UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Sintetizador de sonidos con FPGA

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Redondo Padilla, Rafael

Tutor/a: Andrés Martínez, David de

CURSO ACADÉMICO: 2022/2023

Resumen

Este proyecto consiste en el diseño e implementación de un sintetizador de sonidos en *hardware* programable. El diseño está disponible públicamente para que se pueda estudiar y aprovechar en otras implementaciones. El sintetizador digital trata tanto la síntesis aditiva, en la que se combinan dos osciladores con polifonía reproduciendo un número genérico de voces y síntesis sustractiva, incluyendo filtros de paso bajo, alto, de banda y de rechazo de banda mediante la implementación de las ecuaciones de Chamberlin. La funcionalidad se ha extendido para obtener un control mayor sobre la frecuencia, con parámetros para modificar el semitono o ajustar la afinación.

En cada oscilador se puede seleccionar un tipo de onda básica, onda sierra, triangular, cuadrada o senoidal. En la onda cuadrada también se puede modular el ancho de pulso. Además también se ha trabajado en la incorporación del efecto *portamento* y la gestión de la dinámica del sonido, con el generador de envolventes de amplitud. También dispone de tres osciladores de baja frecuencia, uno para cada oscilador donde modulan la frecuencia o *pitch* y otro que modula el parámetro de la frecuencia de corte o resonancia de los filtros. El sintetizador también contiene un procesador para agregar ruido a la señal. Se puede conmutar la entrada del convertidor analógico digital o de los generadores de onda, para capturar una señal externa y procesarla con el generador de ruido más los filtros.

Se ha elegido como placa de prototipado el dispositivo FPGA de AMD-Xilinx Basys-3, por tanto se ha trabajado con el entorno de desarrollo Vivado ML Standard Edition para implementar el modelo, desarrollado con el lenguaje de descripción VHDL. Para reproducir los sonidos hemos utilizado el dispositivo Pmod I2S2, que incorpora convertidores de analógico a digital y de digital a analógico. La frecuencia de muestreo del sintetizador es 44.1 kHz con una resolución de 24 bits, en estéreo.

Por otro lado, se ha desarrollado una aplicación *software* con interfaz gráfica en lenguaje C++ con el *framework* JUCE. Esta aplicación se ha compilado para un entorno con Windows, pero también como *plug-in* VST3 y se ha comprobado su funcionamiento en un DAW. Además esta aplicación puede ser portable a otros sistemas operativos y a otros formatos de *plug-in*. El propósito principal es controlar los parámetros del *hardware* mediante una conexión UART desde la interfaz gráfica. Además la aplicación también contiene un gestor para almacenar los *presets* y se ha trabajado en la sincronización con la FPGA, tanto en la aplicación *standalone* como en el *plug-in* para recuperar adecuadamente el estado de los parámetros. En la aplicación *standalone* también se ha trabajado en la administración mediante un controlador MIDI, útil como piano y para controlar los parámetros de la interfaz.

Palabras clave: sintetizador de sonidos, síntesis aditiva, síntesis sustractiva, dispositivo hardware programable, FPGA, Vivado, VHDL, JUCE, C++

Abstract

This project consists of the design and implementation of a programmable hardware sound synthesizer. The design is publicly available so that it can be studied and leveraged in other implementations. The digital synthesizer deals with both additive synthesis, in which two oscillators are combined with polyphony playing a generic number of voices, and subtractive synthesis, including low-pass, high-pass, bandpass, and band-reject filters by implementing the equations of Chamberlin. The functionality has been extended for more control over frequency, with parameters to change the semitone or adjust the pitch.

In each oscillator you can select a basic wave type, saw wave, triangle, square or sine wave. In the square wave you can also modulate the pulse width. In addition, work has also been done on the incorporation of the portamento effect and the management of sound dynamics, with the amplitude envelope generator. It also has three low frequency oscillators, one for each oscillator where they modulate the frequency or pitch and another that modulates the cutoff frequency or resonance parameter of the filters. The synthesizer also contains a processor to add noise to the signal. The input of the analog-to-digital converter or wave generators can be switched to capture an external signal and process it with the noise generator plus filters.

The AMD-Xilinx Basys-3 FPGA device has been chosen as the prototyping board, therefore we have worked with the Vivado ML Standard Edition development environment to implement the model, developed with the VHDL description language. To reproduce the sounds we have used the Pmod I2S2 device, which incorporates analog-to-digital and digital-to-analog converters. The synthesizer's sample rate is 44.1 kHz with 24 bit resolution, in stereo.

On the other hand, a software application with a graphical interface in C++ language has been developed with the JUCE framework. This application was compiled for a Windows environment, but also as a VST3 plug-in and has been tested to work in a DAW. Furthermore, this application can be portable to other operating systems and other plug-in formats. The main purpose is to control hardware parameters via a UART connection from the graphical interface. In addition, the application also contains a manager to save the presets and work was done on synchronization with the FPGA, both in the standalone application and in the plug-in to adequately recover the status of the parameters. In the standalone application, work has also been done on the administration through a MIDI controller, useful as a piano and to control the interface parameters.

Keywords: sound synthesizer, additive synthesis, subtractive synthesis, programmable hardware device, FPGA, Vivado, VHDL, JUCE, C++

Índice

Índice de figuras

Índice de tablas

Índice de algoritmos

Índice de acrónimos

AAX	Avid Audio Extension
ADC	Analog to Digital Converter
ADSR	Attack, Decay, Sustain and Release
AU	Audio Unit
AMD	Advanced Micro Devices
ARM	Advanced RISC Machine
BRAM	Block RAM
CLB	Configurable Logic Block
COM	Communication Port
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DAW	Digital Audio Workstation
DSP	Digital Signal Processor
FF	Flip Flops
FIR	Finite Impulse Response
FPGA	Field Programable Gate Array
FTDI	Future Technology Devices International
GPU	Graphics Processing Unit
HLS	High Level Synthesis
Hz	Hertz
IIR	Infinite Impulse Response
IoT	Internet Of Things
I2S	Integrated Inter-IC Sound Bus
JTAG	Joint Test Action Group
JUCE	Jules' Utility Class Extensions
LED	Light Emitting Diode
LFO	Low Frequency Oscillator
LRCK	Left Right Clock
LUT	LookUp Tables
LV2	LADSPA Version 2
MAC	Multiply And Accumulate
MCLK	Master Clock
MIDI	Musical Instrument Digital Interface
MMCM	Mixed-Mode Clock Manager
MP3	MPEG-1 Audio Layer 3
MPU	Multiple Process Unit
OpenCL	Open Computing Language
PLL	Phase-Locked Loop
RCA	Radio Corporation of America (plug and jack)
RTL	Register Transfer Level
SCLK	Serial Clock
SRAM	Static Random Acces Memory
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial bus
VCP	Virtual COM Port
VGA	Video Graphics Array
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
VST	Virtual Studio Technology
WNS	Worst Negative Slack
XML	Extensible Markup Language

1. Introducción

La música es un aspecto importante en las personas, sobre todo en algunas culturas, genera emociones, influye en el estado de ánimo, es terapéutica. Es una forma de manifestación artística con orígenes muy antiguos. Inicialmente se utilizaban instrumentos musicales sencillos, que han ido evolucionando en el tiempo. Con la aparición de la música contemporánea, los sintetizadores de audio se utilizan como instrumento musical, sobre todo en la música electrónica. Por otra parte, además de las composiciones musicales también se utilizan para acompañar bandas sonoras, crear efectos especiales para películas, en sonidos para videojuegos, entre otros.

Entendemos por síntesis de sonido cuando generamos sonidos con medios que no son acústicos. En el caso de síntesis analógica, mediante variaciones de voltaje y la síntesis digital consiste en el procesamiento de las formas de onda cuando son producidos por sistemas digitales como puede ser un computador, donde su salida se dirige a un convertidor digital a analógico. Existen diversas técnicas de síntesis, comenzando desde la síntesis sustractiva que fue el primer tipo de síntesis [1] y es muy común en todos los sintetizadores analógicos, donde las señales son filtradas para eliminar las frecuencias no deseadas. La síntesis aditiva consiste en combinar osciladores para generar nuevas formas de onda, donde la utilización de las envolventes para variar la amplitud de onda son muy importantes. Por otra parte, existe la síntesis por modulación de frecuencia, donde se varía la frecuencia de una señal portadora respecto a otra moduladora [2], por lo que se pueden generar señales más complejas con solo dos osciladores [1]. La síntesis por modulación de amplitud es cuando la señal portadora modifica la amplitud de otra señal de sonido. Además existen muchos otros tipos de síntesis tales como la granular, de pulsos, por modulación en anillo, modulación por impulsos codificados, mediante distorsión de fase o con tablas de ondas también conocida como Wavetable o vectorial [3], síntesis espectral y por modelado físico.

Un sintetizador puede generar distintos tonos y timbres, imitar otros tipos de instrumentos y son generalmente controlados con otros dispositivos como los teclados. Los primeros sintetizadores comerciales datan de los finales de los años 60, cuando Robert Moog creó el primer sintetizador portable con nombre Minimoog [2]. A diferencia de los sintetizadores creados hasta el momento, denominados modulares, no había que cablear manualmente módulos para empezar a generar sonidos aunque no ofrecían muchas posibilidades. Durante la década de los años 70, aunque con precios elevados, estos instrumentos comienzan a producirse en masa por marcas muy conocidas hoy en día como son Roland, Korg o Yamaha, con sintetizadores monofónicos, es decir, que solo podían reproducir una voz y que generaban los sonidos por control de voltaje.

Durante esta época comenzaron a utilizarse de forma masiva, pero generalmente todavía no disponían de memoria para almacenar las configuraciones que generan los sonidos. Su evolución continuó con la polifonía y la incorporación de procesadores muy básicos que podían almacenar configuraciones en memoria, como en el Prophet 5. Posteriormente comienza la digitalización de estos instrumentos, con osciladores estabilizados por microprocesadores y los componentes que eran analógicos pasan a transformarse en digitales, como en el Juno 60. A partir de este momento surge el protocolo MIDI (Musical Instruments Digital Interface), un lenguaje que permite comunicar varios dispositivos entre sí para compartir información [8] [9], por ejemplo un teclado electrónico que envía mensajes a un sintetizador. Empiezan a crearse sintetizadores totalmente digitales y aparecen nuevas formas de síntesis, más memoria para el almacenamiento, que se traduce en una mayor capacidad para generar sonidos.

Sobre los años 90 se empiezan a desarrollar procesadores más potentes, con los que se podían crear formas de onda más parecidas a los instrumentos reales y los sintetizadores entonces incluyen funciones de secuenciadores, cajas de ritmos que utilizaban métodos de muestreo con pequeñas pantallas para obtener la información sobre su configuración.

Por otra parte, se empiezan a utilizar la informática para la música, como secuenciadores o controladores MIDI, pero capaces de procesar y generar sonidos con los conocidos emuladores con tecnología VST (Virtual Studio Technology). Se empiezan a utilizar de forma generalizada los DAW (Digital Audio Workstation), que ofrecen las herramientas para trabajar con producciones musicales en un computador. Hoy en día se están volviendo a utilizar los sintetizadores *hardware* y sus usuarios también comienzan a crear sus propios instrumentos musicales digitales, desde sintetizadores modulares hasta emulaciones sobre dispositivos como una MPU (Multiple Process Unit) o últimamente también con diseños en FPGA (Field Programable Gate Array), con una alta capacidad de procesamiento de señales en paralelo a tiempo real, manteniendo latencias muy bajas.

Existe una amplia comunidad de desarrolladores que comparten el diseño de sus instrumentos. Paralelamente, marcas comerciales han relanzando sintetizadores con componentes electrónicos analógicos, pero incluyendo los avances tecnológicos actuales. Pero también han evolucionado con sintetizadores basados en FPGA, aprovechando al máximo su eficiencia para conseguir un alto grado de polifonía, con esquemas complejos, aumentando el número de osciladores, voces, timbres, presentan una elevada complejidad en su desarrollo y un conocimiento del dispositivo y de sus técnicas.

1.1 Motivación

Los sintetizadores como instrumentos musicales han sido una pasión personal desde hace tiempo y aprender cómo éstos funcionan también ha sido una de mis prioridades que he ido estudiando y profundizando durante mi tiempo libre. Además he realizado varios cursos sobre el tratamiento de señales acústicas, masterización de producciones musicales o el diseño de sonidos basados en síntesis digital en emulaciones software. Por otra parte realicé una adquisición de un sintetizador *hardware*, con el que practicando y gracias a los conocimientos que he ido obteniendo durante los cursos he logrado entender mejor las capacidades que ofrecen y su funcionamiento. También he realizado varias producciones de música electrónica que han salido al mercado a través de sellos discográficos. Saber utilizar un sintetizador es esencial a la hora de realizar estas producciones, tanto para agilizar el proceso de desarrollo como para conseguir expresar de forma adecuada lo que se desea transmitir con los sonidos más adecuados.

Además este proyecto supone un reto para aplicar los conocimientos adquiridos durante el grado, principalmente de asignaturas relacionadas con la rama de ingeniería de computadores, como Diseño de Sistemas Digitales, en la que aprendemos a trabajar sobre dispositivos FPGA. Diseñar la implementación del sintetizador ayuda a la preparación para un entorno profesional, trabajando sobre la estructuración y esquema de componentes para conseguir el funcionamiento deseado.

Aprender a programar un instrumento musical sobre un dispositivo *hardware* con VHDL también ayuda a ampliar el pensamiento a la hora de desarrollar el proyecto, desde otro punto de vista al que estamos acostumbrados a trabajar durante el grado, ya que diseñar sobre este lenguaje de descripción supone pensar de forma distinta a cuando programamos con lenguajes de alto nivel, como se destaca en [4]. Además cuando el proyecto finalice, se puede disponer de una base para continuar aprendiendo, aplicando mejoras al proyecto, estudiando otros diseños y experimentando con nuevas funcionalidades.

1.2 Objetivos

El objetivo principal de este proyecto consiste en diseñar e implementar un sintetizador de sonidos en *hardware* programable. El diseño de este instrumento estará disponible públicamente para que cualquier persona pueda estudiarlo, modificarlo o aprovechar alguna de sus funcionalidades para que puedan ser incluidas en otros dispositivos *hardware*.

Para lograr el objetivo principal, se plantea la siguiente subdivisión de objetivos secundarios:

- Estudiar e implementar los tipos de síntesis aditiva y sustractiva, generando varios tipos de onda en dos osciladores. El sintetizador dispondrá de polifonía siendo capaz de reproducir un número genérico de voces. Para tratar el tipo de síntesis sustractiva, se incluyen filtros de tipo paso bajo, alto, de banda y de rechazo de banda.
- Agregar funcionalidad para crear dinamismo en los sonidos, utilizando osciladores de baja frecuencia capaces de modular tanto la frecuencia de las voces en cada oscilador como la frecuencia de corte o resonancia de los filtros y un generador de envolventes de amplitud.
- Programar una aplicación de escritorio con interfaz gráfica, cuyo objetivo principal consista en poder configurar cada uno de los parámetros del dispositivo *hardware*, a través de una comunicación mediante un puerto serie en el computador.
- Ampliar la funcionalidad de esta aplicación y desarrollar un *plug-in* que se pueda ejecutar sobre un DAW como aplicación VST3. Se deberá conseguir mantener la sincronización del estado del dispositivo *hardware* en la aplicación *Standalone*, así como dar la posibilidad de que ésta pueda ser controlada por dispositivos externos como controladores MIDI. Por otra parte, se deberá establecer un mecanismo para que sea posible recuperar configuraciones almacenadas previamente, tanto en la aplicación *Standalone* como en el *plug-in*.

1.3 Estructura del documento

Este documento está estructurado en varias secciones. La sección 1 presenta los sintetizadores como instrumentos para la generación de sonidos y comenta brevemente las distintas formas de síntesis que existen, además describe la evolución que han tenido estos dispositivos a lo largo del tiempo y las tecnologías que se están integrando actualmente en sintetizadores más modernos.

La sección 1.1 trata la motivación, que describe el interés personal en el tema y el motivo que ha llevado a trabajar en este proyecto, los objetivos en la sección 1.2, donde se destaca el objetivo principal y los objetivos secundarios que han sido necesarios para lograr este objetivo principal.

En la sección 2.1 se explica la arquitectura, en el apartado 2.2 el flujo de diseño para FPGA y en la sección 2.3 se describen las aplicaciones. En la sección 2.4 se comenta el estado de FPGA en sintetizadores actuales y en el apartado 2.5 se realiza una revisión sobre los grandes fabricantes para elegir la placa para el prototipado, analizando cuál es el *hardware* más adecuado para lograr los objetivos de este proyecto.

La sección 3.1 documenta la configuración que se ha escogido para los dispositivos que se han utilizado en el proyecto y en las secciones 3.2, 3.3 y 3.4 se describe el diseño y la estructura del sintetizador y del proyecto. En la sección 4 se desarrolla la implementación en cuatro fases y en la sección 5 se presentan los resultados. El documento cierra manifestando las conclusiones que se han ido obteniendo durante la implementación en la sección 6 y por último se presentan posibles ampliaciones y mejoras para trabajos futuros en la sección 7.

2. Arquitectura y aplicaciones de un FPGA

En esta sección se comenta la arquitectura y el flujo de diseño para FPGA, las aplicaciones, las FPGAs en los sintetizadores de la actualidad y la elección de la placa para el prototipado.

2.1 Arquitectura de FPGA

En este apartado se describe en qué consiste la arquitectura de un FPGA, analizando cuál es su estructura en la que se detalla brevemente sus componentes, las fases del flujo de diseño y la situación de FPGA en sintetizadores de la actualidad. Finalmente se presenta nuestra propuesta, donde hablamos del prototipo que se desea realizar y comentamos los fabricantes que existen de placas de prototipado, analizando y determinando el *hardware* más adecuado para conseguir los objetivos de este proyecto.

Un FPGA es un dispositivo lógico programable compuesto por un conjunto de bloques lógicos comunicados a través de conexiones programables, donde se puede implementar tanto circuitos combinacionales como secuenciales. En otras palabras, es un circuito integrado en blanco, que contiene elementos que permiten implementar un circuito integrado personalizado que realice una función concreta. Sus principales ventajas frente a los circuitos integrados para aplicaciones específicas son que puede puede ser reprogramable, por tanto la convierte en un dispositivo muy flexible para desarrollar nuevos diseños, sus costes de adquisición y que el tiempo de diseño y fabricación son menores. Sus desventajas, son más lentas y consumen más potencia. En su arquitectura más básica, consta al menos de tres bloques funcionales, que son lo bloques lógicos configurables, bloques de entrada y salida y una matriz de interconexiones programables o bloques y líneas de interconexión. La Figura 1 muestra la arquitectura básica de un FPGA.

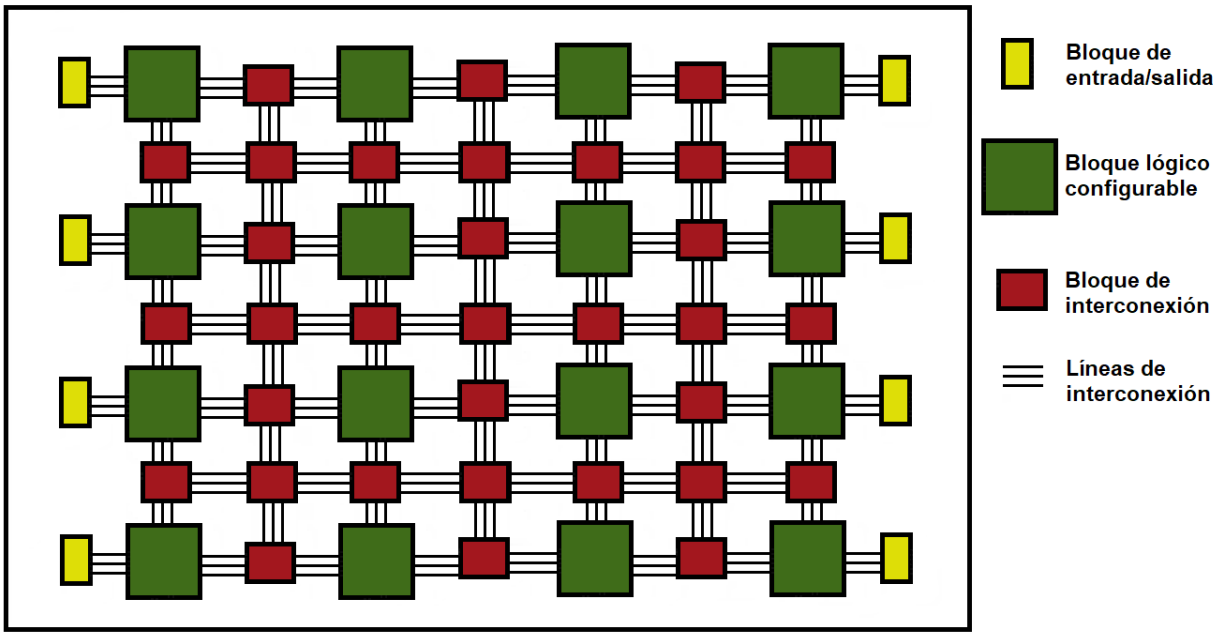

Figura 1: Arquitectura básica de un FPGA. Fuente: [10]

En la estructura básica de un bloque lógico encontramos una n-LUT (Lookup Tables) cuyo propósito es realizar el cómputo de cualquier función de n entradas, simplemente programando la LUT con la tabla de verdad de la función a implementar. La Figura 2 muestra el diagrama de un bloque lógico básico y la Figura 3 muestra un ejemplo de una LUT de 3 entradas.

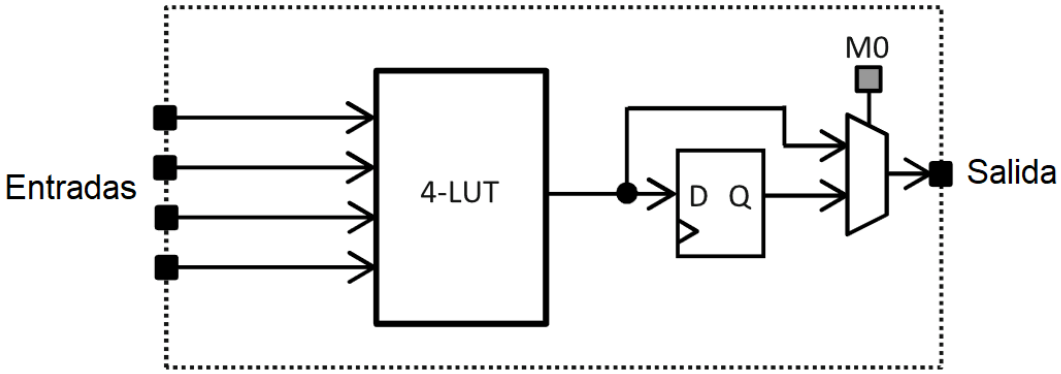

Figura 2: Diagrama de un bloque lógico básico. Fuente: [11]

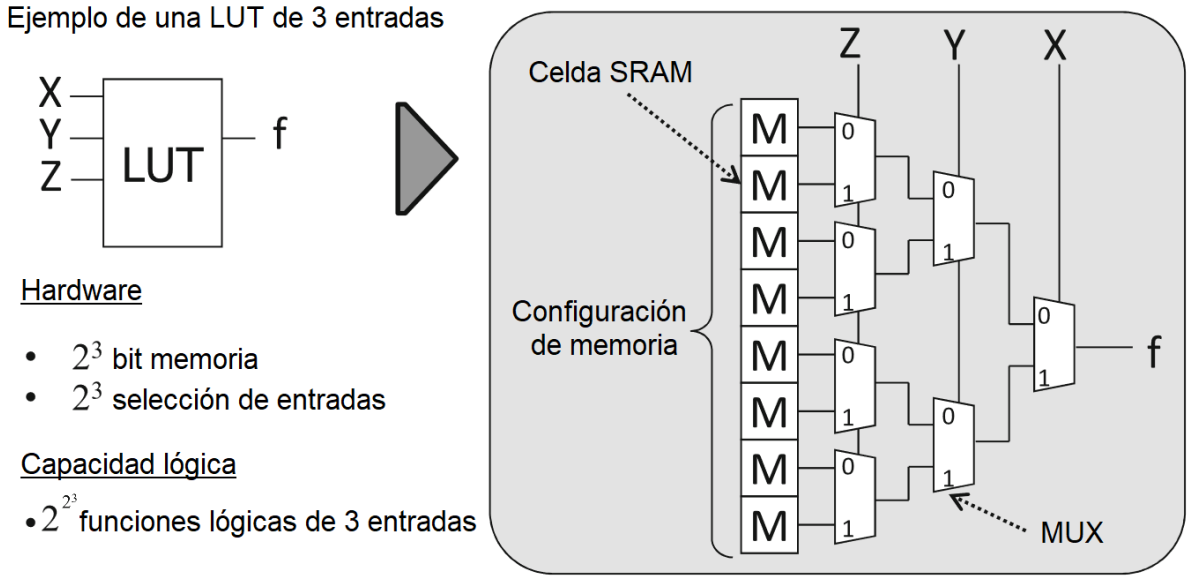

Figura 3: Ejemplo de LUT de 3 entradas. Fuente: [11]

Al aumentar el número de entradas en las LUT podemos implementar una lógica más compleja por cada bloque, por tanto se necesitará un menor número de bloques y de interconexiones entre bloques lógicos para implementar el diseño del circuito. Una LUT más grande tiene más retardo que una más pequeña, aunque se consigue mejorar el retardo porque para implementar funciones complejas se necesitan menos LUT en cascada.

Resumiendo, básicamente un FPGA se divide en tres partes, la primera que consiste en bloques lógicos que realizan circuitos lógicos, la segunda que contiene elementos de entrada y salida y la tercera que consiste en los bloques de conexión. Esta última está compuesta por *Connection Blocks* que conectan las entradas y salidas de los bloques lógicos configurables (CLB) con los canales de comunicación verticales y horizontales, junto con los *Switch Blocks* que permiten cambiar la dirección de encaminamiento entre los canales de comunicación. El encaminamiento segmentado también dispone de canales de longitud variable, lo que permite realizar conexiones directas entre CLBs lejanos y reducir el retardo de cruzar por diferentes transistores de paso para interconectar diferentes segmentos cortos.

Integrar bloques de cómputo heterogéneos, diferentes de los bloques lógicos configurables puede aportar beneficios en la arquitectura, puesto que la implementación de funciones complejas puede requerir de un gran número de bloques, ocupando mayor área y mayor consumo, además deben conectarse entre sí lo que implica una mayor latencia o retardo.

Por otra parte, las FPGAs comerciales también contienen circuitos para funciones específicas, tales como procesadores, bloques de memoria o multiplicadores. Disponer de *hardware* dedicado, como pueden ser bloques de memoria para el almacenamiento interno de datos o multiplicadores para realizar la operación MAC (Multiply And Accumulate) de forma eficiente puede mejorar esta implementación y liberar los recursos de los CLBs para otros fines.

Respecto a la tecnología para su programación, los circuitos en un FPGA son controlados por interruptores programables que están fabricados con distintas tecnologías de semiconductores. Todos los elementos configurables están controlados por bits en una memoria de configuración, de tal forma que cambiando el contenido de la memoria se cambia la interconexión y el modo de funcionamiento de sus elementos. Los tres tipos de tecnologías ampliamente usadas en FPGAs modernas son las memorias flash, antifuse y memorias estáticas (SRAM). Podemos clasificarlas en dos conjuntos:

- Volátiles: están basadas en SRAM y proporcionan una rápida e infinita reconfiguración, es una tecnología *mainstream,* su desventaja es que su contenido se pierde al desconectarlas de la alimentación.
- No volátiles: son las que no pierden su contenido una vez dejan de estar alimentadas.
  - Las memorias flash que son reprogramables aunque solo se pueden reprogramar un número finito de veces.
  - Antifuse que ofrecen retardos bajos y no consumen energía estática, por tanto son una tecnología óptima en el ámbito espacial y militar pero no son reprogramables.

Conocer bien el *hardware* que se está programando es esencial para una utilización eficiente de los recursos disponibles.

2.2 Flujo de diseño para FPGA

El flujo de diseño de un FPGA comprende diferentes niveles de abstracción que representan la descripción de un algoritmo a lo largo de múltiples pasos con el objetivo final de configurar un bitstream para transferirlo al FPGA. Se distingue en dos fases, frontend y backend [12].

Existe una variedad muy amplia de posibilidades para crear descripción de *hardware* para FPGAs, lenguajes, bibliotecas y herramientas enfocadas al diseño frontend. Las FPGAs siguen un enfoque del diseño dirigido al modelado y es común que esté soportado por interfaces gráficas productivas. Por ejemplo, MATLAB Simulink permite generar diseños para FPGA y emularlos simplemente utilizando el ratón del computador. Estas virtualizaciones tienen como objetivo sistemas de cualquier complejidad. También los fabricantes de FPGA proveen de bibliotecas IP para dominios de varias aplicaciones. Por otra parte, las FPGAs pueden ser programadas con lenguajes de programación tradicionales. La mayoría de herramientas de diseño de lenguajes de alto nivel son de C/C++, esto incluye compiladores de fabricantes de FPGA como OpenCL y Vivado HLS. Para mejorar la productividad en dominios específicos, por ejemplo álgebra lineal, existen compiladores source-to-source que componen incluso un nivel de abstracción más alto. En la otra parte del espectro tenemos los lenguajes de descripción de *hardware* de bajo nivel, siendo Verilog o VHDL los más populares.

Estos lenguajes permiten un control total sobre el *hardware* generado. El resultado de la fase de diseño frontend es el nivel de RTL, que pasa al flujo backend. El nivel RTL es básicamente una abstracción donde el estado de toda la información se almacena en registros o cualquier otro tipo de memoria, donde la lógica entre los registros se usa para generar nuevos estados. El nivel RTL describe todos los elementos de memoria, registros o memorias y la lógica utilizada así como las conexiones entre los diferentes elementos de memoria y lógica, es decir, el flujo de datos a través del circuito. La Figura 4 muestra el flujo de un diseño general para FPGA, con los pasos de síntesis e implementación y diferentes niveles de abstracción.

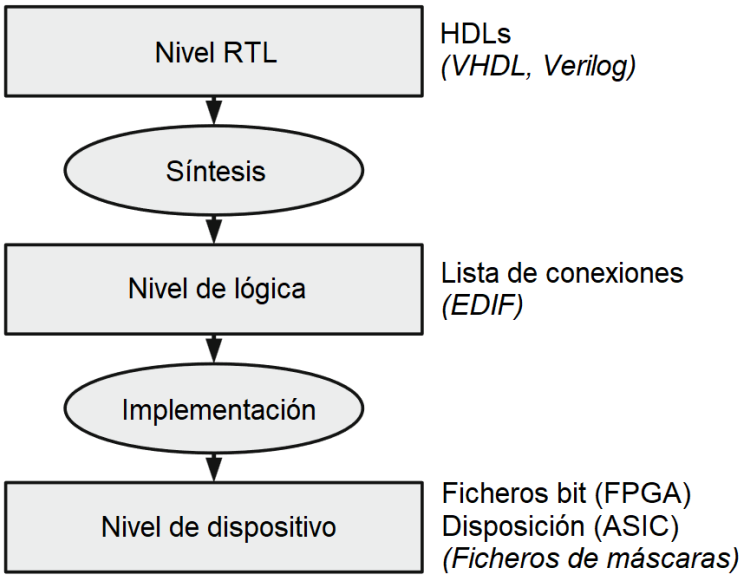

Figura 4: Flujo de diseño general para FPGA, síntesis e implementación

Las especificaciones RTL se hacen con los lenguajes de descripción, estos lenguajes proveen tipos de datos, operaciones lógicas y aritméticas, jerarquías y muchas construcciones conocidas de otros lenguajes de programación, como pueden ser los bucles. Sin embargo, el *hardware* es paralelo de forma inherente y no se ejecuta una única instrucción en un instante determinado sino que muchos módulos funcionan de forma concurrente en un sistema.

Una herramienta de síntesis traduce la descripción RTL en una lista de conexiones, pasando al nivel lógico de abstracción. Es aquí donde cualquier descripción comportamental del nivel RTL se traduce a registros y puertas lógicas para implementar funciones booleanas elementales. La lista de conexiones es un grafo donde los nodos componen los registros y puertas y las aristas las señales de conexiones. En el paso de la síntesis, es posible encontrar opciones para controlar las primitivas que se usarán para implementar una memoria específica o lógica, pero en la mayoría de casos la configuración por defecto funciona adecuadamente y las herramientas de niveles más altos son capaces de ajustar la lógica del proceso de síntesis automáticamente, sin que se tenga que preocupar por los ajustes el diseñador.

La lista de conexiones se optimiza y en un siguiente paso se implementa por primitivas (LUT, bloques de memoria, celdas de entrada y salida) y la red de interconexión de un FPGA. En un primer paso, denominado mapeado de la tecnología, las puertas lógicas se despliegan en LUTs, donde hay distintos algoritmos para minimizar el número de LUTs utilizadas o los retardos de propagación. Después estas primitivas se colocan en el tejido de un FPGA, con algoritmos de agrupamiento y de búsqueda local (simulated annealing [11]) para minimizar la congestión y la distancia entre las conexiones. Entonces se encaminan las primitivas, esto es conocido como el problema de *place and route*, es un proceso muy costoso y se suele resolver con algoritmos de tipo *Pathfinder* [12].

Finalmente el mapeado, colocación y encaminamiento de la lista de conexiones se transfiere a un archivo de configuración, denominado *bitstream*, con la configuración de cada primitiva y los interruptores de la matriz.

2.3 Aplicaciones de un FPGA

Los principales campos de aplicación de FPGAs son muy diversos [14], entre los que destacan:

- Seguridad: permite acelerar la encriptación de datos mejorando la seguridad en sectores de las tecnología de la información clave como los centros de datos o la nube.
- Aeroespacial y defensa: la tecnología antifuse es tolerante a la radiación, óptima para la aplicación en estos campos.
- Prototipado y simulación: permite prototipado rápido, fácil y preciso de software embedido para sistemas en chip y su evaluación.
- Automoción: permiten el procesamiento de la información captada por sensores enviando órdenes a los accionadores del vehículo ofreciendo mayor seguridad a los usuarios.
- Telecomunicaciones: para diseñar sistemas multimedia escalables, de alta calidad en los servicios audiovisuales.
- Electrónica de consumo individual: la flexibilidad que ofrece una FPGA y sus herramientas de programación permite que empresas de dispositivos electrónicos se adapten a cambios acelerados y modas ofreciendo productos de alta calidad en el mercado.
- Centros de datos: ofrecen mayor ancho de banda a los servidores para soportar el incremento de carga de trabajo y evitar que se generen cuellos de botella, se obtienen menores latencias y también muy importante se reduce el consumo energético.
- Computación de alto rendimiento: que cubren campos muy diversos, genoma humano, astrofísica, etc.
- Industria: ideales para aplicaciones robóticas, impresión 3D, automatización de procesos, etc.
- Medicina: para aplicaciones de diagnóstico y monitorización en los que es preciso obtener resultados precisos rápidamente.
- Redes: ofrecen tecnología inalámbrica rápida y a bajo coste, con las que se puede ofrecer infraestructuras que conecten muchos dispositivos y soporten el tráfico masivo de datos con latencias bajas.
- Inteligencia artificial y Machine Learning: por ejemplo para resolver algoritmos complejos como el reconocimiento de voz a partir del aprendizaje de grandes bases de datos.
- Industria 4.0: obteniendo mayor rendimiento y velocidad en el análisis y procesamiento de datos a tiempo real captados por dispositivos IoT.

2.4 Las FPGA en los sintetizadores de la actualidad

Los primeros sintetizadores analógicos eran controlados por voltaje, después empezaron a incluir microprocesadores y componentes digitales. Ahora el desarrollo con FPGA en estos instrumentos está popularizándose. Las FPGA son muy útiles, eficientes para el cálculo y generación de formas de onda con alto rendimiento en el procesamiento de señales, a frecuencias de muestreo altas en paralelo. A diferencia de otros sintetizadores *hardware*, los que están basados en FPGA aportan mucha flexibilidad en su diseño y en el desarrollo de nuevas características personalizadas.

Un ejemplo de sintetizadores en el mercado que aprovechan las altas capacidades de un FPGA pueden ser Waldorf Kyra, con 4096 osciladores simultáneos con 128 voces y ondas con un alto sobre muestreo [5] o el sintetizador Novation Peak, que consta de osciladores de control numérico basados en FPGA que operan a 24 MHz y generan formas de onda idénticas a las producidas por los osciladores analógicos [6]. Según mencionan ambos fabricantes, trabajar con FPGA les ha permitido que el diseño se extienda optimizando la síntesis y son más eficientes que otros tipos de sintetizadores. La Figura 5 muestra el aspecto de un sintetizador *hardware* basado en FPGA.

Figura 5: Sintetizador comercial basado en FPGA, Waldorf Kyra. Fuente: [5]

Además el sintetizador Kyra también puede transmitir el audio al computador a través de USB a una frecuencia de muestreo de hasta 96 kHz y 24 bits, con ocho canales simultáneos en estéreo. Desafortunadamente, no dan mucha información sobre qué modelos de FPGA usan y la cantidad de recursos disponibles, aunque sabemos que se tratan de dispositivos con mucha una gran cantidad de recursos por la variedad en las características que ofrecen, varias formas de onda y tipos de efectos y filtros, generadores de envolventes y modulaciones LFO asignables a distintos parámetros, patrones de arpegiadores, con capacidad de reproducir polifonía con un número muy elevado de osciladores y voces.

Las emulaciones de sintetizadores software están avanzando a mucha velocidad durante la última década aproximadamente, generando el sonido con inteligencia artificial y Machine Learning. Hoy en día no se comercializan todavía sintetizadores *hardware* que se basen en estos avances, pero las FPGA son muy útiles en este campo y pueden acelerar los algoritmos con un bajo coste energético y un alto rendimiento para conseguir resolver problemas complejos.

2.5 Elección de la placa para el prototipado

En esta sección analizamos cuál es la placa de prototipado que pensamos que se adapta mejor a nuestro proyecto, analizando las características que ofrecen para poder determinar cuál es la más adecuada. Pero primero veremos cuáles son los principales fabricantes de FPGAs en el mercado actual.

- Xilinx Inc[1], ahora AMD-Xilinx es un fabricante que produce circuitos integrados con una gran reputación por ser una de las compañías más avanzadas en la producción de alta tecnología. Solo fabrican FPGAs, también FPGA con procesadores que reciben el nombre de Programmable SoC y con procesadores sistólicos para acelerar convoluciones con el nombre ACAP. Disponen de varios modelos, cada uno está enfocado a un uso en concreto. Es junto con Altera líder en el mercado de las FPGAs. Además también producen chips y herramientas para la integración de *software* y *hardware*.
- Altera Corporation[2], es otro de los fabricantes más reconocidos en el mundo de las FPGAs, desde 1984 han producido continuamente productos de gran calidad. Fabrican FPGAs para diferentes usos, incluyendo la industria, microprocesadores o electrónica de consumo. Muchas compañías tecnológicas utilizan sus productos y han diseñado chips para la NASA, para la agencia espacial europea o para uso militar en los Estados Unidos.
- Lattice Semiconductor[3], es otra de las compañías con influencia y se enfoca principalmente en productos que son demasiados complejos que otros fabricantes más pequeños pueden producir, por ejemplo chips que se adapten a las necesidades específicas de alguna compañía en particular. Sin embargo también han tenido éxito al producir chips que se adaptan a diferentes aplicaciones, desde equipamientos para la medicina de alta tecnología hasta tarjetas de video.
- Achronix[4], fabricantes americanos que están en el mercado desde el 2003. Se concentran en la producción de FPGAs muy pequeñas para muchos tipos de aplicaciones, uso militar, electrónica o medicina. También producen chips que se utilizan para las comunicaciones inalámbricas, enfocándose en centros de datos y redes lo que les ha llevado a conseguir su alta reputación.
- QuickLogic Corporation[5], otra compañía que ha logrado muy buenos resultados los últimos años. Sin embargo las FPGAs que producen no son de gran calidad y se enfocan en otro tipo de componentes y la producción de circuitos integrados.
- Microchip Technology[6], desde 1980 producen reconocidos microcontroladores y otros micropocesadores. También producen un rango de FPGAs denominadas Microchip's Multimedia Module, que se utilizan en distintos dispositivos y han sido reconocidas por su gran capacidad y rendimiento.
- Microsemi Corporation[7], desde 1983 han ido perfeccionando sus FPGAs y se han concentrado principalmente en aplicaciones militares y científicas, por tanto sus productos van dirigidos a gobiernos e investigadores.
- Efinix[8], fundada en el año 2000 es hoy en día otro de los principales fabricantes. Están especializados en la fabricación de FPGAs pequeñas pero complejas, principalmente para la industria y electrónica de consumo.

1 AMD-Xilinx: https://www.xilinx.com/
2 FPGAs de Intel: https://www.intel.com/content/www/us/en/products/programmable.html
3 Lattice Semiconductor: https://www.latticesemi.com/
4 Achronix: https://www.achronix.com/
5 QuickLogic Corportaion: https://www.quicklogic.com/
6 Microchip Technology: https://www.microchip.com/
7 Microsemi Corporation: https://www.microsemi.com/
8 Efinix, Inc: https://www.efinixinc.com/

Intel Corporation realizó la adquisición de Altera Corporation en el año 2015 por 16.7 billones de dólares, con el fin de mantener la posición de liderazgo que tenían sus procesadores en los centros de datos, ya que un nuevo y más beneficioso modelo de computación heterogénea con FPGAs les permitían obtener grandes resultados a bajo coste. Integrar la arquitectura de un FPGA ofrece un mayor rendimiento de cómputo con un bajo consumo energético y procesadores con las mismas características o incluso mejores que sus procesadores ARM.

Advanced Micro Devices Inc (AMD) también realizó la adquisición de Xilinx Inc, en el año 2021 por 34 billones de dólares. La computación de alto rendimiento será un pilar cada vez mayor en las principales tendencias del futuro. Si bien las CPU y GPU siguen siendo los motores en los dispositivos de computación de alto rendimiento, la demanda de recursos de computación adaptable son clave para acelerar las cargas de trabajo emergentes. Con esta compra, AMD es líder en la industria de la computación adaptable y de alto rendimiento, además amplia su cartera de productos y conocimientos y mejora la gama de aplicaciones inteligentes para centros de datos y tecnologías para las plataformas de procesamiento en la nube [15].

Como hemos visto en la sección anterior, los sintetizadores *hardware* comerciales ofrecen muchas características muy variadas para generar todo tipo de sonidos. No obstante, el objetivo en este proyecto es el de implementar las funcionalidades básicas que éstos ofrecen, para lograr una base robusta y aprender a realizar un diseño eficiente. En consecuencia, no necesitamos que nuestro dispositivo disponga de una gran cantidad de recursos, además a mayor cantidad mayor es el coste de adquisición del dispositivo y también es habitual que incluyan otras funcionalidades que no llegaríamos a aprovechar en este proyecto.

Después de analizar los dispositivos que ofrecen los principales fabricantes, se ha llegado a la conclusión de que AMD-Xilinx dispone de una gama de productos interesante y ofrece placas que recomiendan para trabajar en proyectos en las universidades [16], donde los precios de la mayoría de éstas no son excesivamente elevados y las características y recursos que ofrecen son más que suficientes. La serie Artix-7 es la que vemos más adecuada para abordar este proyecto, una vez se ha analizado la guía de selección para estas series [17]. La Tabla 1 muestra una comparativa de los recursos disponibles en las las distintas placas de la serie Artix-7.

Tabla 1: Comparativa de recursos disponibles la serie Artix-7 de Xilinx. Fuente: [17]

	XC7A12T	XC7A15T	XC7A25T	**XC7A35T**	XC7A50T	XC7A75T	XC7A100T	XC7A200T
Logic Cells	12,800	16,640	23,360	**33,280**	52,160	75,520	101,440	215,360
Slices	2000	2,600	3,650	**5,200**	8,150	11,800	15,850	33,650
CLB Flip-Flops	16000	20,800	29,200	**41,600**	65,200	94,400	126,800	269,200
Distributed RAM	171	200	313	**400**	600	892	1,188	2,888
Block RAM	20	25	45	**50**	75	105	135	365
Total Block RAM (Kb)	720	900	1,620	**1,800**	2,700	3,780	4,860	13,140
CMTs	3	5	3	**5**	5	6	6	10
Single-Ended I/O	150	250	150	**250**	250	300	300	500
Differential I/O	72	120	72	**120**	120	144	144	240
DSP Slices	40	45	80	**90**	120	180	240	740
GTP Transceiver	2	4	4	**4**	4	8	8	16

Durante este proyecto, de entre las placas disponibles de la serie Artix-7 de Xilinx hemos escogido trabajar con la placa Basys-3 de Diligent Inc.[9], cuya referencia en la Tabla 1 es XC7A35T.

El manual de referencia [7] describe sus características y detalles de funcionamiento. Se trata de una placa de altas capacidades y bajo coste, dispone de conexión USB, 16 *switches* y 16 LEDs, cinco botones, display de cuatro dígitos de siete segmentos, salida VGA y otros puertos a los que se pueden conectar tres puertos Pmod para añadir más funcionalidad concreta, con un reloj a 100 MHz. Por otra parte, incluye un puente de USB a UART (Universal Asynchronous Receiver-Transmitter), que permite que aplicaciones del computador se comuniquen con la placa mediante puertos serie. Para lograr esta comunicación los drivers VCP (Virtual COM port) de "FTDI Chip", tienen como objetivo que el dispositivo USB aparezca como un puerto COM adicional, de forma que nuestro software podrá acceder al USB del mismo modo que si se tratase de un puerto COM estándar. La Figura 6 muestra el aspecto físico de la placa que hemos utilizado en este proyecto.

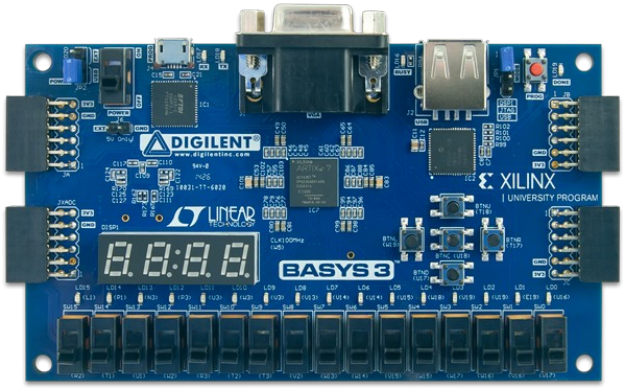

Figura 6: Aspecto físico de la placa de desarrollo de Diligent Basys 3

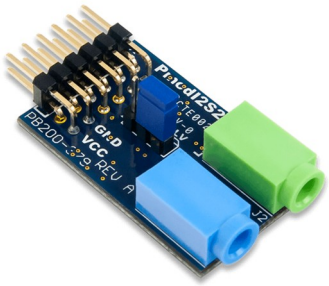

Figura 7: Pmod I2S2 Stereo Audio Input and Output

Para poder reproducir los sonidos que nuestro sintetizador genera, necesitamos un dispositivo que convierta la señal de digital a analógica. En este proyecto utilizamos el módulo Pmod I2S2 [18], que ofrece un convertidor de analógico a digital Cirrus CS5343 Multi-Bit y un convertidor de digital a analógico (DAC) Cirrus CS4344 Stereo [19], cada uno conectado a Jacks de audio de 3.5 mm. Soporta resoluciones de 24 bits por canal a frecuencias de muestreo de entrada de hasta 108 kHz. Durante la fase de implementación, utilizaremos una tarjeta de sonido externa para la realizar la captura de muestras y su posterior análisis. La Figura 7 muestra el aspecto del módulo Pmod I2S2, que conectaremos en el puerto superior izquierdo de la placa.

Como la placa que hemos escogido para este proyecto dispone de un FPGA de AMD-Xilinx, utilizaremos como entorno de desarrollo Vivado ML Standard Edition para implementar nuestro modelo en la FPGA, desarrollando con el lenguaje de descripción VHDL. La licencia para utilizar este entorno es de 2995 dólares, pero cabe destacar que la licencia es gratuita para trabajar con la placa que hemos elegido. Utilizaremos este entorno para realizar el diseño y los procesos de síntesis e implementación y transferir el modelo a la placa. Utilizar el simulador en este entorno es muy útil para comprobar que el *hardware* que estamos describiendo se comporta de la forma esperada, donde podemos comprobar el correcto funcionamiento de nuestro modelo visualizando los resultados en función de los datos de entrada que hemos introducido.

9 Diligent Inc. Basys-3: https://digilent.com/reference/programmable-logic/basys-3/start

3. Configuración y estructura del trabajo

En esta sección se explica la configuración y arquitectura global del sistema junto con la estructura del sintetizador y del proyecto de Vivado.

3.1 Configuración

En el mundo digital las señales de audio son discretas, toman valores finitos. La frecuencia de muestreo es el número de estos valores tomados en el tiempo. En general, está expresado en Hz (hercios) o ciclos por segundo, aunque también se pueden utilizar otras magnitudes. Durante este proyecto se ha trabajado con una frecuencia de muestreo de 44.1 kHz, es decir, generamos 44100 muestras por segundo. Este factor es importante ya que puede influir en la calidad del sonido, de forma que a menor frecuencia se pierde fidelidad. Típicamente para aplicaciones de música se utilizan 44.1 kHz o 48 kHz, si bien es posible encontrar ciertos entornos en los que por la calidad requerida se llega a trabajar hasta con 192 kHz. La Figura 9, Figura 8 y Figura 10 muestran como se ha perdido precisión en la representación de la onda, ya que al reducir la cantidad de muestras tomadas el resultado es aproximado.

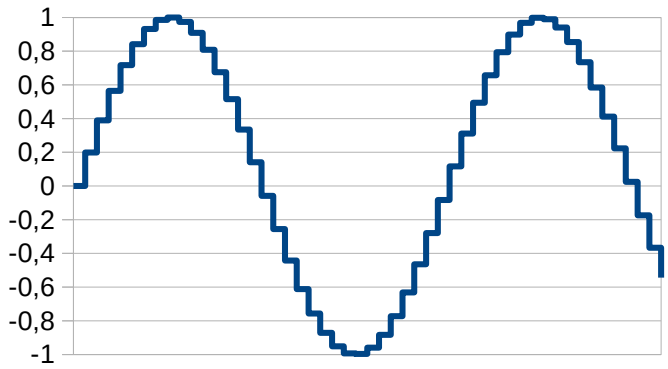

Figura 8: Onda senoidal con frecuencia de muestreo más baja

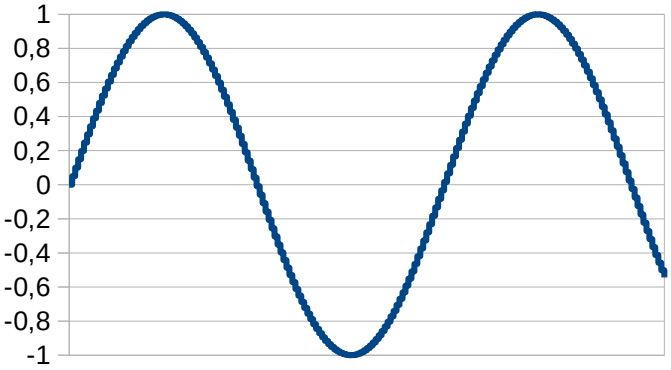

Figura 9: Onda senoidal con frecuencia de muestreo más alta

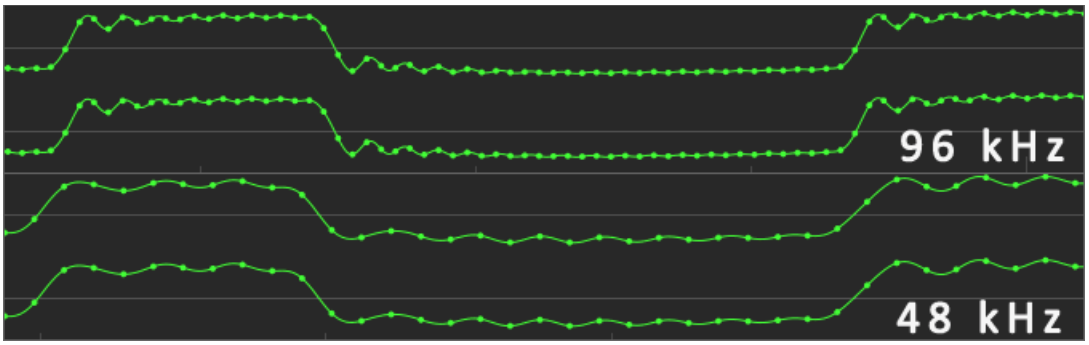

Figura 10: Onda con una frecuencia de muestreo de 48 kHz (inferior) y 96 kHz (superior)

Según el teorema de muestreo de Nyquist-Shannon [13], la frecuencia de muestreo debe ser mayor que el doble de la frecuencia mayor que queremos reproducir, como consecuencia en este sintetizador solo podemos generar sonidos hasta la frecuencia de 22.05 kHz. Pero debido a otras limitaciones de implementación solo generaremos frecuencias hasta los 14 kHz aproximadamente, estas limitaciones radican en que las frecuencias mayores requieren un número menor de ciclos para completar su periodo, por tanto su división solo es distinguible por el número de decimales que estamos perdiendo, como veremos con más detalle en la fase de implementación.

También es importante destacar que frecuencias de muestreo más altas también ayudan a prevenir otras características indeseadas como el *aliasing*, característica que se produce cuando una señal excede la frecuencia de Nyquist, por lo que las muestras se interpretan de forma incorrecta y pueden percibirse como una distorsión. Por otra parte también es muy importante la profundidad o resolución de bits, que se refiere al número de bits que se utilizan para representar cada una de las muestras de la señal. Cuando la cantidad de bits es mayor, mayor es el rango dinámico de la señal, es decir, podemos representar de forma más precisa el valor de la muestra en cada instante.

Para aplicaciones de audio es habitual trabajar con una resolución de 16 bits, aunque también es común encontrar situaciones en las que es recomendable utilizar 32 bits en coma flotante, como por ejemplo para discretizar las grabaciones que provienen de señales analógicas, una mayor resolución ofrece en este caso ventajas al procesar posteriormente la señal. Normalmente no se recomienda emplear resoluciones inferiores a 16 bits, puesto que la pérdida de precisión puede suponer una reducción en la calidad del sonido. En este proyecto se ha escogido trabajar con una tasa de 24 bits por muestra, porque es una precisión suficiente para generar las formas de onda con una fidelidad adecuada.

3.2 Arquitectura global del sistema

La Figura 11 muestra el entorno que hemos utilizado para trabajar con el sintetizador y realizar las pruebas. El dispositivo Basys-3 se encuentra conectado por USB a un computador, al que también conectamos una tarjeta de sonido para capturar las muestras con mayor fidelidad que la que nos ofrece habitualmente la tarjeta de sonido integrada del computador. La tarjeta de audio que hemos utilizado también dispone de unos convertidores Cirrus Logic con una resolución de 16 o 24 bits y una frecuencia de muestreo de hasta 96 kHz, igual que el dispositivo Pmod I2S2. Para escuchar el sintetizador tenemos varias opciones, se pueden conectar unos auriculares a la tarjeta de sonido o conectar unos altavoces desde una salida RCA de la tarjeta, pero también es posible utilizar los altavoces integrados del portátil. El sintetizador está programado en la FPGA y se controla desde una aplicación con una interfaz gráfica en el computador, mediante mensajes que se transmiten y reciben por cable USB. Esta aplicación soporta conexiones de dispositivos *hardware* MIDI para su gestión y por otra parte, también es posible conectar cualquier dispositivo reproductor de audio, como por ejemplo un reproductor de MP3 o un *smartphone* a la entrada de audio del Pmod I2S2.

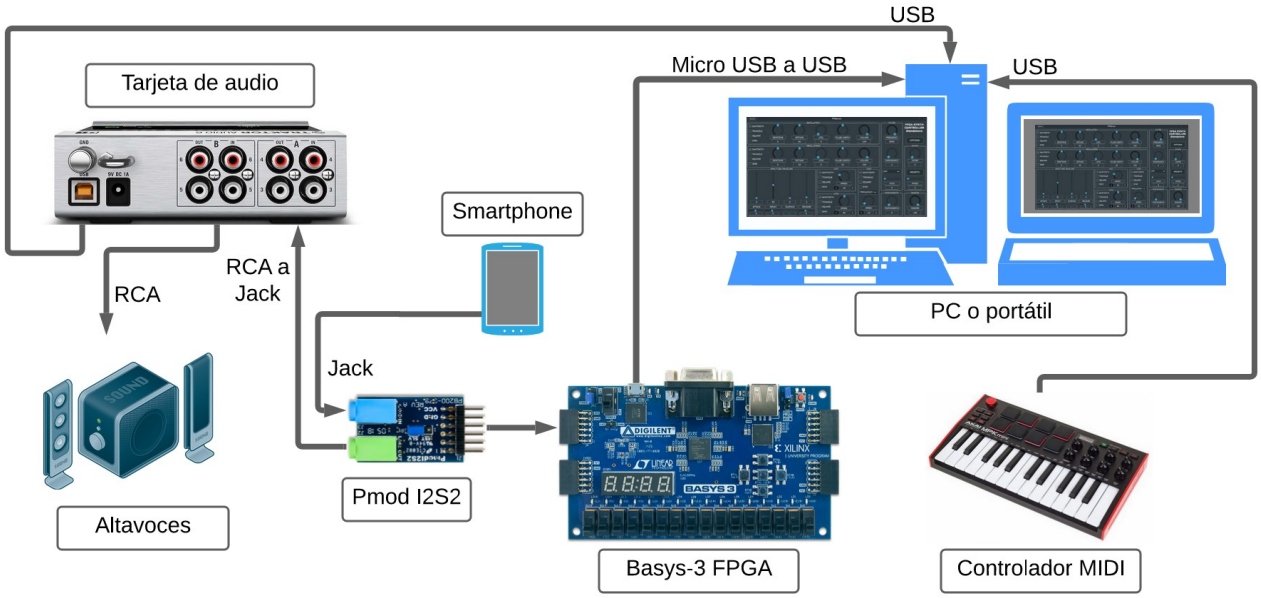

Figura 11: Arquitectura del sistema

Tanto la tarjeta de audio como la placa Basys-3 junto con el dispositivo Pmod se pueden alimentar por USB, aunque ambos pueden alimentarse con una fuente externa de la misma manera que el controlador MIDI *hardware*.

3.3 Estructura del sintetizador

En cuanto al sintetizador, constará de dos osciladores donde cada uno puede reproducir distintos tipos de formas de ondas básicas. Debido a que nuestro sintetizador es polifónico, cada oscilador puede generar varias formas de onda simultáneas, donde también es posible diferenciar independientemente los semitonos o realizar pequeños ajustes de afinación. La suma total de todas estas voces se dirigirá al generador de ruido.

Por otra parte, disponemos de dos LFOs independientes para modular la frecuencia de cada voz de un oscilador. De igual forma que con los osciladores, cada oscilador de baja frecuencia puede generar cualquier tipo de onda básica y también es posible controlar su amplitud.

Adicionalmente el generador de envolventes de amplitud se encargará de modular la amplitud de cada una de las ondas, previamente a la entrada del generador de ruido. La salida del generador de ruido es dirigida a los filtros, donde ambos solo procesarán la señal en el caso de que alguno de éstos se encuentre activado. Los filtros que vamos a implementar son los de paso bajo, alto, de banda y de rechazo de banda y también contienen otro oscilador de baja frecuencia para modular la frecuencia de corte o la resonancia.

Finalmente la salida del filtro se dirige al componente que transmite los datos al DAC. La Figura 12 muestra un esquema de funcionamiento de nuestro sintetizador, donde además de generar los sonidos de forma interna también podemos seleccionar la señal recibida por la entrada estéreo del dispositivo Pmod I2S2. Si seleccionamos esta entrada, la señal será procesada por el generador de ruido y los filtros, antes de llegar al DAC.

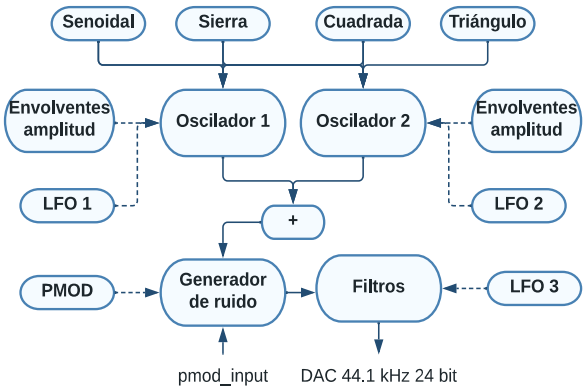

Figura 12: Esquema de funcionamiento del sintetizador

Nótese como la envolvente de amplitud está conectada con cada oscilador, pero consta de un solo componente que procesa cada una de las voces de un oscilador. Se requiere este diseño ya que la polifonía exige tratar la amplitud de cada onda de forma distinta, pero si se tratase de un sintetizador monofónico se podría reducir la utilización de recursos realizando el procesamiento sobre una suma global. Sin embargo esta estructura va a suponer un mayor consumo de procesadores de señal digital, proporcional al número de voces. De forma análoga sucede con las LFOs, que constan de un solo componente para cada tipo de onda y que generan cada una de las muestras de cada oscilador de baja frecuencia.

3.4 Estructura del proyecto

Respecto a la estructura del proyecto de Vivado, en la Figura 13 podemos encontrar el esquema que se ha planteado y que nos puede ayudar a comprender mejor cómo se ha realizado el diseño.

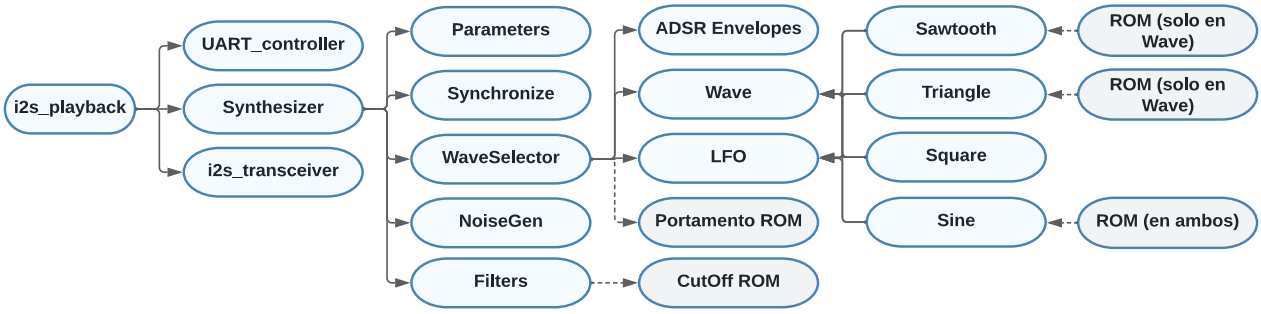

Figura 13: Estructura del sintetizador en el proyecto de Vivado

Los diseños *hardware* son típicamente estructurales, es decir, éstos se basan en definir o reutilizar componentes y especificar su interconexión. Por tanto, una metodología clásica para el diseño es *bottom-up*, donde comenzamos por definir los componentes básicos necesarios para diseñar otros componentes más complejos y de esta forma diseñar todo el sistema.

El componente con el nivel más alto es i2s_playback, es decir, el *top module* del proyecto. Éste se encarga de realizar las conexiones necesarias con UART_controller, Synthesizer y i2s_tranceiver. UART_controller se encarga de gestionar la recepción y envío de datos con la aplicación interfaz, mientra que i2s_transceiver obtiene los datos recibidos desde el convertidor analógico digital, así como transmite los datos del sonido generado por el sintetizador al convertidor digital analógico.

Synthesizer es el componente principal del sintetizador y se encarga de gestionar las conexiones entre varios componentes. Por una parte tenemos el componente Parameters, que sirve para actualizar los valores para cada uno de los parámetros internos del sintetizador, recibiendo el dato desde UART_controller y el componente Synchronize, donde se preparan los datos que deben ser enviados al puerto serie cuando la aplicación del computador lo requiere. Por otra parte contiene los componentes que actúan como procesadores de señal, bien sea mediante la generación de ruido o la aplicación de los filtros. Synthesizer también se encarga de realizar la suma de todas las voces de cada oscilador y aplicar su correspondiente volumen y mezcla. Por último, éste aplicará las operaciones para obtener el volumen general, cuyo resultado es el que entra al generador de ruido.

WaveSelector se encarga de transmitir a Synthesizer cada una de las ondas de los osciladores y por tanto recibe todos los parámetros de configuración del sintetizador necesarios. Disponemos de distintos componentes independientes para cada tipo de onda, cada uno de éstos contiene una máquina de estados para generar las muestras de cada onda en distintos ciclos de reloj y de esta forma aprovechar al máximo los recursos disponibles. Como se ha comentado previamente en este capítulo, cada oscilador de baja frecuencia también es un componente distinto para cada tipo de onda, siguiendo el mismo modo de funcionamiento que los componentes que generan las ondas audibles.

El componente adsr_envelopes también consta de una máquina de estados para calcular en distintos ciclos la amplitud de cada una de las ondas, donde WaveSelector aplica las operaciones de multiplicación necesarias para finalmente transmitir los datos tras la modulación a Synthesizer.

Algunos datos se encuentran almacenados en memoria, como es el caso de los valores de tiempo de deslizamiento entre distintas frecuencias para el *portamento*, valores precalculados para la frecuencia de corte en los filtros, los incrementos de cada muestra para las ondas sierra, triángulo y senoidal o valores para indexar el incremento en la onda senoidal, en función de su frecuencia. El paquete synthesizer_package contiene las constantes y los tipos que el resto de componentes necesitan y que éstos importan.

4. Desarrollo del proyecto

La implementación se ha desarrollado en cuatro fases, añadiendo más funcionalidades de forma incremental a medida que se han ido consiguiendo los objetivos en cada etapa.

4.1 Fase 1: Ondas básicas, reproducción y control desde el computador

En primer lugar se han generado los cuatro tipos de ondas básicas y se ha comprobado mediante la simulación que las formas de ondas son las esperadas, para más tarde plantear el diseño para que se pueda seleccionar la frecuencia de cada nota. Una vez se ha verificado que los periodos de distintas frecuencias son correctos, se ha planteado el diseño para el envío de los datos al convertidor digital a analógico y se ha comprobado que los sonidos se reproducen correctamente, mediante la grabación y la visualización en una aplicación externa en la que podemos analizar las formas de onda.

Seguidamente se han generado los componentes para enviar y recibir los datos a través de UART, al mismo tiempo que se ha compilado una pequeña aplicación en el lenguaje C para transmitir y recibir los datos en el computador mediante el puerto serie.

4.1.1 Los cuatro tipos de ondas básicas

Para comenzar se han generado las formas de onda sierra, triangular, cuadrada y senoidal, pero sin una frecuencia determinada, siendo la frecuencia de reloj la que ha establecido la frecuencia de cada nota, es decir, en cada flanco de subida del reloj del sistema se ha actualizado el valor de la muestra.

La Figura 15 muestra la forma de onda de cada tipo, a una frecuencia de 441 Hz. En primer lugar describimos el componente de la onda sierra, donde los valores de cada muestra eran de 24 bits sin signo y no nos preocupaba el desbordamiento, ya que por su forma simplemente debemos ir incrementando el valor de la muestra y cuando desborda volverá a su valor mínimo. No obstante el Algoritmo 1 muestra el proceso utilizado para la generación de la onda sierra, donde se tiene en consideración el número de ciclos para completar el periodo a una frecuencia determinada, que explicamos con más detalle en la sección 4.1.2.

Algoritmo 1: Onda sierra

```
si contador < ciclos_periodo entonces
  contador = contador + 1
  sierra = sierra + incremento
si no
  contador = 0
  sierra = MIN
fin si
```

El Algoritmo 2 trata la onda triangular, que tiene que sumar al valor de la muestra el doble que en el caso de la sierra hasta alcanzar su valor máximo, es decir a la mitad del periodo para completar la onda a una frecuencia determinada, para empezar a restar y así sucesivamente. La señal que recibe el nombre sumar está simplemente para indicar la operación a realizar con el incremento, por esta razón cuando llegamos a la mitad del periodo su valor debe cambiar.

Algoritmo 2: Onda triangular

```
si contador < ciclos_periodo / 2 entonces
  si sumar = TRUE entonces
    triangular = triangular + incremento
  si no
    triangular = triangular - incremento
  fin si
  contador = contador + 1
si no
  contador = 0
  sumar = not sumar
fin si
```

La onda cuadrada es en principio la más sencilla de todas, puesto que debe mantenerse un valor máximo o mínimo durante un número de ciclos determinado. Sin embargo al principio seguimos esta idea pero no presentaba una onda cuadrada en simulación sino una onda triangular, por tanto decidimos agregar una muestra con valor cero en cada transición. Posteriormente nos dimos cuenta de que este comportamiento no es el que sucede cuando se transmiten los datos al DAC mediante el análisis desde Audacity y que en la configuración analógica del simulador, podemos seleccionar la interpolación en modo mantener para que la simulación se corresponda con lo que realmente se va a reproducir.

El Algoritmo 3 muestra esta última implementación sin valores intermedios, porque el diseño es más sencillo y además no presenta un problema de desajuste en la afinación de la frecuencia, es decir no alteramos el periodo para completar la onda si insertamos un ciclo más, por tanto así los ajustes de afinación son exactamente iguales en todos los tipos de onda.

Algoritmo 3: Onda cuadrada

```
si contador < ciclos_periodo entonces
  si nivel_alto = TRUE entonces
     cuadrada = MAX
  si no
     cuadrada = MIN
  fin si
  contador = contador + 1
si no
  contador = 0
  nivel_alto = not nivel_alto
fin si
```

El Algoritmo 4 trata el diseño de la onda senoidal en el que se toma una serie de valores de una tabla que previamente hemos calculado y tenemos una señal, que en el siguiente algoritmo recibe el nombre contador, para actualizar el índice para obtener el valor de la muestra en cada instante. La señal con el nombre índice varía en función de la frecuencia de la nota, siendo un número más grande para frecuencias altas y más pequeño para frecuencias bajas.

Por otra parte, como veremos en la sección 4.1.2, la tabla de los datos consta de unos valores que son simétricos y por tanto podemos dividir en cuatro partes. Por este motivo tenemos la señal denominada etapa, puesto que en cada etapa recorremos el vector hacia delante o hacia atrás y asignamos al valor de la onda el dato en positivo o negativo.

Algoritmo 4: Onda senoidal

```
si etapa = PRIMER_CUADRANTE entonces
  si contador + índice < tamaño_tabla entonces
    contador = contador + índice
  si no
    contador = tamaño_tabla
    etapa =  SEGUNDO_CUADRANTE
  fin si
  senoidal = dato_tabla(contador)
o si etapa = SEGUNDO_CUADRANTE entonces
  si contador - índice > 0 entonces
    contador = contador - índice
  si no
    contador = 0
    etapa = TERCER_CUADRANTE
  fin si
  senoidal = dato_tabla(contador)
o si etapa = TERCER_CUADRANTE entonces
  si contador + índice < tamaño_tabla entonces
    contador = contador + índice
  si no
    contador = tamaño_tabla
    etapa = CUARTO_CUADRANTE
  fin si
  senoidal = -(dato_tabla(contador))
o si etapa = CUARTO_CUADRANTE entonces
  si contador - índice > 0 entonces
    contador = contador - índice
  si no
    contador = 0
    etapa = PRIMER_CUADRANTE
  fin si
  senoidal = -(dato_tabla(contador))
fin si
```

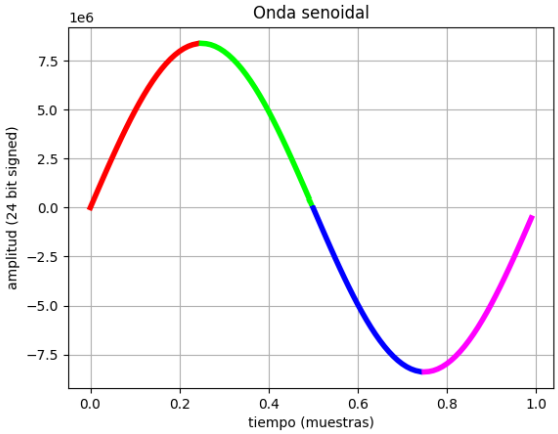

Figura 14: Cuadrantes onda senoidal

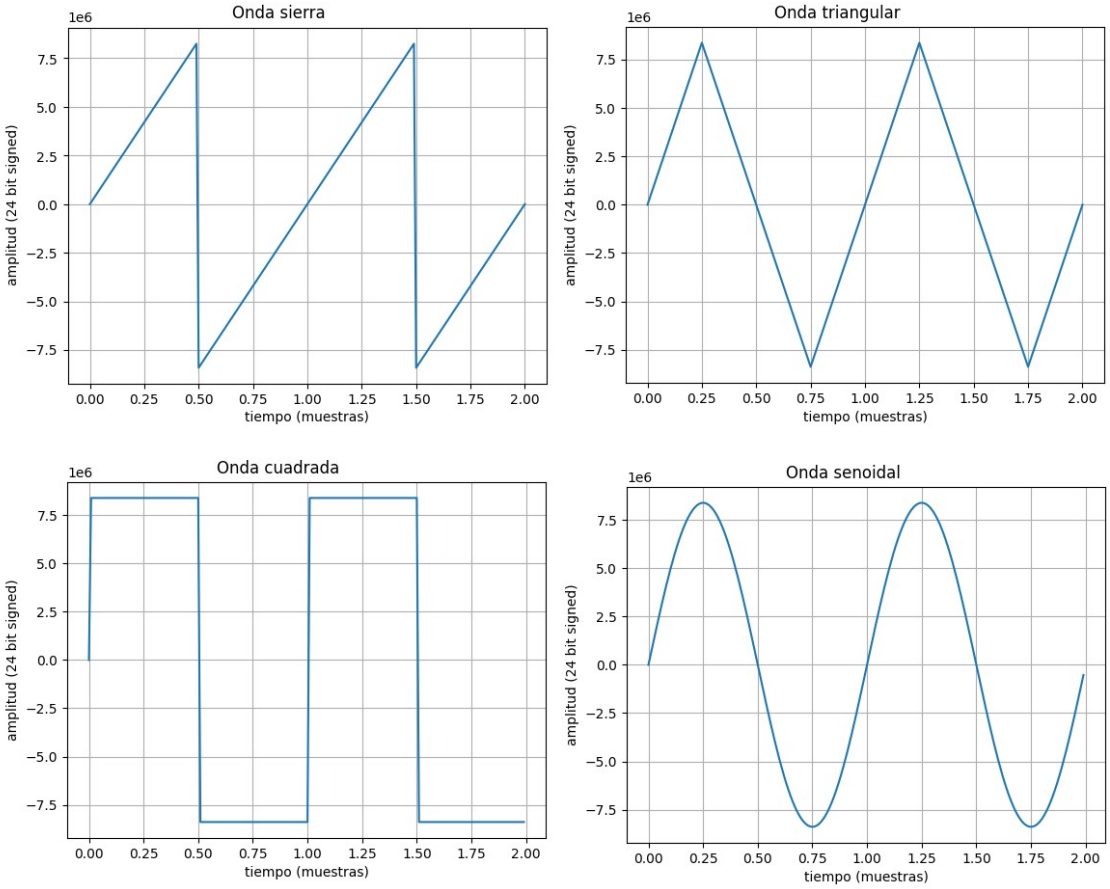

Figura 15: Formas de ondas básicas de cada oscilador

4.1.2 Selección de frecuencias y tonos

Pero para trabajar con el sintetizador en mayor grado, hemos de ser capaces de seleccionar entre distintos tonos o frecuencias de una nota determinada. En la sección 4.4.1 vemos cómo utilizamos tanto el teclado del computador como un controlador MIDI para cambiar las notas en la aplicación del computador, mientras que en esta sección vemos el desarrollo del proyecto con VHDL.

Como ya hemos visto, en un segundo actualizaremos el valor de 44100 muestras, pero para saber cuál es la cantidad de ciclos que necesita una onda de una frecuencia concreta, realizamos el cociente de la frecuencia de muestreo entre la frecuencia de la nota, como muestra la ecuación (1), para obtener el número de ciclos para completar un periodo.

$$Ciclos_{nota} = \frac{Frecuencia_{muestreo}}{Frecuencia_{nota}} \quad (1)$$

En la implementación inicial, hemos formado una tabla de 128 valores porque es el número de notas que un teclado MIDI puede generar, siendo la frecuencia más baja de 8.18 Hz y las más alta de 12.54 kHz [27]. Sin embargo, la cantidad de valores de la tabla se extenderá cuando tengamos que incorporar el efecto de deslizamiento, porque debemos conocer los ciclos de más frecuencias. Cabe destacar que necesitamos almacenar estos valores en una tabla porque la operación de la división en *hardware* dedicado es muy costosa, excepto cuando se divide por potencias de dos, de forma que es más conveniente precalcular los datos puesto que son siempre los mismos.

En los componentes generadores de ondas, se ha incluido una señal cuya función es la de contar el número de ciclos para determinar cuando se ha completado el periodo, por lo tanto volvemos a generar el valor inicial. No obstante, calcular el periodo de esta forma conduce a un cierto error que se traduce en un desajuste en la frecuencia de una nota, como muestra la Tabla 2, ya que debemos truncar el periodo. Este error se va incrementando a medida que las frecuencias de una nota son superiores, de forma que el error se hace más notable en las frecuencias altas.

Tabla 2: Periodo de varias frecuencias

Frecuencia (Hz)	Ciclos
440	**100**,22
441	**100**
442	**99**,77
882	**50**
12500	**3**,528
13000	**3**,392
13500	**3**,266

Para evitar los denominados clics digitales, un efecto indeseado que se produce cuando una onda no comienza o finaliza con valor cero o muy cercano, tuvimos en cuenta que éste es el valor inicial cuando se empieza a generar muestras, pero también que no va a terminar de generarlas, incluso cuando se deja de pulsar una tecla hasta que se complete el periodo de la nota, porque en esta etapa todavía no disponíamos del generador de envolventes de amplitud, que mitiga este defecto donde el sonido de los clics era bastante notable.

Por otra parte, a excepción de la onda cuadrada donde el valor de la muestra es el valor máximo o mínimo de la resolución de bits definida, se ha incluido una tabla de datos que contiene los valores de incremento de la muestra en las ondas sierra y triangular, que hemos obtenido en C++ y cuyo código fuente se encuentra disponible en el Anexo 9.1. Como veremos en la sección 4.1.3, el valor de los datos que el DAC espera es de 24 bits con signo, pero los datos de incremento de esta tabla son de 40 bits, donde para ganar precisión disponemos de 24 bits para la parte entera y 16 bits para la parte decimal. Esto se conoce como notación Q, una forma de especificar números binarios en coma fija y que denotamos como Q24.16. Utilizamos coma fija para poder obtener una mayor precisión en los cálculos, puesto que las divisiones para obtener el incremento no son números enteros en ningún caso y nunca se alcanzaría el valor máximo. El formato coma flotante puede ser más preciso pero es demasiado costoso e innecesario y las operaciones en coma fija son más sencillas y rápidas. Cabe destacar que aunque los valores que se trasmiten al DAC son tanto negativos como positivos, los valores en esta tabla son sin signo y siempre positivos, ya que simplemente indican el valor de incremento de la muestra para cada ciclo del periodo. El Anexo 9.2 muestra una tabla de valores con las notas, frecuencias, incrementos en decimal y valores en coma fija hexadecimal.

En el caso especial de la onda senoidal, no guardamos los valores de incremento de una muestra sino que tenemos directamente un valor en concreto. A mayor número de muestras, mayor es la precisión para definir la forma de la onda pero también la cantidad de recursos que necesitamos. Sin embargo, la onda senoidal presenta una simetría no solo en su parte positiva y negativa sino también en las dos primeras mitades de la parte positiva. Por tanto aprovechando esta simetría, almacenamos solo los valores del primer cuarto, donde hemos incluido una cantidad de 4096 valores y debido a que es un tamaño suficientemente grande hemos creado un componente para que estos valores se almacenen en Block RAM, liberando de esta forma los recursos de LUTs. Hemos escogido el valor 4096 porque 16384 es una cantidad más que suficiente para obtener una precisión que ofrece una buena fiabilidad en la actualización de la muestra.

Por otra parte, aunque solo en el caso de la onda senoidal, el número de ciclos también se utiliza como índice en una nueva tabla que almacena el valor para obtener el incremento del índice para extraer el dato concreto de la otra tabla. En el Anexo 9.3 se adjunta el *script* de Python que hemos utilizado para obtener los valores de las muestras y en el Anexo 9.4 se han extraído en C++ los valores para obtener la suma del índice de la tabla con los valores de una muestra, es decir, para saber cuanto tenemos que sumar al valor que indexa la tabla que obtiene el valor en concreto.

4.1.3 Reproducción del sonido, el DAC

Ahora el sintetizador es capaz de generar distintas formas de onda con determinadas frecuencias y procedemos a realizar el diseño y las pruebas para capturar la salida del DAC. Por tanto hemos buscado previamente información sobre el protocolo estándar I2S (Integrated Interchip Sound) [20] publicado por primera vez en 1986, puesto que es el que utiliza el dispositivo Pmod.

El protocolo es un estándar para interconectar circuitos de audio digital, donde el bus I2S separa las señales de datos y del reloj. El bus consiste en al menos la señal de reloj serie, un reloj de palabra (o selección de palabra) y al menos una línea de datos multiplexados. También puede incluir un reloj maestro y una señal multiplexada de entrada para obtener una comunicación bidireccional. El reloj serie debe operar a una frecuencia múltiplo de la frecuencia de muestreo. Este multiplicador depende del número de bits por canal y del número de canales. La selección de palabra permite conocer al circuito receptor si los datos están siendo enviados al canal uno o dos, porque se pueden enviar dos canales por la misma línea de datos. En el caso de audio estéreo, la especificación establece que el canal izquierdo se transmite en el flanco de bajada del reloj de selección de palabra y el canal derecho en el flanco de subida [20]. La selección de palabra tiene un ciclo de reloj del 50% y debe tener la misma frecuencia que la frecuencia de muestreo. El primer bit transmitido después de una transacción del reloj de palabra es el bit menos significativo de la palabra anterior. La Figura 16 muestra un esquema de temporización básica del protocolo.

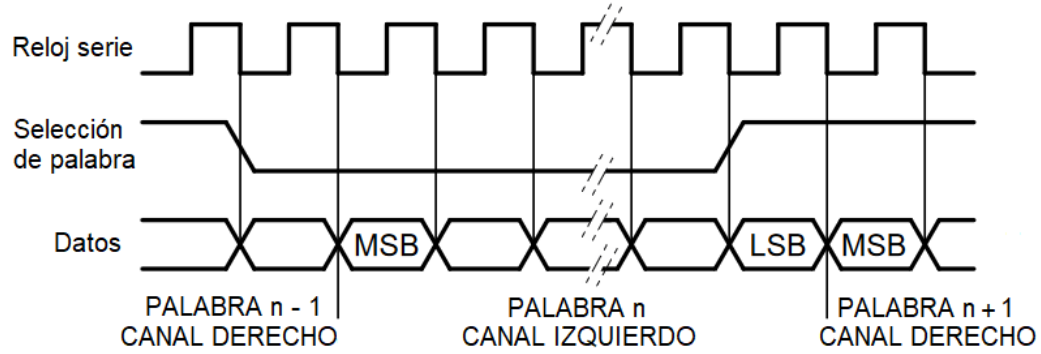

Figura 16: Temporización básica en el protocolo I2S. Fuente: [20]

Después de conocer el funcionamiento básico de este protocolo, hemos encontrado un proyecto para Vivado disponible en los repositorios de Github [18], realizado por Diligent para varias de sus placas donde ofrecen una guía de iniciación rápida para configurar un sistema de redirección de audio, de la entrada a la salida a una frecuencia de muestreo de 44.1 kHz donde se necesitan tres señales de control generadas por el sistema que son:

- Un reloj maestro (MCLK), a una frecuencia aproximada de 22.579 MHz.
- Un reloj serie (SCLK), el cual conmuta a cada 8 ciclos de MCLK.
- Una señal de selección de palabra, que conmuta a cada 64 ciclos de SCLK.

Sin embargo, hemos encontrado otro proyecto para Vivado en el foro de DigiKey [21], donde se presenta un proyecto que está adaptado para nuestra placa Basys-3 en el que el Pmod I2S2 está insertado en el conector JA. Se proporciona un proyecto de Vivado y VHDL para crear un sistema de reproducción de audio, donde la entrada del convertidor analógico digital se redirige a la salida del convertidor digital analógico y mediante los *switches* es posible controlar la señal *reset*, así como ajustar el volumen de salida.

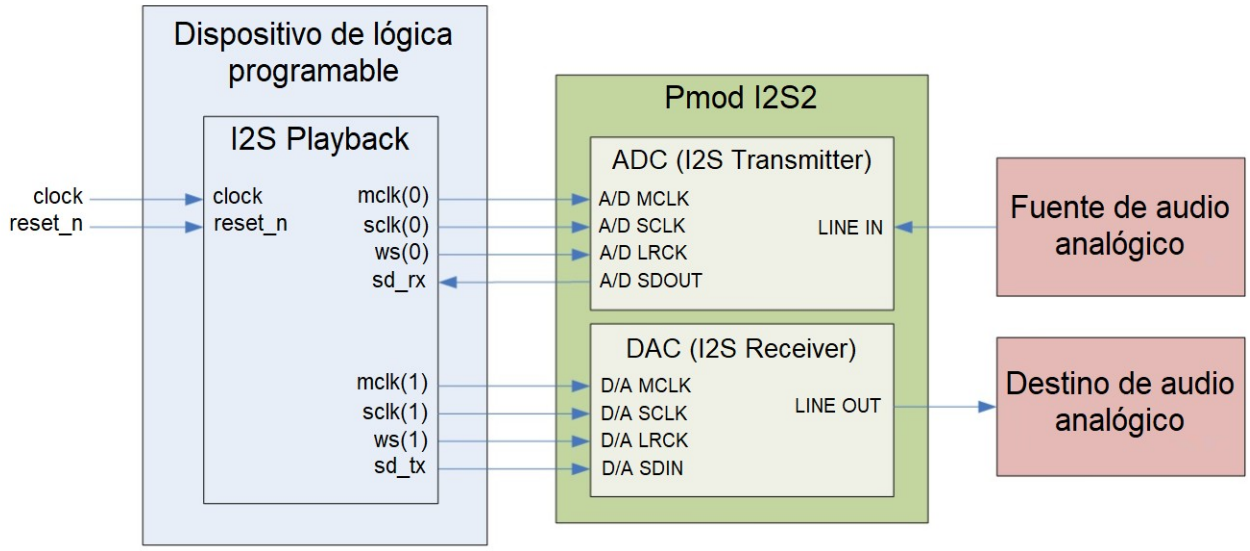

Figura 17: Diagrama de bloques del sistema. Fuente: [21]

La Figura 17 muestra un diagrama de bloques del sistema en este proyecto. El componente I2S Transceiver se encarga de manejar la recepción y transmisión según la especificación de bus del estándar I2S. El diseño del componente I2S Playback configura el componente Transceiver de la siguiente forma:

- El dispositivo Pmod I2S utiliza unos convertidores analógico a digital y digital analógico, en el que el componente Transceiver interfiere con éstos directamente. Ambos soportan una resolución de 24 bits, de forma que el Transceiver está configurado para una resolución de 24 bits (con signo) mediante su parámetro genérico d_width.
- El diseño de I2S Playback utiliza la frecuencia de muestreo de 44.1 kHz, en consecuencia lo más apropiado es usar el ratio de SCLK/LRCK de 64 y para MCLK/LRCK de 256 [19].
- Como la frecuencia para seleccionar la palabra es de 44.1 kHz, la frecuencia del reloj serie es 64 veces ésta y la frecuencia del reloj maestro es cuatro veces más. Por consiguiente, tenemos una frecuencia de reloj maestro de 11.29 MHz.

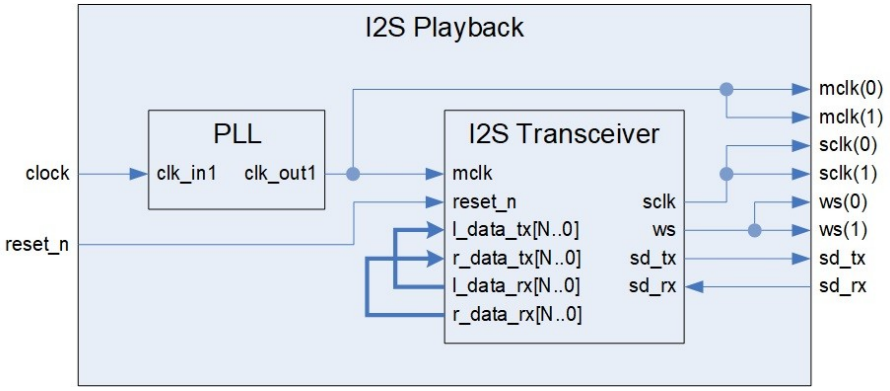

Figura 18: Arquitectura interna de I2S Playback. Fuente: [21]

La Figura 18 muestra un esquema donde se observa la arquitectura interna del diseño de I2S Playback, que consiste en el componente I2S Transceiver junto con un componente PLL (Phase-Locked Loop). Basys-3 incluye un único oscilador a 100 MHz y el reloj de entrada puede impulsar MMCM (Mixed-Mode Clock Manager) o un PLL para generar los relojes de varias frecuencias. El componente Transceiver requiere un reloj maestro para operar, del cual deriva el reloj serie y la señal de selección de palabra, necesarias para la interfaz I2S. El reloj maestro de 11.29 MHz se genera utilizando este PLL. La frecuencia que genera la salida en el PLL se puede configurar y se genera a partir de la ecuación (2) [28]. Para conseguir una frecuencia aproximada de 11.29 MHz hemos configurado el PLL con un F_{input} igual a 100 MHz, M igual a 46, D igual a 81.5 y O igual a 5, obteniendo un F_{output} de 11.28834 MHz.

$$F_{output} = \left(\frac{F_{input} * M}{D * O} \right) \tag{2}$$

Tabla 3: Descripción de puertos de I2S Playback.

Puerto	Tamaño	Modo	Tipo de datos	Descripción
clock	1	in	std_logic	Reloj del sistema
reset_n	1	in	std_logic	Señal reset asíncrona
mclk	2	out	std_logic_vector	Reloj maestro
sclk	2	out	std_logic_vector	Reloj serie
ws	2	out	std_logic_vector	Word select
sd_tx	1	out	std_logic	Transmitir datos
sd_rx	1	in	std_logic	Recibir datos

La Tabla 3 muestra la descripción de los puertos de I2S Playback. Las señales del reloj maestro y de selección de palabra son puertos de entrada en nuestro componente Synthesizer, puesto que no comenzamos a generar las muestras de cada onda hasta que no llega un flanco de subida de la señal de selección de palabra. Por otro lado, se utiliza el reloj maestro como señal de reloj para evitar problemas de temporización, puesto que el reloj del sistema funciona a 100 MHz y no es múltiplo del reloj maestro. Finalmente hemos integrado en este proyecto los generadores de onda que ya hemos visto y se ha procedido a comprobar que la reproducción del sonido es la esperada. Una característica del DAC que creemos que es conveniente destacar es que parece tener un cierto error cuando se transmiten datos que se encuentran muy distantes, por ejemplo con una onda sierra cuando pasa al valor mínimo tras alcanzar el máximo o con la onda cuadrada, donde los valores siempre se encuentran en un extremo. La Figura 19 muestra una ilustración en la que se observan discrepancias en los valores mínimos de la onda sierra capturada desde el Pmod I2S2, respecto a la sierra digital de Audacity.

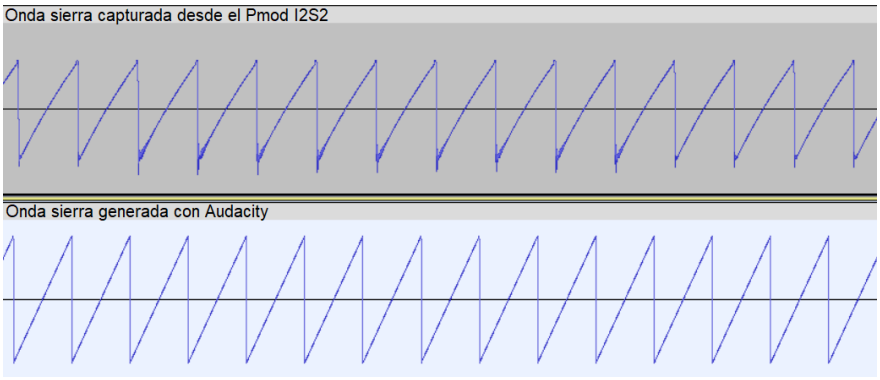

Figura 19: Comparación de la onda sierra, digital frente a capturada del Pmod I2S2

Para descartar que se trata de un problema con la toma de muestras, se ha encaminado la onda sierra perfecta generada por el *software* Audacity, desde una salida a otra entrada de la tarjeta de sonido que además también incluye unos convertidores muy similares, para comprobar que este problema no sucede en la captura de las muestras, por tanto podemos deducir que se trata de un problema relacionado directamente con el dispositivo Pmod I2S2. Por otra parte, se podría mitigar este error creando una muestra intermedia con un valor a cero, sin embargo estamos alterando de esta forma el número de ciclos para completar su periodo con lo que incurriríamos en un desajuste en la frecuencia de la nota.

Seguidamente hemos procedido a desarrollar un programa simple para la recepción y envío de datos desde una aplicación del computador, donde poder seleccionar entre distintas frecuencias o notas sobre la marcha y analizar la respuesta cuando las estamos cambiando.

4.1.4 Transmisión y recepción de datos con UART

La transmisión y recepción de los datos está basada en UART, un protocolo para el intercambio de datos en serie entre dos dispositivos. Es un protocolo simple en el que se utilizan dos hilos para transmitir y recibir datos en ambas direcciones. La comunicación puede ser:

- Simplex, cuando los datos se envían solo en una dirección
- Semiduplex, donde cada extremo puede comunicar datos pero solo uno al mismo tiempo
- Full duplex, cuando ambos pueden transmitir simultáneamente

Aunque hoy en día este protocolo se está sustituyendo por otros que en lugar de comunicarse por un puerto serie usan tecnologías como Ethernet, es un protocolo soportado por nuestro dispositivo Basys-3 y que podemos utilizar a través del USB. UART se sigue utilizando en aplicaciones que requieren velocidades más bajas porque es simple, económico y fácil de integrar.

En UART se toman bytes de datos pero se transmiten bits individuales de forma secuencial y el receptor forma los bits en bytes completos. Es asíncrono y el transmisor y receptor no comparten la misma señal de reloj, por tanto ambos deben transmitir a la misma velocidad con el fin de mantener la temporización de los bits. Esta velocidad se mide en baudios, que es el número de veces que una señal de comunicaciones serie cambia de estado. Si la señal cambia una vez por cada bit de datos entonces un bit por segundo equivale a un baudio. Las velocidades en baudios más comunes en UART son 4800, 9600, 19.2k, 57.6k, y 115.2k y como se he comentado ambos deben establecer la misma, pero también deben tener la misma estructura y trama.

Las tramas en UART contienen un bit de inicio y de parada, los bits de los datos y opcionalmente un bit de paridad [22]. Debido a su naturaleza asíncrona, el transmisor necesita indicar que va a enviar bits de datos,con el bit de inicio. Por otra parte, el bit de parada indica que los datos han terminado y debe quedar a la espera. Los bits de datos se transmiten inmediatamente después del bit de inicio y pueden contener de cinco a nueve bits, pero lo más habitual son siete u ocho bits. El bit menos significativo se transmite primero de forma habitual. El bit opcional de paridad se utiliza para la detección de errores y se inserta entre el último bit de datos y el bit de parada. El valor del bit de paridad depende de si se ha definido una paridad par o impar. En la paridad par, el bit se ajusta para que el número de unos de la trama sea par y en la paridad impar para que sea impar. La Figura 20 muestra un ejemplo de una trama completa en UART.

bit de inicio	bits de datos	bit de paridad	bit de parada
(1 bit)	(de 5 a 9 bits)	(1 bit o nada)	(1 o 2 bits)

Figura 20: Ejemplo de una trama en UART

Al buscar información sobre este protocolo hemos encontrado un tutorial [23], en un blog de una comunidad que se dedica al aprendizaje de *hardware*, en el que se trabaja con la transmisión y recepción de datos con UART en la placa Basys-3 y sobre un proyecto en Vivado con VHDL.

Por tanto, hemos seguido los pasos para construir un proyecto nuevo que, tras comprobar que su funcionamiento es correcto, hemos terminado integrando en el proyecto con los generadores de ondas y el DAC. Este proyecto utiliza una configuración de 115.2k baudios, un bit de inicio y otro de parada, ocho bits de datos y sin bit de paridad. El proyecto consta de:

- UART_controller, es el componente de nivel más alto y conecta el diseño con el exterior.
- button_debounce, conecta los botones para transmitir con el resto del proyecto y prevé que actúe múltiples veces con una sola pulsación
- UART, combina los componentes para transmitir y recibir
- UART_tx, contiene toda la lógica para transmitir
- UART_rx, contiene toda la lógica para recibir

Veamos con más detalle los componentes para transmitir y recibir los datos. La Figura 21 muestra el diagrama de bloques del transmisor de UART, donde cada bloque azul representa un proceso.

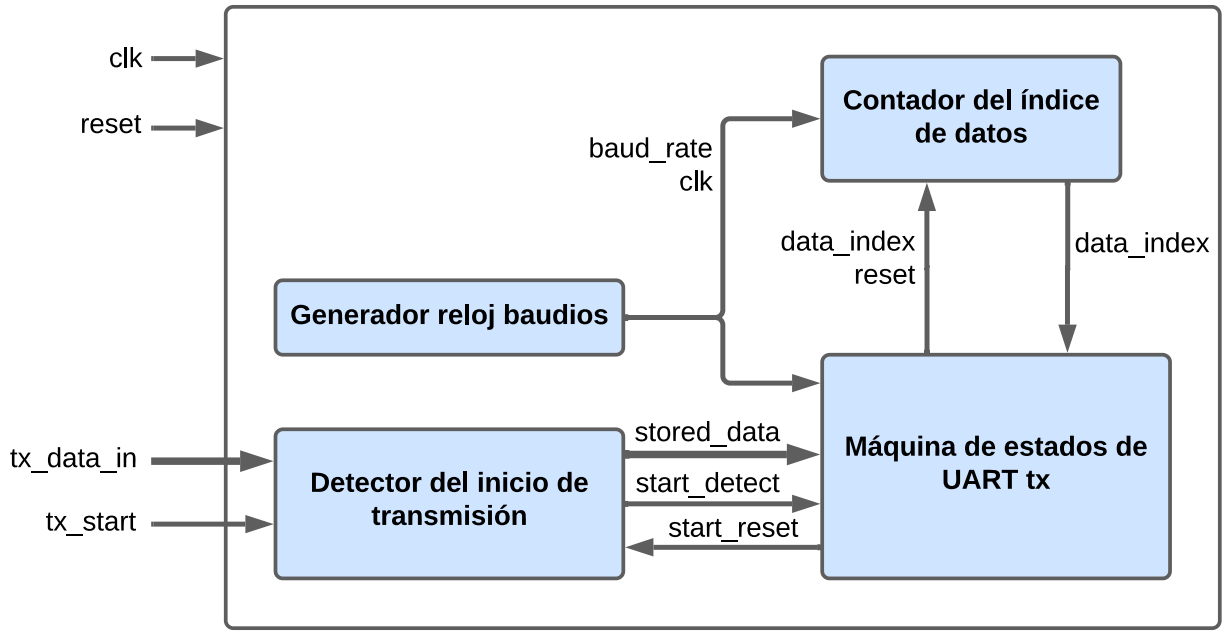

Figura 21: Diagrama de bloques del transmisor de UART

- El generador del reloj de baudios mantiene una señal que cuenta el número de ticks de baud_clk_ticks del reloj maestro, se trata de una constante que refleja el ratio entre el reloj maestro y la tasa de baudios que hemos fijado.
- El detector del inicio de transmisión funciona a la frecuencia del reloj maestro y captura los cambios en la señal que indican un inicio de transmisión, es necesario para la máquina de estados. También se encarga de almacenar en una señal los datos que se van a transmitir.
- El contador del índice de datos es un contador simple de cero a siete que funciona a la frecuencia de la tasa de baudios y se utiliza en la máquina de estados para recorrer el vector de datos almacenados y enviar los bits uno a uno.

La Figura 22 muestra un diagrama de la máquina de estados del componente transmisor.

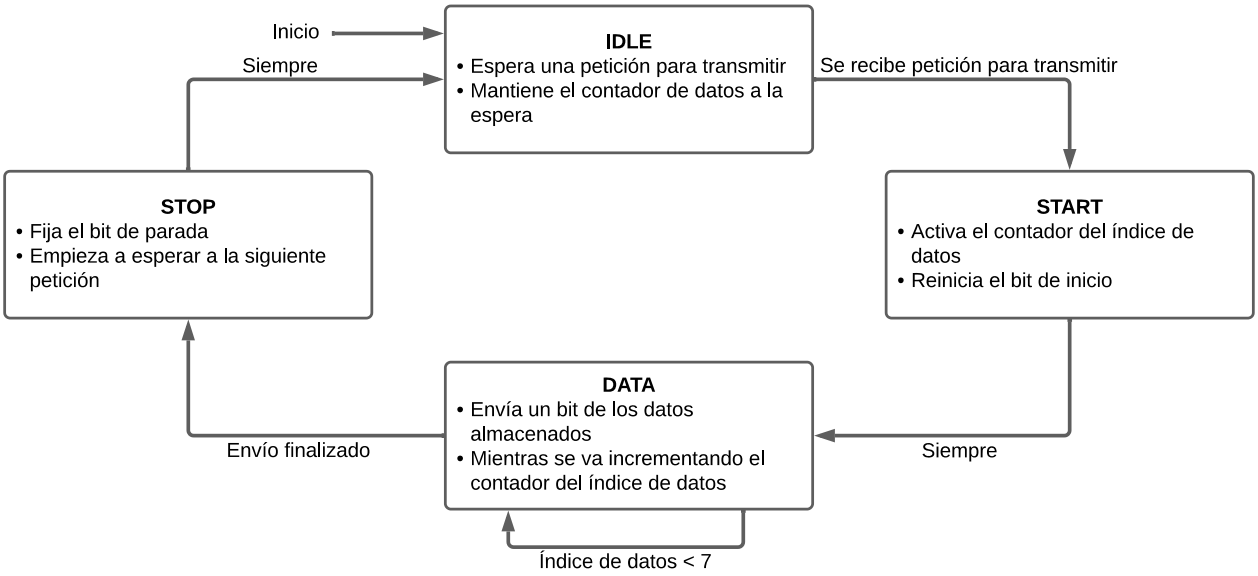

Figura 22: Diagrama de la máquina de estados del transmisor UART

El componente receptor de datos tiene otro enfoque, puesto que los contadores se encuentran en la máquina de estados. Tiene una señal de reloj que es 16 veces más rápida que el reloj de la tasa de baudios. Esto es necesario para fijar un punto de captura durante la mitad de la duración en la que se está recibiendo un bit. La Figura 23 muestra un diagrama de la máquina de estados en el componente receptor.

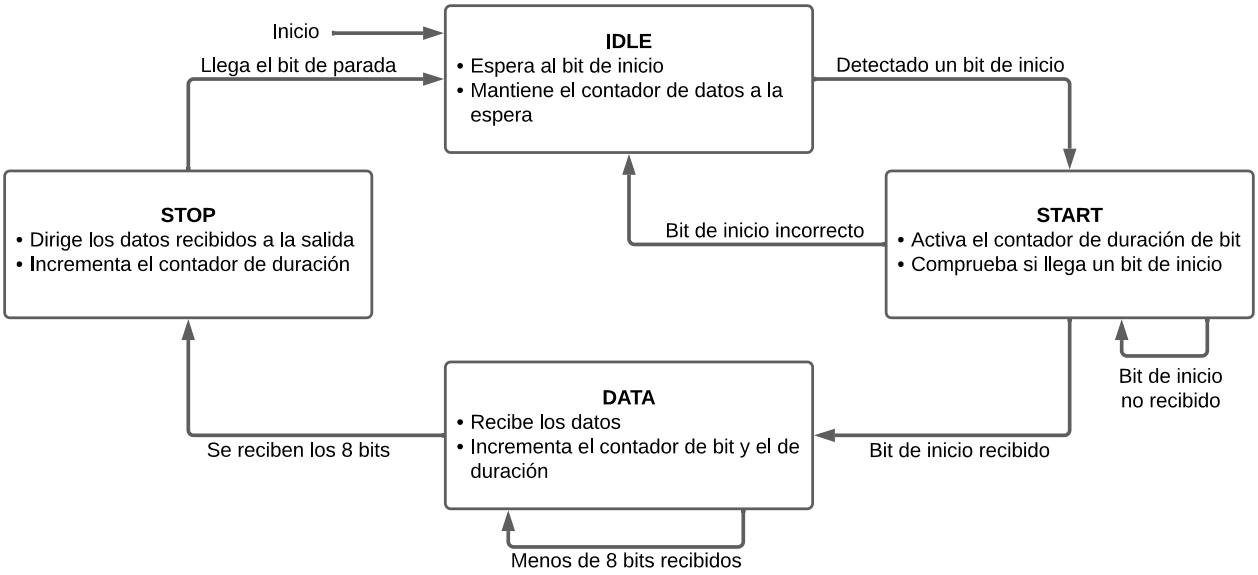

Figura 23: Diagrama de la máquina de estados del receptor UART

Para comprobar que el diseño es correcto se ha utilizado la aplicación Tera Term, configurando el puerto serie, que se puede obtener desde el administrador de dispositivos, junto con la tasa de baudios, la cantidad de bits para datos y el bit de parada e indicando que no hay bit de paridad. Se ha verificado que el funcionamiento es correcto y después se ha programado una aplicación en el lenguaje C++ que nos servirá como base para la aplicación con interfaz, cuyo código fuente está en el Anexo 9.5. Se ha estudiado la comunicación serie [24] antes de desarrollar esta aplicación.

4.2 Fase 2: Osciladores, polifonía y optimización de los recursos

En la segunda fase, donde ya tenemos un sintetizador monofónico en el que podemos seleccionar entre varias notas distintas mediante pulsaciones en el teclado del computador, hemos creado un segundo oscilador donde podemos cambiar el tipo de onda y también hemos desarrollado la polifonía, de forma que cada oscilador puede crear al mismo tiempo más de una forma de onda.

Puesto que ya tenemos varias formas de onda, debemos realizar las operaciones necesarias para realizar la mezcla de las muestras, desde las voces de cada oscilador hasta la suma de los dos osciladores. En este momento también se han añadido los parámetros para controlar los niveles de volumen de cada oscilador, así como del volumen general o *Master* del sintetizador. Después hemos incorporado dos nuevos parámetros, para cambiar los semitonos de cada oscilador de forma independiente y realizar pequeños ajustes de afinación en las notas. Una vez se ha logrado este objetivo, donde controlamos de forma más precisa la frecuencia de la onda, hemos procedido a incluir un parámetro que crea el efecto de *portamento*, donde la frecuencia de la nota actual se desliza desde la frecuencia de la nota que previamente sonaba. Por otra parte hemos ampliado la funcionalidad del componente que genera la onda cuadrada, de forma que podemos modificar el ancho de pulso.

4.2.1 Osciladores, polifonía y mezcla

En este apartado hablamos en primer lugar de cómo se han incorporado varias voces en un solo oscilador para después incorporar un segundo oscilador y tratar la mezcla de todas las muestras.

El primer planteamiento fue el de duplicar el número de instancias de cada componente generador de muestras o incluir un bucle en cada uno de éstos en función del número de voces. Sin embargo de esta forma estamos indicando que necesitamos generar todos los valores en un mismo ciclo y por este motivo se duplica el número de recursos utilizados, por ejemplo en el caso de los datos que se almacenan en memoria Block RAM como son los valores de incremento de la onda sierra y triangular o las muestras de la onda senoidal, porque en un mismo ciclo solo podemos acceder a un dato de la memoria. Emplear este enfoque supone alcanzar el máximo de recursos disponibles en nuestra placa para pocas voces, por tanto necesitamos plantear otro diseño.

Los componentes que generan los tipos de onda no empiezan a realizar su trabajo hasta que no llega un flanco de subida del reloj de selección de palabra, con un periodo igual a la frecuencia de muestreo. No obstante el reloj de estos componentes, el reloj maestro funciona a 11.29 MHz, es decir tenemos 256 ciclos disponibles para realizar los cálculos de todas las muestras antes de que llegue otro flanco de subida del reloj de selección de palabra. Un planteamiento más adecuado es por tanto aprovechar todos los ciclos disponibles hasta que se requiere generar nuevas muestras. Siguiendo esta idea, hemos incluido una máquina de estados en cada uno de los componentes de cada tipo de onda y en el componente WaveSelector tenemos un proceso que activa la señal *start* cuando llega un flanco de subida del reloj de selección de palabra. Esta señal se encamina a un puerto de entrada de cada generador y solo estará activa durante un ciclo del reloj maestro. La máquina de estados de los generadores consta de tres estados:

- WAIT_START, es el estado inicial y se espera a la activación de *start* para pasar al estado de lectura de datos.
- READING, es un estado necesario para esperar a obtener los datos de memoria porque al cambiar la dirección del dato, éste no está disponible hasta el siguiente ciclo.
- WORKING, donde vamos incrementando el índice de la onda en la que trabajamos en este ciclo y cambiamos al estado de lectura, hasta que llegamos al número total de voces para volver al estado inicial.

El estado de lectura solo existe en las ondas sierra, triangular y senoidal, aunque no en la onda cuadrada porque este tipo de onda no requiere obtener datos desde memoria. En otro proceso es donde se generan las muestras de cada onda, que solo funcionará cuando una señal *enable* esté activa y que solo estará activa cuando el estado es el de trabajando. Por otra parte, la onda sierra y triangular también incluye otro estado de espera, para que una vez se active la señal *start* pasar a este estado en el que esperan su turno, de forma que cuando el generador de un tipo termina es entonces cuando comienza el siguiente. Para lograr este objetivo tienen de un contador de ciclos de espera en función del número de voces. Por tanto, con este enfoque primero actúan las ondas senoidal y cuadrada simultáneamente, después la onda sierra y finalmente la onda triangular.

Con esta solución hemos conseguido elevar el número de voces de manera significativa en esta fase donde todavía no hemos implementado LFOs y por tanto la cantidad de datos para las tablas de los incrementos o para obtener el índice de salto en la tabla de datos en la onda senoidal son menores y almacenadas en LUTs. Mientras que en la primera aproximación se superaba el límite de 20800 LUTs disponibles para solo seis voces, aprovechar los ciclos disponibles entre distintos flancos de la selección de palabra reduce en gran cantidad la utilización, como muestra la Tabla 4.

Tabla 4: Utilización de recursos en función del número de voces

Voces	LUTs	FF	BRAM	DSPs
***Sin** la optimización con la máquina de estados*				
6	**21520**	1664	18	36
***Con** la optimización con la máquina de estados*				
6	3546	1820	3	9
8	3681	2284	3	11
10	4871	2758	3	13
12	5559	3233	3	15
14	6291	3697	3	17
16	6542	4162	3	19
18	7003	4642	3	21
20	7649	5118	3	23
22	8217	5577	3	25
24	8649	6060	3	27
26	9807	6522	3	29
28	10453	7002	3	31
30	10820	7455	3	33
32	11373	7938	3	35

Seguidamente hemos aprovechado estos datos para obtener una recta de regresión lineal con la que podemos estimar la cantidad de recursos necesarios para un número determinado de voces.

En nuestro sintetizador, los recursos que más utilizamos son LUT y DSP, por este motivo hemos obtenido una línea de tendencia para cada tipo de recurso. La Figura 24 muestra los resultados de cada tendencia, donde en primer lugar se observa muy poca dispersión de los puntos respecto de la recta. En consecuencia los coeficientes de determinación tienen un valor muy próximo a uno en el caso de la línea para LUT y de uno para DSP, lo que indica la buena calidad de este modelo en la predicción de la utilización de estos recursos en función del número de voces. El eje Y izquierdo muestra la cantidad de LUT mientras que el eje Y derecho la cantidad de DSP. EL eje X muestra el número de voces.

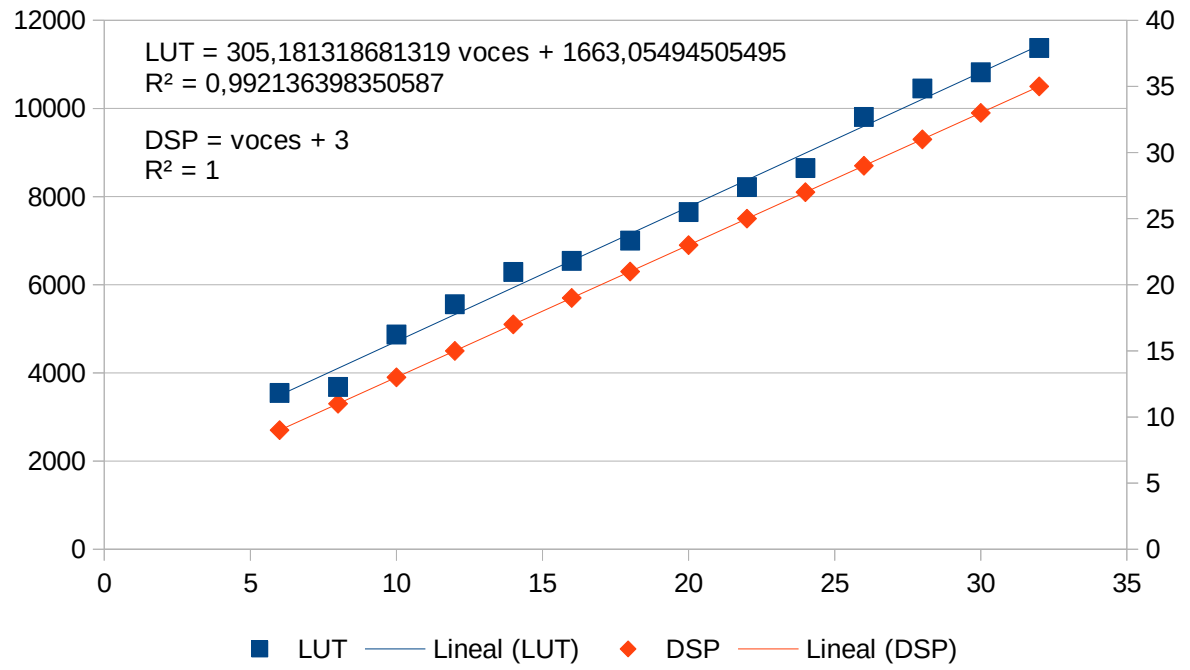

Figura 24: Recta de regresión de la utilización de recursos LUT y DSP

Según las ecuaciones que el modelo ofrece, podemos obtener hasta un máximo de 87 voces para alcanzar el límite de recursos DSP y de 62 voces antes de alcanzar el límite de LUTs, por tanto 62 voces sería el límite máximo que podemos obtener en esta fase, donde destacamos que todavía no hemos implementado la gestión de la dinámica del sonido que incrementará la utilización de los recursos.

Ahora que podemos obtener un número variable de voces de forma eficiente, procedemos a incluir un segundo oscilador. Para lograr este objetivo solamente tenemos que incrementar el número de voces y el total se reparte entre el número de osciladores, en consecuencia como el sintetizador tiene dos osciladores el número genérico de la constante del número de voces debe ser par.

El componente Synthesizer tiene un proceso donde para cada oscilador se realiza por una parte la suma de todas sus voces y por otra se lleva la cuenta de las voces que actualmente tienen una muestra con valor distinto a cero, que conocemos gracias al vector *note_enable* y que se gestiona en la sección 4.3.1, para realizar la operación de multiplicación que es necesaria para normalizar el volumen del oscilador sin llegar a un desbordamiento de la señal. Al contar el número de voces que suenan actualmente obtenemos siempre el máximo volumen en el oscilador, por ejemplo supongamos que el oscilador soporta hasta seis voces pero solo suenan dos, gracias a esta cuenta podemos dividir la suma total entre dos, o lo que sería equivalente podemos multiplicarlo por un 50%, de esta forma nunca desbordaremos la resolución de bits que hemos establecido. Utilizamos coma fija para realizar este proceso puesto que se trata de una multiplicación por un número decimal. Para esto tenemos una tabla de valores constantes que nos indica el valor por el que tenemos que multiplicar para la cuenta de voces, que extraemos de otra tabla con los valores en coma fija con una precisión de 16 bits. De esta forma, multiplicamos la suma de las ondas por el número decimal multiplicado por dos elevado a 16 desplazando el resultado 16 bits a la derecha de forma que operamos unicamente con valores enteros. Por ejemplo para 2 voces, multiplicamos la suma de ambas por 32768 y desplazamos el resultado 16 bits. A mayor cantidad de bits mayor será la precisión en el resultado, siendo 16 bits un valor adecuado para el propósito que queremos lograr. Las multiplicaciones para obtener el volumen del oscilador se realizan de forma análoga.

Por otra parte, antes de sumar los dos osciladores debemos extender su signo duplicando el bit de mayor peso puesto que se trata de valores con signo y la suma de ambos puede desbordar con lo que obtendríamos un error en el resultado. Los osciladores se suman y multiplican con la misma metodología que se ha utilizado para combinar las voces de un oscilador. Finalmente en esta fase se encamina la suma total a la señal del DAC, pero en la siguiente se encaminará al generador de ruido una vez esté implementado.

4.2.2 Ajustes de afinación

Para obtener un mayor control de la frecuencia de las notas en cada oscilador se han incorporado al sintetizador los parámetros *semitone* y *finetune,* con los que se puede ajustar con más precisión la frecuencia de forma independiente para cada oscilador. Por otra parte, también hemos incluido el efecto *portamento*, que solo es aplicable al oscilador cuando su LFO no está activa.

4.2.2.1 Semitono

Cada oscilador contiene un parámetro donde se puede ajustar el tono de la nota hasta una escala máxima superior o inferior, es decir, hasta un máximo de 12 semitonos. Con este ajuste podemos reproducir distintas notas en cada oscilador con una misma pulsación en el teclado.

Cambiar el semitono no modifica la frecuencia en sí, sino que modifica el número de la nota en el componente Parameters y por tanto al cambiar el código también cambia la frecuencia, es decir el periodo de cada onda al generar todas las voces de un oscilador. Cabe destacar que el parámetro se puede modificar sin necesidad de soltar y volver a pulsar la nota en el teclado y este cambio debe verse reflejado inmediatamente. Simplemente cambiar el código en el vector *note_code* es suficiente para que funcione todo adecuadamente porque los generadores tienen como entrada el número de ciclos para completar el periodo y éste se obtiene de una tabla en función del código de la nota.

Sin embargo, cabe destacar que si pulsamos notas en los extremos de un teclado y reducimos o aumentamos el semitono, podríamos obtener un código de nota que se salga del rango permitido, que recordamos se debe encontrar entre cero y 128. Por este motivo necesitamos una variable auxiliar en el proceso donde actualizamos los valores de Parameters para finalmente asignar el nuevo código de la nota al vector solo si es un valor válido. Además necesitamos dos señales, una por oscilador, para guardar el último valor del semitono puesto que necesitamos restar el valor actual menos el valor guardado para saber qué cantidad sumar o restar al semitono. Por ejemplo, los valores del parámetro de la interfaz se encuentran en el rango [-12,12] y el valor actual es 11. Si ahora nos llega una actualización del semitono a 12, debemos restar 12 menos 11 para sumar uno al código de la nota. El último valor que hemos guardado y el valor del semitono recibido no es obligatoriamente una diferencia de uno, ya que en la interfaz además de utilizar el parámetro rotatorio también podemos introducir con el teclado un nuevo valor, pero con el método de la resta nos aseguramos que el funcionamiento siempre va a ser correcto.

4.2.2.2 Fine tune

Continuando con los ajustes de la frecuencia, cada oscilador dispone de un segundo parámetro de forma que podemos ajustar con más precisión la nota con una pequeña afinación. Este parámetro recibe el nombre de *fine tune*, o también lo hemos nombrado *detune* ya que desafinar un oscilador respecto a otro consigue el efecto que recibe este nombre y da una sensación de expansión de la señal estéreo.

El parámetro de la interfaz puede tener un valor mínimo de cero y un valor máximo de 100, siendo 50 el valor central donde no se aplica este efecto. El componente selector de ondas contiene un proceso en el que se aplica la ecuación (3).

$$detune(wavenow) = ((50 - parámetro) * ciclos(nota(wavenow)))/1024 \quad (3)$$

Restamos 50 al valor del parámetro que obtenemos de la interfaz porque es el valor central y si el valor es menor tiene que sumar ciclos al periodo mientras que si es mayor tiene que restarlos. En el valor central, el resultado de la multiplicación será cero por tanto no se aplica este efecto. Ciclos es el vector del cual obtenemos el periodo para completar la frecuencia de la nota sobre la que se está trabajando en el ciclo actual. Desplazamos el resultado 10 bits para reducir la cantidad de ciclos *detune* que modificarán la frecuencia, en otras palabras, este valor es una constante con la que definimos la cantidad máxima de afinación que permitimos al usuario. Además, es adecuado que sea un valor potencia de dos para economizar la utilización de recursos.

Recordamos que las notas con frecuencia más bajas necesitan un mayor número de ciclos para completar el periodo de onda, por tanto necesitan también más cantidad de ciclos para ajustar la afinación. No obstante, mediante esta ecuación nos aseguramos que esto se va a cumplir porque la operación se realiza en función del periodo de la nota. Pero cabe destacar que aunque es cierto que la proporción para todas las notas es correcta, las notas con las frecuencia más altas tienen menos ciclos y por tanto todavía menos ciclos para afinar la nota, de manera que con las notas de escalas más altas no disponemos de gran precisión para realizar ajustes en la afinación.

4.2.2.3 Portamento

El parámetro *portamento* también está relacionado con la frecuencia de la nota, ya que se produce un desplazamiento progresivo desde la última nota hasta la actual. En consecuencia, la cantidad de ciclos para completar el periodo de la nota debe disminuir de forma gradual si la nota anterior es más baja y debe aumentar si la nota anterior tiene una frecuencia más alta. El tiempo que tarda está definido por el valor del parámetro de la interfaz y debe ser proporcional para todas las notas, por ejemplo si el valor de *portamento* es 50, el tiempo para ir desde la frecuencia 110 Hz a 440 Hz debe ser igual que el tiempo para ir desde la frecuencia 220 Hz a 440 Hz.

Por este motivo, necesitamos conocer el número de ciclos que debemos esperar a la frecuencia del reloj maestro para incrementar o reducir un ciclo al periodo para completar una onda. Como la nota con frecuencia más baja tiene un periodo de 5391 ciclos y la de frecuencia más alta tiene un periodo de un ciclo, la diferencia más grande puede ser como máximo 5390. Por tanto, incluimos una tabla de 5390 valores que indican el número de ciclos de espera para ir desde la nota anterior con una determinada frecuencia hasta la frecuencia actual. Como es una cantidad suficientemente grande almacenaremos estos valores en Block RAM. Pero primero debemos conocer cuál es la última nota pulsada, por tanto en el componente Parameters que es donde actualizamos los valores de todos los parámetros, pero también donde detectamos las pulsaciones del teclado, guardamos en una señal el código de la nota. Como el código de la nota también es una señal y su valor no cambia hasta que el proceso finaliza, asignaremos el valor de la nota actual en el siguiente ciclo en el que se vuelva a pulsar una nota, consiguiendo de esta forma almacenar la nota previa. Es importante destacar que necesitamos almacenar en un vector el valor de la última nota para cada oscilador, puesto que debido al parámetro semitono pueden ser distintas. Este vector se encamina al puerto de entrada del componente selector de ondas, ya que es éste quien tiene el proceso donde se va modificando el periodo de la nota actual. En este proceso si no hay *portamento*, el periodo de una onda será el número de ciclos de esa nota más los ciclos del ajuste de afinación, pero si el valor de *portamento* es mayor a cero entonces se realizan los cálculos.

En el proceso de cálculo primero se comprueba si la nota se acaba de pulsar, pero de esta tarea se encarga el generador de envolventes en la sección 4.3.1, para inicializar el periodo de la nota actual con los ciclos de la última nota pulsada y también inicializamos el valor de los ciclos de espera que obtenemos desde memoria.

La dirección para obtener el dato de los ciclos de espera será la diferencia en valor absoluto del periodo de la nota previa menos la nota actual. Además este valor debe multiplicarse por el valor del parámetro de la interfaz en coma fija. En los ciclos posteriores y mientras la nota está activada, contamos en una variable, que es un vector para cada onda, si hemos alcanzado la cantidad de ciclos de espera para variar el periodo de la nota actual, hasta que llegamos al periodo de la nota actual con la suma de los ciclos de afinación.

Hemos obtenido la tabla de valores de los ciclos de espera con un programa en C++, cuyo código fuente se encuentra disponible en el Anexo 9.6, donde guardamos la espera para la diferencia de todas las posibles combinaciones de cada nota.

4.2.3 Onda cuadrada con ancho de pulso variable

Para modificar el tono de una nota también se puede variar el ancho de pulso, es decir, modificar la duración del tiempo que el dato se mantiene a nivel alto y a nivel bajo durante un periodo para completar la onda, aunque esto solo es posible para la onda cuadrada debido a su forma.

El parámetro de la interfaz funciona de forma análoga al del ajuste afinación, con un valor mínimo de cero y un valor máximo de 100, siendo 50 el valor central en el que no se aplica modificación. Para lograr este objetivo, simplemente tenemos que realizar una multiplicación en coma fija de los ciclos para completar un periodo y del parámetro de la interfaz, para obtener el número de ciclos que se mantiene a nivel alto y una resta del periodo menos esta multiplicación para obtener los ciclos a nivel bajo. Cuando el parámetro se encuentra en su valor central seguimos obteniendo una onda cuadrada simétrica, es decir, con el mismo número de ciclos a nivel alto y bajo, pero si el valor es menor de 50 tendremos menos ciclos a nivel alto, más a nivel bajo y viceversa.

Por otra parte, cuando se ha terminado la implementación, hemos realizado la toma de muestras para verificar el correcto funcionamiento y hemos observado que los valores obtenidos en el DAC no se ajustan exactamente a los valores en simulación, que muestra la Figura 25. Los valores que hemos capturado reducen el valor del nivel alto cuando reducimos el ancho de pulso y también aumentan el nivel bajo a medida que incrementamos el valor del parámetro. No obstante, parece que este comportamiento es normal en cualquier sintetizador que modula el ancho de pulso.

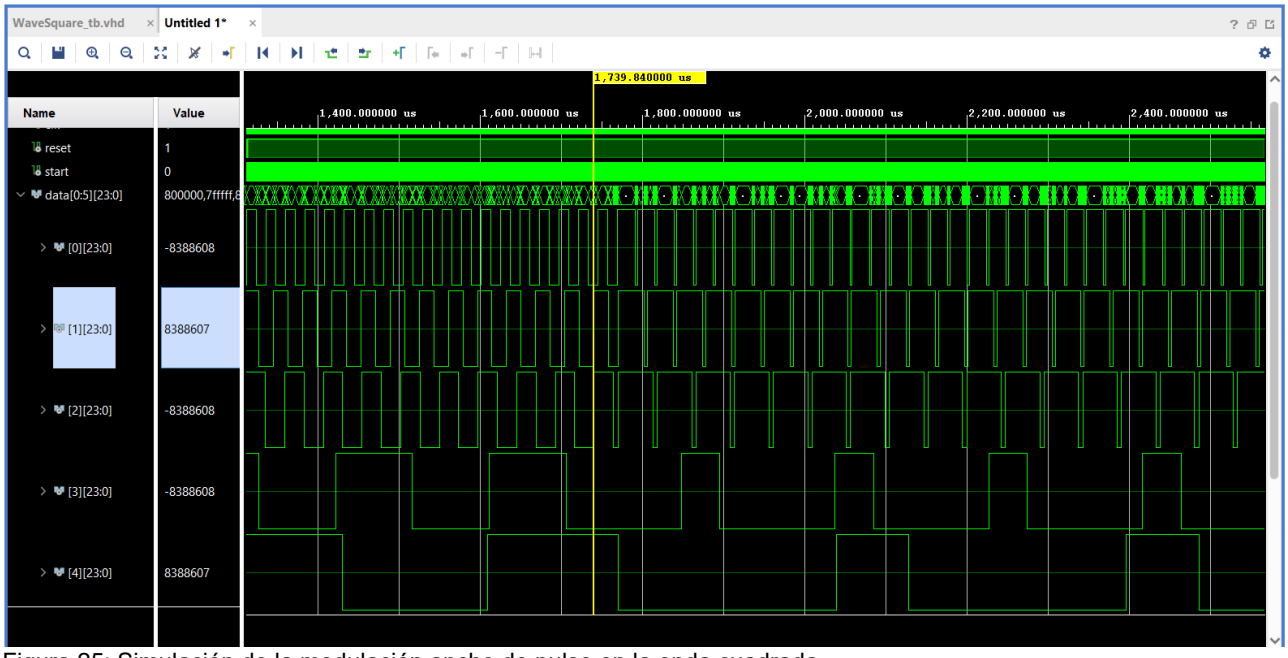

Figura 25: Simulación de la modulación ancho de pulso en la onda cuadrada

4.3 Fase 3: Gestión de la dinámica en el sonido

En esta fase hemos trabajado en las modulaciones, diseñando en primer lugar el generador de las envolventes de amplitud para controlar de forma dinámica el valor de amplitud de cada muestra en las voces de un oscilador. Posteriormente hemos diseñado los componentes para el oscilador de baja frecuencia, contienen los cuatro tipos de ondas básicas como los osciladores del sintetizador y modifican la frecuencia de cada voz en éstos, esto es también conocido como modulación de *pitch*. Cuando se ha verificado que el funcionamiento es correcto, se han incorporado los filtros de estado variable con los que mediante unas ecuaciones se puede incorporar simultáneamente un filtro de paso bajo, de paso alto, de banda y de rechazo de banda. Mientras se ha estudiado este componente realizando las simulaciones y pruebas en la placa para lograr optimizar los recursos, hemos aprovechado la LFO para modular la frecuencia de corte y la resonancia de los filtros. Por último se ha trabajado en el componente que genera ruido y en la gestión de la señal de entrada para los filtros, que puede conmutar entre el la entrada del Pmod o del propio sintetizador.

4.3.1 Generador de envolventes de amplitud

El generador de envolventes es uno de los elementos más importantes de un sintetizador, ya que mediante éste podemos modular de forma dinámica los valores para distintos parámetros. Es un elemento que en principio se puede asociar a varias fuentes, como por ejemplo la frecuencia de la nota o también denominado *pitch*, aunque en este proyecto hemos establecido la modulación a la amplitud de la onda unicamente. El generador ADSR consta de cuatro fases [1]:

- Ataque, es la fase en la que la amplitud de la onda se encuentra en el valor mínimo hasta que se obtiene una pulsación del teclado y éste va incrementando hasta el valor máximo y se cambia a la siguiente fase.
- Decaimiento, es la fase que define el tiempo que se tarda para modificar la amplitud del valor máximo hasta el valor establecido en la siguiente fase. Si dejamos de pulsar la tecla, se cambia directamente a la fase de liberación, pero si se mantiene cambiamos a la etapa siguiente cuando se alcanza el valor máximo.
- Sostenimiento, es la cantidad de amplitud que se mantiene mientras se sigue pulsando la tecla, hasta que ésta se suelta y se cambia a la siguiente fase.
- Liberación, es el tiempo que tarda en disminuir la amplitud desde el valor que tiene en la fase de sostenimiento hasta cero.

El generador de envolventes es conocido habitualmente como ADSR *envelopes*, por sus siglas en inglés de cada etapa *Attack*, *Decay*, *Sustain* y *Release* y ofrece la posibilidad de crear un rango amplio de sonidos, desde sonidos ambientales como *pads* cuando aumentamos la fase de ataque hasta *plucks* con un sonido más corto reduciendo la fase de decaimiento y sin sostenimiento [5].

La Figura 26 muestra una ilustración de las etapas que se han descrito anteriormente. En nuestro generador las variaciones siguen un comportamiento lineal, aunque las líneas podrían ser unas curvas que tengan un comportamiento exponencial, con formas tanto cóncavas como convexas. Sin embargo este enfoque más avanzado requiere un diseño más complicado en el que hemos de llevar una cuenta de la cantidad de ciclos para realizar la variación, además de tener en cuenta en qué instante nos encontramos para saber si los ciclos de espera se incrementan de manera más rápida o lenta.

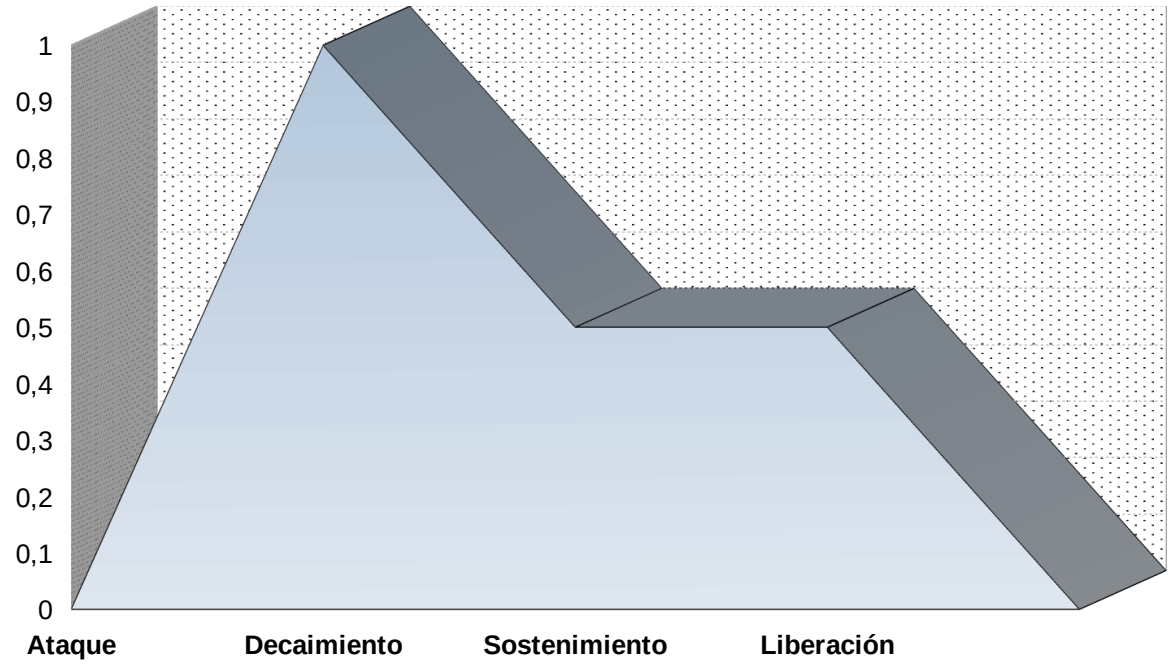

Figura 26: Ilustración del generador de envolventes ADSR

Para desarrollar esta función hemos creado el componente generador de envolventes que recibe el nombre adsr_env, contiene una máquina de estados exactamente igual que las que incluyen todos los componentes generadores de onda. Esta máquina de estados es necesaria porque debemos calcular el valor de amplitud de forma independiente para cada onda, pero además el generador se encarga de detectar una nueva pulsación del teclado y tiene que estar sincronizado con el componente selector de ondas que también contiene la misma máquina de estados.

Contiene otro proceso que se encarga de realizar los cálculos de cada etapa:

- *Attack*, en este estado si la nota no está pulsada cambiamos a la fase *release*. Sabemos que la nota está pulsada debido a que la señal note_play, recibida desde el componente Parameters, está a nivel alto cuando pulsamos una tecla y a nivel bajo cuando se suelta. Si está pulsada y el dato de amplitud es cero, entonces significa que la nota actual se acaba de pulsar y se dirige a un puerto de salida porque el parámetro portamento así lo requiere, para inicializar el periodo de la frecuencia de la nota previa y los ciclos de espera. Mientras está pulsada incrementamos el valor de amplitud de esta nota hasta llegar al valor máximo. El parámetro *attack* de la interfaz nos indica la cantidad de ciclos de espera para realizar el incremento. Por este motivo guardamos la cuenta en un vector para cada onda.
- *Decay*, donde cambiamos a la fase *release* si la nota no está pulsada y en caso contrario se reduce el valor de amplitud, teniendo en cuenta los ciclos de espera, hasta que llega al valor de sostenimiento.
- *Sustain*, en esta etapa no se hace nada hasta que detectamos que se ha dejado de pulsar la nota y cambiamos al siguiente estado.
- *Release*, donde cambiamos a la fase attack cuando se detecta que una nota está pulsada, en caso contrario se va reduciendo el valor de amplitud hasta que el dato es cero y cambia al estado *attack*, además de cambiar el valor de la señal que indica que una nota está sonando. Esta señal se encamina a un puerto de salida puesto que es necesaria en varios componentes, por ejemplo en el selector de ondas para realizar los cálculos del efecto *portamento,* solo si la nota está activa, o en Synthesizer para llevar la cuenta de voces activas y realizar la operación de multiplicación para aplicar el volumen del oscilador.

En la interfaz gráfica, además de los parámetros ADSR tenemos disponibles otro *slider* para variar el tiempo de las fases de ataque, decaimiento y liberación. Este parámetro actúa como si fuera un multiplicador de forma que obtenemos un rango mayor para seleccionar un valor. En Parameters se aplica la operación correspondiente cuando se actualiza el valor, tanto del *slider* del generador como del multiplicador, pero estás señales no salen del componente y solo se dirigen a puertos de salida los valores del generador.

4.3.2 Osciladores de baja frecuencia

Los osciladores de baja frecuencia también aportan dinámica en el sonido y de manera análoga a un generador de envolventes también puede estar asociado a distintos destinos. En este proyecto utilizamos una modulación de frecuencia o *pitch,* de forma independiente en cada oscilador, como destino para aplicar las modulaciones con LFOs.

Pero a diferencia de los generadores de envolventes, donde los parámetros ADSR son los que nos ayudan a controlar la modulación, en el caso de las LFOs son los valores de cada muestra los que la controlan y nosotros podemos controlar esa modulación variando la frecuencia del oscilador de baja frecuencia y su amplitud. Este es el motivo de que reciban el nombre de “baja frecuencia”, primero porque no son audibles como los otros osciladores, ya que se encuentran fuera del rango que un humano puede normalmente escuchar y segundo porque sus frecuencias son mucho más bajas respecto a los osciladores normales del sintetizador, estando por debajo de los 20 Hz.

Por tanto nuestro sintetizador consta en principio de una LFO para cada oscilador que modula el *pitch* y en la sección 4.3.3 incluimos otra LFO que modula la frecuencia y resonancia de los filtros. El tipo de onda de la LFO se puede seleccionar como en los otros osciladores y contiene también los mismos tipos. Se ha diseñado un componente para cada tipo de onda y de igual forma que en los osciladores básicos, cada tipo contiene la misma máquina de estados para calcular cada onda en ciclos distintos, con la única diferencia de que cambia al estado inicial cuando la cuenta de LFO es tres, puesto que el número de LFO en este sintetizador es constante.

Otra diferencia importante es que los valores de cada muestra están en el rango cero y uno, coma fija y 24 bits. El motivo de asignar estos valores se debe a que de esta forma podemos multiplicar el periodo de la onda directamente por el valor de la LFO, para obtener un porcentaje. Los valores para obtener el periodo de cada onda y el incremento de la muestra se encuentran definidos en el paquete synthesizer_package y por tanto almacenados en LUT. Los datos para indexar el valor de la muestra en la onda senoidal también están el LUT, pero los datos de cada muestra siguen estando en Block RAM puesto que la cantidad de datos no cambia, ocupa 3 bloques de memoria frente a 2000 LUT y necesitamos aprovechar este recurso para otros fines. Este cambio es debido a que las frecuencias de cada onda para la LFO son fijas, pueden tener un valor desde la nota cero hasta la 128, no pueden tener *portamento* y por tanto no se requiere otras frecuencias, ciclos del periodo o valores de incremento, más allá de los 128 valores de cada nota. Por otra parte las LFOs no se pueden fijar a un tempo determinado, porque ampliaría mucho el rango de posibles frecuencias, así que 128 es la cantidad máxima en esta implementación.

Sin embargo, sí que podemos cambiar la frecuencia de la nota mientras está activada mediante el parámetro de la interfaz gráfica y por esta razón la onda LFO sierra y triangular debe controlar los posibles desbordamientos para reiniciar su contador. Esto también sucedía para los osciladores básicos cuando incorporamos el parámetro de semitono, puesto que con éste se puede reducir la frecuencia por tanto puede descuadrarse el valor de muestra actual respecto al contador, llegar a desbordar y terminar creando una forma de onda distinta a la esperada. Los otros tipos de onda no presentan este problema y por otra parte la onda cuadrada de la LFO mantiene un ancho de pulso simétrico.

El componente selector de ondas contiene un proceso para activar una señal, cuya función es la de advertir a cada componente LFO que una nota cualquiera de un oscilador básico está sonando, o que la entrada que hemos seleccionado es la del Pmod, de forma que los componentes de LFO solo estarán activos en estos casos. Se ha diseñado de esta forma principalmente para conseguir una sincronización en las dos LFOs de los osciladores, pero también para que nuevas pulsaciones de una tecla tengan siempre el mismo efecto. En un futuro se podría incluir un botón en la interfaz para que active o desactive esta opción, donde simplemente deberíamos realizar el proceso de activación si el botón de la interfaz está activado.

Puesto que cada LFO afectará a la frecuencia de una nota, es decir a los ciclos para completar el periodo, debemos extender el diseño del proceso en el componente selector en el que se calculan los ciclos cuando se aplicaba el efecto *portamento*. Primero se comprueba si se aplica la LFO, por tanto no es posible aplicar *portamento* junto LFOs simultáneamente. Para cada voz, se multiplica el valor de su periodo por el dato de la muestra de la LFO y este valor por su valor de amplitud.

Existen cuatro modos de aplicación en una LFO, donde el periodo puede variar desde frecuencias más bajas o desde frecuencias más altas. La Tabla 5 muestra una descripción de cada modo y la fórmula que se aplica en cada caso. LFO es el valor tras las operaciones descritas anteriormente.

Tabla 5: Descripción de los modos de una LFO

Modo	Fórmula	Descripción
0	Periodo - LFO	Los ciclos varían desde frecuencias más bajas
1	Periodo - Periodo - LFO	Igual que el anterior pero con una escala más alta
2	Periodo + LFO	Los ciclos varían desde frecuencias más altas
3	Periodo + Periodo + LFO	Igual que el anterior pero con una escala más baja

Cabe destacar que es importante ajustar la cantidad de bits de las variables en las que guardamos la multiplicación del valor de la muestra de la LFO y su amplitud, para que Vivado entienda mejor cómo se debe realizar el diseño y la implementación en la utilización de recursos DSP. Por otra parte también es importante comprobar que el valor del periodo no va a exceder de la cantidad de ciclos para la frecuencia más baja o más alta, puesto que con estas operaciones puede suceder y por tanto se debe asignar la variación solo si se encuentra en el rango.

Los valores de incremento de muestras para una LFO se han calculado con un *script* en C++ que se encuentra en el Anexo 9.7. Los valores para indexar el dato que obtiene la muestra de la onda senoidal son los mismos que para la onda básica, que hemos calculado en la sección 4.1.2.

4.3.3 Filtros de estado variable

Entrando en la síntesis sustractiva, incluimos los filtros cuya función es reducir las frecuencias no deseadas. El filtrado se refiere al tratamiento de la señal que afecta al espectro de frecuencias, amplificando o atenuando unas frecuencias determinadas.

Un filtro digital no es más que una función matemática con una salida en función de las entradas. En este proyecto hemos estudiado los filtros de estado variable de Chamberlin [25], implementado las ecuaciones de su diseño donde se puede obtener simultáneamente las salidas de cuatro tipos de filtros, paso bajo, paso alto, de banda y de rechazo de banda.

La estructura de los filtros de Chamberlin está basada en filtros IIR (Respuesta de Impulso Infinita) que presentan una latencia más baja que los filtros FIR (Respuesta de Impulso Finita), tienden a tener una mejor respuesta de magnitud y también requieren menos coeficientes y variables [26].

Además de ser más estable, la estructura de los filtros de Chamberlin también permite controlar la frecuencia de corte, es decir el límite o corte en la respuesta frecuencial y el factor de calidad Q o resonancia, que sirve para enfatizar o suprimir las frecuencias que están justo encima o debajo de la frecuencia de corte.

No obstante estos filtros también presentan limitaciones, la frecuencia de corte máxima está en función de la frecuencia de muestreo y en nuestra implementación no puede ser superior a 8 kHz. Esta afirmación se demuestra de forma matemática en [26], que trata los límites de estabilidad en los filtros de Chamberlin.

La Figura 27 muestra un ejemplo de aplicación de un filtro de paso bajo en una onda senoidal de 441 Hz con una frecuencia de corte de 800 Hz, en la que incrementamos la resonancia desde uno hasta tres. En la ilustración podemos observar varias cosas, primero el desfase en el tiempo que es debido a la realimentación necesaria en los valores de las ecuaciones donde no obtenemos el primer valor de la salida hasta el ciclo siguiente y segundo se observa, como para valores de Q altos, determinados valores de la muestra pueden exceder de la cantidad de bits establecida, por tanto debemos limitar estos valores para no incurrir en un desbordamiento.

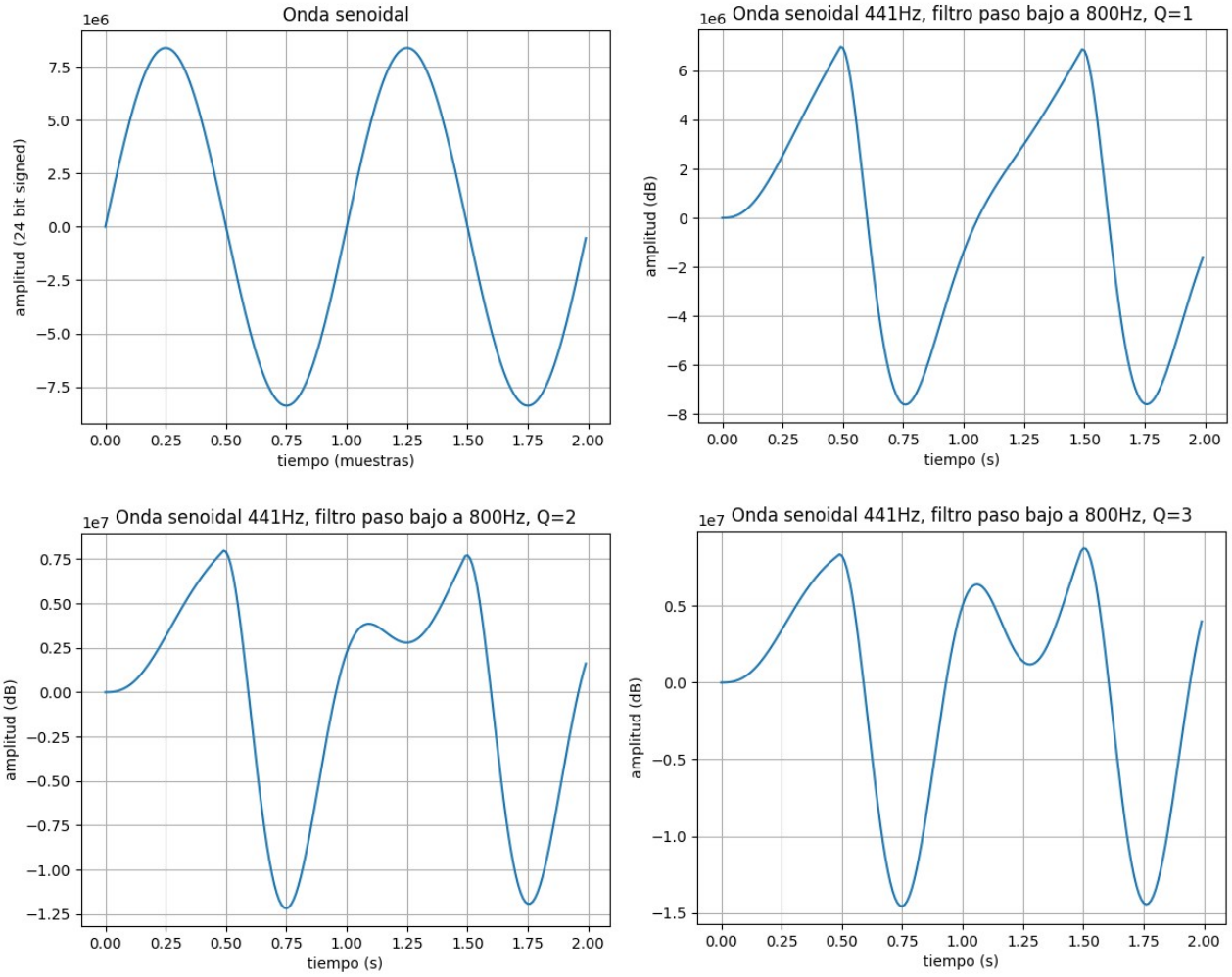

Figura 27: Aplicación de un filtro de paso bajo a 800 Hz con distinta resonancia

Para la implementación de los filtros hemos creado el componente Filters, que obtiene los datos de entrada desde el componente generador de ruido y su salida es dirigida al DAC.

La implementación está basada en las ecuaciones:

- (4), se obtiene el valor del filtro de paso bajo, donde D1 es el retardo asociado con la salida del filtro paso banda y D2 es el retardo asociado con la salida del filtro paso bajo.
- (5), obtenemos el valor del filtro de paso alto, esta ecuación tiene como entrada el valor de los generadores de onda, que pueden haber sido procesados por el generador de ruido. Q1 es el valor que calculamos en (9).
- (6), donde se obtiene el valor del filtro de banda. F1 es el valor que calculamos en (8).
- (7), que obtiene el valor del filtro de rechazo de banda, es la suma de cálculos anteriores.
- (8), donde se calculan unos valores que son constantes, para una determinada frecuencia de corte y en función de la frecuencia de muestreo.
- (9), que calcula la inversa del valor del parámetro de la resonancia, valores constantes.

$$Filtro_{paso\,bajo} = D2 + F1 * D1 \tag{4}$$

$$Filtro_{paso\,alto} = Entrada - Filtro_{paso\,bajo} - Q1 * D1 \tag{5}$$

$$Filtro_{paso\,banda} = F1 * Filtro_{paso\,alto} + D1 \tag{6}$$

$$Filtro_{rechazo\,banda} = Filtro_{paso\,alto} + Filtro_{paso\,bajo} \tag{7}$$

$$F1 = 2 * \sin\left(\frac{\pi * Frecuencia_{corte}}{Frecuencia_{muestreo}}\right) \tag{8}$$

$$Q1 = \left(\frac{1}{resonancia}\right) \tag{9}$$

Puesto que para calcular los valores de F1 se utiliza el cálculo del seno, guardamos los valores en memoria para frecuencias de corte desde cero hasta 8 kHz, que hemos calculado previamente en un *script* de Python en el Anexo 9.8, con incrementos de 1.0 Hz. Por otra parte se han calculado los valores de Q1, que se encuentran en el rango [0.8,17.09] y que se incrementan en 0.1 hasta la cantidad de 10 y en uno a partir de este valor. Los valores menores a 0.8 no producen la salida esperada en la implementación aplicando estas fórmulas en VHDL. Sin embargo con los *scripts* de Python que hemos desarrollado para obtener las gráficas el comportamiento sí es correcto, pero desconocemos cuál es la causa de esta discrepancia y es el motivo de que la resonancia obtenga este valor mínimo. Por otra parte aunque puede soportar valores más grandes para Q1, 17.09 es el valor máximo porque a partir de este valor se empiezan a obtener varias muestras con valores superiores a la cantidad de bits máxima, que tenemos que limitar y por tanto se pierde el efecto.

El componente Filters contiene como puertos de salida los datos filtrados en 24 bits con signo y una señal para indicar que el procesamiento de los filtros ha finalizado. Como puertos de entrada contiene la señal *start*, que indica cuando debe empezar el procesamiento de los filtros, una señal que indica si debe realizarse el procesamiento, es decir la entrada seleccionada es la del Pmod o es la entrada del sintetizador pero al menos se ha generado una onda, los valores de frecuencia, resonancia y el tipo de filtro de la interfaz gráfica más el modo de la LFO y el valor de la muestra. Por otra parte, contiene un proceso donde según el estado en que nos encontramos realiza la operación necesaria.

Los estados son los siguientes:

- LOAD_DATA_INI, donde indicamos en la señal de finalización que los cálculos actuales no han terminado y actualizamos la dirección, con la frecuencia de corte, para obtener el dato de F1 desde memoria.
- WAIT_DATA_INI, estado en el que esperamos un ciclo para tener el dato de memoria.
- IDLE, es el estado inicial en el que esperamos a que la señal señal *start* esté activa. En ese caso se comprueba que hay seleccionado algún tipo de filtro y que se debe realizar el procesamiento, en caso contrario reiniciamos todos los valores de cada filtro e indicamos que el filtro ha terminado. Si hay que procesar, comprobaremos que la LFO del filtro está activada, si no lo está pasaremos al cálculo del filtro paso bajo. Si lo está comprobamos cual es el destino de la modulación, que puede ser la resonancia o la frecuencia, para actualizar los operandos necesarios para las operaciones de los siguientes estados.
- LFO_MULT0, es donde obtenemos el dato de frecuencia o resonancia tras la modulación de la LFO, es decir tras la multiplicación, que nos servirá como índice para las tablas.
- LOAD_DATA_LFO, actualizamos la dirección de memoria con el dato anterior.
- WAIT_DATA_LFO, insertamos el ciclo de espera para que la frecuencia de corte esté disponible en el siguiente estado.
- LFO_MULT1, finalmente actualizamos los valores de F1 y Q1.
- LOWPASS, realizamos la operación del filtro de paso bajo.
- HIGHPASS, realizamos la operación del filtro de paso alto.
- BANDPASS, realizamos la operación del filtro de paso banda.
- NOTCH, realizamos la operación del filtro de rechazo de banda e indicamos en la señal de terminación que el procesamiento ha finalizado, para volver al estado LOAD_DATA_INI.

El dato de salida de audio está multiplexado y se selecciona en función del tipo de filtro y de si ha habido un desbordamiento para 24 bits, porque los valores de cada tipo de filtro requieren cuatro veces este tamaño por las operaciones que realizan, en cuyo caso lo dejamos en el límite máximo o mínimo de 24 bits con signo. La señal que indica la terminación se necesita para el componente Synthesizer, para calcular los datos del siguiente canal, se explica en detalle en la sección 4.3.4.

Por otra parte a diferencia de las LFOs que se aplican solo al *pitch* de cada oscilador, para la LFO de los filtros se puede modular tanto la frecuencia como la resonancia pero solo tiene dos modos. Un modo modula desde el valor del parámetro mínimo hasta el valor seleccionado actualmente en la interfaz y el otro modo modula desde el valor máximo hasta el actual.

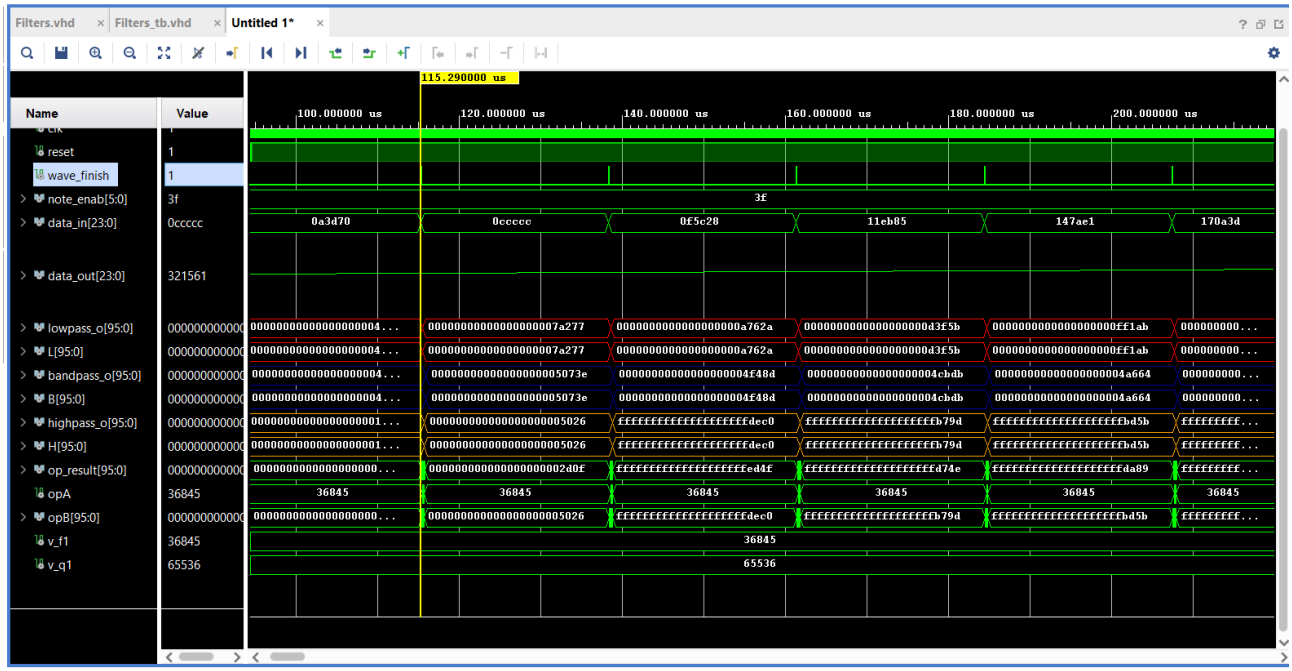

Figura 28: Simulación de los filtros de estado variable en Vivado

Cabe destacar que la implementación se podría optimizar para minimizar la utilización de recursos DSP si incorporamos un nuevo estado en el que se realicen las operaciones de multiplicación que se realizan en los estados de cada filtro y vamos actualizando el valor de los operandos, siguiendo la misma estructura que hemos utilizado para el cálculo de la LFO.

Sin embargo, mientras que la simulación muestra que los valores de salida son correctos, cuando se transfiere este diseño a la placa no se comporta de la manera esperada, obteniendo valores de salida incorrectos.

No se ha llegado a detectar cuál es la causa de este error, ya que como muestra la Figura 28 los valores sí que son correctos si comparamos los datos en cada filtro con y sin optimización, por otra parte no parece tener un error de temporización ya que el reloj de este componente es el reloj maestro pero los datos no se actualizan hasta que llega el flanco de subida de la selección de palabra. La Figura 29 muestra otra simulación de los filtros pero con la aplicación de una LFO, que modula la frecuencia de corte por tanto los valores de F1 y con una resonancia fija de uno, donde los valores de salida también son los esperados.

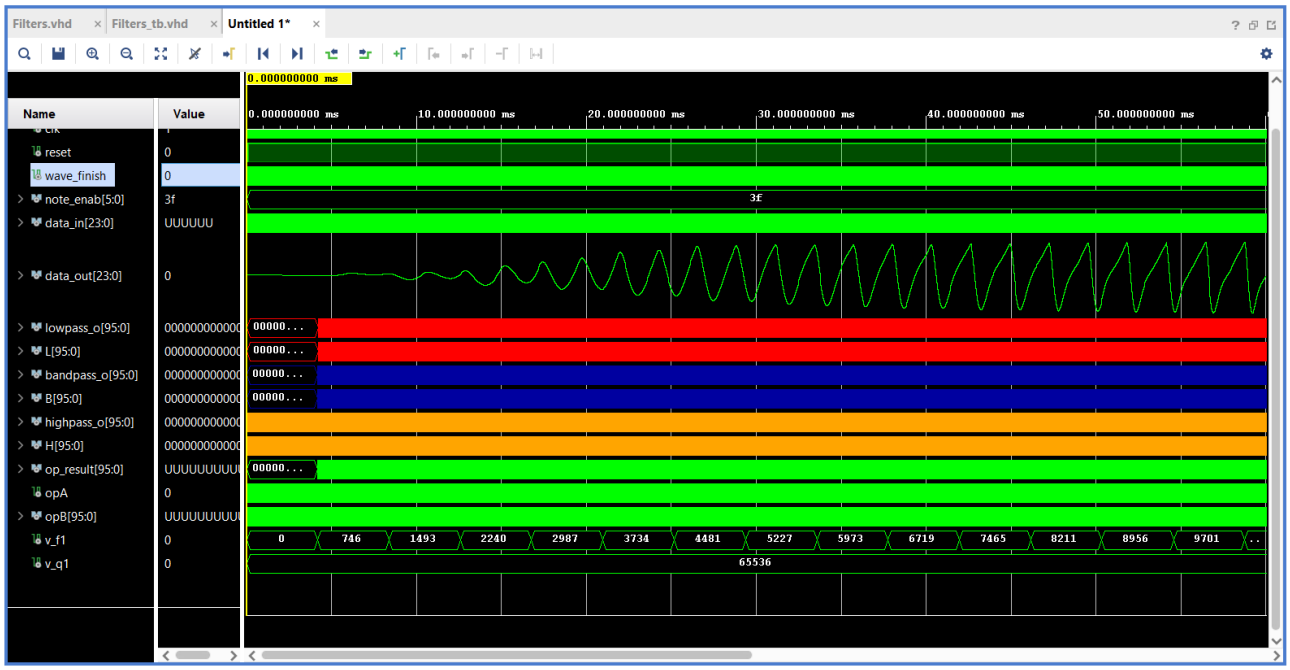

Figura 29: Simulación de los filtros con LFO en Vivado

4.3.4 Generador de ruido

Los generadores de ruido se utilizan para diversos tipos de aplicaciones, entre los que se están los sintetizadores de sonidos y se utilizan sumados a las frecuencias de una onda, que es habitual que se apliquen para dar color al sonido.

Existen diversos tipos de ruido pero habitualmente los sintetizadores incluyen ruido blanco, que se caracteriza por tener una distribución uniforme en el espectro y una respuesta plana, es decir, una intensidad constante para todas las frecuencias y el ruido rosa que no tiene una respuesta plana.

No obstante, la implementación de estos tipos requiere la generación de valores aleatorios que no es sencilla en un diseño en VHDL y cuyo desarrollo supone una dificultad elevada, para el tiempo que se dispone para la generación de este componente. Por este motivo se ha decidido diseñar el componente que recibe el nombre NoiseGen, cuya implementación es más sencilla pero genera un tipo de ruido distinto a los típicamente conocidos. La interfaz gráfica dispone de un parámetro en el rango de cero a 100, de forma que en el valor mínimo no se aplica ruido, desde uno hasta 99 se genera un ruido que simplemente agrega armónicos a la señal y con el máximo valor se genera un ruido cuyos valores son más aleatorios, pero que están en función del dato de entrada.

Igual que con el componente de los filtros, tiene como puertos de entrada una señal que indica cuándo debe empezar a procesar los datos y otra señal que indica si debe procesarlos, además del valor del parámetro de la interfaz y los datos de audio. Su salida es una señal de 24 bits con signo con los datos de audio después de su tratamiento. Contiene una tabla de 100 elementos en la que hemos guardado los valores para incrementar o reducir cada muestra de la señal de audio, en función del valor del parámetro de la interfaz. Además contiene un proceso donde realizamos los cálculos, en el que si el valor del parámetro es el máximo aplicamos la función lógica OR exclusiva de los 20 bits de menor peso de la señal de audio y el valor de incremento obtenido de la tabla. Si el parámetro no es el máximo pero es mayor a cero, obtenemos directamente el valor de incremento de la tabla.

Después tenemos otra señal para sumar o restar el incremento, que vamos variando a cada ciclo y una variable temporal para realizar el cálculo previamente a la asignación en el dato de salida, puesto que la operación puede desbordar. Realizamos la comprobación de desbordamiento donde solo podemos sumar si el bit de signo en la señal de audio es negativo o es positivo pero el de la suma también y solo podemos restar, si el bit de signo es positivo o es negativo pero el de la resta también.

Recordamos que la señal de entrada de este componente se puede conmutar desde la interfaz y puede ser la entrada del Pmod o las ondas generadas en el sintetizador. Esta función se gestiona desde el componente Synthesizer, contiene una máquina de estados cuyo propósito es indicar cuando deben empezar a trabajar los componentes del filtro y del generador de ruido. Contiene cinco estados con la siguiente funcionalidad:

- NOISE_L, indica que debe a empezar a trabajar el generador de ruido y se asigna al dato de su señal de entrada directamente el canal izquierdo del dato recibido desde el Pmod.
- FILTER_L, desactiva la señal para que trabaje el generador de ruido y activa la del filtro. Se mantiene en este estado hasta que el filtro indique que ha terminado, ya que el filtro requiere varios ciclos para el procesamiento y cuando termina se desactiva la señal para indicar que el filtro debe trabajar y se asigna al canal izquierdo de audio el dato del filtro.
- NOISE_R, activa la señal *start* para el generador de ruido y prepara su dato de entrada, en función de señal que indica si la entrada es del Pmod o del sintetizador.
- FILTER_R, detiene la activación de *start* para el generador de ruido y activa la del filtro. Se queda a la espera, para que cuando termine se asigne la salida del filtro al canal derecho. Pero ahora si la entrada es del sintetizador, se asigna la salida del filtro al canal izquierdo también porque la configuración del sintetizador es estéreo, aunque en este sintetizador se duplica el valor de la señal. En cualquier caso, cambiamos al estado inicial.
- IDLE, es el estado inicial en el que esperamos a que los generadores de onda terminen. Si la entrada de audio es la del Pmod, cambiamos al estado NOISE_L y en caso contrario al estado NOISE_R.

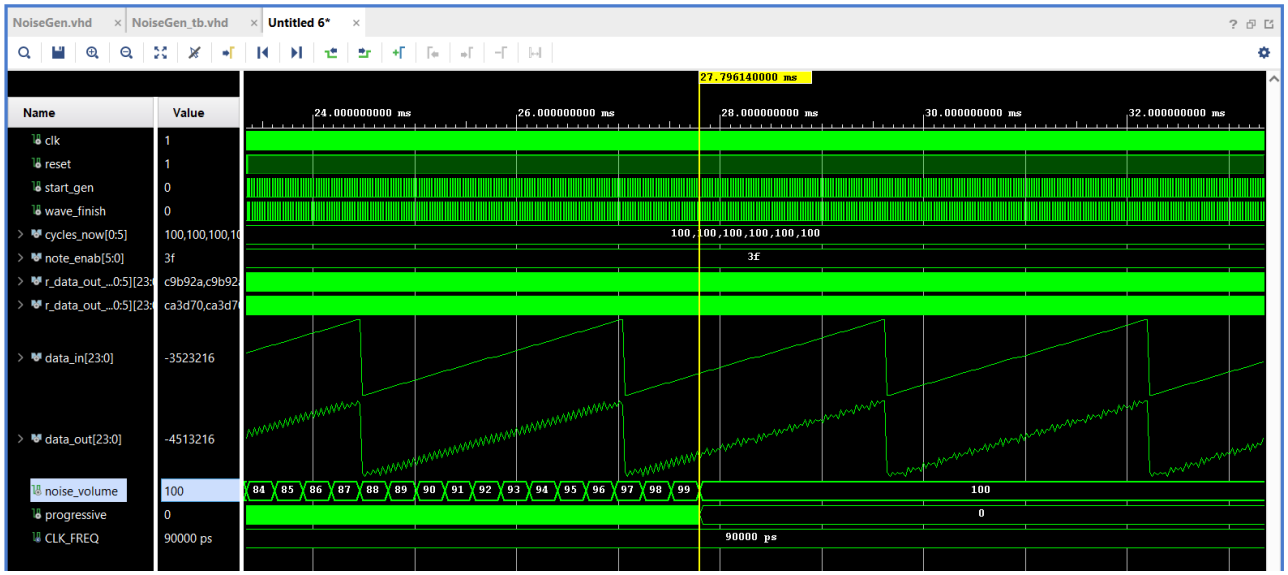

Figura 30: Simulación del generador de ruido en una onda sierra

La Figura 30 muestra una simulación en la que se observa de manera visual el comportamiento de la mezcla del ruido en una onda sierra, verificando que cuando el parámetro de ruido se encuentra en su valor máximo la forma de onda es más aleatoria y similar a una mezcla con ruido blanco.

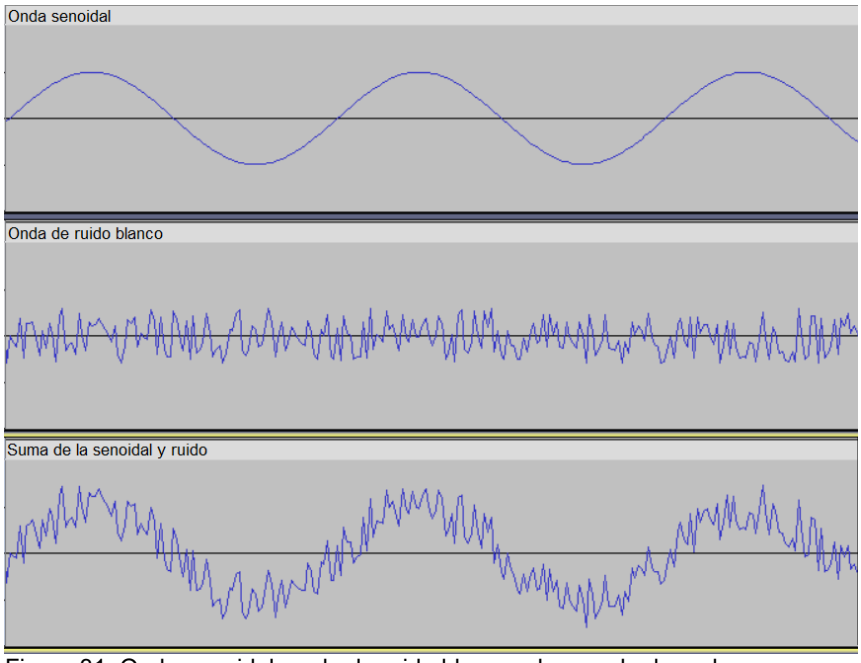

Figura 31: Onda senoidal, onda de ruido blanco y la mezcla de ambas

En la Figura 31 se puede observar como sería una implementación del generador de ruido blanco en una onda senoidal ampliada, donde se observa con más claridad como los valores de ruido son totalmente aleatorios y por tanto la suma de ambas ondas.

4.4 Fase 4: Interfaz gráfica, aplicación *Standalone* y *plug-in* VST3

En la última fase de implementación, hemos trabajado en la programación de una aplicación en el lenguaje C++ junto con el *framework* JUCE, ofreciendo una interfaz gráfica que gestiona todos los parámetros del sintetizador. Se ha compilado como *plug-in* VST3 y se ha verificado que su ejecución es la esperada en un DAW. También se ha incorporado un menú para gestionar las configuraciones que podemos almacenar, además hemos establecido un mecanismo para sincronizar el estado de los parámetros de la aplicación y la placa cuando se ésta se lanza a ejecución. Por otra parte se ha añadido la programación necesaria para que en la aplicación *Standalone* se puedan controlar los distintos parámetros con un controlador *hardware* MIDI externo conectado al computador.

4.4.1 La aplicación de escritorio

Para agilizar el proceso de desarrollo de la aplicación hemos trabajado con el framework JUCE. Se trata de un framework de código abierto basado en C++, utilizado ampliamente en aplicaciones de audio y desarrollo de *plug-ins*. Es multiplataforma y permite crear aplicaciones de escritorio, aplicaciones para móviles y plug-ins en todos los formatos que utilizan los DAW, incluyendo VST y VST3 (para Windows), AU y AUv3 (para MacOs/iOs), AAX (para Pro Tools) y LV2 (para Linux).

Su instalación es sencilla, basta con descomprimir un archivo en disco e incluye una aplicación, de nombre Projucer, con la que podemos generar y configurar proyectos para Xcode, Visual Studio, Android Studio, Code::Blocks y archivos make para Linux. Como hemos trabajado en un entorno con Windows, hemos generado una solución para Visual Studio. Debido a su portabilidad, permite que nos enfoquemos principalmente en el desarrollo de la aplicación sin tener que preocuparnos por las diferencias entre sistemas operativos o formatos de los plug-ins.

Además existe mucha documentación, tutoriales, ejemplos que nos ayudan en el desarrollo de la aplicación y una amplia comunidad que responde a dudas y preguntas en su foro oficial.

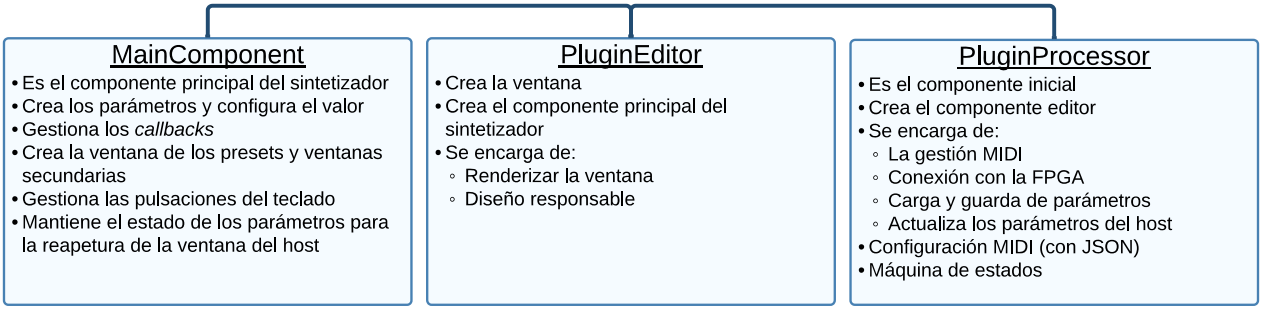

Figura 32: Diagrama de clases de la aplicación

La Figura 32 muestra un esquema que resume cuál es la estructura en la solución para Visual Studio. La clase PluginProcessor es el componente de entrada cuando se ejecuta la aplicación, crea PluginEditor y éste se encarga de crear MainComponent, el componente donde se gestionan los parámetros de la aplicación. Esta es la estructura habitual que sigue una aplicación en JUCE que comparte el mismo código para una aplicación *standalone* y *plug-in* VST3. El resultado final de la interfaz gráfica se muestra en la Figura 33.

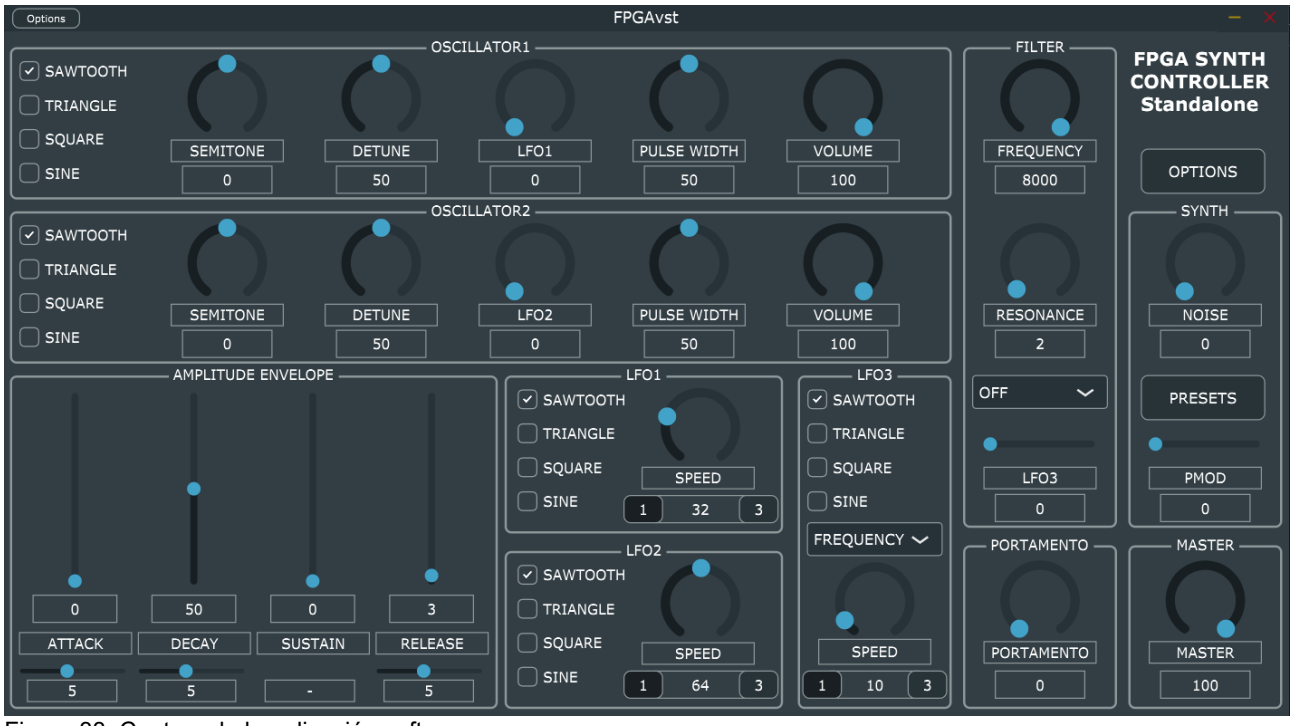

Figura 33: Captura de la aplicación software

4.4.1.1 PluginProcessor

Como PluginProcessor es el primer componente, se encarga de mantener una comunicación con el *host* cuando funciona como un *plug-in*, independientemente del formato. En primer lugar debe transmitir si esta aplicación se trata un:

- Instrumento virtual, son los que generan sonido y solo puede haber uno por pista.
- Efecto de sonido, son los que procesan el sonido, pueden haber varios por pista.
- Efecto MIDI, alteran los datos MIDI que le entran a un instrumento, pueden haber varios.

En nuestro caso consiste en un instrumento virtual aunque no produzca sonido, porque si en una futura ampliación queremos recibir los datos de audio directamente, bien sea por cable o de forma inalámbrica, podemos transmitirlos al DAW en la misma pista que hemos insertado el instrumento, evitándonos el encaminamiento para recibir en el DAW la salida del DAC en una pista, además del posible ruido adicional que el convertidor puede generar. Por otra parte también comunica al host otros datos relevantes, como si tiene alguna entrada de audio, si es estéreo, si acepta datos MIDI, el número de parámetros, si produce mensajes MIDI o el propio nombre del *plug-in*, entre otros.

Esta clase contiene la función hasEditor, que indica si la aplicación contiene una ventana, porque puede no incluir una interfaz y el propio DAW crea un espacio donde incluye todos los parámetros. Como hemos creado también una aplicación *standalone*, debemos obligatoriamente crear el editor que gestionará la ventana principal. Cuando el editor cree el componente principal del sintetizador y éste todos los parámetros en la interfaz, realizará una llamada a una función de esta clase para conectar con la FPGA y realizar la sincronización, que veremos en detalle en la sección 4.4.4, a no ser que funcione en modo *plug-in* donde la conexión se realiza en el constructor de esta clase.

Pero antes de conectar busca si en la ruta del ejecutable se encuentra un fichero de configuración que guarda el número de puerto serie al que conectar, en caso contrario se utiliza uno por defecto. Por otra parte, también en el constructor, se define la función parameterChanged como *callback* para una lista de parámetros con un identificador concreto, se declara el estado inicial para la máquina de estados que actualiza los valores de los controles y se lee el fichero de configuración que almacena los canales y controles que han sido mapeados a un controlador MIDI, si la aplicación funciona en modo *standalone*, todo esto se detalla en la sección 4.4.4.

El Anexo 9.9 contiene los datos de los identificadores en valor hexadecimal, que asignamos como *string* al crear los controles. Los valores son los mismos para un control, tanto en los componentes Parameters y Sinchronize de VHDL como en los controles en la aplicación. También almacenamos un HashMap, una clase de JUCE que se trata del típico diccionario con el par clave-valor pero con unas funciones propias para obtener un valor por clave, crear o actualizar el valor de una clave, entre otras, donde la clave es también este identificador y que utilizamos para mantener los datos de cada parámetro actualizados. Esta estructura de datos es necesaria porque en versión *plug-in*, MainComponent se destruye cuando la ventana se cierra y se vuelve a crear al abrirla y debemos inicializar los controles con los valores previos, por eso se guardan en PluginProcessor.

La función processBlock es la encargada de la gestión del audio, pero también de la recepción de datos MIDI, para cada mensaje filtramos su contenido y si se trata de una pulsación del teclado lo comunicamos a la FPGA en las dos versiones, pero si se trata de un control de un parámetro se transmitirá solo si la aplicación es *standalone*, puesto que el *host* ya se encarga del gestionar el mapeado de los controles MIDI a los parámetros.

Además esta clase también se encarga de comunicar al DAW el valor de cada parámetro cuando se guarda un proyecto, para recibir el estado en la carga del proyecto, que vemos con más detalle en la sección 4.4.2 puesto que existen algunas diferencias entre las dos versiones. También se encarga de comunicar al *host* la disposición de los parámetros, que son enlazados en la creación de cada componente en MainComponent, como veremos en la sección 4.4.1.3, porque todos los parámetros no estarán disponibles en el *host* y comunicaremos solo los parámetros que queramos que sean automatizables desde el DAW.

Por otro lado, contiene la función setParamValue que es donde se ha desarrollado la máquina de estados cuyo propósito es actualizar los valores de los controles de la interfaz en la sincronización y en la carga de *presets*, donde también realiza el envío de datos al puerto serie. Esto se verá con más detalle en la sección 4.4.4.

Contiene también otras funciones auxiliares, para obtener la ruta del ejecutable donde deberemos guardar el fichero de *presets* y en la que se creará el fichero de configuración de la aplicación, o la función para leer este fichero de configuración y actualizar el puerto serie.

En el destructor de esta clase cerraremos la conexión con el puerto serie y por último, si se ha mapeado algún parámetro a un controlador MIDI lo almacenaremos en el fichero de configuración.

4.4.1.2 PluginEditor

La clase PluginEditor se crea una vez PluginProcessor ha comprobado que es una aplicación con interfaz gráfica, como es nuestro caso. En el constructor se definirá el tamaño de la ventana y se crea el componente principal, que contiene cada uno de los controles y sus callbacks.

Esta componente simplemente contiene las funciones para realizar el renderizado de la aplicación, así como gestionar un diseño responsable mediante la función resized, en la que podemos volver a definir el tamaño de cada control cuando la ventana principal cambia de tamaño. En este caso, como todo el diseño está gestionado desde MainComponent, esta función simplemente redefine el tamaño de la ventana principal que no puede ser modificado, independientemente de la resolución de pantalla.

4.4.1.3 MainComponent

Esta clase es la que contiene todos los elementos de nuestra interfaz gráfica, es decir, los *sliders*, los botones, los menús y las selecciones de *radio buttons,* así como las subclases que crean la ventana para seleccionar los *presets*. Además contiene una referencia a PluginProcessor, para poder realizar las llamadas a las funciones necesarias.

En el constructor definimos todos los elementos y sus etiquetas en la interfaz, con su identificador correspondiente y les damos el valor que hemos obtenido en la inicialización de PluginProcessor. Si se trata de un *plug-in*, el *callback* en PluginProcessor para la actualización de los parámetros modifica estos valores, por tanto al volver a abrir la ventana los valores estarán actualizados. Por otra parte también se define el estilo de cada componente, se enlazan con los parámetros del *host* para que éste conozca a qué control le corresponde en la interfaz y se almacenan en estructuras de datos referencias a éstos, necesarias por ejemplo para establecer la asignación de controles MIDI o en la actualización de los controles en la máquina de estados. Cuando se termina de crear los controles, se realiza la conexión con la FPGA si se trata de la versión *standalone*.

Cabe destacar que hemos creado una clase para nuestros *sliders*, que heredan de la clase *Slider*, puesto que hemos añadido funcionalidad extra que no ofrece el control básico de JUCE. Cuando pulsamos el botón derecho sobre un *slider* rotatorio, se abre una pequeña ventana para indicar si queremos quedar a la escucha de aprendizaje de este parámetro, se explica en la sección 4.4.4.

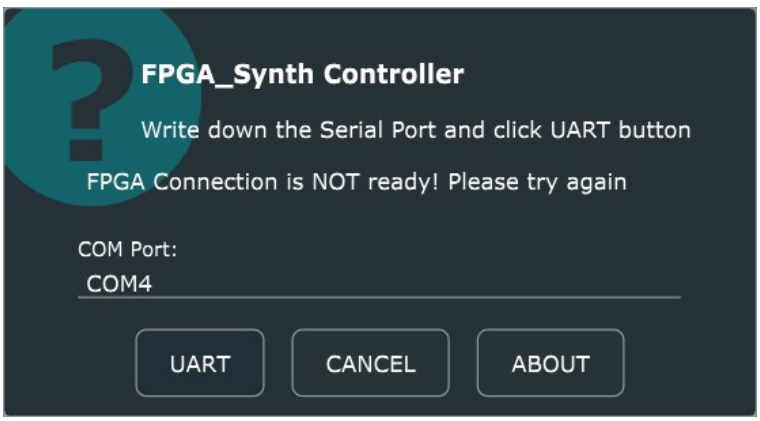

Figura 34: Ventana de opciones de la aplicación software

También tiene el componente que crea una ventana flotante al pulsar sobre el botón de opciones. Esta ventana nos indica si la conexión al puerto serie se ha realizado con éxito y permite realizar una nueva conexión indicando el número de puerto. Si la conexión es satisfactoria, el número de puerto se guarda en el fichero de configuración de la aplicación, que se lee cada vez que se lanza a ejecución para que la sincronización se pueda realizar correctamente en el puerto adecuado. La Figura 34 muestra una captura de la ventana de opciones.

Por otra parte esta clase también hereda de KeyListener y de Timer, el primero para sobrescribir una función en nuestra clase y detectar las pulsaciones de teclas cuando la ventana mantiene el foco y el segundo, para tener un *callback* temporizado que utilizamos también para detectar las pulsaciones del teclado, aunque la aplicación no tenga el foco. De esta forma podemos utilizar el piano simulado aunque estemos navegando por la ventana de los *presets*. La función se encarga de comprobar por cada tecla relevante si está pulsada para insertarla en un vector y enviar el dato a la FPGA. Finalmente comprobaremos el vector de teclas pulsadas y si no lo está, indicaremos la liberación a la placa. Veamos el Algoritmo 5 para entender mejor el funcionamiento.

Algoritmo 5: Detección del teclado en la función temporizada

```
teclas_piano[15] = { A, W, S, … } // en realidad son los códigos ASCII
bucle de teclas_piano
  si juce::teclapulsada(teclas_piano[iteración]) entonces
    enviar_dato = TRUE
    si teclas_piano[iteración] en vector_pulsadas entonces
      enviar_dato = FALSE
    fin si
    si enviar_dato entonces
      insertar(vector_pulsadas, teclas_piano[iteración])
      enviar_pulsación_a_FPGA() // se envía 1 byte, el identificador de la tecla pulsada
    fin si
  fin si
fin bucle
bucle de vector_pulsadas
  si no juce::teclapulsada(vector_pulsadas[iteración]) entonces
    eliminar(vector_pulsadas, vector_pulsadas[iteración])
    enviar_liberación_a_FPGA() // se envían 2 bytes, 0x04 que indica note off + id de la tecla
  fin si
fin bucle
```

Por otra parte, destacamos que los parámetros que no han sido enlazados con el *host* requieren indicar una función de *callback* en esta clase que se ejecutará cuando su valor es actualizado. Por este motivo los botones del modo de cada LFO y los selectores de onda tienen este *callback,* en el que cuando su valor cambia se notifica a la FPGA.

Por último, es importante para que la aplicación termine de forma ordenada, que en el destructor cerremos la ventana que contiene los *presets* si sigue abierta, porque esta y la ventana principal pueden estar abiertas simultáneamente y es conveniente liberar memoria.

La Figura 35 muestra una ilustración en la que indicamos la correspondencia de las teclas de un computador con las teclas de un piano, que simulamos. Utilizamos 15 teclas, aunque una escala contiene 12, para tener un poco más de flexibilidad ya que el teclado del computador es limitado. Por este motivo, usamos la tecla z para bajar una escala y la tecla x para incrementarla. Como las escalas de los extremos pueden tener teclas que se salgan del rango, la programación *hardware* comprueba si el dato recibido para esta escala es una nota válida.

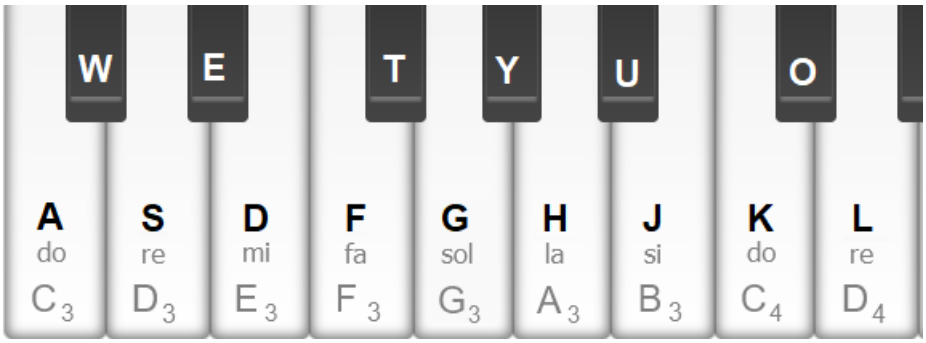

Figura 35: Asignación de las teclas en el piano simulado

4.4.2 El plug-in VST3

Uno de los principales motivos que hemos tenido en cuenta para escoger este framework, además de la portabilidad, es la facilidad que éste ofrece para compilar con un mismo código compartido una aplicación de escritorio al mismo tiempo que un *plug-in* para un DAW. Adicionalmente también nos permite elegir los formatos de *plug-in* para compilar, no obstante hemos trabajo con el formato VST3 y Ableton Live como DAW, realizando las pruebas necesarias en este entorno.

El funcionamiento del *plug-in* debe ser en principio, exactamente igual que cuando ejecutamos la aplicación *standalone*, sin embargo existen algunas pequeñas diferencias que debemos tener en cuenta si nuestro objetivo es que el comportamiento sea el mismo en ambas aplicaciones.

Como hemos visto previamente, PluginProcessor es el primer componente y crea el editor, que a su vez crea MainComponent. Esto es así en ambas versiones, pero una diferencia principal es que en la versión *standalone* esto solo sucede una vez, mientras que en el *plug-in* si cerramos la ventana el editor se destruirá y por tanto también MainComponent. Es muy importante tener este aspecto en cuenta, por ejemplo porque si volvemos a abrir la ventana del *plug-in*, volveremos a crear los controles y debemos establecer su último valor y no el inicial. Además cuando lanzamos un *plug-in* en el DAW, la ventana se crea y se destruye una primera vez sin notificarnos nada. Es el comportamiento normal en el DAW y no podemos modificarlo.

Por otra parte, el *plug-in* no necesita simular el teclado MIDI con el teclado del computador, puesto que la mayoría de los DAWs ya incorporan esta funcionalidad, si no lo hacen siempre encaminan la entrada MIDI para que nuestra aplicación reciba mensajes MIDI, de manera que controlaremos el teclado de esta forma. El DAW ya ofrece muchas opciones para lograr este objetivo, desde los denominados clips MIDI hasta mapeados de *hardware*. Es el motivo de que el *plug-in* no incorpore la detección de pulsaciones en el computador, de forma que a través de una macro que hemos definido indicaremos, en tiempo de compilación, las secciones de código que deben ser incluidas en cada versión.

Otro aspecto importante a tener en cuenta es la conexión con la FPGA. Mientras que en la versión *standalone* necesitamos indicar a la placa que hemos iniciado la aplicación para que nos envíe el estados de los parámetros y configurar los controles de la interfaz, en el *plug-in* esto sucede a la inversa de forma que cuando lanzamos el *plug-in* a una nueva pista del DAW, los controles están en su estado inicial y estos datos se envían a la FPGA.

Es importante seguir este procedimiento, puesto que al cargar un proyecto del DAW que hemos guardado previamente, nuestro *plug-in* comunica al DAW el estado de los parámetros para poder establecerlos de nuevo en la carga. En este momento sincronizamos con la FPGA enviándole el estado de los controles de la interfaz. Veamos en el Algoritmo 6 el desarrollo.

Algoritmo 6: Conexión de la aplicación con la FPGA

MainComponent (constructor)	*PluginProcessor (constructor)*
creamos los controles de la interfaz control.valor(pvalues[control.id]) **#si standalone** PluginProcessor→FPGA_Connection() sincronizamos.. (enviar byte 0x01 + recibir) **#fin si standalone**	Inicializamos el hashmap pvalues leemos el fichero de configuración **si no** juce::existeFichero() **entonces** puerto_com = “COM4” **fin si** **#si no standalone** FPGA_Connection() sincronizamos.. (enviar valores de pvalues) **#fin si standalone**

Cuando guardamos un proyecto, comunicamos al *host* el estado con la función getState y cuando cargamos el proyecto recuperamos el estado con la función setSate, puesto que el *framework* ya se encarga de este funcionamiento. No obstante, nosotros debemos preocuparnos de indicar qué parámetros son los que queremos guardar y recuperar. Cabe destacar que los parámetros que recuperamos están normalizados en el rango [0,1], pero aunque la documentación no menciona nada sobre esto, los foros oficiales siempre son una fuente de ayuda óptima [29].

4.4.3 Almacenamiento de la configuración, los presets

Entendemos por *presets* los elementos que almacenan una configuración, es decir, los valores de los parámetros que hemos almacenado previamente. Se ha desarrollado un sistema para guardar y recuperar configuraciones que pueda ser utilizado en las dos versiones de la aplicación, aunque sabemos que el formato de *plug-in* ofrece posibilidades para almacenar *presets* y el DAW puede gestionarlos, no podríamos llegar a compartirlos sin este desarrollo. Es algo típico que los *plug-ins* comerciales también desarrollen un gestor de *presets* propio en su aplicación.

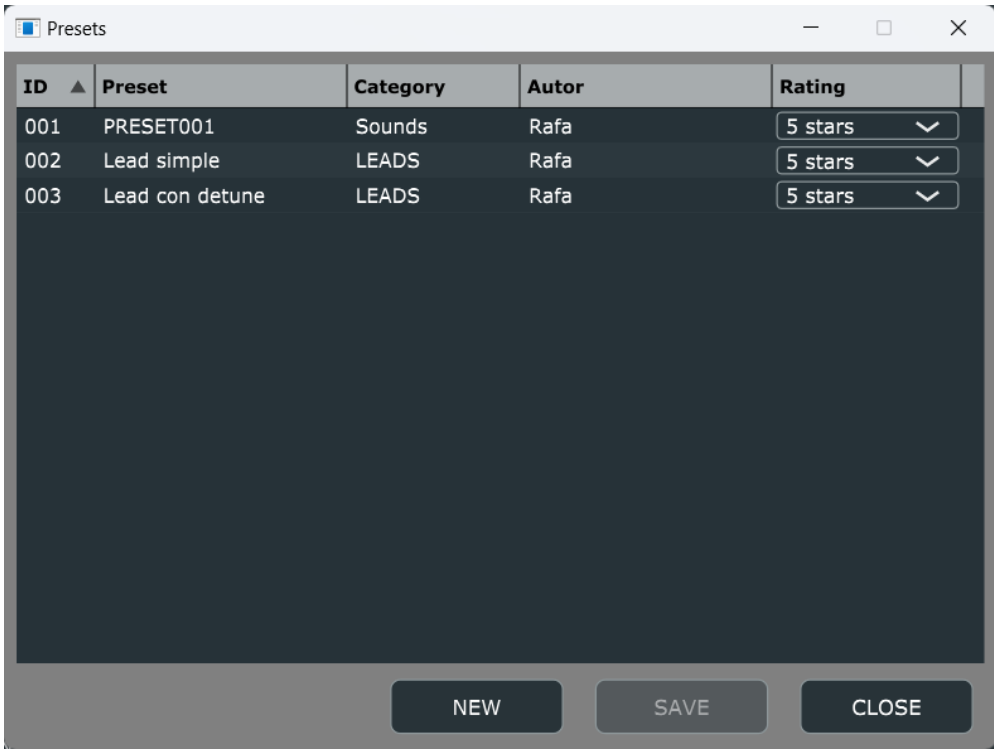

Figura 36: Ventana de presets de la aplicación software

La Figura 36 muestra el aspecto del gestor de presets en nuestra aplicación, que nos permite crear un nuevo *preset* desde cero, hasta un máximo de 1000 elementos, pero también sobrescribir la configuración de un *preset* previamente guardado. Si se crea o modifica un *preset*, se abre una ventana flotante en la que debemos introducir los datos como el nombre, la categoría y el autor.

Los *presets* se guardan en un fichero en formato xml, puesto que JUCE ofrece facilidades para el filtrado de este tipo de ficheros. Si cuando pulsamos el botón de *presets* el fichero no se encuentra junto al ejecutable, se lanzará otra ventana flotante que indica el error y la ruta completa donde el fichero debería encontrarse. El Anexo 9.10 contiene una posible configuración con un solo *preset*.

Por tanto cuando pulsamos sobre el botón de *presets* en la interfaz, se crea una nueva ventana en BasicWindow y ésta crea el componente de la tabla que incluimos dentro de la ventana. La tabla toma todos los datos del fichero xml, es decir, el tamaño y nombre de cada columna, así como los datos de cada fila. Se almacenan los datos en una estructura de datos y se crean los botones con sus correspondientes funciones de *callback*. Por otra parte, este componente también se encarga de realizar el renderizado de la tabla y de los botones de la ventana.

Cuando pulsamos sobre el botón guardar o el botón nuevo y confirmamos, se actualiza o se crea un elemento de la estructura de datos de la tabla, mientras recorremos el vector de componentes de MainComponent del cual obtenemos el valor de cada control utilizando su referencia.

En el destructor del componente de la tabla, que se ejecuta al cerrar la ventana de *presets*, si se ha modificado algún elemento entonces se procede a modificar el fichero xml. Cabe destacar que el fichero debe existir previamente y que no se comprueba si no existe para crear uno nuevo.

4.4.4 Sincronización con la FPGA y dispositivos MIDI

Para realizar la sincronización que se ha visto en la sección 4.4.2, hemos incluido una máquina de estados en la aplicación. Por otra parte, el proyecto de Vivado tiene el componente Synchronize, que se encarga de preparar todos los datos que son transmitidos desde la FPGA al computador cuando éste se lo indica. Este componente tiene como puerto de entrada una señal que recibe desde Parameters y que indica cuando empezar a preparar los datos. Cuando esto sucede, activa una señal que indica al componente UART que debe transmitir y va cambiando de estados para actualizar el dato de entrada a transmitir en UART.

La aplicación por tanto, solo en la versión *standalone*, una vez conecta con la FPGA envía un byte con el identificador para sincronizar y queda a la espera en un bucle para recibir los datos. Una vez los ha recibido envía cada dato a la función setParamValue, con una máquina de estados que es igual que la del componente Parameters en VHDL.

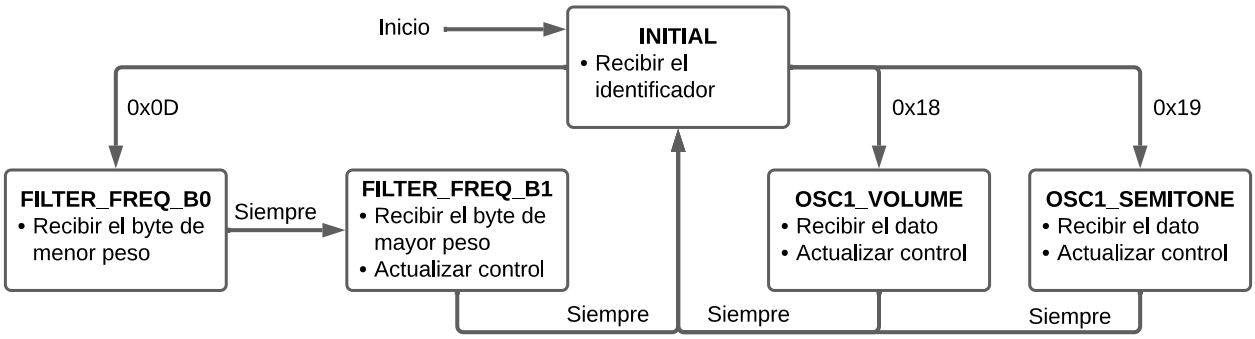

Figura 37: Máquina de estados de la aplicación

La Figura 37 muestra un diagrama de la máquina de estados de la aplicación. En ésta se cambia del estado inicial al estado de cada parámetro, en el que actualizamos el control de la interfaz. El parámetro del filtro es una excepción porque recibimos su valor en dos bytes y se necesitan dos estados para actualizar el control. Cuando hemos recibido el segundo byte aplicamos la ecuación (10) antes de actualizar el control.

$$Parámetro = byte_{MSB} * 256 + byte_{LSB} \tag{10}$$

Por otra parte, en esta sección también comentamos la gestión de controladores MIDI *hardware*. Como hemos visto previamente, solo en la versión *standalone* hemos añadido esta funcionalidad extra porque en el *plug-in* es el DAW el que se encarga de realizar los mapeados.

Primero nuestra clase personalizada del *slider* tiene un *callback* para la selección de la opción del menú flotante para cada *slider*. En este caso actualizamos una variable booleana para indicar en una iteración de processBlock en PluginProcessor, que el parámetro que hemos seleccionado debe aprenderse la próxima vez que se reciba un dato desde un controlador MIDI. Por otra parte, también guardamos una referencia al componente que hemos indicado que vamos a mapear.

En processBlock, para cada mensaje MIDI recibido, si se trata de un mensaje de un controlador se comprueba primero si estamos a la espera de mapear un control de la interfaz. En caso afirmativo, actualizamos una tabla *hash* donde se almacenan para cada canal como clave, un vector como valor con los 128 posibles controles y su correspondiente identificador, que inicialmente estarán vacíos. Que sea un vector de 128 posiciones se debe a que de esta forma, aunque la mayoría de posiciones en memoria no están aprovechadas, tenemos un acceso directo cuando recibimos el número de control del dispositivo MIDI por tanto podemos obtener rápidamente a qué identificador de la interfaz ha sido mapeado. Es importante mantener la velocidad en el procesamiento de mensajes MIDI, aunque a costa de desperdiciar un poco de memoria, si no queremos introducir latencia en la actualización de los controles. Veamos primero el Algoritmo 7 de la gestión MIDI.

Algoritmo 7: Gestión de los mensajes MIDI

```
bucle por cada mensaje MIDI
  obtener los datos de canal, control y valor, filtrados por JUCE
  si juce::esControlador() entonces
    si mapeando entonces // la variable booleana que actualizamos en el callback del slider
      actualizar la estructura de datos (en adelante el hash midi_cc)
      mapeando = FALSE
    si no
      si midi_cc.contiene(canal) entonces
        id = midi_cc[canal][control] // el identificador, posiblemente es un string vacío
        si referencias.contiene(id) entonces
          referencia[id].actualizar(valor)
        fin si
      fin si
    fin si
  fin si
fin bucle
```

Como ya hemos comentado en secciones anteriores, nótese como además ahora necesitamos obtener las referencias de cada control de la interfaz para actualizarlo, que hemos insertado en el constructor de MainComponent. La sección de código donde se acceden a estas referencias solo se ejecuta en modo *standalone*, por tanto no necesitamos vaciar esta estructura en el destructor.

Si se accediese en el *plug-in* necesitaríamos eliminar las referencias puesto que si cerrásemos la ventana, estaríamos apuntando a direcciones que no son válidas.
Veamos ahora el Algoritmo 8 de actualización de la estructura de datos.

Algoritmo 8: Actualización de la estructura de datos para mapeado MIDI

```
recibimos como argumentos el canal y el número de control (del dispositivo MIDI)
si no midi_cc.contiene(canal) entonces
  creamos un vector de 128 posiciones con un string vacío
  añadimos el canal como clave a midi_cc, con este vector
fin si
vector = midi_cc[canal]
vector[control] = id // id es la referencia al componente que guardamos en el callback de slider
actualizamos midi_cc con los nuevos datos

// ahora es importante revisar si este identificador ya estaba mapeado a otro control
bucle canales midi_cc
  si canal_argumento != canal_iteración entonces
    vector = midi_cc[canal_iteración]
    // para no tener que iterar, hemos guardado en otro hash el último control de cada id
    si vector[ultimo_control[id]] = id entonces
      vector[ultimo_control[id]] = “”
      actualizamos midi_cc con los nuevos datos // es decir, eliminamos su control anterior
    fin si
  fin si
fin bucle
ultimo_control[id] = control
escribir_Json = TRUE
```

Actualizamos el fichero de configuración solo si hemos mapeado un control de un *hardware* MIDI a un parámetro de la interfaz. Este fichero se leerá cuando se ejecuta la aplicación para actualizar las estructuras de datos, aunque su existencia no es obligatoria.

Cabe destacar que la gestión MIDI también se podría haber realizado desde la parte *hardware* con VHDL, conectando el dispositivo MIDI por USB a la placa, sin embargo deberíamos implementar el filtrado de mensajes en un nuevo componente. Si por el contrario lo hacemos en esta aplicación el *framework* JUCE ya se encarga de realizar todo el filtrado del protocolo MIDI, de forma que nos agiliza el desarrollo del conjunto del proyecto.

Por último comentar que todo el código fuente está disponible de forma pública en el repositorio de GitHub[10], el directorio *hardware* incluye el diseño HDL y *software,* la aplicación que hemos visto en esta sección desarrollada en JUCE, junto con el fichero Projucer para poder crear una solución en distintos entornos de desarrollo. También tiene el fichero de bitstream y el proyecto de Vivado en un fichero comprimido.

10 GitHub, código fuente: https://github.com/rarepa/fpga_project

5. Resultados

En esta sección se presentan los resultados de las simulaciones y de la implementación. Además se ha incluido un manual que explica cómo conectar los dispositivos y el software necesario para lanzar a ejecución la aplicación del sintetizador.

5.1 Simulaciones

Durante la fase de implementación, se han realizado las simulaciones de los generadores de onda y de los generadores de LFOs. También se ha simulado el generador de ruido y el componente de los filtros. El funcionamiento del resto de componentes se ha comprobado en la transferencia a la placa, mediante la toma de muestras de audio y el análisis en el *software* Audacity. La Figura 38 muestra la simulación de la onda senoidal y la Figura 39 muestra la forma de la LFO sierra.

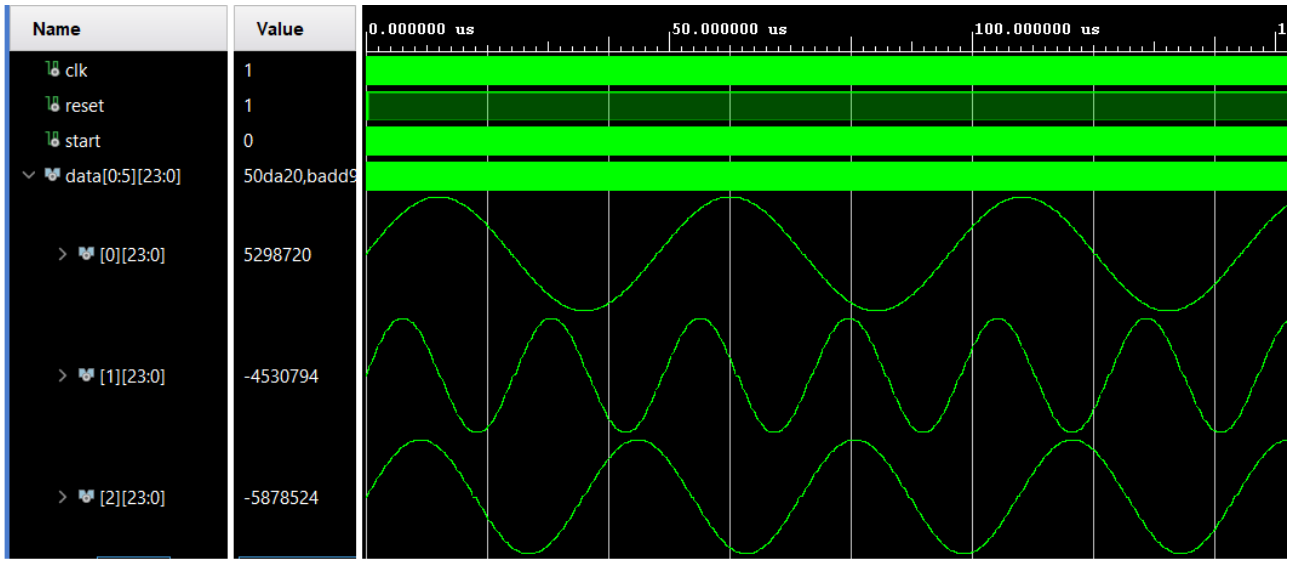

Figura 38: Simulación del generador de onda senoidal

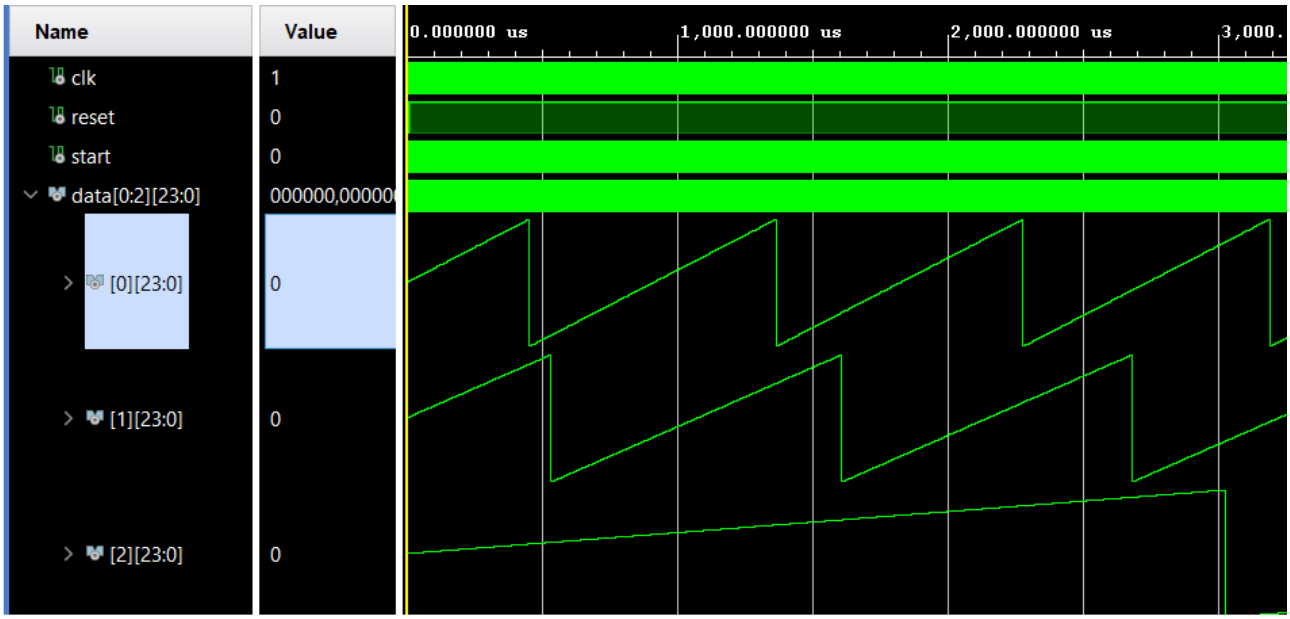

Figura 39: Simulación del generador LFO sierra

5.2 Resultados de implementación

Vivado nos muestra en una tabla la utilización y disponibilidad de los recursos, que se presenta en la Tabla 6, para una polifonía de seis voces. La Figura 40 muestra la utilización en una gráfica, expresada en porcentaje. Aunque BRAM es el recurso más utilizado, si incrementamos las voces éste se mantiene constante mientras que los recursos LUT y DSP aumentan.

Tabla 6: Utilización de los recursos global

Recurso	Utilización	Disponibilidad	Porcentaje utilización
LUT	7938	20800	38,16
LUTRAM	15	9600	0,16
FF	3710	41600	8,92
BRAM	34,50	50	69,00
DSP	33	90	36,67
IO	19	106	17,92
BUFG	2	32	6,25
MMCM	1	5	20,00

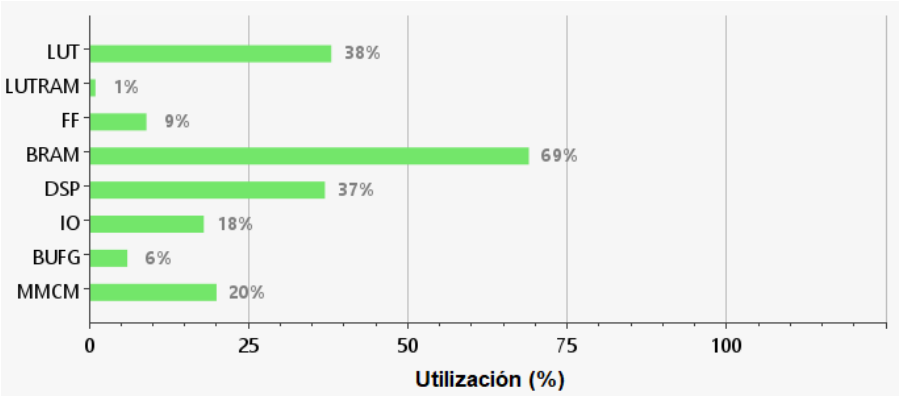

Figura 40: Gráfica de utilización global de los recursos

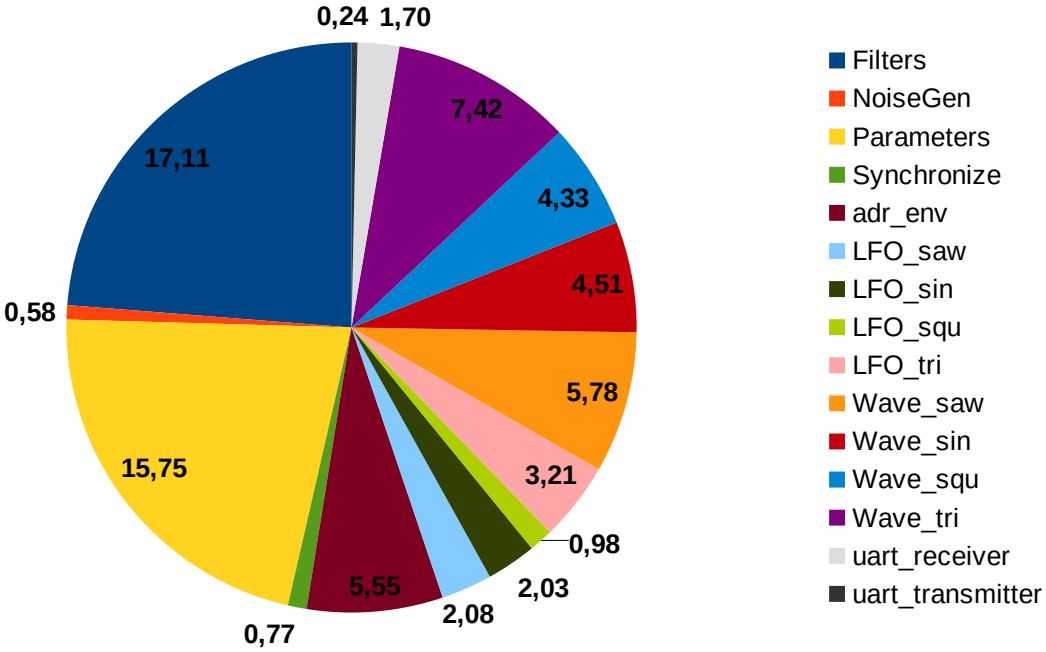

Figura 41: Porcentaje de utilización de LUT por componente

La Figura 41 muestra la utilización de LUT por componente, donde se ve que Filters y Parameters son los componentes que están consumiendo más recursos de este tipo. La Figura 42 muestra la utilización de BRAM. Se han descrito las tablas que contienen las muestras en cada tipo de onda con una memoria ROM y se ha optimizado almacenándolas como BRAM, de forma que podemos relacionar la cantidad de información para componente con el número de BRAM asignado, puesto que conocemos el número de direcciones y en cada una la cantidad de bits almacenada.

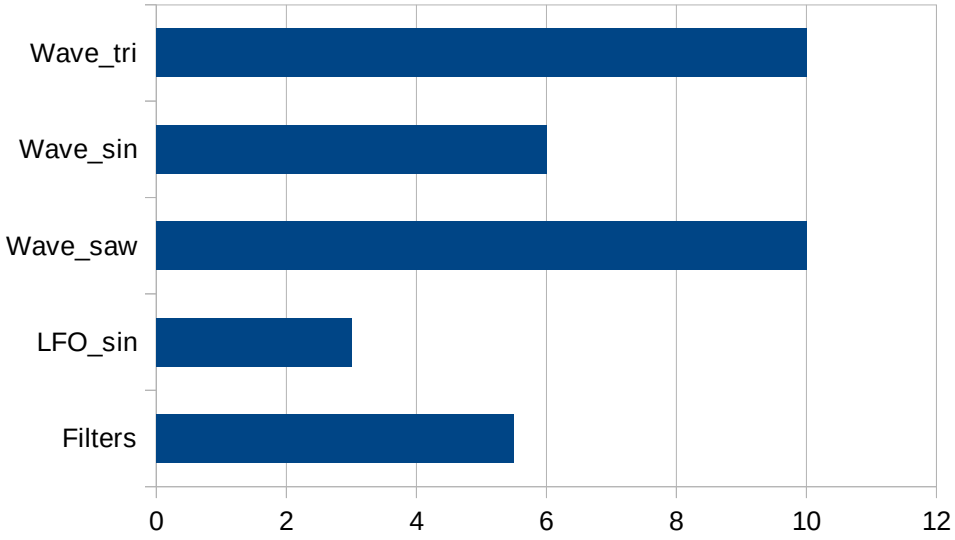

Figura 42: Utilización de BRAM por componente

La Figura 43 muestra el desglose de la utilización de DSP, donde vemos que WaveSelector es el componente que más los necesita debido a la cantidad de operaciones que realiza. Los DSP se utilizan para las operaciones MAC y WaveSelector requiere dos operaciones para el cálculo de la LFO de cada oscilador, dos más para la inicialización del periodo de *portamento* y los ajustes de afinación, además de una multiplicación para cada voz para el cálculo del valor de amplitud del ADSR. Filters necesita calcular cada tipo de filtro, ya que no hemos podido optimizarlo, más la modulación de LFO. La onda cuadrada necesita una multiplicación para variar el ancho de pulso en el nivel alto y otra en el nivel bajo. En las versiones iniciales se replicaban componentes, por lo que el consumo de estos recursos se disparaba. En la versión optimizada, como se ha comentado anteriormente, se utiliza una máquina de estados para reutilizar el mismo componente en ciclos de reloj consecutivos. Al tener mucho margen para obtener los cálculos, podemos hacerlo así y limitar el consumo de recursos, tanto DSP como BRAM.

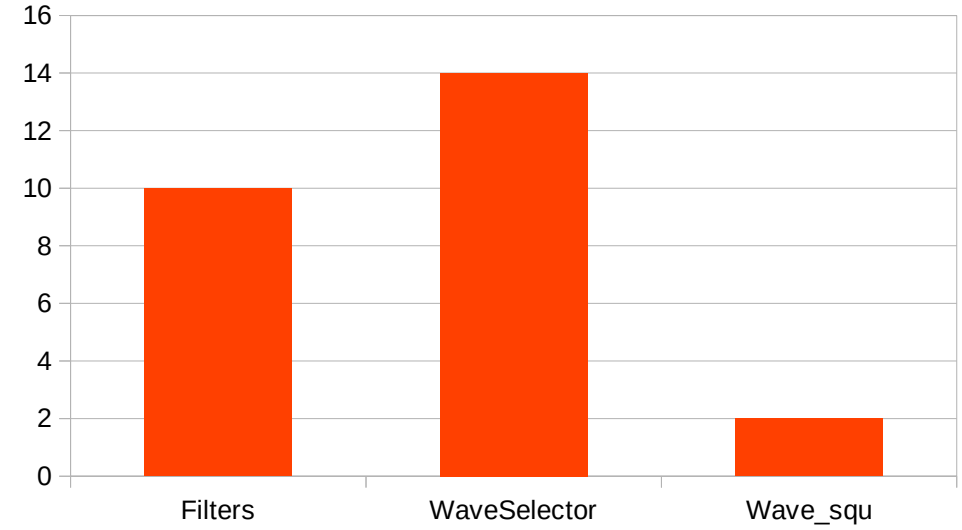

Figura 43: Utilización de DSP por componente

La Tabla 7 muestra el desglose de los datos para la utilización por componente.

Tabla 7: Utilización de recursos por componente

	LUTs	FFs	BRAMs	DSPs
i2s_playback	***7938***	***3710***	***34,5***	***33***
i2s_clock	**0**	**0**	**0**	**0**
i2s_transceiver	**176**	**211**	**0**	**0**
Synthesizer	**7421**	**3441**	**34,5**	**33**
Filters	1270	364	5,5	10
NoiseGen	43	25	0	0
Parameters	1169	335	0	0
Synchronize	57	24	0	0
WaveSelector	4480	2616	29	14
adr_env	*412*	*143*	*0*	*0*
LFO_saw	*154*	*171*	*0*	*0*
LFO_sin	*151*	*118*	*3*	*0*
LFO_squ	*73*	*102*	*0*	*0*
LFO_tri	*238*	*174*	*0*	*0*
Wave_saw	*429*	*457*	*10*	*0*
Wave_sin	*335*	*239*	*6*	*0*
Wave_squ	*321*	*216*	*0*	*2*
Wave_tri	*551*	*461*	*10*	*0*
UART	**144**	**58**	**0**	**0**
receiver	126	30	0	0
transmitter	18	28	0	0

Respecto al análisis de la temporización, se ha obtenido un WNS (Worst Negative Slack) de 51.25 ns. En la ecuación (11) se ha calculado la frecuencia máxima de funcionamiento, que es la inversa del periodo requerido, de 88.6 ns, menos el valor de WNS. Se alcanzan 22.67 MHz cuando hemos solicitado 11.28 MHz, así que aún hay margen para poder incorporar más funcionalidad a costa de reducir la frecuencia sin empeorar la ruta crítica.

$$F_{Máx} = \left(\frac{1}{T_{Req} - WNS} \right) = \left(\frac{1}{88.6 - 51.25} \right) = 26{,}77\ MHz \tag{11}$$

La Tabla 8 contiene un resumen del consumo estimado y La Figura 44 muestra la estimación de consumo en el chip, diferenciado por tipo de recurso.

Tabla 8: Resumen del consumo

Consumo total	0,218 W
Temperatura de unión	26,1 ºC
Margen térmico	58,9 ºC (11,7W)
ϴJA Efectivo	5,0 ºC/W

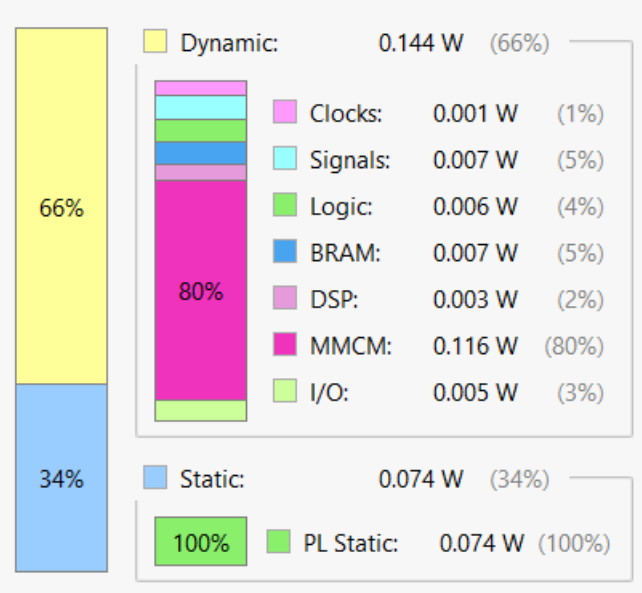

Figura 44: Consumo en el chip

Respecto a estos resultados, se puede observar como el consumo en una FPGA es mucho menor si lo comparamos con una CPU o una GPU, lo que la hace más eficiente desde el punto de vista de rendimiento por vatio. La implementación tiene un consumo total de 0,218 W mientras que una CPU está sobre los 40-50 W en estado de reposo.

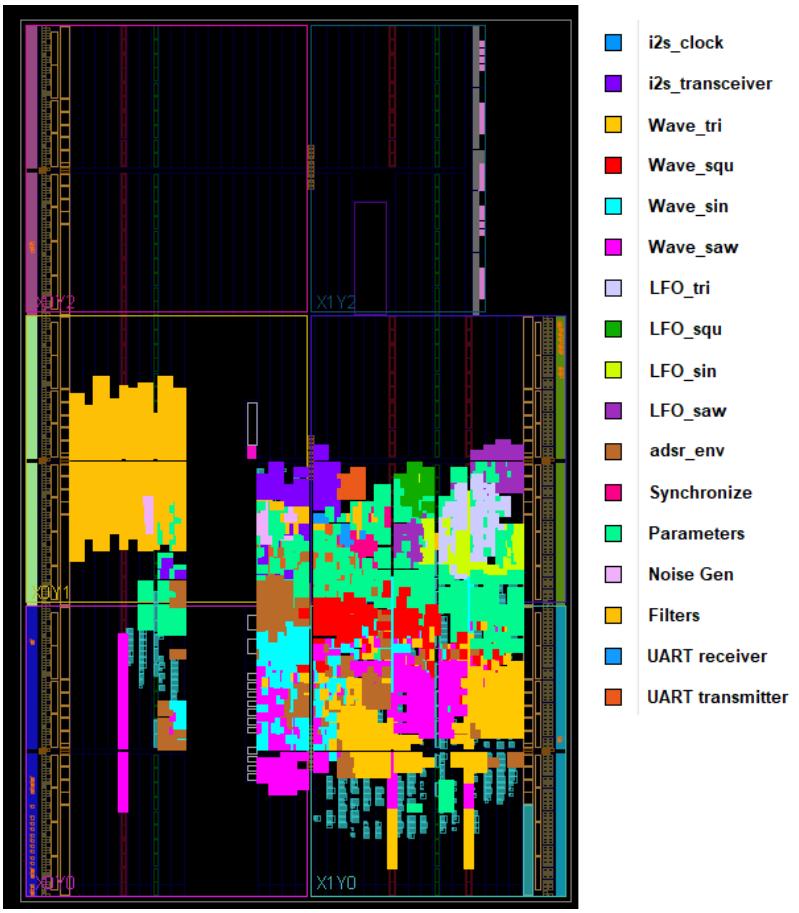

Figura 45: Diseño de la implementación

La Figura 45 muestra el diseño de la implementación, donde se observa en color cuánto espacio ocupa cada componente en silicio. Esto es útil por ejemplo para realizar comparaciones y ver qué se puede tratar de optimizar, qué es más importante optimizar o verificar si hay más espacio, entre otras.

5.3 Manual de setup

En esta sección se explica el proceso que hemos seguido para lanzar a ejecución el sintetizador y capturar muestras de audio, en Windows. Necesitaremos seguir los siguientes pasos:

1. Conectar el dispositivo Pmod I2S2 al puerto JA de la placa Basys 3.
2. Conectar el puerto JTAG (micro USB) de la placa a un puerto USB del computador.
3. Conectar la línea de salida del Pmod I2S2, de color verde, con un cable Jack estéreo a una entrada de una tarjeta de sonido externa, que típicamente es RCA. Cabe destacar que la tarjeta externa no es imprescindible, ya que se puede conectar con dos extremos Jack a una entrada de la tarjeta interna del PC o pórtatil, sin embargo una tarjeta externa consigue habitualmente más calidad de sonido.
4. Adicionalmente se puede conectar un controlador MIDI a una entrada USB del PC, algunos dispositivos también se pueden conectar por Bluetooth.
5. De forma opcional también se puede conectar un reproductor de música o un dispositivo similar a la entrada del Pmod I2S2, de color azul, con un cable Jack.
6. Descargar desde Github el proyecto de Vivado del fichero comprimido.
7. Descomprimir y abrir el proyecto en Vivado.
8. Realizar los procesos de síntesis, implementación y generación del bitstream.
9. Transferir el diseño a la placa desde el Hardware Manager. Alternativamente a los pasos 6-8, se puede descargar el fichero de bitstream y utilizar Vivado Lab para la transferencia.
10. Descargar y descomprimir el *framework* JUCE.
11. Descargar el directorio *software* desde el repositorio de Github de este proyecto.
12. Abrir el fichero con extensión .jucer en Projucer y generar una solución para Visual Studio.
13. Abrir la solución y compilar la aplicación. En el fichero de cabecera de PluginProcessor.h está la macro STANDALONE, que debemos establecer a *true* o *false*, siendo *false* para la versión *plug-in*. Hay que tener cuidado con compilar solo la solución Standalone o VST3 y no toda la solución.
14. Instalar los drivers[11] para la gestión del puerto COM virtual en Windows.

A partir de este momento tenemos dos posibles opciones de ejecución.

Tabla 9: Pasos de ejecución, según la versión de la aplicación

Standalone	*Plug-in* VST3
• Descargar y ejecutar el software Audacity o similar. • Seleccionar la entrada adecuada para el dispositivo de grabación en la configuración de audio. • Seleccionar “Comenzar monitorización”, desde el botón con el icono del micrófono. • Ejecutar la aplicación *standalone*. • Pulsar el teclado del PC para hacer sonar las notas, o bien desde un piano MIDI.	• Abrir un DAW o un host que soporte instrumentos VST3, podemos encontrar software gratuito como VSTHost[12]. • Cargar el *plug-in*. • En un DAW podemos seleccionar la entrada de audio en una nueva pista. También se puede utilizar Audacity para la monitorización. • Utilizar el teclado emulado que nos proporciona el host, o bien un controlador MIDI para hacer sonar las notas.

11 VCP drivers: https://ftdichip.com/drivers/vcp-drivers/
12 VSTHost: https://www.hermannseib.com/english/vsthost.htm

En ambos casos, debemos seleccionar el puerto COM para la conexión UART si nuestro puerto es distinto de “COM4”. Podemos comprobar cuál es nuestro puerto desde el “Administrador de dispositivos” de Windows. Si vemos que el número es distinto, debemos pulsar sobre el botón de opciones de la aplicación, escribir el puerto correcto y pulsar en el botón UART. Este paso solo se debe hacer una vez ya que si la conexión ha sido satisfactoria, se guarda en la configuración.

Si queremos lanzar a ejecución la aplicación en un sistema operativo diferente, hay que actualizar la gestión de los mensajes mediante el puerto serie, puesto que en este proyecto se ha utilizado la biblioteca de Windows.

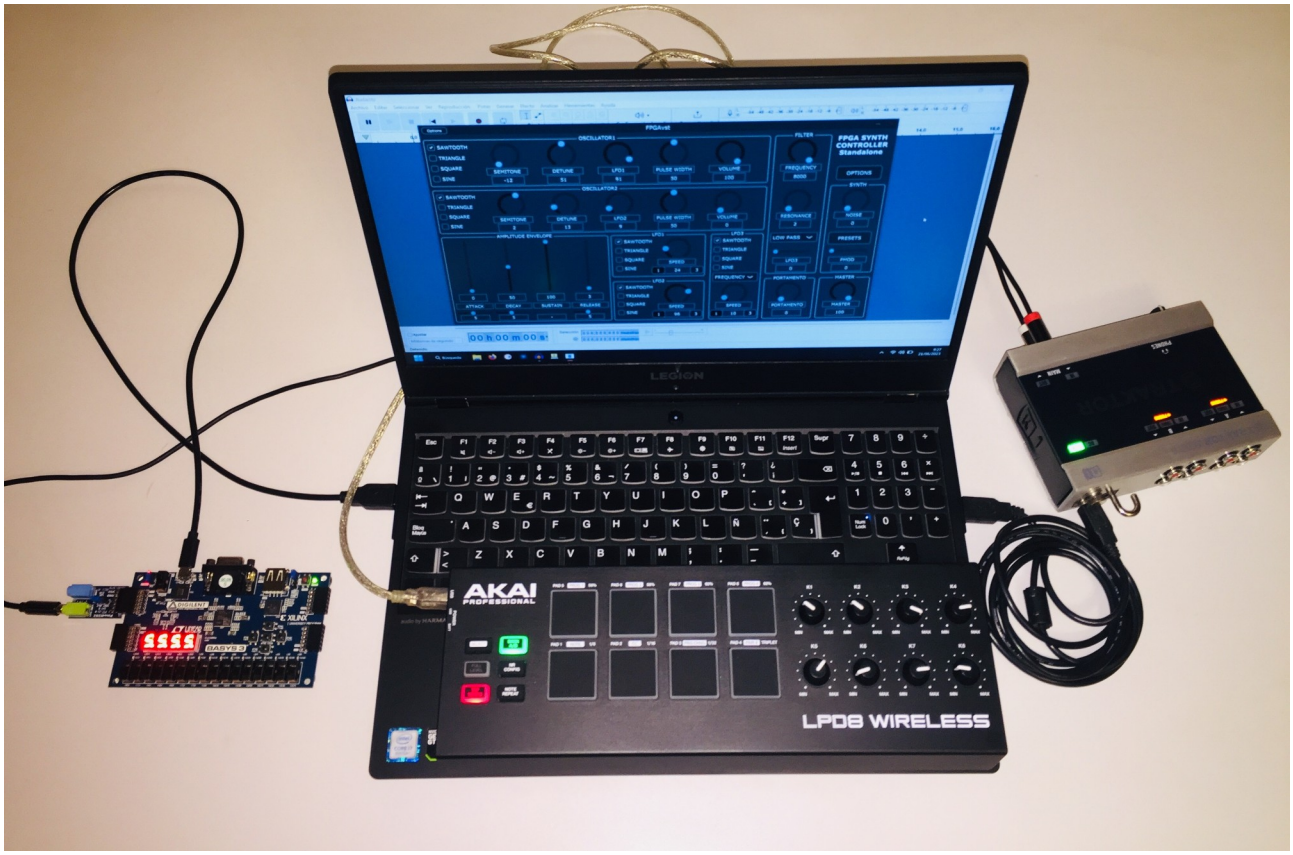

Figura 46: Captura del sistema en funcionamiento

La Figura 46 muestra la imagen donde hemos tomado una captura del sistema en funcionamiento, en modo *standalone* junto con el software Audacity y con un controlador MIDI conectado por USB.

En el repositorio de GitHub[13] se encuentra el directorio *samples*, con muestras de audio en las que se puede escuchar las capacidades del sintetizador, demostrando el funcionamiento de cada tipo de onda a distintas frecuencias, la polifonía y mezcla, el efecto de *portamento*, la aplicación del generador de envolventes de amplitud con distintos valores de ADSR, las modulaciones de LFO con varios tipos de onda, la aplicación del generador de ruido y el procesamiento de cada filtro, incluyendo la modulación por LFO de la frecuencia de corte y la resonancia.

13 Muestras de audio: https://github.com/rarepa/fpga_project/tree/main/samples

6. Conclusiones

A lo largo de este trabajo nos hemos propuesto como objetivo implementar un sintetizador de sonidos que funcione de forma eficiente en un dispositivo FPGA. Hemos conseguido que de forma satisfactoria el sintetizador logre generar varios tipos de onda con polifonía, trabajando la síntesis aditiva pero también la sustractiva mediante la aplicación de filtros, además conseguir dinámica en los sonidos implementando los osciladores de baja frecuencia y el generador de envolventes de amplitud, logrando una utilización óptima de los recursos disponibles. Este proyecto puede servir para que cualquier persona analice su diseño y pueda aprovechar los componentes que hemos utilizado, pero también para continuar con el trabajo y añadir más funcionalidad o mejoras.

En primer lugar, se ha comprobado que plantear un diseño adecuado es muy importante para una gestión óptima de la utilización de los recursos, ya que en la fase en la que hemos implementado la polifonía nos dimos cuenta que el diseño que habíamos planteado alcanzaba una utilización máxima de los recursos. Mientras que instanciar varias veces los componentes que generan cada forma de onda para añadir más voces supone alcanzar el límite de recursos para tan solo seis voces en la placa sobre la que hemos trabajado, añadir una máquina de estados en cada uno de estos componentes y que generen las muestras de una voz en ciclos distintos ayuda a conseguir una optimización importante, hasta el punto de poder generar hasta más de 32 voces de polifonía.

Por otra parte, sería importante volver a estudiar el diseño para determinar la frecuencia de cada nota, si bien esta se ajusta de forma adecuada muy correcta para frecuencias bajas, a medida que utilizamos frecuencias más altas estamos perdiendo una precisión importante, donde reproducir frecuencias por encima de los 14 kHz aproximadamente no es posible, aunque cabe destacar que estamos hablando de la frecuencia fundamental y no de los armónicos que se generan.

Durante la fase de implementación, en la etapa de la construcción de los filtros y del generador de ruido, se ha llegado a la conclusión de que si las señales internas que se generan tienen una profundidad de 24 bits, sería adecuado que el DAC tenga una configuración superior, por ejemplo de 32 bits. Se puede compensar la pérdida de amplitud con un componente que se encargue de realizar algún tipo de compresión o simplemente amplificar la señal, normalizando la señal global. Trabajando de este modo se pueden conseguir ventajas, por ejemplo las resonancias de los filtros no se verían limitadas, tampoco cuando se transforma la señal para agregar ruido y los valores de la señal se encuentran próximos al límite de bits, puesto que de esta forma disponemos de un techo sobre el que trabajar la señal sin tener que recortarla.

Por otra parte se ha comprobado que las simulaciones no siempre reflejan el comportamiento real en el *hardware*, obteniendo unos resultados correctos al intentar economizar la utilización de los recursos en el componente de los filtros, pero que no funciona de forma adecuada al transferirlo a la placa Basys-3.

Por último destacar que este sintetizador supone una base robusta sobre la que seguir trabajando y que hay mucho rango para incorporar nuevas funcionalidades, como por ejemplo otros tipos de síntesis, o trabajar sobre posibles mejoras. Puesto que la polifonía en este sintetizador se puede gestionar de forma genérica es posible reducir el número de voces para poder trabajar en nuevas características, incluso con una placa que no disponga de gran cantidad de recursos.

7. Posibles ampliaciones y mejoras

Como se ha mencionado en el apartado anterior, la capacidad para añadir nuevas funciones en este proyecto es muy amplia, pero también es posible mejorar algunas cualidades. Veamos cuáles son las posibles mejoras.

- Revisar la discrepancia entre la simulación de los filtros cuando se preparan los operandos para realizar una sola multiplicación en cada ciclo y su mal funcionamiento en la placa de prototipado, donde el procesamiento de los filtros no funciona de la forma esperada.
- Analizar el motivo que causa que el pin R2 afecte a la recepción de datos de UART, ya que no se recibe ningún bit en el pin A18 cuando el primero se incluye en el fichero *constraints*.
- Estudiar una posible mejora para determinar los ciclos necesarios para las frecuencias de una nota, ajustando de forma más precisa la afinación en frecuencias altas.
- Configurar un bitstream no volátil que se mantenga cuando se pierde la alimentación.

Respecto a las posibles ampliaciones, destacamos las que creemos que son más interesantes al mismo tiempo que son alcanzables, siendo algunas de éstas de aplicación casi trivial.

- Trabajar en la portabilidad para la recepción y envío de datos mediante el puerto serie en la aplicación del computador, adaptándolo a otros sistemas operativos como Linux.
- Incluir un parámetro para amplificar la señal recibida por el dispositivo Pmod I2S2, puesto que es posible que se utilice algún aparato que no ofrezca el nivel de entrada adecuado.
- Incorporar un control para ajustar la fase de los osciladores de baja frecuencia, donde éste simplemente modifique el valor de inicialización de los contadores de cada LFO, de forma que ampliamos las posibilidades para las modulaciones.
- Sincronizar las LFOs a un tempo definido por el usuario.
- Generar otros tipos de ruido, como pueden ser el blanco o rosa.
- Agregar un ecualizador parámetrico o por bandas, con el ancho de banda ajustable.
- Incluir nuevos efectos de procesamiento, como delay, panning, reverberación, distorsiones como saturación, clipping o overdrive, chorus, echo y demás.
- Ofrecer la posibilidad para elegir el encaminamiento de los efectos, es decir, establecer el orden en el que éstos se aplican.
- Estudiar la viabilidad de la transmisión del audio al computador en formato binario, a través de un módulo USB que ofrezca velocidades de transmisión superiores a las transferencias en UART, ya que una que mantener una latencia baja es un requisito prioritario.
- Aplicar mayor funcionalidad para el componente generador de envolventes, donde poder modular otros parámetros además de la amplitud, como pueden ser el *pitch* o la frecuencia de corte y resonancia de los filtros.
- Incluir curvas logarítmicas para estas envolventes, con formas cóncava o convexa ya que las posibilidades de modulación que éstas ofrecen son mayores a la clásica lineal.
- Gestionar la amplitud de una nota mediante la presión con la que ésta se ha pulsado en un dispositivo MIDI, esto es comúnmente conocido como *velocity*.

Pero cabe destacar que aunque enumeramos solamente estas ampliaciones, las posibilidades de incluir muchas más características son muy extensas, desde la incorporación de nuevas formas de síntesis, como por ejemplo granular, vectorial o por modulación de frecuencia, hasta incorporar más funcionalidad como arpegiadores o diseñar una configuración para que podamos controlar la FPGA con dispositivos MIDI a través de Bluetooth 4.0 o una red inalámbrica, así como gestionar la configuración de los parámetros mediante una aplicación del teléfono móvil o *tablet*.

8. Referencias bibliográficas

[1] A. Sola Salguero. Diseño y tutorial de un sintetizador con la herramienta software Reaktor, Gandía 2010, pp 4, 37, 45.

[2] P. Matrínez Serrano. Programación de un sintetizador MIDI usando el microcontrolador MSP430FR2355, Sevilla, 2020, pp xi, 6, 9.

[3] J. Cañada Toledo. Diseño e implementación hardware de un sintetizador digital modular de audio, Madrid, 2017, pp. 20-22.

[4] B. Mealy and F. Tappero. Free Range VHDL, 1.23 ed. USA: Free Range Factory, 2023, pp 5-7.

[5] M. Caballero, H. Steinbrink, Waldorf Kyra Manual, Germany, 2019, pp 3, 136.

[6] Focusrite, Novation Peak Guía de usuario, Reino Unido, 2017, pp 4.

[7] Diligent. Basys 3 FPGA Board Reference Manual, rev. C Pullman, WA: Diligent, 2016, pp 1-19.

[8] The MIDI Manufacturers Association. The complete MIDI 1.0 Detailed Specification, , third edition, 2014, Los Angeles, CA, pp 1-3.

[9] D. Miles Huber. The MIDI Manual, 3rd Edition, Routledge, 2012, pp 1-5.

[10] P. Jiménez, I. M. Bernal. Diseño e implementación de un sistema de adquisición, compresión y almacenamiento de imágenes empleando VHDL y FPGAs, Ecuador, pp 1-2.

[11] M. Amagasaki, Y. Shibata, Principles and Structures of FPGAs, 2018 Springer, Singapore, pp 1-60.

[12] D. Koch, F. Hanning, D. Ziener, FPGAs for Software Programmers, 2016 Springer, Switzerland, pp 10-20.

[13] M. van Kooten, V. Sarkovic, Nyquist-Shanon sampling theorem, 2019, pp 1.

[14] A. Aurobindo, S. Kakatkar, N. Mehendale, A Review on Applications of FPGAs, Mumbai, Maharashtra, 2022 pp 3-10.

[15] D. Prairie, L. Graves, “AMD Completes Acquisition of Xilinx”, 14 de Febrero 2022, Disponible en: https://www.amd.com/en/newsroom/press-releases/2022-2-14-amd-completes-acquisition-of-xilinx.html (Último acceso: 22 de junio de 2023)

[16] Advanced Micro Devices, Inc, “Recommended FPGA and SOC Development Boards for Universities”, Disponible en: https://www.xilinx.com/support/university/xup-boards.html (Último acceso: 22 de junio de 2023)

[17] Xilinx, “7 Series Product Selection Guide”, XMP101 (v1.8), 8 de Abril, Disponible en: https://docs.xilinx.com/v/u/en-US/7-series-product-selection-guide (Último acceso: 22 de junio de 2023)

[18] Diligent, “Pmod I2S2 Reference Manual”, Disponible en: https://digilent.com/reference/pmod/pmodi2s2/reference-manual (Último acceso: 22 de junio de 2023)

[19] Cirrus Logic, “CS5343/44 Product Data Sheet”, DS687F5, Marzo 2015, Disponible en: https://statics.cirrus.com/pubs/proDatasheet/CS5343-44_F5.pdf (Último acceso: 22 de junio de 2023)

[20] Philips Semiconductors, “I2S bus specification”, 5 de Junio 1996, Disponible en: https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf (Último acceso: 22 de junio de 2023)

[21] Digikey, “I2S Pmod Quick Start (VHDL)”, Marzo 2021, Disponible en: https://forum.digikey.com/t/i2s-pmod-quick-start-vhdl/13065 (Último acceso: 22 de junio de 2023)

[22] E. Peña, M. G. Legaspi, “UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter”, Vol 54 No 4, Diciembre 2020, Disponible en: https://www.analog.com/media/en/analog-dialogue/volume-54/number-4/uart-a-hardware-communication-protocol.pdf (Último acceso: 22 de junio de 2023)

[23] A. Subdin, “UART Interface in VHDL for Basys3 Board”, 1 de Noviembre 2020, Disponible en: https://www.hackster.io/alexey-sudbin/uart-interface-in-vhdl-for-basys3-board-eef170 (Último acceso: 22 de junio de 2023)

[24] Egmont, “Serial Communication in Win32”, Disponible en: http://www.egmont.com.pl/addi-data/instrukcje/standard_driver.pdf (Último acceso: 22 de junio de 2023)

[25] H. Chamberlin, Musical Applications of Microprocessors, 2nd edition, 1985, Indiana, USA, pp. 481-499.

[26] S. Saponara, A. De Gloria, Applications in Electronics Pervading Industry, Environment and Society, 2019, Springer, Switzerland, pp 469-475.

[27] Inspired Acoustics, MIDI note numbers and center frequencies, Disponible en: https://www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies (Último acceso: 22 de junio de 2023)

[28] AMD-Xilinx, “7 Series FPGAs Clocking Resources – User Guide”, UG472 (v1.14), 30 de Julio de 2018, Disponible en: https://docs.xilinx.com/v/u/en-US/ug472_7Series_Clocking (Último acceso: 22 de junio de 2023)

[29] JUCE Forum, “Can’t Write/Restore Integer plugin parameter”, Septiembre 2019, Disponible en: https://forum.juce.com/t/can-t-write-restore-integer-plugin-parameter/35414 (Último acceso: 22 de junio de 2023)

9. Anexos

9.1 Cálculo de los incrementos de muestra de las ondas sierra y triangular

```
#include <iostream>
#include <iomanip>
#include <stdint.h>
#include <math.h>
#include <vector>

using namespace std;

double getFreq(uint8_t midiNote) {
    const double A4_FREQ = 440;
    const int32_t A4_MIDI_NOTE = 69;
    return A4_FREQ * pow(2.0, ((double) (((int32_t) midiNote) - A4_MIDI_NOTE)) / 12.0);
}


uint64_t getInc(uint8_t midiNote) {
    const uint32_t SAMPLE_RATE = 44100; // nuestra frecuencia de muestreo es 44.1 kHz
    double freq = getFreq(midiNote); // pasamos la nota midi y nos devuelve su frecuencia
    double inc = (freq * 16777216) / SAMPLE_RATE; // (freq * 2^24 / 44100) * 2^16 -> Q24.16
    uint64_t ret = round(inc * 65536); // en 64 bits (necesitamos 40 y no cabe en 32)
    return ret;
}


int main() {
    uint8_t index = 9;
    uint8_t scale = 0;
    vector<string> notes = {"C","C#","D","D#","E","F","F#","G","G#","A","A#","B"}; // notas escala
    cout << setprecision(2) << fixed; // imprimirá double con una precisión de 2 dígitos
    for (uint8_t n = 0; n < 128; n++) { // setw = 10 dígitos hex * 4 bits/1 digito -> 40 bits
        string s = n > 20 ? notes[index] + to_string(scale) : "XX";
        scale = index == 11 ? scale + 1 : scale;
        cout << "\t\tx\"" << hex << setw(10) << setfill('0') << getInc(n) << "\",   -- note " << dec <<
setw(3) << setfill(' ') << ((int) n);
        cout << " - " << setw(3) << setfill(' ') << s << " - " << setw(8) << setfill(' ') << getFreq(n) <<
endl;
        if (n > 20) index = index < 11 ? index + 1 : 0;
    }
    return 0;
}
```

9.2 Tabla de incremento para las 128 notas, en decimal y hexadecimal en coma fija

Nota	Frecuencia	Incremento decimal	Coma fija hexadecimal
0	8.18	0.0001853922656609	000c265db8
1	8.66	0.0001964162634473	000cdf516e
2	9.18	0.0002080957822544	000da34494
3	9.72	0.0002204698013904	000e72de96
4	10.30	0.0002335796179938	000f4ed0d2
5	10.91	0.0002474689848590	001037d732
6	11.56	0.0002621842564567	00112eb8cf
7	12.25	0.0002777745436379	0012344895
8	12.98	0.0002942918775368	00134965fd
9	13.75	0.0003117913832200	00146efdcb
10	14.57	0.0003303314636608	0015a60ad5
11	15.43	0.0003499739946543	0016ef96dd
12	16.35	0.0003707845313217	00184cbb70
13	17.32	0.0003928325268947	0019bea2db
14	18.35	0.0004161915645088	001b468929
15	19.45	0.0004409396027807	001ce5bd2c
16	20.60	0.0004671592359876	001e9da1a4
17	21.83	0.0004949379697180	00206fae65
18	23.12	0.0005243685129133	00225d719d
19	24.50	0.0005555490872757	0024689129
20	25.96	0.0005885837550736	002692cbfb
21	27.50	0.0006235827664399	0028ddfb97
22	29.14	0.0006606629273216	002b4c15ab
23	30.87	0.0006999479893086	002ddf2db9
24	32.70	0.0007415690626434	00309976df
25	34.65	0.0007856650537893	00337d45b6
26	36.71	0.0008323831290176	00368d1251
27	38.89	0.0008818792055615	0039cb7a59
28	41.20	0.0009343184719753	003d3b4348
29	43.65	0.0009898759394360	0040df5cc9
30	46.25	0.0010487370258266	0044bae33a
31	49.00	0.0011110981745514	0048d12253
32	51.91	0.0011771675101472	004d2597f5
33	55.00	0.0012471655328798	0051bbf72d
34	58.27	0.0013213258546431	0056982b55
35	61.74	0.0013998959786171	005bbe5b72
36	65.41	0.0014831381252868	006132edbe
37	69.30	0.0015713301075786	0066fa8b6c
38	73.42	0.0016647662580352	006d1a24a2
39	77.78	0.0017637584111229	007396f4b2
40	82.41	0.0018686369439505	007a768690
41	87.31	0.0019797518788719	0081beb993
42	92.50	0.0020974740516533	008975c675
43	98.00	0.0022221963491029	0091a244a5
44	103.83	0.0023543350202945	009a4b2fea

45	110.00	0.0024943310657596	00a377ee5a
46	116.54	0.0026426517092862	00ad3056ab
47	123.47	0.0027997919572343	00b77cb6e4
48	130.81	0.0029662762505737	00c265db7d
49	138.59	0.0031426602151573	00cdf516d9
50	146.83	0.0033295325160704	00da344944
51	155.56	0.0035275168222458	00e72de963
52	164.81	0.0037372738879010	00f4ed0d20
53	174.61	0.0039595037577438	01037d7326
54	185.00	0.0041949481033065	0112eb8cea
55	196.00	0.0044443926982058	012344894a
56	207.65	0.0047086700405889	0134965fd5
57	220.00	0.0049886621315193	0146efdcb5
58	233.08	0.0052853034185724	015a60ad55
59	246.94	0.0055995839144685	016ef96dc9
60	261.63	0.0059325525011474	0184cbb6f9
61	277.18	0.0062853204303146	019bea2db1
62	293.66	0.0066590650321408	01b4689289
63	311.13	0.0070550336444916	01ce5bd2c6
64	329.63	0.0074745477758020	01e9da1a40
65	349.23	0.0079190075154876	0206fae64c
66	369.99	0.0083898962066130	0225d719d3
67	392.00	0.0088887853964115	0246891294
68	415.30	0.0094173400811779	02692cbfaa
69	440.00	0.0099773242630385	028ddfb969
70	466.16	0.0105706068371449	02b4c15aaa
71	493.88	0.0111991678289371	02ddf2db91
72	523.25	0.0118651050022947	0309976df3
73	554.37	0.0125706408606291	0337d45b63
74	587.33	0.0133181300642815	0368d12512
75	622.25	0.0141100672889833	039cb7a58d
76	659.26	0.0149490955516041	03d3b43480
77	698.46	0.0158380150309752	040df5cc97
78	739.99	0.0167797924132260	044bae33a6
79	783.99	0.0177775707928231	048d122528
80	830.61	0.0188346801623558	04d2597f54
81	880.00	0.0199546485260771	051bbf72d3
82	932.33	0.0211412136742898	056982b554
83	987.77	0.0223983356578741	05bbe5b723
84	1046.50	0.0237302100045894	06132edbe6
85	1108.73	0.0251412817212582	066fa8b6c6
86	1174.66	0.0266362601285630	06d1a24a24
87	1244.51	0.0282201345779665	07396f4b1a
88	1318.51	0.0298981911032082	07a76868ff
89	1396.91	0.0316760300619505	081beb992f
90	1479.98	0.0335595848264521	08975c674c
91	1567.98	0.0355551415856462	091a244a51
92	1661.22	0.0376693603247116	09a4b2fea8
93	1760.00	0.0399092970521542	0a377ee5a5
94	1864.66	0.0422824273485796	0ad3056aa8
95	1975.53	0.0447966713157482	0b77cb6e45
96	2093.00	0.0474604200091789	0c265db7cb

97	2217.46	0.0502825634425165	0cdf516d8b
98	2349.32	0.0532725202571261	0da3449448
99	2489.02	0.0564402691559330	0e72de9634
100	2637.02	0.0597963822064163	0f4ed0d1ff
101	2793.83	0.0633520601239009	1037d7325e
102	2959.96	0.0671191696529042	112eb8ce98
103	3135.96	0.0711102831712924	12344894a1
104	3322.44	0.0753387206494231	134965fd50
105	3520.00	0.0798185941043084	146efdcb4a
106	3729.31	0.0845648546971592	15a60ad551
107	3951.07	0.0895933426314964	16ef96dc8b
108	4186.01	0.0949208400183578	184cbb6f96
109	4434.92	0.1005651268850330	19bea2db17
110	4698.64	0.1065450405142522	1b4689288f
111	4978.03	0.1128805383118661	1ce5bd2c68
112	5274.04	0.1195927644128326	1e9da1a3fe
113	5587.65	0.1267041202478018	206fae64bc
114	5919.91	0.1342383393058084	225d719d30
115	6271.93	0.1422205663425848	2468912943
116	6644.88	0.1506774412988463	2692cbfaa0
117	7040.00	0.1596371882086168	28ddfb9695
118	7458.62	0.1691297093943183	2b4c15aaa1
119	7902.13	0.1791866852629929	2ddf2db915
120	8372.02	0.1898416800367156	309976df2d
121	8869.84	0.2011302537700659	337d45b62e
122	9397.27	0.2130900810285044	368d12511f
123	9956.06	0.2257610766237322	39cb7a58cf
124	10548.08	0.2391855288256652	3d3b4347fc
125	11175.30	0.2534082404956038	40df5cc977
126	11839.82	0.2684766786116168	44bae33a60
127	12543.85	0.2844411326851695	48d1225285

9.3 Valores para las muestras de la onda senoidal

```
import numpy as np
from numpy import sin, pi

x = np.linspace(0,1,8192+2) # 8192 muestras (quitaremos los dos valores con 0's)
lut = sin(pi*x)*((2**(24-1))-1) # 2**23-1 (24 bits DAC)

for i, val in enumerate(lut):
    if i % 10: # cada diez líneas, salto
        print('x"'f"{int(val):06X}"'", end=', ') # 6 dígitos hex mínimo
    else:
        print('x"'f"{int(val):06X}"'", end=', ')
        print('')
```

9.4 Valores para obtener el índice de la tabla de muestras de la onda senoidal

```
#include <iostream>
#include <stdint.h>

using namespace std;

uint32_t getIndex(uint16_t sample_cycle) {
    return 16384 / sample_cycle; // 16384 es el tamaño de la tabla de valores * 4
}

int main() {
    for (uint16_t n = 3; n < 5392; n++) { // 5391 son los ciclos de la frecuencia más baja
        cout << getIndex(n) << ",";
        if (n < 1650) {
            if ((n - 3) % 25 == 0)
                cout << endl;
        } else {
            if ((n - 3) % 50 == 0)
                cout << endl;
        }
    }
    return 0;
}
```

9.5 Aplicación simple para la recepción y transmisión de datos con UART en Windows

```
#include <Windows.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
   HANDLE hComm;
   BOOL Status;
   DCB dcbSerialParams = { 0 };
   COMMTIMEOUTS timeouts = { 0 };
   char SerialBuffer[75] = { 0 };
   DWORD BytesWritten = 0;
   DWORD dwEventMask;
   char ReadData;
   DWORD NoBytesRead;
   unsigned char loop = 0;
   char ComPortName[] = "\\\\.\\COM4";

   hComm = CreateFile(ComPortName, GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, 0, NULL);
   if (hComm == INVALID_HANDLE_VALUE) {
      printf("\n Port can't be opened\n\n");
      return 0;
   }

   //Setting the Parameters for the SerialPort
   dcbSerialParams.DCBlength = sizeof(dcbSerialParams);
   Status = GetCommState(hComm, &dcbSerialParams);
   if (Status == FALSE) {
      printf("\nError to Get the Com state\n\n");
      CloseHandle(hComm);
      return 0;
   }

   dcbSerialParams.BaudRate = CBR_115200;
   dcbSerialParams.ByteSize = 8;
   dcbSerialParams.StopBits = ONESTOPBIT;
   dcbSerialParams.Parity = NOPARITY;
   Status = SetCommState(hComm, &dcbSerialParams);
   if (Status == FALSE) {
      printf("\nError to Setting DCB Structure\n\n");
      CloseHandle(hComm);
      return 0;
   }
```

```
    // Setting Timeouts
    timeouts.ReadIntervalTimeout = 50;
    timeouts.ReadTotalTimeoutConstant = 50;
    timeouts.ReadTotalTimeoutMultiplier = 10;
    timeouts.WriteTotalTimeoutConstant = 50;
    timeouts.WriteTotalTimeoutMultiplier = 10;
    if (SetCommTimeouts(hComm, &timeouts) == FALSE) {
        printf("\nError to Setting Time outs");
        CloseHandle(hComm);
        return 0;
    }

    unsigned char synchronize = 0x01; // indicamos con un byte a uno que hay que sincronizar
    Status = WriteFile(hComm, &synchronize, 1, &BytesWritten, NULL);
    if (Status == FALSE) {
        printf("\nFail to Written");
        CloseHandle(hComm);
        return 0;
    }

    printf("\nNumber of bytes written to the serial port = %d\n\n", BytesWritten);

    Status = SetCommMask(hComm, EV_RXCHAR);
    if (Status == FALSE) {
        printf("\nError to in Setting CommMask\n\n");
        CloseHandle(hComm);
        return 0;
    }

    Status = WaitCommEvent(hComm, &dwEventMask, NULL);
    if (Status == FALSE) {
        printf("\nError! in Setting WaitCommEvent()\n\n");
        CloseHandle(hComm);
        return 0;
    }

    do { // recibimos todos los datos que la FPGA ha preparado en Synchronize
        Status = ReadFile(hComm, &ReadData, sizeof(ReadData), &NoBytesRead, NULL);
        SerialBuffer[loop] = ReadData;
        ++loop;
    } while (NoBytesRead > 0);
    --loop;

    printf("Number of bytes received = %d\n", loop);

    for (int index = 0; index < loop; ++index) {
        printf("Receiving: 0x%02X - %d\n", SerialBuffer[index], SerialBuffer[index]);
    }

    CloseHandle(hComm);
    return 0;
}
```

9.6 Ciclos de espera para el efecto portamento

```
#include <iostream>
#include <stdint.h>
#include <math.h>
#include <vector>

using namespace std;

double getFreq(uint8_t midiNote) {
    const double A4_FREQ = 440;
    const int32_t A4_MIDI_NOTE = 69;
    return A4_FREQ * pow(2.0, ((double) (((int32_t) midiNote) - A4_MIDI_NOTE)) / 12.0);
}

int main() {
    vector<int> ciclos;
    vector<int> tabla(16384, 0);
    for (uint16_t n = 0; n < 128; n++) {
        //cout << 44100 / getFreq(n) << endl;
        ciclos.push_back((int)(44100 / getFreq(n)));
    }

    for (int i = 0; i < ciclos.size(); i++) {
        for (int j = 0; j < ciclos.size(); j++) {
            if (abs(ciclos[i] - ciclos[j]) > 0) {
                //cout << "De la nota " << i << " (" << ciclos[i] << ") a la nota " << j << " (" << ciclos[j] << ")
con diferencia de ciclos = "
                //<< abs(ciclos[i] - ciclos[j]) << ": hay que esperar " << 44100 / abs(ciclos[i] - ciclos[j]) <<
" ciclos" << endl;
                tabla[abs(ciclos[i] - ciclos[j])] = (44100/8) / abs(ciclos[i] - ciclos[j]);
                //tabla[abs(ciclos[i] - ciclos[j])] = 44100 / abs(ciclos[i] - ciclos[j]);
            }
        }
    }

    for (int i = 0; i < 5391; i++) { // la diferencia más grande es de 5390
        //cout << i << ": " << tabla[i] << endl;
        cout << tabla[i] << ",";
        if (i % 50 == 0) cout << endl;
    }

    return 0;
}
```

9.7 Cálculo de los valores de incremento para una LFO

```
#include <iostream>
#include <iomanip>
#include <stdint.h>
#include <math.h>

using namespace std;

double getFreq(uint8_t midiNote) {
    const double A4_FREQ = 440;
    const int32_t A4_MIDI_NOTE = 69;
    return A4_FREQ * pow(2.0, ((double) (((int32_t) midiNote) - A4_MIDI_NOTE)) / 12.0);
}

int main() {
    double freq_inc = 20 / (double)128; // 20 Hz distribuidos en 128 notas
    double freq = freq_inc;
    for (uint16_t n = 0; n < 128; n++) {
        int cycles = 44100 / freq;
        double inc = 1 / (double)cycles;
        cout << int(inc * pow(2,24)) << ",";
        freq += freq_inc;
    }
    return 0;
}
```

9.8 Cálculo de los valores de F1 y Q1 para los filtros

```
import numpy as np
import math

def fc(frequency):
  F1 = 2 * math.sin(((math.pi * frequency) / 44100))
  return F1

# cálculo de F1, con frecuencia de corte desde 0 Hz hasta 8 kHz
values = ""
cut_freq = 0
for i in range (8000): # hasta 8 kHz
  freq = fc(cut_freq) * pow(2,16)
  values += str(int(freq)) + ", "
  cut_freq += 1 # sumamos 1 Hz
  if i % 10 == 0:
    values += "\n"

print(values)
```

```
# cálculo de Q1, con resonancia desde 0.8 hasta 17.09
resonance = 0.8

values = ""
for i in range (101):
  Q1 = 1/resonance
  values += str(int(Q1 * pow(2, 16))) + ", "

  if resonance >= 10:
    resonance += 1.0
  else:
    resonance += 0.1

  if i % 10 == 0:
    values += "\n"

print(values)
```

9.9 Identificadores de los parámetros en valor hexadecimal

Decimal	Hexadecimal	Parámetro
1	0x01	sincronizar
2	0x02	escala--
3	0x03	escala++
4	0x04	note off
5	0x05	master volumen
6	0x06	attack
7	0x07	decay
8	0x08	sustain
9	0x09	release
10	0x0A	attack mult
11	0x0B	decay mult
12	0x0C	release mult
13	0x0D	filtro freq_b0
14	0x0E	filtro Q
15	0x0F	filtro lfo
16	0x10	filtro tipo
17	0x11	lfo1 tipo
18	0x12	lfo1 freq
19	0x13	lfo2 tipo
20	0x14	lfo2 freq
21	0x15	noise volumen
22	0x16	pmod input
23	0x17	osc1 tipo
24	0x18	osc1 volumen
25	0x19	osc1 semitono
26	0x1A	osc1 detune
27	0x1B	osc1 lfo
28	0x1C	osc2 tipo
29	0x1D	osc2 volumen
30	0x1E	osc2 semitono
31	0x1F	osc2 detune
32	0x20	osc2 lfo
33	0x21	portamento
34	0x22	lfo1 modo
35	0x23	lfo2 modo
36 – 51	0x24 – 0x33	teclado software
52	0x34	osc1 pulse
53	0x35	osc2 pulse
54	0x36	lfo3 tipo
55	0x37	lfo3 freq
56	0x38	lfo3 modo
57	0x39	lfo3 destino
128 – 255	0x80 – 0xFF	teclado hardware

9.10 Ejemplo de fichero presets en xml

```
<?xml version="1.0" encoding="UTF-8"?>

<DEMO_TABLE_DATA>
 <COLUMNS>
  <COLUMN columnId="1" name="ID" width="50"/>
  <COLUMN columnId="2" name="Preset" width="160"/>
  <COLUMN columnId="3" name="Category" width="110"/>
  <COLUMN columnId="4" name="Autor" width="160"/>
  <COLUMN columnId="5" name="Rating" width="120"/>
 </COLUMNS>
 <DATA>
  <ITEM ID="001" Preset="PRESET001" Category="Sounds" Autor="Rafa" Rating="1"
      v1="0x17" v2="0x00" v3="0x19" v4="0x02" v5="0x1A" v6="0x32" v7="0x1B"
      v8="0x09" v9="0x34" v10="0x32" v11="0x18" v12="0x64" v13="0x1C"
      v14="0x01" v15="0x1E" v16="0x02" v17="0x1F" v18="0x32" v19="0x20"
      v20="0x09" v21="0x35" v22="0x32" v23="0x1D" v24="0x64" v25="0x06"
      v26="0x00" v27="0x0A" v28="0x05" v29="0x07" v30="0x32" v31="0x0B"
      v32="0x05" v33="0x08" v34="0x64" v35="0x09" v36="0x01" v37="0x0C"
      v38="0x05" v39="0x11" v40="0x01" v41="0x12" v42="0x66" v43="0x00"
      v44="0x00" v45="0x00" v46="0x02" v47="0x60" v48="0x00" v49="0x23"
      v50="0x00" v51="0x36" v52="0x0a" v53="0x00" v54="0x16" v55="0x00"
      v56="0x02" v57="0x00" v58="0x00" v59="0x64" v60="0x00" v61="0x40"
      v62="0x1f" v63="0x0D" v64="0x00" v65="0x0E" v66="0x00" v67="0x0F"
      v68="0x00" v69="0x10" v70="0x00" v71="0x15" v72="0x00" v73="0x16"
      v74="0x00"/>
 </DATA>
</DEMO_TABLE_DATA>
```

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**			X	
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**		X		
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

El objetivo que podemos relacionar en mayor grado con este proyecto es el ODS 12 “Producción y consumo responsables”. El sintetizador está basado en un dispositivo hardware reprogramable, lo que nos permite modificar o añadir funcionalidades sin necesidad de crear uno nuevo, de forma que podemos reciclarlo sin tener que utilizar *harware* nuevo. Esto incluye por ejemplo arreglar los fallos de programación o incluir ampliaciones y extensiones. En consecuencia, esto nos permite enfrentarnos a la “obsolescencia programada”, alargar la vida del producto y también evitamos producir desperdicios. Además estos dispositivos tienen un consumo energético bajo, entonces se puede adaptar el diseño para que el sintetizador funcione de forma independiente, es decir sin el uso de un computador, de manera que fomentaríamos un consumo energético más responsable.

Otro objetivo que puede contribuir, aunque en menor medida, es el ODS 9 “Industria, innovación e infraestructuras”. Como el diseño que hemos implementado sigue una filosofía de código abierto, estamos facilitando que otras personas puedan continuar con el proyecto a partir de donde lo hemos dejado, para adaptarlo a sus necesidades o innovar con un planteamiento de nuevas funcionalidades, entre otras. Esto no sería posible si tuvieran que comenzar el diseño desde cero, en lugar de partir de un diseño que es funcional y está verificado.