            end if;
        end if;

    end process;

    LRCK <= LRCK_int;  -- Assign LRCK
    SCLK <= SCLK_int;  -- Assign SCLK
end Logic;
```

Apéndice D

Módulo operacional de números de punto flotante.

Nombre del archivo: SingleAddProduct.vhd

Descripción: El siguiente código describe una configuración para FPGA que al recibir tres números codificados con precisión simple según la especificación IEEE 754, realice la operación Número 1 + (Número 2 * Número 3).

```
----------------------------------------------------------------------
-- Escuela Superior Politécnica del Litoral (ESPOL)
-- Facultad de Ingeniería en Electrónica y Computación (FIEC)
-- Final undergraduate project: VHDL Human Voice Filtering
--
-- Degree to obtain: Electronics and Automation Engineer
--
-- Made by: Ángel Román Velóz & Alexis Mora Roca
-- Contact: (+593) 98 817 2232 & (+593) 99 587 0443
--
-- Parent Module:   BPF2channels.vhd
-- Module Name: SingleAddProduct.vhd
--
-- Description: Performs the operation N1+(N2*N3).
-- Comment:         Will be used recursively by BPF to
calculate the filter output.
--
-- Dependencies:    IEEE standard library for standard
behaviour.
--                      IEEE_PROPOSED library for floating
point number operations.
----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Dependencies for FLOAT_PKG
use IEEE.NUMERIC_STD.ALL;
library IEEE_PROPOSED;
use IEEE_PROPOSED.FIXED_FLOAT_TYPES.all;
use IEEE_PROPOSED.FIXED_PKG.all;
use IEEE_PROPOSED.FLOAT_PKG.ALL;

entity SingleAddProduct is
    generic(
        float_width     : integer := 32  -- 32bits for single
precision floating number
        );
    port(
```

```
        N1      : in std_logic_vector(float_width-1 downto
0);  -- input with IEEE754 format
        N2      : in std_logic_vector(float_width-1 downto
0);  -- input with IEEE754 format
        N3      : in std_logic_vector(float_width-1 downto
0);  -- input with IEEE754 format
        R       : out std_logic_vector(float_width-1 downto
0)  -- output with IEEE754 format
        );
end SingleAddProduct;

-- Advise: To use FLOAT_PKG operations, data must match with
the corresponding floating
--         point type specified in package. Input and output
data type were left as
--         STD_LOGIC_VECTOR for better data handling between
modules and to perform
--         simulations as FLOAT_PKG types are not supported.

architecture Logic of SingleAddProduct is

    -- float32 data type matchs IEEE754 format
    signal N1fp     : float32;  -- N1 with float32 data type
    signal N2fp     : float32;  -- N2 with float32 data type
    signal N3fp     : float32;  -- N3 with float32 data type
    signal Rfp      : float32;  -- R with float32 data type

begin

    N1fp <= to_float(N1);   -- Convert N1 to float32 data
type
    N2fp <= to_float(N2);   -- Convert N2 to float32 data
type
    N3fp <= to_float(N3);   -- Convert N3 to float32 data
type

    Rfp <= N1fp + (N2fp * N3fp);  -- Performs R = N1 + (N2
* N3)

    R <= to_slv(Rfp);  -- Convert float32 to STD_LOGIC_VECTOR
and assign to R

end Logic;
```

Apéndice E

Módulo de filtro pasa-banda de dos canales.

Nombre del archivo: BPF2channels.vhd

Descripción: El siguiente código provee la configuración para la FPGA de realizar la operación de un filtro pasa-banda de dos canales de audio, izquierdo y derecho.

```
----------------------------------------------------------------------
-- Escuela Superior Politécnica del Litoral (ESPOL)
-- Facultad de Ingeniería en Electrónica y Computación (FIEC)
-- Final undergraduate project: VHDL Human Voice Filtering
--
-- Degree to obtain: Electronics and Automation Engineer
--
-- Made by: Ángel Román Velóz & Alexis Mora Roca
-- Contact: (+593) 98 817 2232 & (+593) 99 587 0443
--
-- Parent Module:   TopDriver.vhd
-- Module Name: BPF2channels.vhd
--
-- Description: Calculate the band-pass filter output.
-- Comment:         Designed BPF is divided into second-
order stages and made
--                      with synchronous sequential logic.
--                      BPF.vhd, as designed, will take
2+4*(filter_order/2) clock
--                      times to perfom the system output
after receiving 'Start'
--                      signal. Be careful editting the
script when trying to
--                  enhance the filter with a higher order.
--                      More states and coefficients should
be added if increasing
--                      band-pass filter order.
--
-- Dependencies:    IEEE standard library for standard
behaviour.
----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity BPF2channels is
    generic(
```

```
        float_width     : integer := 32   -- 32bits IEEE754
single precision format
        );
    port(
        Clk     : in std_logic;  -- Clock signal
        Reset_N : in std_logic;  -- Low logic asynchronous
reset
        -- Left Channel
        L_Start : in std_logic;   -- Input to start left
channel bandpass algorithm
        L_Rk        :   in   std_logic_vector(float_width-1
downto 0);  -- Left audio signal input
        L_Yk        :   out   std_logic_vector(float_width-1
downto 0) := (others => '0'); -- Left Output
        Ldone       : out std_logic := '0';    -- Output
notification of ready left channel output
        -- Right Channel
        R_Start : in std_logic;   -- Input to start left
channel bandpass algorithm
        R_Rk        :   in   std_logic_vector(float_width-1
downto 0);  -- Rigth audio signal input
        R_Yk        :   out   std_logic_vector(float_width-1
downto 0) := (others => '0'); -- Right Output
        Rdone       : out std_logic := '0'    -- Output
notification of ready right channel output
        );
end BPF2channels;

-- As this BPF module uses a Single Precision format for
math operations, it's high resources
-- dependent. This uses a lots of logic cells, so Xilinx Ise
software will ask for a minimum
-- clock time of about 70 [ns] for this module. If not
provided as asked, this module won't
-- operate as intended.

architecture Logic of BPF2channels is

    -- SingleAddProduct performs the operation N1+(N2*N3)
    component SingleAddProduct is
        port(
            N1      :   in   std_logic_vector(float_width-1
downto 0);
            N2      :   in   std_logic_vector(float_width-1
downto 0);
            N3      :   in   std_logic_vector(float_width-1
downto 0);
            R       :   out   std_logic_vector(float_width-1
downto 0)
            );
```

```
    end component;
    -- Further information about this module's behavior in
SingleAddProduct.vhd

    type state is (  -- Create state data type
        Ta,  -- Initial State

    TbL,Tc1L,Tc2L,Tc3L,Tc4L,Td1L,Td2L,Td3L,Td4L,Te1L,Te2L,T
e3L,Te4L,

    Tf1L,Tf2L,Tf3L,Tf4L,Tg1L,Tg2L,Tg3L,Tg4L,Th1L,Th2L,Th3L,
Th4L,
        Ti1L,Ti2L,Ti3L,Ti4L,Tj1L,Tj2L,Tj3L,Tj4L,TkL,     --
Left channel

    TbR,Tc1R,Tc2R,Tc3R,Tc4R,Td1R,Td2R,Td3R,Td4R,Te1R,Te2R,T
e3R,Te4R,

    Tf1R,Tf2R,Tf3R,Tf4R,Tg1R,Tg2R,Tg3R,Tg4R,Th1R,Th2R,Th3R,
Th4R,
        Ti1R,Ti2R,Ti3R,Ti4R,Tj1R,Tj2R,Tj3R,Tj4R,TkR      --
Right channel
        );
    signal T        : state := Ta;  -- Assing state type to
an internal signal

    -- Internal signals for SingleAddProduct component
    signal N1,N2,N3,R   :    std_logic_vector(float_width-1
downto 0);

    -- Internal signals to store delays
    -- LEFT CHANNEL
    signal L_rkd0,L_rkd1,L_rkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal L_wkd0,L_wkd1,L_wkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal L_pkd0,L_pkd1,L_pkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal L_qkd0,L_qkd1,L_qkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal L_skd0,L_skd1,L_skd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal L_tkd0,L_tkd1,L_tkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal L_ukd0,L_ukd1,L_ukd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal L_vkd0,L_vkd1,L_vkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal L_ykd0,L_ykd1,L_ykd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
```

```
    -- RIGHT CHANNEL
    signal R_rkd0,R_rkd1,R_rkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal R_wkd0,R_wkd1,R_wkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal R_pkd0,R_pkd1,R_pkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal R_qkd0,R_qkd1,R_qkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal R_skd0,R_skd1,R_skd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal R_tkd0,R_tkd1,R_tkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal R_ukd0,R_ukd1,R_ukd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal R_vkd0,R_vkd1,R_vkd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    signal R_ykd0,R_ykd1,R_ykd2 :
std_logic_vector(float_width-1 downto 0) := (others => '0');
    -- xkdi format stands for x[k-i] to represent delays
    -- As a 4th-order BPF has two stages, w[k] is added as
the first stage output
    -- w[k] will be second stage input too.

    -- Each stage output follows the equation:
    --  q[k] = an_0*d[k] + an_1*d[k-1] + an_2*d[k-2] -
bn_1*q[k-1] - bn_2*q[k-2]
    --  Where q: stage output, d: stage input, n: stage
number.
    -- an correspond to numerator coefficients vector of
q[k]/d[k] of nth-stage
    -- bn correspond to denominator coefficients vector of
q[k]/d[k] of nth-stage
    -- bn_0 must be 1 as it is the output q[k] coefficient.

    -- Advise: To keep the SingleAddProduct module with no
modifications, denominator
    -- coefficients bn_1 and bn_2 were negated in design so
the output equation will
    -- now follow:
    --  q[k] = an_0*d[k] + an_1*d[k-1] + an_2*d[k-2] + (-
bn_1*q[k-1]) + (-bn_2*q[k-2])

    -- THE FOLLOWING CONSTANT COEFFICIENTS MATCH FOR A 16TH
ORDER BAND-PASS FILTER WITH
    --  LOW CUTOFF FREQUENCY OF 100 Hz AND HIGH CUTOFF
FREQUENCY OF 3 KHz.
    -- DON'T FORGET TO REPLACES VALUES IF DIFFERENT CUTOFF
FREQUENCIES ARE DESIRED.
```

```
    -- Coefficients are declared as constants as their values
won't change.
    -- Stage 1
    constant a1_0   : std_logic_vector(float_width-1 downto
0) := "00111101101110110110110011010111";
    constant a1_2   : std_logic_vector(float_width-1 downto
0) := "10111101101110110110110011010111";
    constant b1_1   : std_logic_vector(float_width-1 downto
0) := "00111111111100101100010001100011";
    constant b1_2   : std_logic_vector(float_width-1 downto
0) := "10111111011011101010111000100001";
    -- Stage 2
    constant a2_0   : std_logic_vector(float_width-1 downto
0) := "00111101101110110110110011010111";
    constant a2_2   : std_logic_vector(float_width-1 downto
0) := "10111101101110110110110011010111";
    constant b2_1   : std_logic_vector(float_width-1 downto
0) := "00111111111111111101100010110110l";
    constant b2_2   : std_logic_vector(float_width-1 downto
0) := "10111111011111111011001011001010O";
    -- Stage 3
    constant a3_0   : std_logic_vector(float_width-1 downto
0) := "00111101101101011010011110111100";
    constant a3_2   : std_logic_vector(float_width-1 downto
0) := "10111101101101011010011110111100";
    constant b3_1   : std_logic_vector(float_width-1 downto
0) := "00111111111111110001101111111110";
    constant b3_2   : std_logic_vector(float_width-1 downto
0) := "10111111011111110001110101100l011";
    -- Stage 4
    constant a4_0   : std_logic_vector(float_width-1 downto
0) := "00111101101101011010011110111100";
    constant a4_2   : std_logic_vector(float_width-1 downto
0) := "10111101101101011010011110111100";
    constant b4_1   : std_logic_vector(float_width-1 downto
0) := "00111111111001001010010001111111";
    constant b4_2   : std_logic_vector(float_width-1 downto
0) := "10111111010100011001110011001101";
    -- Stage 5
    constant a5_0   : std_logic_vector(float_width-1 downto
0) := "00111101101100011001010100000111";
    constant a5_2   : std_logic_vector(float_width-1 downto
0) := "10111101101100011001010100000111";
    constant b5_1   : std_logic_vector(float_width-1 downto
0) := "00111111111111101001100111111100";
    constant b5_2   : std_logic_vector(float_width-1 downto
0) := "10111111011111101001101l011101100";
    -- Stage 6
    constant a6_0   : std_logic_vector(float_width-1 downto
0) := "00111101101100011001010100000111";
```

```
    constant a6_2   : std_logic_vector(float_width-1 downto
0) := "10111101101100011001010100000111";
    constant b6_1   : std_logic_vector(float_width-1 downto
0) := "00111111110110110000101101100101";
    constant b6_2   : std_logic_vector(float_width-1 downto
0) := "10111111001111011010010100010010";
    -- Stage 7
    constant a7_0   : std_logic_vector(float_width-1 downto
0) := "00111101101011110111110100101110";
    constant a7_2   : std_logic_vector(float_width-1 downto
0) := "10111101101011110111110100101110";
    constant b7_1   : std_logic_vector(float_width-1 downto
0) := "00111111111111100100100000011110";
    constant b7_2   : std_logic_vector(float_width-1 downto
0) := "10111111011111001001001101010011";
    -- Stage 8
    constant a8_0   : std_logic_vector(float_width-1 downto
0) := "00111101101011110111110100101110";
    constant a8_2   : std_logic_vector(float_width-1 downto
0) := "10111101101011110111110100101110";
    constant b8_1   : std_logic_vector(float_width-1 downto
0) := "00111111110101100100001010010001";
    constant b8_2   : std_logic_vector(float_width-1 downto
0) := "10111111001100111001011010011111";
    --  (*)  Keep  in  mind  denominator  coefficients  were
previously negated and these
    --     negated values were the ones which were declared
as constants.
    -- (*) an_1 was equal to zero in all stages so is not
included
    -- (*) bn_1 must be equal to 1 as it corresponds to stage
output.

begin

    process(Clk,Reset_N)      --   Sensitivity   list   for
synchronous sequential logic
    begin
        if Reset_N = '0' then   -- Asynchronous low logic
reset
            T <= Ta;                    -- Returns to first
stage
            -- CLEAR LEFT CHANNEL
            L_Yk <= (others => '0');    -- Clear BPF output
            Ldone <= '0';                   -- Clear notify
output
            L_rkd0 <= (others => '0');  -- Clear internal
r[k] signal
            L_rkd1 <= (others => '0');  -- Clear internal
r[k-1] signal
```

```
            L_rkd2 <= (others => '0');   -- Clear internal
r[k-2] signal
            L_wkd0 <= (others => '0');   -- Clear internal
w[k] signal
            L_wkd1 <= (others => '0');   -- Clear internal
w[k-1] signal
            L_wkd2 <= (others => '0');   -- Clear internal
w[k-2] signal
            L_pkd0 <= (others => '0');   -- Clear internal
p[k] signal
            L_pkd1 <= (others => '0');   -- Clear internal
p[k-1] signal
            L_pkd2 <= (others => '0');   -- Clear internal
p[k-2] signal
            L_qkd0 <= (others => '0');   -- Clear internal
q[k] signal
            L_qkd1 <= (others => '0');   -- Clear internal
q[k-1] signal
            L_qkd2 <= (others => '0');   -- Clear internal
q[k-2] signal
            L_skd0 <= (others => '0');   -- Clear internal
s[k] signal
            L_skd1 <= (others => '0');   -- Clear internal
s[k-1] signal
            L_skd2 <= (others => '0');   -- Clear internal
s[k-2] signal
            L_tkd0 <= (others => '0');   -- Clear internal
t[k] signal
            L_tkd1 <= (others => '0');   -- Clear internal
t[k-1] signal
            L_tkd2 <= (others => '0');   -- Clear internal
t[k-2] signal
            L_ukd0 <= (others => '0');   -- Clear internal
u[k] signal
            L_ukd1 <= (others => '0');   -- Clear internal
u[k-1] signal
            L_ukd2 <= (others => '0');   -- Clear internal
u[k-2] signal
            L_vkd0 <= (others => '0');   -- Clear internal
v[k] signal
            L_vkd1 <= (others => '0');   -- Clear internal
v[k-1] signal
            L_vkd2 <= (others => '0');   -- Clear internal
v[k-2] signal
            L_ykd0 <= (others => '0');   -- Clear internal
y[k] signal
            L_ykd1 <= (others => '0');   -- Clear internal
y[k-1] signal
            L_ykd2 <= (others => '0');   -- Clear internal
y[k-2] signal
```

```
            -- CLEAR RIGHT CHANNEL
            R_Yk <= (others => '0');    -- Clear BPF output
            Rdone <= '0';                  -- Clear notify
output
            R_rkd0 <= (others => '0');   -- Clear internal
r[k] signal
            R_rkd1 <= (others => '0');   -- Clear internal
r[k-1] signal
            R_rkd2 <= (others => '0');   -- Clear internal
r[k-2] signal
            R_wkd0 <= (others => '0');   -- Clear internal
w[k] signal
            R_wkd1 <= (others => '0');   -- Clear internal
w[k-1] signal
            R_wkd2 <= (others => '0');   -- Clear internal
w[k-2] signal
            R_pkd0 <= (others => '0');   -- Clear internal
p[k] signal
            R_pkd1 <= (others => '0');   -- Clear internal
p[k-1] signal
            R_pkd2 <= (others => '0');   -- Clear internal
p[k-2] signal
            R_qkd0 <= (others => '0');   -- Clear internal
q[k] signal
            R_qkd1 <= (others => '0');   -- Clear internal
q[k-1] signal
            R_qkd2 <= (others => '0');   -- Clear internal
q[k-2] signal
            R_skd0 <= (others => '0');   -- Clear internal
s[k] signal
            R_skd1 <= (others => '0');   -- Clear internal
s[k-1] signal
            R_skd2 <= (others => '0');   -- Clear internal
s[k-2] signal
            R_tkd0 <= (others => '0');   -- Clear internal
t[k] signal
            R_tkd1 <= (others => '0');   -- Clear internal
t[k-1] signal
            R_tkd2 <= (others => '0');   -- Clear internal
t[k-2] signal
            R_ukd0 <= (others => '0');   -- Clear internal
u[k] signal
            R_ukd1 <= (others => '0');   -- Clear internal
u[k-1] signal
            R_ukd2 <= (others => '0');   -- Clear internal
u[k-2] signal
            R_vkd0 <= (others => '0');   -- Clear internal
v[k] signal
            R_vkd1 <= (others => '0');   -- Clear internal
v[k-1] signal
```

```
            R_vkd2 <= (others => '0');   -- Clear internal
v[k-2] signal
            R_ykd0 <= (others => '0');   -- Clear internal
y[k] signal
            R_ykd1 <= (others => '0');   -- Clear internal
y[k-1] signal
            R_ykd2 <= (others => '0');   -- Clear internal
y[k-2] signal
            -- Add or delete signals clearing if desired
filter is different.
        elsif Clk'event and Clk = '1' then  -- Synchronous
behaviour
            Ldone <= '0';   -- Keep notify output to low
unless left output data is ready
            Rdone <= '0';   -- Keep notify output to low
unless right output data is ready
            case T is       -- Synchronous sequential logic
behaviour
                when Ta =>  -- Initial State
                    if L_Start = '1' then
                        T <= TbL;   -- To next state after
receiving Start signal (Left channel)
                    elsif R_Start = '1' then
                        T <= TbR;   -- To next state after
receiving Start signal (Right channel)
                    else
                        T <= Ta;    -- Keep on same state if
not Start signal yet
                    end if;
                -- LEFT CHANNEL BPF
                when TbL =>  -- Swaping Delay State
                    T <= Tc1L;  -- To next state
                    -- R[k] delay and input
                    L_rkd2 <= L_rkd1;  -- r[k-2] <= r[k-1]
                    L_rkd1 <= L_rkd0;  -- r[k-1] <= r[k]
                    L_rkd0  <=  L_Rk;       --  r[k]  <=  R[k]
(Input)
                    -- W[k] delay
                    L_wkd2 <= L_wkd1;  -- w[k-2] <= w[k-1]
                    L_wkd1 <= L_wkd0;  -- w[k-1] <= w[k]
                    -- P[k] delay
                    L_pkd2 <= L_pkd1;  -- p[k-2] <= p[k-1]
                    L_pkd1 <= L_pkd0;  -- p[k-1] <= p[k]
                    -- Q[k] delay
                    L_qkd2 <= L_qkd1;  -- q[k-2] <= q[k-1]
                    L_qkd1 <= L_qkd0;  -- q[k-1] <= q[k]
                    -- S[k] delay
                    L_skd2 <= L_skd1;  -- s[k-2] <= s[k-1]
                    L_skd1 <= L_skd0;  -- s[k-1] <= s[k]
                    -- T[k] delay
```

```
                    L_tkd2 <= L_tkd1;  -- t[k-2] <= t[k-1]
                    L_tkd1 <= L_tkd0;  -- t[k-1] <= t[k]
                    -- U[k] delay
                    L_ukd2 <= L_ukd1;  -- u[k-2] <= u[k-1]
                    L_ukd1 <= L_ukd0;  -- u[k-1] <= u[k]
                    -- V[k] delay
                    L_vkd2 <= L_vkd1;  -- v[k-2] <= v[k-1]
                    L_vkd1 <= L_vkd0;  -- v[k-1] <= v[k]
                    -- Y[k] delay
                    L_ykd2 <= L_ykd1;  -- y[k-2] <= y[k-1]
                    L_ykd1 <= L_ykd0;  -- y[k-1] <= y[k]
                -- From here, all states will calculate R =
N1 + (N2*N3) recursively
                -- Each stage takes 5 clock times including
output saving
                -- Last clock of a stage is shared with first
clock of next stage
                when Tc1L =>
                    T <= Tc2L;  -- To next state
                    -- Stage 1: Numerator
                    N1 <= (others => '0');
                    N2 <= a1_0;
                    N3 <= L_rkd0;
                    -- Performs R = a1_0 * r[k]
                when Tc2L =>
                    T <= Tc3L;  -- To next state
                    N1 <= R;
                    N2 <= a1_2;
                    N3 <= L_rkd2;
                    -- Performs R = previous_R + a1_2 * r[k-
2]
                when Tc3L =>
                    T <= Tc4L;  -- To next state
                    -- Stage 1: Denominator
                    N1 <= R;
                    N2 <= b1_1;
                    N3 <= L_wkd1;
                    -- Performs R = previous_R + (-b1_1) *
w[k-1]
                when Tc4L =>
                    T <= Td1L;  -- To next state
                    N1 <= R;
                    N2 <= b1_2;
                    N3 <= L_wkd2;
                    -- Performs R = previous_R + (-b1_2) *
w[k-2]
                when Td1L =>
                    T <= Td2L;  -- To next state
                    -- Save previous_R to wkd0 as w[k] =
previous_R
```

```
                    -- When solving previous_R you'd get:
                    -- w[k] = a1_0 * r[k] + a1_2 * r[k-2] +
(-b1_1) * w[k-1] + (-b1_2) * w[k-2]
                    L_wkd0 <= R;
                    -- Stage 2: Numerator
                    N1 <= (others => '0');
                    N2 <= a2_0;
                    N3 <= R;  -- same as w[k] or wkd0
                    -- Performs R = a2_0 * w[k]
                when Td2L =>
                    T <= Td3L;  -- To next state
                    N1 <= R;
                    N2 <= a2_2;
                    N3 <= L_wkd2;
                    -- Performs R = previous_R + a2_2 * w[k-
2]
                when Td3L =>
                    T <= Td4L;  -- To next state
                    -- Stage 2: Denominator
                    N1 <= R;
                    N2 <= b2_1;
                    N3 <= L_pkd1;
                    -- Performs R = previous_R + (-b2_1) *
p[k-1]
                when Td4L =>
                    T <= Te1L;  -- To next state
                    N1 <= R;
                    N2 <= b2_2;
                    N3 <= L_pkd2;
                    -- Performs R = previous_R + (-b2_2) *
p[k-2]
                when Te1L =>
                    T <= Te2L;  -- To next state
                    -- Save previous_R to pkd0 as p[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- p[k] = a2_0 * w[k] + a2_2 * w[k-2] +
(-b2_1) * p[k-1] + (-b2_2) * p[k-2]
                    L_pkd0 <= R;
                    -- Stage 3: Numerator
                    N1 <= (others => '0');
                    N2 <= a3_0;
                    N3 <= R;  -- same as p[k] or pkd0
                    -- Performs R = (a3_0) * p[k]
                when Te2L =>
                    T <= Te3L;  -- To next state
                    N1 <= R;
                    N2 <= a3_2;
                    N3 <= L_pkd2;
```

```
                    -- Performs R = previous_R + a3_2 * p[k-
2]
                when Te3L =>
                    T <= Te4L;  -- To next state
                    -- Stage 3: Denominator
                    N1 <= R;
                    N2 <= b3_1;
                    N3 <= L_qkd1;
                    -- Performs R = previous_R + (-b3_1) *
q[k-1]
                when Te4L =>
                    T <= Tf1L;  -- To next state
                    N1 <= R;
                    N2 <= b3_2;
                    N3 <= L_qkd2;
                    -- Performs R = previous_R + (-b3_2) *
q[k-2]
                when Tf1L =>
                    T <= Tf2L;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- q[k] = a3_0 * p[k] + a3_2 * p[k-2] +
(-b3_1) * q[k-1] + (-b3_2) * q[k-2]
                    L_qkd0 <= R;
                    -- Stage 4: Numerator
                    N1 <= (others => '0');
                    N2 <= a4_0;
                    N3 <= R;  -- same as q[k] or qkd0
                    -- Performs R = (a4_0) * q[k]
                when Tf2L =>
                    T <= Tf3L;  -- To next state
                    N1 <= R;
                    N2 <= a4_2;
                    N3 <= L_qkd2;
                    -- Performs R = previous_R + a4_2 * q[k-
2]
                when Tf3L =>
                    T <= Tf4L;  -- To next state
                    -- Stage 4: Denominator
                    N1 <= R;
                    N2 <= b4_1;
                    N3 <= L_skd1;
                    -- Performs R = previous_R + (-b4_1) *
s[k-1]
                when Tf4L =>
                    T <= Tg1L;  -- To next state
                    N1 <= R;
                    N2 <= b4_2;
                    N3 <= L_skd2;
```

```
                    -- Performs R = previous_R + (-b4_2) *
s[k-2]
                when Tg1L =>
                    T <= Tg2L;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- s[k] = a4_0 * q[k] + a4_2 * q[k-2] +
(-b4_1) * s[k-1] + (-b4_2) * s[k-2]
                    L_skd0 <= R;
                    -- Stage 5: Numerator
                    N1 <= (others => '0');
                    N2 <= a5_0;
                    N3 <= R;  -- same as s[k] or skd0
                    -- Performs R = (a5_0) * s[k]
                when Tg2L =>
                    T <= Tg3L;  -- To next state
                    N1 <= R;
                    N2 <= a5_2;
                    N3 <= L_skd2;
                    -- Performs R = previous_R + a5_2 * s[k-
2]
                when Tg3L =>
                    T <= Tg4L;  -- To next state
                    -- Stage 5: Denominator
                    N1 <= R;
                    N2 <= b5_1;
                    N3 <= L_tkd1;
                    -- Performs R = previous_R + (-b5_1) *
t[k-1]
                when Tg4L =>
                    T <= Th1L;  -- To next state
                    N1 <= R;
                    N2 <= b5_2;
                    N3 <= L_tkd2;
                    -- Performs R = previous_R + (-b5_2) *
t[k-2]
                when Th1L =>
                    T <= Th2L;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- t[k] = a5_0 * s[k] + a5_2 * s[k-2] +
(-b5_1) * t[k-1] + (-b5_2) * t[k-2]
                    L_tkd0 <= R;
                    -- Stage 6: Numerator
                    N1 <= (others => '0');
                    N2 <= a6_0;
                    N3 <= R;  -- same as t[k] or tkd0
                    -- Performs R = (a6_0) * t[k]
```

```
                when Th2L =>
                    T <= Th3L;  -- To next state
                    N1 <= R;
                    N2 <= a6_2;
                    N3 <= L_tkd2;
                    -- Performs R = previous_R + a6_2 * t[k-
2]
                when Th3L =>
                    T <= Th4L;  -- To next state
                    -- Stage 6: Denominator
                    N1 <= R;
                    N2 <= b6_1;
                    N3 <= L_ukd1;
                    -- Performs R = previous_R + (-b6_1) *
u[k-1]
                when Th4L =>
                    T <= Ti1L;  -- To next state
                    N1 <= R;
                    N2 <= b6_2;
                    N3 <= L_ukd2;
                    -- Performs R = previous_R + (-b6_2) *
u[k-2]
                when Ti1L =>
                    T <= Ti2L;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- u[k] = a6_0 * t[k] + a6_2 * t[k-2] +
(-b6_1) * u[k-1] + (-b6_2) * u[k-2]
                    L_ukd0 <= R;
                    -- Stage 7: Numerator
                    N1 <= (others => '0');
                    N2 <= a7_0;
                    N3 <= R;  -- same as u[k] or ukd0
                    -- Performs R = (a7_0) * u[k]
                when Ti2L =>
                    T <= Ti3L;  -- To next state
                    N1 <= R;
                    N2 <= a7_2;
                    N3 <= L_ukd2;
                    -- Performs R = previous_R + a7_2 * u[k-
2]
                when Ti3L =>
                    T <= Ti4L;  -- To next state
                    -- Stage 7: Denominator
                    N1 <= R;
                    N2 <= b7_1;
                    N3 <= L_vkd1;
                    -- Performs R = previous_R + (-b7_1) *
v[k-1]
```

```
                when Ti4L =>
                    T <= Tj1L;  -- To next state
                    N1 <= R;
                    N2 <= b7_2;
                    N3 <= L_vkd2;
                    -- Performs R = previous_R + (-b7_2) *
v[k-2]
                when Tj1L =>
                    T <= Tj2L;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- v[k] = a7_0 * u[k] + a7_2 * u[k-2] +
(-b7_1) * v[k-1] + (-b7_2) * v[k-2]
                    L_vkd0 <= R;
                    -- Stage 8: Numerator
                    N1 <= (others => '0');
                    N2 <= a8_0;
                    N3 <= R;  -- same as v[k] or vkd0
                    -- Performs R = a8_0 * v[k]
                when Tj2L =>
                    T <= Tj3L;  -- To next state
                    N1 <= R;
                    N2 <= a8_2;
                    N3 <= L_vkd2;
                    -- Performs R = previous_R + a8_2 * v[k-
2]
                when Tj3L =>
                    T <= Tj4L;  -- To next state
                    -- Stage 8: Denominator
                    N1 <= R;
                    N2 <= b8_1;
                    N3 <= L_ykd1;
                    -- Performs R = previous_R + (-b8_1) *
y[k-1]
                when Tj4L =>
                    T <= TkL;  -- To next state
                    N1 <= R;
                    N2 <= b8_2;
                    N3 <= L_ykd2;
                    -- Performs R = previous_R + (-b8_2) *
y[k-2]
                when TkL =>
                    T <= Ta;  -- To initial state
                    -- Save previous_R to ykd0 as y[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- y[k] = a8_0 * v[k] + a8_2 * v[k-2] +
(-b8_1) * y[k-1] + (-b8_2) * y[k-2]
                    L_ykd0 <= R;
```

```
                -- If there is no more stages, BPF output is
ready
                    L_Yk <= R;      --  Send  previous_R  =
y[k] to output Yk
                    Ldone <= '1';   -- Notify Yk output is
ready with 'done' signal
                -- RIGHT CHANNEL BPF
                when TbR =>  -- Swaping Delay State
                    T <= Tc1R;  -- To next state
                    -- R[k] delay and input
                    R_rkd2 <= R_rkd1;  -- r[k-2] <= r[k-1]
                    R_rkd1 <= R_rkd0;  -- r[k-1] <= r[k]
                    R_rkd0  <=  R_Rk;       --  r[k]  <=  R[k]
(Input)
                    -- W[k] delay
                    R_wkd2 <= R_wkd1;  -- w[k-2] <= w[k-1]
                    R_wkd1 <= R_wkd0;  -- w[k-1] <= w[k]
                    -- P[k] delay
                    R_pkd2 <= R_pkd1;  -- p[k-2] <= p[k-1]
                    R_pkd1 <= R_pkd0;  -- p[k-1] <= p[k]
                    -- Q[k] delay
                    R_qkd2 <= R_qkd1;  -- q[k-2] <= q[k-1]
                    R_qkd1 <= R_qkd0;  -- q[k-1] <= q[k]
                    -- S[k] delay
                    R_skd2 <= R_skd1;  -- s[k-2] <= s[k-1]
                    R_skd1 <= R_skd0;  -- s[k-1] <= s[k]
                    -- T[k] delay
                    R_tkd2 <= R_tkd1;  -- t[k-2] <= t[k-1]
                    R_tkd1 <= R_tkd0;  -- t[k-1] <= t[k]
                    -- U[k] delay
                    R_ukd2 <= R_ukd1;  -- u[k-2] <= u[k-1]
                    R_ukd1 <= R_ukd0;  -- u[k-1] <= u[k]
                    -- V[k] delay
                    R_vkd2 <= R_vkd1;  -- v[k-2] <= v[k-1]
                    R_vkd1 <= R_vkd0;  -- v[k-1] <= v[k]
                    -- Y[k] delay
                    R_ykd2 <= R_ykd1;  -- y[k-2] <= y[k-1]
                    R_ykd1 <= R_ykd0;  -- y[k-1] <= y[k]
                -- From here, all states will calculate R =
N1 + (N2*N3) recursively
                -- Each stage takes 5 clock times including
output saving
                -- Last clock of a stage is shared with first
clock of next stage
                when Tc1R =>
                    T <= Tc2R;  -- To next state
                    -- Stage 1: Numerator
                    N1 <= (others => '0');
                    N2 <= a1_0;
                    N3 <= R_rkd0;
```

```
                    -- Performs R = a1_0 * r[k]
                when Tc2R =>
                    T <= Tc3R;  -- To next state
                    N1 <= R;
                    N2 <= a1_2;
                    N3 <= R_rkd2;
                    -- Performs R = previous_R + a1_2 * r[k-
2]
                when Tc3R =>
                    T <= Tc4R;  -- To next state
                    -- Stage 1: Denominator
                    N1 <= R;
                    N2 <= b1_1;
                    N3 <= R_wkd1;
                    -- Performs R = previous_R + (-b1_1) *
w[k-1]
                when Tc4R =>
                    T <= Td1R;  -- To next state
                    N1 <= R;
                    N2 <= b1_2;
                    N3 <= R_wkd2;
                    -- Performs R = previous_R + (-b1_2) *
w[k-2]
                when Td1R =>
                    T <= Td2R;  -- To next state
                    -- Save previous_R to wkd0 as w[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- w[k] = a1_0 * r[k] + a1_2 * r[k-2] +
(-b1_1) * w[k-1] + (-b1_2) * w[k-2]
                    R_wkd0 <= R;
                    -- Stage 2: Numerator
                    N1 <= (others => '0');
                    N2 <= a2_0;
                    N3 <= R;  -- same as w[k] or wkd0
                    -- Performs R = a2_0 * w[k]
                when Td2R =>
                    T <= Td3R;  -- To next state
                    N1 <= R;
                    N2 <= a2_2;
                    N3 <= R_wkd2;
                    -- Performs R = previous_R + a2_2 * w[k-
2]
                when Td3R =>
                    T <= Td4R;  -- To next state
                    -- Stage 2: Denominator
                    N1 <= R;
                    N2 <= b2_1;
                    N3 <= R_pkd1;
```

```
                    -- Performs R = previous_R + (-b2_1) *
p[k-1]
                when Td4R =>
                    T <= Te1R;  -- To next state
                    N1 <= R;
                    N2 <= b2_2;
                    N3 <= R_pkd2;
                    -- Performs R = previous_R + (-b2_2) *
p[k-2]
                when Te1R =>
                    T <= Te2R;  -- To next state
                    -- Save previous_R to pkd0 as p[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- p[k] = a2_0 * w[k] + a2_2 * w[k-2] +
(-b2_1) * p[k-1] + (-b2_2) * p[k-2]
                    R_pkd0 <= R;
                    -- Stage 3: Numerator
                    N1 <= (others => '0');
                    N2 <= a3_0;
                    N3 <= R;  -- same as p[k] or pkd0
                    -- Performs R = (a3_0) * p[k]
                when Te2R =>
                    T <= Te3R;  -- To next state
                    N1 <= R;
                    N2 <= a3_2;
                    N3 <= R_pkd2;
                    -- Performs R = previous_R + a3_2 * p[k-
2]
                when Te3R =>
                    T <= Te4R;  -- To next state
                    -- Stage 3: Denominator
                    N1 <= R;
                    N2 <= b3_1;
                    N3 <= R_qkd1;
                    -- Performs R = previous_R + (-b3_1) *
q[k-1]
                when Te4R =>
                    T <= Tf1R;  -- To next state
                    N1 <= R;
                    N2 <= b3_2;
                    N3 <= R_qkd2;
                    -- Performs R = previous_R + (-b3_2) *
q[k-2]
                when Tf1R =>
                    T <= Tf2R;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
```

```
                    -- q[k] = a3_0 * p[k] + a3_2 * p[k-2] +
(-b3_1) * q[k-1] + (-b3_2) * q[k-2]
                    R_qkd0 <= R;
                    -- Stage 4: Numerator
                    N1 <= (others => '0');
                    N2 <= a4_0;
                    N3 <= R;  -- same as q[k] or qkd0
                    -- Performs R = (a4_0) * q[k]
                when Tf2R =>
                    T <= Tf3R;  -- To next state
                    N1 <= R;
                    N2 <= a4_2;
                    N3 <= R_qkd2;
                    -- Performs R = previous_R + a4_2 * q[k-
2]
                when Tf3R =>
                    T <= Tf4R;  -- To next state
                    -- Stage 4: Denominator
                    N1 <= R;
                    N2 <= b4_1;
                    N3 <= R_skd1;
                    -- Performs R = previous_R + (-b4_1) *
s[k-1]
                when Tf4R =>
                    T <= Tg1R;  -- To next state
                    N1 <= R;
                    N2 <= b4_2;
                    N3 <= R_skd2;
                    -- Performs R = previous_R + (-b4_2) *
s[k-2]
                when Tg1R =>
                    T <= Tg2R;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- s[k] = a4_0 * q[k] + a4_2 * q[k-2] +
(-b4_1) * s[k-1] + (-b4_2) * s[k-2]
                    R_skd0 <= R;
                    -- Stage 5: Numerator
                    N1 <= (others => '0');
                    N2 <= a5_0;
                    N3 <= R;  -- same as s[k] or skd0
                    -- Performs R = (a5_0) * s[k]
                when Tg2R =>
                    T <= Tg3R;  -- To next state
                    N1 <= R;
                    N2 <= a5_2;
                    N3 <= R_skd2;
                    -- Performs R = previous_R + a5_2 * s[k-
2]
```

```
                when Tg3R =>
                    T <= Tg4R;  -- To next state
                    -- Stage 5: Denominator
                    N1 <= R;
                    N2 <= b5_1;
                    N3 <= R_tkd1;
                    -- Performs R = previous_R + (-b5_1) *
t[k-1]
                when Tg4R =>
                    T <= Th1R;  -- To next state
                    N1 <= R;
                    N2 <= b5_2;
                    N3 <= R_tkd2;
                    -- Performs R = previous_R + (-b5_2) *
t[k-2]
                when Th1R =>
                    T <= Th2R;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- t[k] = a5_0 * s[k] + a5_2 * s[k-2] +
(-b5_1) * t[k-1] + (-b5_2) * t[k-2]
                    R_tkd0 <= R;
                    -- Stage 6: Numerator
                    N1 <= (others => '0');
                    N2 <= a6_0;
                    N3 <= R;  -- same as t[k] or tkd0
                    -- Performs R = (a6_0) * t[k]
                when Th2R =>
                    T <= Th3R;  -- To next state
                    N1 <= R;
                    N2 <= a6_2;
                    N3 <= R_tkd2;
                    -- Performs R = previous_R + a6_2 * t[k-
2]
                when Th3R =>
                    T <= Th4R;  -- To next state
                    -- Stage 6: Denominator
                    N1 <= R;
                    N2 <= b6_1;
                    N3 <= R_ukd1;
                    -- Performs R = previous_R + (-b6_1) *
u[k-1]
                when Th4R =>
                    T <= Ti1R;  -- To next state
                    N1 <= R;
                    N2 <= b6_2;
                    N3 <= R_ukd2;
                    -- Performs R = previous_R + (-b6_2) *
u[k-2]
```

```
                when Ti1R =>
                    T <= Ti2R;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- u[k] = a6_0 * t[k] + a6_2 * t[k-2] +
(-b6_1) * u[k-1] + (-b6_2) * u[k-2]
                    R_ukd0 <= R;
                    -- Stage 7: Numerator
                    N1 <= (others => '0');
                    N2 <= a7_0;
                    N3 <= R;  -- same as u[k] or ukd0
                    -- Performs R = (a7_0) * u[k]
                when Ti2R =>
                    T <= Ti3R;  -- To next state
                    N1 <= R;
                    N2 <= a7_2;
                    N3 <= R_ukd2;
                    -- Performs R = previous_R + a7_2 * u[k-
2]
                when Ti3R =>
                    T <= Ti4R;  -- To next state
                    -- Stage 7: Denominator
                    N1 <= R;
                    N2 <= b7_1;
                    N3 <= R_vkd1;
                    -- Performs R = previous_R + (-b7_1) *
v[k-1]
                when Ti4R =>
                    T <= Tj1R;  -- To next state
                    N1 <= R;
                    N2 <= b7_2;
                    N3 <= R_vkd2;
                    -- Performs R = previous_R + (-b7_2) *
v[k-2]
                when Tj1R =>
                    T <= Tj2R;  -- To next state
                    -- Save previous_R to qkd0 as q[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- v[k] = a7_0 * u[k] + a7_2 * u[k-2] +
(-b7_1) * v[k-1] + (-b7_2) * v[k-2]
                    R_vkd0 <= R;
                    -- Stage 8: Numerator
                    N1 <= (others => '0');
                    N2 <= a8_0;
                    N3 <= R;  -- same as v[k] or vkd0
                    -- Performs R = a8_0 * v[k]
                when Tj2R =>
                    T <= Tj3R;  -- To next state
```

```
                    N1 <= R;
                    N2 <= a8_2;
                    N3 <= R_vkd2;
                    -- Performs R = previous_R + a8_2 * v[k-
2]
                when Tj3R =>
                    T <= Tj4R;  -- To next state
                    -- Stage 8: Denominator
                    N1 <= R;
                    N2 <= b8_1;
                    N3 <= R_ykd1;
                    -- Performs R = previous_R + (-b8_1) *
y[k-1]
                when Tj4R =>
                    T <= TkR;  -- To next state
                    N1 <= R;
                    N2 <= b8_2;
                    N3 <= R_ykd2;
                    -- Performs R = previous_R + (-b8_2) *
y[k-2]
                when TkR =>
                    T <= Ta;  -- To initial state
                    -- Save previous_R to ykd0 as y[k] =
previous_R
                    -- When solving previous_R you'd get:
                    -- y[k] = a8_0 * v[k] + a8_2 * v[k-2] +
(-b8_1) * y[k-1] + (-b8_2) * y[k-2]
                    R_ykd0 <= R;
                -- If there is no more stages, BPF output is
ready
                    R_Yk <= R;     -- Send previous_R = y[k]
to output Yk
                    Rdone <= '1';   -- Notify Yk output is
ready with 'done' signal
                when others =>
                    T <= Ta;  -- To inital state
            end case;
        end if;
    end process;

    -- Port map SingleAddProduct component to perform R = N1
+ (N2 * N3)
    SingleCalc: SingleAddProduct port map (N1,N2,N3,R);

end Logic;
```

Apéndice F

Módulo convertidor de tipo de dato arreglado con signo a simple precisión.

Nombre del archivo: Fixed2Float.vhd

Descripción: Usado antes de ingresar un dato al filtro pasa-banda para convertir el tipo de dato del receptor I^2S de complemento a la base dos a simple precisión.

```
----------------------------------------------------------------------
-- Escuela Superior Politécnica del Litoral (ESPOL)
-- Facultad de Ingeniería en Electrónica y Computación (FIEC)
-- Final undergraduate project: VHDL Human Voice Filtering
--
-- Degree to obtain: Electronics and Automation Engineer
--
-- Made by: Ángel Román Velóz & Alexis Mora Roca
-- Contact: (+593) 98 817 2232 & (+593) 99 587 0443
--
-- Parent Module:   TopDriver.vhd
-- Module Name: Fixed2Float.vhd
--
-- Description: Convert number from fixed to floating point
format.
-- Comment:         Used  to  convert  I2S  two's  complement
output (receiver)
--                      to  floating  point  and  match  BPF
input format.
--
-- Dependencies:    IEEE  standard  library  for  standard
behaviour.
--                      IEEE_PROPOSED library for floating
point number operations.
----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Dependencies for FLOAT_PKG
use IEEE.NUMERIC_STD.ALL;
library IEEE_PROPOSED;
use IEEE_PROPOSED.FIXED_FLOAT_TYPES.all;
use IEEE_PROPOSED.FIXED_PKG.all;
use IEEE_PROPOSED.FLOAT_PKG.ALL;

entity Fixed2Float is
    generic(
        fixed_width     : integer := 24;  -- 24bits for I2S
two's complement output format
```

```
        float_width     : integer := 32     -- 32bits for
single precision floating number
        );
    port(
        FixedInput      : in std_logic_vector(fixed_width-1
downto 0);  -- From I2S
        FloatOutput     : out std_logic_vector(float_width-
1 downto 0)  -- To BPF
        );
end Fixed2Float;

-- Advise: To use FLOAT_PKG operations, data must match with
the corresponding floating
--         point type specified in package. Input and output
data type were left as
--         STD_LOGIC_VECTOR for better data handling between
modules and to perform
--         simulations as FLOAT_PKG types are not supported.

architecture Logic of Fixed2Float is

    signal Fixed_int    : sfixed(fixed_width-1 downto 0);  -
- Signed fixed with no fraction
    signal Float_int    : float32;                         -
- Single precision IEEE754 format

begin

    Fixed_int <= to_sfixed(FixedInput, fixed_width-1, 0);  -
- Convert input to sfixed
    Float_int <= to_float(Fixed_int);                      -
- Convert sfixed to float
    FloatOutput <= to_slv(Float_int);            -- Convert
float to std_logic_vector

end Logic;
```

Apéndice G

Módulo convertidor de tipo de dato de simple precisión a arreglado con signo.

Nombre del archivo: Float2Fixed.vhd

Descripción: Usado antes de ingresar un dato al transmisor I^2S para convertir el tipo de dato del filtro pasa-banda de simple precisión a complemento a la base dos.

```
----------------------------------------------------------------
-----------------------
-- Escuela Superior Politécnica del Litoral (ESPOL)
-- Facultad de Ingeniería en Electrónica y Computación (FIEC)
-- Final undergraduate project: VHDL Human Voice Filtering
--
-- Degree to obtain: Electronics and Automation Engineer
--
-- Made by: Ángel Román Velóz & Alexis Mora Roca
-- Contact: (+593) 98 817 2232 & (+593) 99 587 0443
--
-- Parent Module:   TopDriver.vhd
-- Module Name: Float2Fixed.vhd
--
-- Description: Convert number from floating point to fixed
format.
-- Comment:         Used  to  convert  BPF  floating  point
output to I2S two's
--                       complement format (transmitter)
--
-- Dependencies:    IEEE  standard  library  for  standard
behaviour.
--                       IEEE_PROPOSED library for floating
point number operations.
----------------------------------------------------------------
-----------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Dependencies for FLOAT_PKG
use IEEE.NUMERIC_STD.ALL;
library IEEE_PROPOSED;
use IEEE_PROPOSED.FIXED_FLOAT_TYPES.all;
use IEEE_PROPOSED.FIXED_PKG.all;
use IEEE_PROPOSED.FLOAT_PKG.ALL;

entity Float2Fixed is
    generic(
        float_width     : integer  :=  32;    --  32bits  for
single precision floating number
```

```
        fixed_width     : integer := 24   -- 24bits for I2S
two's complement output format
        );
    port(
        FloatInput      : in std_logic_vector(float_width-1
downto 0);  -- From BPF
        FixedOutput     : out std_logic_vector(fixed_width-
1 downto 0)  -- To I2S
        );
end Float2Fixed;

-- Advise: To use FLOAT_PKG operations, data must match with
the corresponding floating
--         point type specified in package. Input and output
data type were left as
--         STD_LOGIC_VECTOR for better data handling between
modules and to perform
--         simulations as FLOAT_PKG types are not supported.

architecture Logic of Float2Fixed is

    signal Float_int    : float32;                          -
- Single precision IEEE754 format
    signal Fixed_int    : sfixed(fixed_width-1 downto 0);  -
- Signed fixed with no fraction

begin

    Float_int <= to_float(FloatInput);                     --
Convert input to float
    Fixed_int <= to_sfixed(Float_int, fixed_width-1, 0);  -
- Convert float to sfixed
    FixedOutput <= to_slv(Fixed_int);           -- Convert
sfixed to std_logic_vector

end Logic;
```

ANEXOS

Anexo 1

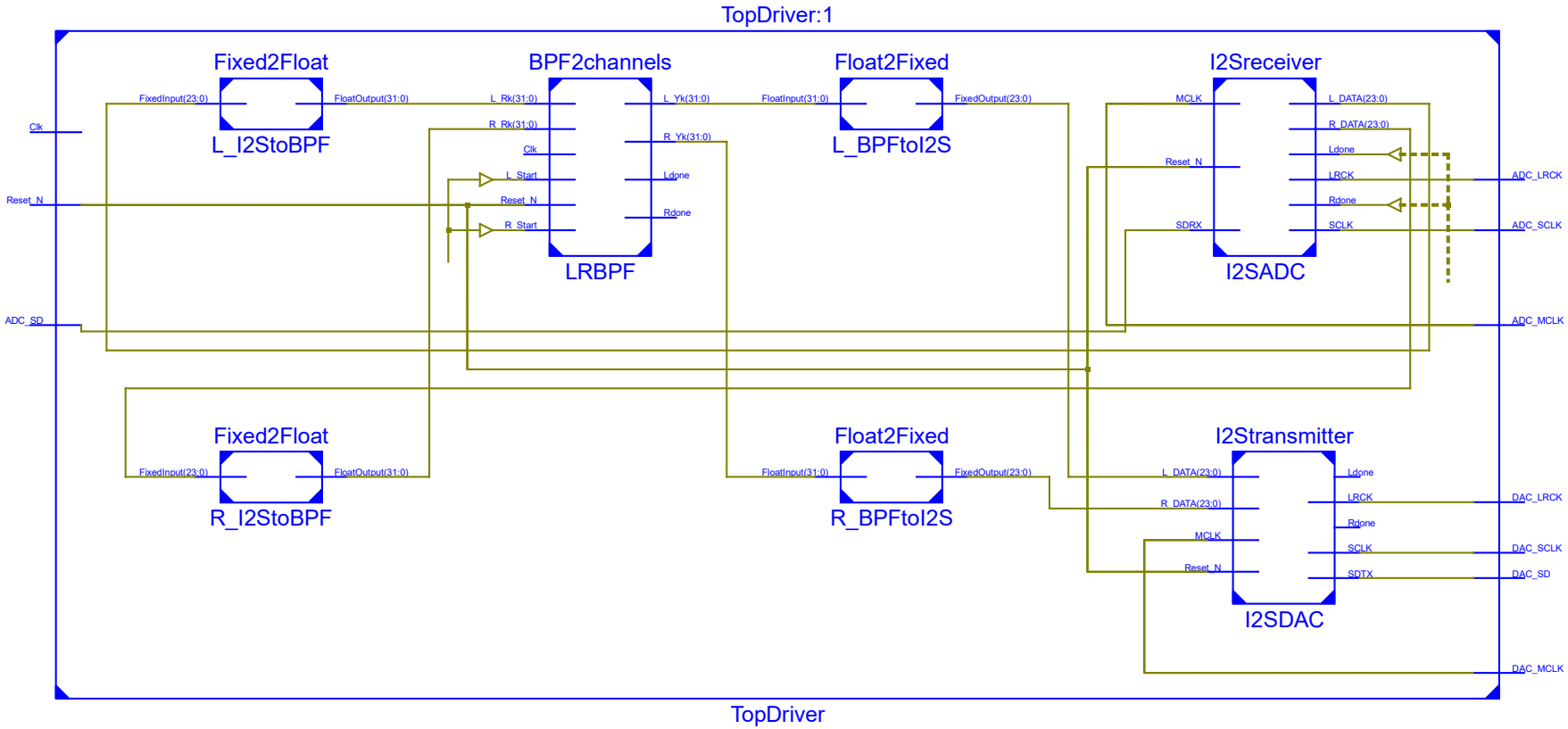

Figura 4.0.1 Diagrama RTL del controlador principal