Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Diseño e implementación de un sintetizador de audio en un dispositivo Zynq-7000 SoC

Autor: Antonio Gómez Navarro
Tutor: Hipólito Guzmán Miranda

Dpto. de Ingenieria Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Diseño e implementación de un sintetizador de audio en un dispositivo Zynq-7000 SoC

Autor:
Antonio Gómez Navarro

Tutor:
Hipolito Guzmán Miranda
Profesor titular

Dpto. de Ingenieria Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Trabajo Fin de Grado: Diseño e implementación de un sintetizador de audio en un dispositivo Zynq-7000 SoC

Autor: Antonio Gómez Navarro

Tutor: Hipólito Guzmán Miranda

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

Agradecimientos

En primer lugar agradecer a mi familia el apoyo y la confianza. En segundo lugar, agradecer a todos mis amigos, en especial a mis compañeros de piso, el haber estado a mi lado durante tanto tiempo. Por último agradezco a mi tutor su atención y amabilidad.

Antonio Gómez Navarro
Sevilla, 2021

Resumen

El mundo del audio digital ha sufrido un gran avance durante los últimos años promovido por la aparición de nuevas tecnologías y el aumento de la capacidad de procesamiento de los dispositivos electrónicos.

La idea de este proyecto surge de la motivación por explorar la implementación de técnicas de síntesis digital en dispositivos modernos APSoC (*All Programmable System-on-Chip*). Para ello se realizará el diseño y la implementación de Zynth, un sintetizador polifónico y parametrizable basado en la serie Zynq de Xilinx.

Abstract

The world of digital audio has undergone a great advance during the last years promoted by the appearance of new technologies and the increase of the processing capacity of electronic devices.

The idea of this project arises from the motivation to explore the implementation of digital synthesis techniques in modern APSoC (All Programmable System-on-Chip) devices. For this purpose, the design and implementation of Zynth, a polyphonic and parametrizable synthesizer based on Xilinx's Zynq series, will be made.

Índice Abreviado

Índice

1 Introducción

Mi sintetizador modular tiene la capacidad de crear más de siete millones de combinaciones sonoras distintas. Para reproducirlas todas, el ser humano debería escucharlas de manera ininterrumpida durante doscientos diez años.

Robert Moog, 1974

Es bien sabida la importancia que tuvo la electrónica en el s. XX, cuyo crecimiento fue encontrando multitud de aplicaciones en el mundo contemporáneo. La capacidad de construir mecanismos lógicos que funcionen a partir de electricidad ha sido fundamental para engendrar año tras año nuevas generaciones de sistemas, cada vez más potentes e inteligentes. Estos avances han propiciado un cambio significativo en muchos aspectos de la sociedad, y la música no iba a ser una excepción.

La llegada de la música electrónica supuso toda una revolución en la manera en la que los músicos podían interactuar con los instrumentos e interpretar sus ideas mediante el sonido. Los primeros instrumentos electrónicos estaban construidos usando circuitos analógicos; con el paso del tiempo, el auge de la electrónica digital —propiciado por la invención del transistor, del circuito integrado y, más adelante, del microprocesador— cambió por completo la forma en la que podíamos "jugar" con el sonido y trajo consigo multitud de nuevas oportunidades [1-3].

Esta revolución en la música continua en la actualidad, ya que el progreso de la tecnología y el desarrollo de chips modernos con mayor capacidad de procesamiento nos ofrece cada vez más posibilidades. La idea de este proyecto nace de la motivación por explorar nuevas formas de generación de sonido mediante síntesis digital, usando para ello tecnologías modernas muy prometedoras como son los dispositivos SoC (*System-on-Chip*). En este capítulo haremos primero una introducción a los orígenes de la producción musical electrónica y luego definiremos el planteamiento del proyecto.

1.1 Breve historia de los sintetizadores

1.1.1 Orígenes

El primer sintetizador de música eléctrico fue creado por Elisha Gray (coinventor del teléfono) en 1876. Elisha descubrió que podía controlar circuitos electromagnéticos para crear osciladores básicos de una nota. Estos osciladores eran capaces de producir tonos sinusoidales a partir de la vibración producida a la frecuencia de resonancia del circuito. Este concepto fue desarrollándose hasta llegar a lo que se conoce comúnmente como Oscilador Controlado por Tensión, o VCO[1], en el cual la frecuencia de oscilación es proporcional al voltaje de entrada del circuito. Estos osciladores, lejos de ser perfectos[2], sirvieron para la creación de la primera generación de sintetizadores. Filtros analógicos, amplificadores de tubos de vacío y otros circuitos fueron usados para modificar las formas de onda generadas, permitiendo la variación del timbre musical del instrumento.

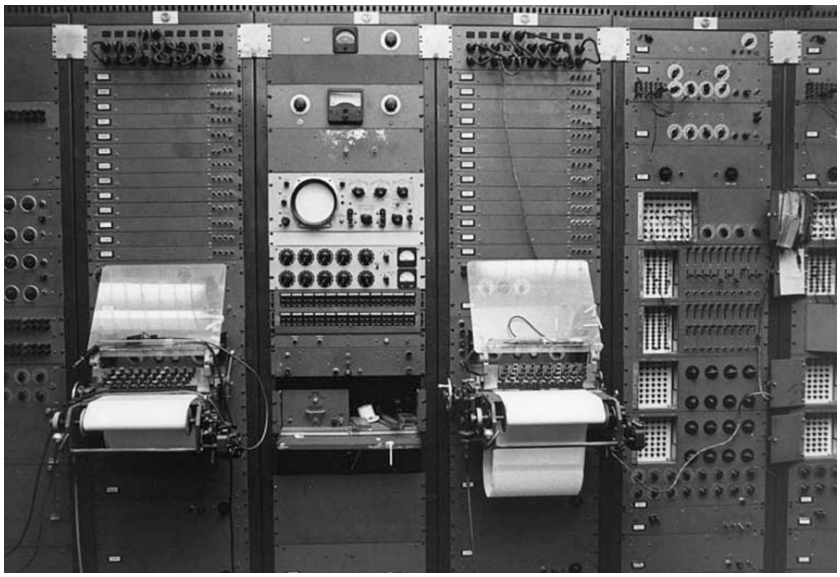

Figura 1.1 Terminales de papel perforado del RCA Mark I (Fuente: [4]).

Un ejemplo de sintetizadores pioneros fueron los Mark I y II de *Radio Corporation of America* (RCA) [4]. En los años 50, RCA era uno de los mayores conglomerados de entretenimiento en EEUU. Con la invención de los Mark buscaban reducir costes en las sesiones de grabación mediante la automatización de los arreglos musicales, ahorrándose de esta manera el coste de contratar una orquesta. La máquina resultante fue una monstruosa colección de componentes modulares que ocupaban una habitación entera en la Universidad de Columbia. El "instrumento" era como un ordenador analógico de la época, cuya única entrada de datos era una máquina de escribir donde se escribían las partituras en código binario en forma de agujeros en un papel (véase la figura 1.1). Esto último lo convertía en el primer sintetizador electrónico programable. El sonido era generado por una serie de osciladores de tubos de vacío que ofrecían una polifonía de cuatro voces. El sonido generado

[1] Del inglés, Voltage Controlled Oscillator.

[2] Si bien su diseño y componentes empleados fueron perfeccionándose con el tiempo, estos se veían afectados en mayor o menor medida por las no-linealidades propias de los circuitos eléctricos, de manera que incluso una variación en la temperatura podía modificar la frecuencia de oscilación.

era redirigido manualmente a los distintos componentes, una técnica que sería adoptada en los posteriores sintetizadores modulares. Cada módulo es una etapa de procesamiento del sonido, es el usuario el que decide como interconectar dichos módulos para producir un sonido característico.

1.1.2 Años 60-80

La llegada de los circuitos integrados de estado sólido, más baratos y confiables que los de válvulas de vacío, hizo que los sintetizadores de RCA y similares se volvieran obsoletos. Una de las figuras más importantes de esta nueva generación de sintetizadores analógicos fue la de Robert Moog [5]. Moog estuvo inspirado por la idea de controlar la tensión de los VCO para variar el tono de los osciladores (hasta entonces, los sintetizadores empleaban osciladores con frecuencias fijas asociadas a cada nota). De esta manera, fue creando circuitos controlados por tensión hasta que creó el primer sintetizador Moog modular.

Figura 1.2 El primer prototipo vendido de un sintetizador Moog en 1964 (Fuente: wikipedia.es).

Los diseños de Moog ganaron mucha popularidad e hizo que más músicos se interesaran por este tipo de instrumentos. No obstante, aún siendo más pequeño que sus predecesores, el público pedía sintetizadores más pequeños todavía y fáciles de transportar. Esto trajo consigo la llegada del Minimoog, a finales de 1970, considerado como el sintetizador más clásico de la historia [6]. Un poco antes de él salió el VCS3, de EMS (Electronic Music Studios), que fue el primer sintetizador portátil [7].

Figura 1.3 Minimoog, 1970 (Fuente: vintagesynth.com).

Pronto muchas empresas —como Roland, Yamaha y Sequential Circuits— vieron un mercado prometedor en este tipo de instrumentos y se lanzaron a fabricar sus propios modelos. A su vez, artistas de renombre empezaron a utilizar sintetizadores en sus interpretaciones, popularizándolos aún más.

1.1.3 Años 80-actualidad

Los primeros microprocesadores aparecieron en la década de los 70 y marcaron el inicio de la era digital. Para finales de esta década comenzaron a aparecer los primeros sintetizadores digitales controlados por microprocesadores. Estos sistemas buscaban recrear los sonidos característicos de los sintetizadores analógicos, con diseños mucho más compactos. Los primeros que surgieron se trataban de sintetizadores híbridos. Uno de los más famosos de esta época fue el Prophet-5 de Sequential Circuits, fabricado en 1978, que fue un sintetizador analógico que tenía memoria para guardar las configuraciones del usuario [8].

Figura 1.4 Prophet-5, 1978 (Fuente: vintagesynth.com).

Otro ejemplo de sintetizador híbrido fue el Roland Juno-60, el cual usaba Osciladores Controlados Digitalmente (DCO) en vez de los análogos VCO. Los DCOs, que eran osciladores análogicos controlados digitalmente, aseguraban la estabilidad de la afinación a diferencia de los VCO. El Juno-60 fue el primer sintetizador analógico (todo menos el control de los osciladores era analógico) que se podía llevar a un escenario, encenderlo y tocar con total confianza de que el instrumento estaría afinado [9].

Figura 1.5 Roland Juno-60, 1982 (Fuente: vintagesynth.com).

El DX7 de Yamaha, lanzado en 1983, se considera el primer sintetizador verdaderamente digital [10]. Utiliza un tipo de síntesis desarrollada por el profesor John Chowning en la Universidad de Stanford en la década de 1970, que Yamaha denominó "Modulación de Frecuencia" (FM) y que consiste en variar la frecuencia de una señal (denominada

portadora) con respecto a una segunda (denominada moduladora), generando finalmente una onda modulada en frecuencia [11].

Figura 1.6 Yamaha DX7, 1983 (Fuente: vintagesynth.com).

En los años 80 la popularidad de los sintetizadores solo fue en aumento, ya que empezaron a salir modelos con precios más asequibles debidos, no solo al avance tecnológico, sino a la irrupción en el mercado de nuevos competidores y a la economía de producción a escala. Por esta época también aparece el protocolo MIDI, el cual es un estándar que permite la comunicación entre sistemas de audio de distintos fabricantes.

Los primeros sistemas digitales estaban muy limitados por la velocidad y la memoria disponible de los microprocesadores. Los siguientes años estuvieron marcados por un crecimiento exponencial del procesamiento de los microprocesadores, lo que ha permitido la creación de instrumentos cada vez más complejos, compactos y baratos.

1.2 Planteamiento del proyecto

La idea de este proyecto surge de la motivación por explorar la implementación de técnicas de síntesis digital en dispositivos modernos *System-on-Chip*. Para ello usaré la serie Zynq de Xilinx para crear lo que he bautizado como Zynth (Zynq + Synth). En el capítulo 2 explicaremos en detalle en que consiste este dispositivo, de momento adelantaremos que se compone de un procesador ARM de propósito general y una FPGA. La naturaleza de procesamiento en paralelo de la FPGA la convierte en una plataforma muy atractiva para la generación de tonos polifónicos. El sistema puede beneficiarse del uso de la FPGA en el procesamiento de audio para disminuir la latencia, la cual es un factor decisivo en el diseño de cualquier instrumento digital.

Los instrumentos digitales son considerados sistemas de tiempo real estricto (hard real-time): es absolutamente necesario que la respuesta se produzca dentro del límite especificado. La latencia[a] límite se sitúa en torno a los 20ms, a partir del cual se degrada considerablemente la sincronización háptico-acústica.

[a] Entendemos como latencia el tiempo transcurrido desde que se pulsa una tecla hasta que se produce un sonido.

1.2.1 Requisitos

El **objetivo** del proyecto es construir un sintetizador polifónico que reciba comandos MIDI por puerto serie y que pueda ser configurado mediante una interfaz humano-máquina.

Requisitos funcionales

- F.1: Generación de ondas periódicas.
  - F.1.1: Generación de onda senoidal.
  - F.1.2: Generación de onda cuadrada.
  - F.1.3: Generación de onda triangular.
  - F.1.4: Generación de onda diente de sierra.
- F.2: Ondas configurables en amplitud.
- F.3: Ondas configurables en frecuencia.
- F.4: Generación de armónicos.
  - F.4.1: Generación del 2do armónico.
  - F.4.2: Generación del 4to armónico.
  - F.4.3: Tipo de onda (senoidal, cuadrada...) de los armónicos configurable.
  - F.4.4: Armónicos configurables en amplitud.
- F.5: Modulación en amplitud mediante un oscilador de baja frecuencia (LFO).
  - F.5.1: LFO configurable en amplitud.
  - F.5.2: LFO configurable en frecuencia.
  - F.5.3: Tipo de onda (senoidal, cuadrada...) del LFO configurable.
- F.6: Generación de una envolvente acústica.
  - F.6.1: Tiempo de ataque configurable.
  - F.6.2: Tiempo de decaimiento configurable.
  - F.6.3: Tiempo de sostenimiento configurable.
  - F.6.4: Tiempo de extinción configurable.
- F.7: Mezcla (suma) de las notas generadas.
- F.8: Recepción de comandos MIDI.

Requisitos de prestaciones

- P.1.1: La profundidad del audio será de 16 bits.
- P.1.2: La frecuencia de muestreo será de 48kHz.
- P.2: Existirán 256 niveles de amplitud disponibles.
- P.3: El sistema debe poder generar ondas en el rango audible (entre 20Hz y 20kHz).
- P.4: Los armónicos generados tendrán las mismas prestaciones que las ondas fundamentales (P.1-3).

- P.5: El LFO tendrá las mismas prestaciones que las ondas fundamentales (P.1-3).
- P.6: Existirán 16 valores disponibles para los parámetros de configuración.
- P.7: La polifonía será de 8 voces.
- P.8.1: La recepción de comandos MIDI será mediante UART.
- P.8.2: Los comandos disponibles serán NOTEON y NOTEOFF.

Requisitos de operación

- O.1: Sintetizador configurable mediante una interfaz humano-máquina.
  - O.1.1: Interfaz gráfica mediante una pantalla LCD de 16x2 caracteres.
  - O.1.2: Control de la interfaz mediante un encoder rotativo incremental con botón.
- O.2: Salida de audio a través de auriculares.

1.2.2 Metodología

El proyecto de Zynth está regido además por una serie de valores que son los siguientes:

1. El proyecto será realizado de acuerdo a la filosofía de código libre. Todo el código estará disponible en un repositorio público alojado en Github[3].
2. Todo el proyecto será realizado desde cero, escribiendo cada módulo en bajo nivel (VHDL/C) y sin usar ninguna IP de terceros.
3. A lo largo del desarrollo del mismo se utilizarán una serie de buenas prácticas como el control de versiones, la verificación unitaria y el uso de *scripts* para automatizar procesos.

En los siguientes capítulos se hará una introducción teórica a los distintos aspectos que de los que se compone el proyecto, para finalmente explicar la implementación del mismo en el capítulo 5.

[3] *https://github.com/RTLogik/Zynth*

2 Plataforma hardware

En este capítulo hablaremos sobre el *All Programmable System-on-Chip* (APSoC) Zynq-7000 de Xilinx, el dispositivo físico donde se realizará la implementación del sintetizador digital. Describiremos sus características principales y que lo hace tan especial. También hablaremos de la PYNQ-Z2, que es la placa de evaluación de Zynq que utilizaremos.

2.1 Zynq-7000 SoC

La característica fundamental de los dispositivos Zynq es que combinan un procesador doble núcleo ARM Cortex-A9 con una FPGA de la serie 7 de Xilinx, más concretamente una Artix-7. Para abreviar se referirá de ahora en adelante a la parte del procesador como PS (del inglés *Processing System*) y a la parte de la FPGA como PL (del inglés *Programmable Logic*). Ambas partes están conectadas entré si mediante interfaces AXI, las cuales son un estándar en la industria y proporcionan comunicación *on-chip* con baja latencia y gran ancho de banda.

2.1.1 Processing System (PS)

El sistema de procesamiento de Zynq no consiste solo en un procesador ARM, sino que engloba a un conjunto de unidades de procesamiento que forman una *Application Processing Unit* (APU), interfaces externas de conexión a periféricos, interfaces de memoria, memoria cache, circuitos de generación de reloj y bloques para la interconexión de dichos elementos.

Application Processing Unit (APU)

La APU se compone principalmente de dos núcleos de procesamiento ARM, cada uno con unidades computacionales asociadas: un motor de procesamiento multimedia NEON y una unidad de punto flotante (FPU); una unidad de gestión de memoria (MMU); y una memoria caché de nivel 1 (en dos secciones para instrucciones y datos). La APU también contiene una memoria caché de nivel 2, y otra memoria en chip (OCM). Por último, una Unidad de Control de Snoop[1] (SCU) forma un puente entre los núcleos ARM y las memorias caché de nivel 2 y OCM.

[1] El "snooping" es uno de los varios mecanismos para garantizar la coherencia de la caché, es decir, para gestionar la consistencia de los datos en los distintos recursos compartidos de la caché.

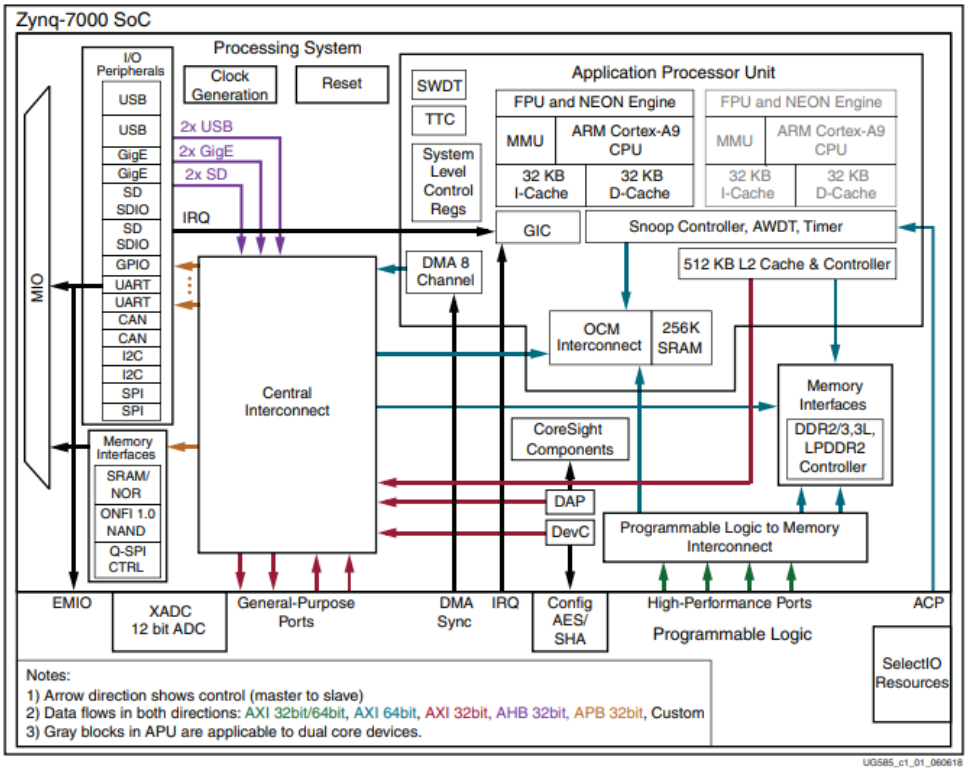

Figura 2.1 Diagrama funcional del PS (Fuente: [12].

Interfaces externas

La comunicación entre el PS y las interfaces externas se logra principalmente a través del MIO (*Multiplexed Input/Output*), que proporciona 54 pines de conectividad flexible, lo que significa que el mapeo entre periféricos y pines se puede definir según sea necesario. Ciertas conexiones también se pueden realizar a través del EMIO (*Extended MIO*), que pasa a través del PL y comparte sus recursos de E/S. El EMIO puede utilizarse cuando se requiere una extensión más allá de los 54 pines, o como método de interconexión del PS con un bloque periférico implementado en el PL.

2.1.2 Programmable Logic (PL)

El PL tiene la estructura general de una FPGA, basada en Artix-7 para la gama de bajo coste; y en Kintex-7 para la gama media-alta [13]. Los elementos básicos de una FPGA son:

- **Configurable Logic Blocks (CLBs):** como su nombre indica, son pequeños bloques que agrupan elementos de lógica que puede ser configurada por el usuario. Están conectados entre ellos y con otros elementos mediante interconexiones programables. Cada CLB contiene dos slices.
- **Slices:** contiene los recursos necesarios para implementar circuitos combinacionales y secuenciales. Estan compuestos de LUTs y Flip-Flops.
- **Look-Up Table (LUT):** capaces de implementar funciones lógicas, registros y memorias ROM o RAM.

- **Flip-Flop:** o biestable, es un circuito con dos estados posibles y capaz de almacenar información. Es la unidad básica de memoria equivalente a un registro de 1 bit.
- **Matrices de interconexión:** están situadas junto a cada CLB y permiten el rutado de señales entre los distintos elementos de la FPGA.
- **Input/Output Blocks (IOBs):** proporcionan una interfaz con el exterior permitiendo la conexión de los recursos de la FPGA a "pads" que están directamente conectados a pines físicos del chip.

Junto con estos elementos de propósito general, cada modelo de FPGA puede incluir recursos especiales. Para el caso de Zynq, estos recursos adicionales son:

- **Block RAM (BRAM):** se trata de bloques dedicados de memoria, más optimizados y compactos que la alternativa de utilizar memoria distribuida implementada en LUTs. Cada bloque BRAM puede almacenar hasta 36kB, pudiendose configurar el ancho de palabra.
- **DSP48E1:** es un elemento especializado en la implementación de aritmética de alta velocidad con palabras de mediano-gran tamaño. Es un recurso muy útil para aplicaciones de procesamiento de la señal donde se requiera de cálculos intensivos (multiplicaciones sobre todo) sobre un flujo de datos.
- **XADC:** conversores análogico-digital de 12 bits capaces de muestrear señales analógicas a 1Msps.

Deberemos aprovechar estos recursos especializados y utilizarlos en nuestro diseño siempre que podamos. Para una descripción detallada del funcionamiento de cada componente debemos acudir a [12].

2.2 Ventajas de los SoC

Los SoC presentan varias características que los convierten en una solución bastante atractiva dentro del mercado de los sistemas embebidos. Una de las ventajas de Zynq es el acoplamiento de un procesador y de la lógica programable en un mismo dispositivo físico.

Al juntar lógica programable con microprocesadores de propósito general en un mismo chip obtenemos los mejor de los dos mundos: la flexibilidad del software con el procesamiento inherentemente paralelo de una FPGA. Los enlaces AXI de gran rendimiento entre el PS y el PL permiten explotar al máximo las diferentes propiedades de estos dos recursos. Esto se debe a la reducción de la sobrecarga de comunicación entre ellos en comparación con una alternativa de componentes discretos.

A continuación se enumeran algunas ventajas de los sistemas basados en SoC:

1. Menor gasto de área; al estar todo integrado en un mismo chip se reduce el área ocupada en una placa de circuito impreso.
2. Menor consumo de energía que una implementación equivalente con componentes discretos.
3. Mayor velocidad de comunicación, ya que todas las comunicaciones (entre el procesador y la FPGA) se realizan dentro del chip entre partes que están físicamente muy próximas.

4. El desempeño de un procesador *hard-core* (esto es, implementado en silicio) es mucho mayor que su equivalente *soft-core* (implementado en la lógica programable de la FPGA).

2.3 PYNQ-Z2

PYNQ es un proyecto de código abierto de Xilinx consistente en una imagen de arranque de Linux que incluye librerías de Python que permiten programar fácilmente dispositivos Zynq en alto nivel haciendo uso del framework Jupyter. El objetivo de PYNQ es que ingenieros de software sin conocimientos de lenguajes de descripción de hardware puedan hacer uso de las capacidades que ofrece Zynq [14].

Existen varias placas compatibles con PYNQ, la que usaremos en este proyecto sera la PYNQ-Z2 de TUL. Aunque dicha placa forme parte del proyecto PYNQ, no es obligatorio usar la imagen de PYNQ en ella. De hecho, en este proyecto se usará la PYNQ-Z2 como una placa de evaluación de Zynq sin hacer uso de la capa de abstracción de Python, programando el PL usando VHDL y el PS usando C *bare-metal* (esto último significa que no haré uso de ningún sistema operativo).

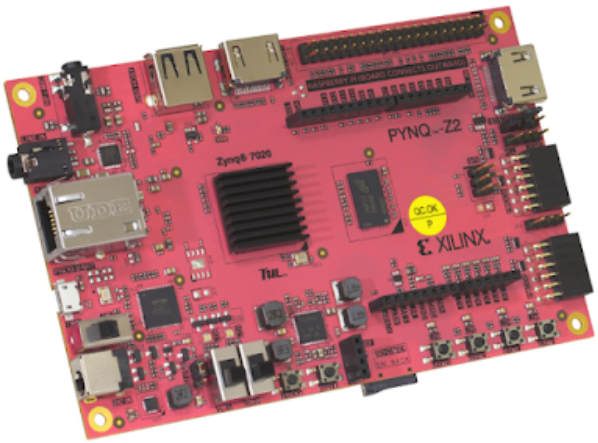

Figura 2.2 PYNQ-Z2 (Fuente: [15]).

Esta placa cuenta con una gran selección de periféricos: memoria RAM DDR3 de 512MB, MicroSD, conectores HDMI de E/S, USB host y Ethernet, multitud de pines GPIO, botones, leds, interruptores... [15]. Realmente los únicos periféricos que utilizaremos en este proyecto serán los pines GPIO, el codec de audio y el jack de salida para auriculares (también hay un jack para entrada de linea).

2.3.1 Codec de audio

Un codec de audio es un dispositivo que transforma señales de audio analógicas a señales digitales y viceversa. A diferencia de un conversor genérico, los conversores de audio están especializados en trabajar a bajas frecuencias (recordemos que la máxima frecuencia sonora audible esta a 20kHz) y con mucha resolución. Además de esta conversión, un codec de audio se caracteriza por incluir funcionalidades especiales dedicadas al tratamiento de audio, como un DSP integrado que permita tratar las señales, control de volumen, mezcladora de canales, circuito de *bias* para la conexión de un micrófono de condensador y entradas/salidas adaptadas para señales de linea o micrófono tanto mono como estéreo.

El codec de audio presente en la PYNQ-Z2 es el modelo ADAU1761 de Analog Devices. El ADAU1761 es un códec de audio estéreo de baja potencia con procesamiento de audio digital integrado que admite grabación y reproducción estéreo de 48 kHz a 14 mW desde una fuente analógica de 1.8 V. Los ADC y DAC de audio estéreo admiten frecuencias de muestreo de 8 kHz a 96 kHz, así como un control de volumen digital [16].

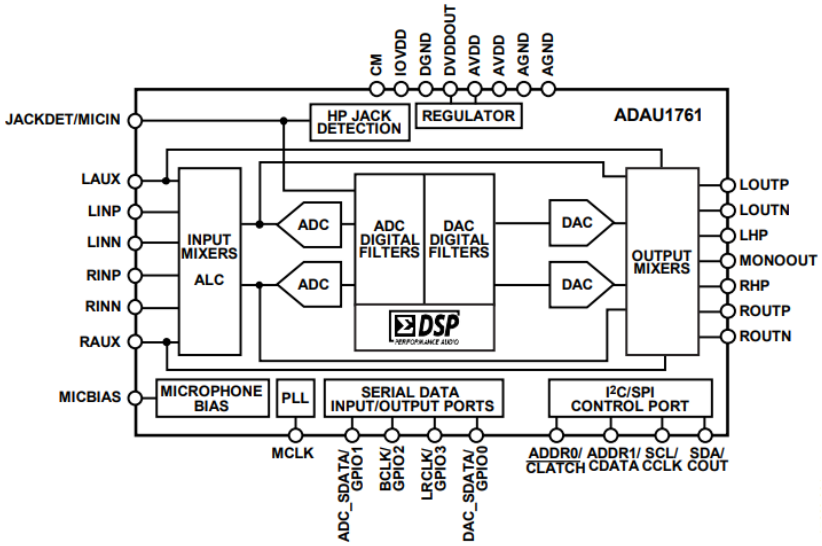

Figura 2.3 Diagrama de bloques funcional del ADAU1761 (Fuente: [16]).

El codec consta de 66 registros internos que deberemos modificar para configurarlo a nuestra medida. Esto se hará mediante el puerto de control, que permitirá la lectura/escritura de cada registro interno mediante el protocolo I2C (también es posible mediante SPI). El PS será el responsable de realizar la modificación de los registros del codec para su configuración.

Por otro lado, el flujo de datos de audio de entrada/salida se realiza mediante el puerto serie utilizando el protocolo de comunicación I2S, el cual es una variación de I2C para señales estereo de audio. La señal de audio estará en formato de complemento a 2 y codificada mediante modulación por impulsos codificados (o PCM, *Pulse Code Modulation*).

En el capitulo 5 hablaremos en más detalle sobre los protocolos I2C e I2S y sobre cómo se realiza la comunicación con el codec.

3 Metodología de codiseño HW/SW

Una de las primeras cosas que debería hacerse cuando se inicia un proyecto de mediana complejidad es establecer cual va a ser el flujo de diseño y una serie de buenas prácticas que te acompañen a lo largo del desarrollo del mismo. Conocer bien los procesos a seguir y las herramientas disponibles te ayudará a ser más productivo y a tener mayor control de la calidad.

Dada la cantidad de horas que he empleado estudiando y perfeccionando mis pautas de diseño en este proyecto y debido a que este tipo de co-diseños software/hardware no se enseñan en la carrera, he creído conveniente dedicar un capitulo explicando lo que he aprendido al respecto durante todos estos meses. Quizá este texto le resulte útil a alguien que quiera abordar un proyecto de este tipo y no sepa por donde empezar.

Este capítulo no pretende ser una guía detallada de todos los elementos que se mostrarán a continuación, sino mas bien una introducción a los aspectos generales que se deberían de tener en cuenta.

3.1 Aspectos básicos en el diseño de sistemas basados en Zynq

Generalmente, en cualquier proyecto de diseño electrónico, la primera fase consiste en definir el comportamiento deseado del sistema. Se trata de describir el sistema a nivel funcional y crear un conjunto de especificaciones a partir de una serie de requisitos.

La siguiente fase de diseño del sistema será realizar una apropiada partición de las funcionalidades entre software y hardware y definir las interfaces entre ambas secciones. Es posible que la partición que realicemos en primera instancia deba ser ajustada o modificada más adelante en el proyecto.

Una vez particionado el sistema, el desarrollo de hardware y de software puede ser realizado en paralelo, hasta cierto punto. A medida que avancemos en el desarrollo la verificación de software, de hardware y de ambos a la vez se convierte en una parte fundamental del proceso de diseño.

3.1.1 Aproximación de arriba a abajo

El diseño de la arquitectura de un sistema suele ser realizado mediante una aproximación de arriba a abajo. Esto quiere decir que empezamos definiendo la interfaz y funcionalidad del sistema de más alto nivel para luego identificar los subsistemas o las tareas que lo componen,

así como las interacciones entre dichos subsistemas. Al principio no es necesario especificar detalles técnicos acerca de cada subsistema, solamente su funcionalidad y su conexión con el exterior (interfaz de entrada y salida) como si se tratara de un modelo de caja negra. Dependiendo de la complejidad del diseño, estos subsistemas pueden ser descompuestos a su vez en otros subsistemas, descendiendo cada vez más en el nivel de jerarquía. El resultado final es una representación abstracta de componentes y transacciones. Una vez definidos todos los componentes podemos trabajar en cada uno de ellos por separado para luego ir integrandolos progresivamente, validando cada componente antes de escalar al siguiente nivel de jerarquía.

Una de las ventajas de este método de aproximación al diseño de sistemas es que podemos enfocarnos en cada momento en el diseño de módulos bien definidos y de baja o media complejidad, pudiéndonos abstraer del sistema al completo y de su complejidad subyacente. Además, si el equipo esta compuesto por más de una persona, cada miembro del grupo puede dedicarse al diseño de un modulo distinto, aumentando la productividad. Como inconveniente cabe mencionar que a veces las descripciones de los modelos de caja negra que realizamos en primera instancia pueden ser demasiado vagas o irrealizables en la práctica. Para ello será necesaria la experiencia y el conocimiento de la tecnología empleada.

3.1.2 Partición del sistema

Una vez definidos los bloques fundamentales de los que se compone nuestro sistema, debemos decidir cuales de ellos deben ser implementados en hardware, cuales en software, y como va a ser la comunicación entre estos. Generalmente, el software (en el PS) será usado para implementar tareas generales de procesamiento que sean secuenciales, sistemas operativos, aplicaciones de usuario e interfaces gráficas (GUI); mientras que las partes del diseño que requieran un flujo de datos computacionalmente intensivo serán realizadas en el PL. Además, cualquier algoritmo de software en el que se pueda identificar un paralelismo significativo (problemas que se puedan dividir de manera eficiente en múltiples tareas paralelas) será un candidato para su implementación en el PL, correspondiéndose con un modelo de "coprocesador" en el que las tareas paralelas y computacionalmente intensivas pueden descargarse del procesador al hardware para lograr un aumento general del rendimiento.

Los componentes de hardware implementados en la lógica programable de la FPGA son, por norma general, mucho más rápidos debido a la característica de procesamiento paralelo de las FPGA. Sin embargo, estos componentes tienden a ser más caros ya que ocupan mayor área de silicio. Por otro lado, los componentes de software implementados en un procesador de propósito general son más baratos tanto para crear y mantener (más facilidad de diseño y por tanto menor coste temporal) como en términos de silicio “empleado” (en este caso nos referimos a la cantidad de memoria usada). Por contraparte, estos componentes son también mas lentos debido al procesamiento de tareas inherentemente secuencial.

La lógica programable de la FPGA es adecuada para la implementación de problemas que pueden ser eficientemente divididos en múltiples tareas paralelas. Estas tareas suelen ser repetitivas y estáticas por naturaleza, como la multiplicación de vectores en un algoritmo de procesamiento de imagen. Por otro lado, los problemas que sean más dinámicos e impredecibles serán mas adecuados para su implementación en software dada la facilidad

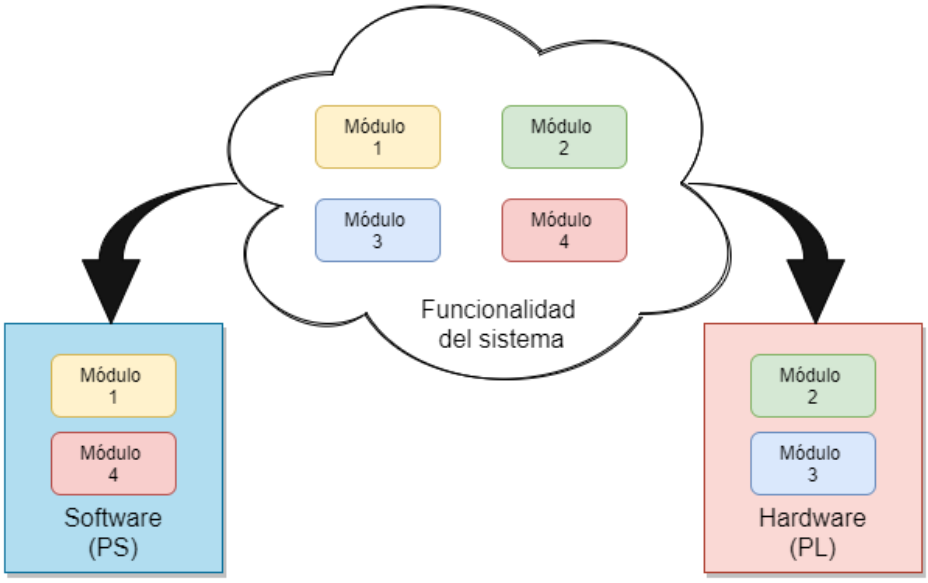

Figura 3.1 Partición software/hardware del sistema.

de ramificación condicional con estructuras como if-else y el uso de funciones[1].

Otro factor a considerar es el formato numérico que queremos usar para un determinado proceso. Por norma general, los procesadores presentan más soporte para cálculos con números en coma flotante, debido a las unidades dedicadas para coma flotante (FPUs, del ingles *Floating Point Units*) que acompañan en muchas ocasiones el procesador. Las FPGAs pueden soportar también cálculos de este tipo pero requieren gran cantidad de lógica para implementarlos. Sería adecuado implementar cálculos en coma flotante y de alta precisión en una FPGA de gran tamaño, donde el impacto de la huella en el silicio sea menor. Cabe destacar que hoy día muchas FPGAs cuentan con bloques DSP capaces de realizar operaciones de coma flotante, reduciendo así el uso de LUTs requeridas.

3.1.3 Bloques IP

Un bloque IP (*Intellectual Property*) es un módulo de hardware con una funcionalidad específica. La funcionalidad de estos bloques está garantizada ya que han sido previamente verificados, suponiendo un ahorro importante de tiempo al diseñador, el cual puede incorporar un modulo de este tipo a su diseño sabiendo que va a funcionar sin necesidad de testearlo. Solo será necesario comprender como es la interfaz del módulo para poder interactuar con él.

En términos de bloques IP, los hay de dos tipos:

- **Soft IP:** permiten al usuario final modificar el IP hasta cierto punto. El formato del IP puede presentarse como código fuente en HDL, donde el usuario deberá realizar la síntesis e implementación, o como una *netlist* a nivel de puertas lógicas, donde ya se ha realizado la síntesis y el usuario solo deberá realizar la implementación. En

[1] Aunque este tipo de estructuras están presentes tambíen en HDL, son menos versátiles y deben ser usadas con cautela, teniendo en cuenta siempre el circuito derivado de la síntesis.

este último caso el grado de personalización será menor. Un ejemplo de soft IP seria un procesador Microblaze implementado en lógica programable.

- **Hard IP:** en este caso el bloque IP ha pasado por el proceso de síntesis e implementación y no es posible para el usuario final realizar ningún tipo de modificación. Una vez el IP ha pasado por el proceso de *place and route*, este solo puede ser integrado en un dispositivo específico (en muchas ocasiones esta integración se realiza directamente en el silicio del chip durante el proceso de manufactura). Este tipo de IP presentan la ventaja de que son deterministas en términos de rendimiento y área de silicio requerida. Un ejemplo de hard IP seria el procesador de ARM presente en el chip de Zynq.

Existen tres métodos principales para obtener IPs:

1. Creados "a mano" a partir de código HDL o de alguna herramienta de alto nivel como Vitis HLS, Matlab HDL Coder o System Generator for DSP.
2. De las librerías ofrecidas por los vendedores de FPGAs junto con sus herramientas de desarrollo.
3. Obtenidos de proveedores de terceros, los cuales pueden ser comerciales (Synopsys, Rambus...) o open-source (GRLIB, OpenCores...).

El mayor inconveniente en crear nuestro propio IP en HDL es que se requiere de bastante experiencia de diseño para obtener una solución óptima, sobretodo en diseños complejos. El tiempo de desarrollo puede ser elevado y frustante. Sin embargo, si queremos tener el máximo control sobre el resultado, tenemos grandes restricciones de hardware, de reloj, o la funcionalidad de nuestro módulo es muy específica, escribir nuestros propios IP en HDL será la mejor opción.

Cuando creas un catálogo propio de IPs, ya sea para uso personal o comercial, es muy importante validar cada módulo y asegurarte de que funciona correctamente. Del mismo modo, es muy importante documentar cada módulo para que cualquier persona (incluido tú mismo dentro de un tiempo) sepa como funciona.

3.1.4 Toolchain

Se conoce como *toolchain* al conjunto de herramientas de software que nos va a permitir desarrollar, integrar e implementar nuestro diseño. En el ámbito de los sistemas embebidos estas herramientas suelen ser proporcionadas por los fabricantes de chips y son exclusivas para los productos de su catálogo. Para algunos chips existen también alternativas open-source. Xilinx se muestra reacia a liberalizar su tecnología de generación del *bitstream*, lo que nos obliga a utilizar su software propietario[2]. Las herramientas que nos proporciona Xilinx son Vivado, para el desarrollo de hardware; y Vitis, para el desarollo de software.

[2] Si bien existen fases del desarollo como la simulación o edición de ficheros de texto que podemos externalizar, en este trabajo usaremos exclusivamente las herramientas proporcionadas por Xilinx.

Vivado

Vivado es un entorno de desarrollo integrado (IDE) que nos permitirá editar código HDL, crear restricciones de diseño, sintetizar el código en circuitos a nivel de transferencia de registros, implementar dichos circuitos en el silicio de nuestro dispositivo (mediante un proceso conocido como *place and route*) y generar el *bitstream* con el que programaremos la FPGA. En cada paso del proceso podremos realizar análisis y generar reportes de rendimiento, consumo de energía, área, etcétera.

Algunas de las funcionalidades que ofrece Vivado son:

- **Simulator:** nos permite simular nuestro sistema y visualizar las formas de onda resultantes.
- **System Integrator:** permite a los diseñadores agregar IPs y conexionarlos de manera fácil y sencilla como si fuese un diagrama de bloques.
- **IP Packager:** permite empaquetar nuestros propios IPs para después poder añadirlos a nuestra biblioteca personal.

Vitis

Vitis es un IDE basado en Eclipse utilizado para el desarrollo de aplicaciones software donde podremos editar y compilar código en C/C++ y cargarlo en el dispositivo. El programa viene acompañado de librerías estándar y drivers y ofrece además funcionalidades de debugging y de creación de imágenes para la programación de la memoria flash. Vitis posee una extensión, Vitis HLS, que permite traducir código C, C++ o SystemC a lenguaje de descripción de hardware (HDL), acelerando el tiempo de desarrollo y "acercando" el diseño de hardware a ingenieros de software.

Existen dos tipos de proyectos en Vitis, *Platform Project* y *System Project*, más adelante cuando veamos el flujo de diseño detallaremos como trabajar con ambos tipos de proyectos.

3.1.5 Herramientas de validación

Conforme el diseño va aumentando en tamaño y complejidad, la validación se convierte en un paso fundamental para poder seguir construyendo nuestro sistema sobre unos cimientos que sabemos que son estables y seguros. Es importante detectar los errores en fases tempranas para poder disminuir el impacto que supondría su corrección. A continuación vamos a ver las herramientas de las que disponemos para la validación de Zynq:

Validación de hardware

La primera toma será siempre la simulación funcional. Para ello deberemos crear un banco de pruebas (*testbench*) para cada módulo que queramos verificar, donde simulemos las entradas y monitorizemos las salidas, valorando si son las adecuadas o no. Si el módulo se comporta como esperábamos podemos pasar a los siguientes pasos que son la síntesis y la implementación (*place and route*). Vivado nos permite realizar simulaciones post-síntesis y post-implementación, las cuales tienen en consideración el circuito resultante y como esta implementado en el silicio para valorar detalles más avanzados como el *skew* y el *jitter* del reloj. Es muy común que estas simulaciones no arrojen los resultados deseados, revelando condiciones de carrera o bucles combinacionales que permanecían ocultos, obligándonos a replantear el diseño. Cuanto más avanzados estemos en el proceso de

síntesis e implementación, mas largas serán las iteraciones que debamos realizar para corregir los errores observados en simulación. Es por esto que es recomendable detectar la mayor cantidad de fallos en fases tempranas del diseño.

Vivado también nos permite monitorizar el comportamiento de nuestro diseño en vivo, es decir, cargado en la FPGA y en funcionamiento. Para ello, deberemos introducir en nuestro sistema lo que se conoce como ILA[3] Cores. Los bloques ILA pueden ser introducidos en el PL después de la sintesis o inferidos directamente en el código RTL. Pueden ser configurados para monitorizar, mediante sondas, las interfaces y señales de nuestro diseño. Podremos añadir tantos como queramos (teniendo en cuenta que consumen área), cada uno asociado a un dominio de reloj: la frecuencia de muestreo de las sondas será la de la señal de reloj conectada al ILA Core. Estos cores van conectados a un hub el cual se comunica con Vivado a través de JTAG. Cada core contiene un conjunto programable de comparadores los cuales podemos configurar en tiempo de ejecución desde Vivado para que hagan de trigger y muestren las señales de onda capturadas cuando se cumplan las condiciones que especifiquemos. Estas condiciones pueden ser sencillas, como comprobar cuando una señal (o bit) toma un determinado valor; o complejas, como una maquina de estados que precise de una cierta secuencia de acontecimientos para generar un trigger.

Validación de software

En Vitis, como en otros muchos IDE, podremos realizar la depuración de aplicaciones mediante GDB (GNU Debugger) el cual ofrece amplias facilidades para rastrear y modificar la ejecución de programas informáticos. Con el depurador podremos interrumpir la ejecución del programa cuando ocurra un evento determinado (mediante el uso de watchpoints, breakpoints...), pudiendo monitorizar y modificar los valores de las variables internas de los programas, de los registros de la CPU y de la memoria. Además, podremos hacer que la CPU ejecute las instrucciones paso a paso para poder comprobar el correcto funcionamiento de nuestros algoritmos.

Validación de software/hardware

Es posible la validación de ambos dominios funcionando la vez mediante el empleo de *cross-triggers*. Podemos programar triggers que vayan del PS al PL y viceversa. Esto permite al usuario utilizar puntos de interrupción en el software para activar la captura de datos de hardware y, a la inversa, puntos de interrupción de hardware que pueden detener la depuración de aplicaciones en el entorno de desarrollo de software.

3.2 Flujo de diseño

Entendemos como flujo de diseño a los pasos que se sucederán en el proceso de desarrollo de un trabajo. A continuación veremos como este flujo se trata de un proceso iterativo entre hardware y software, donde iremos perfeccionando el diseño o añadiendo nuevas funcionalidades en cada iteración.

3.2.1 Creación de la plataforma hardware

Por regla general (sobretodo si el equipo esta formado por una sola persona), se suele empezar realizando el diseño del hardware. Usaremos Vivado para diseñar los bloques

[3] *Integrated Logic Analyzer*

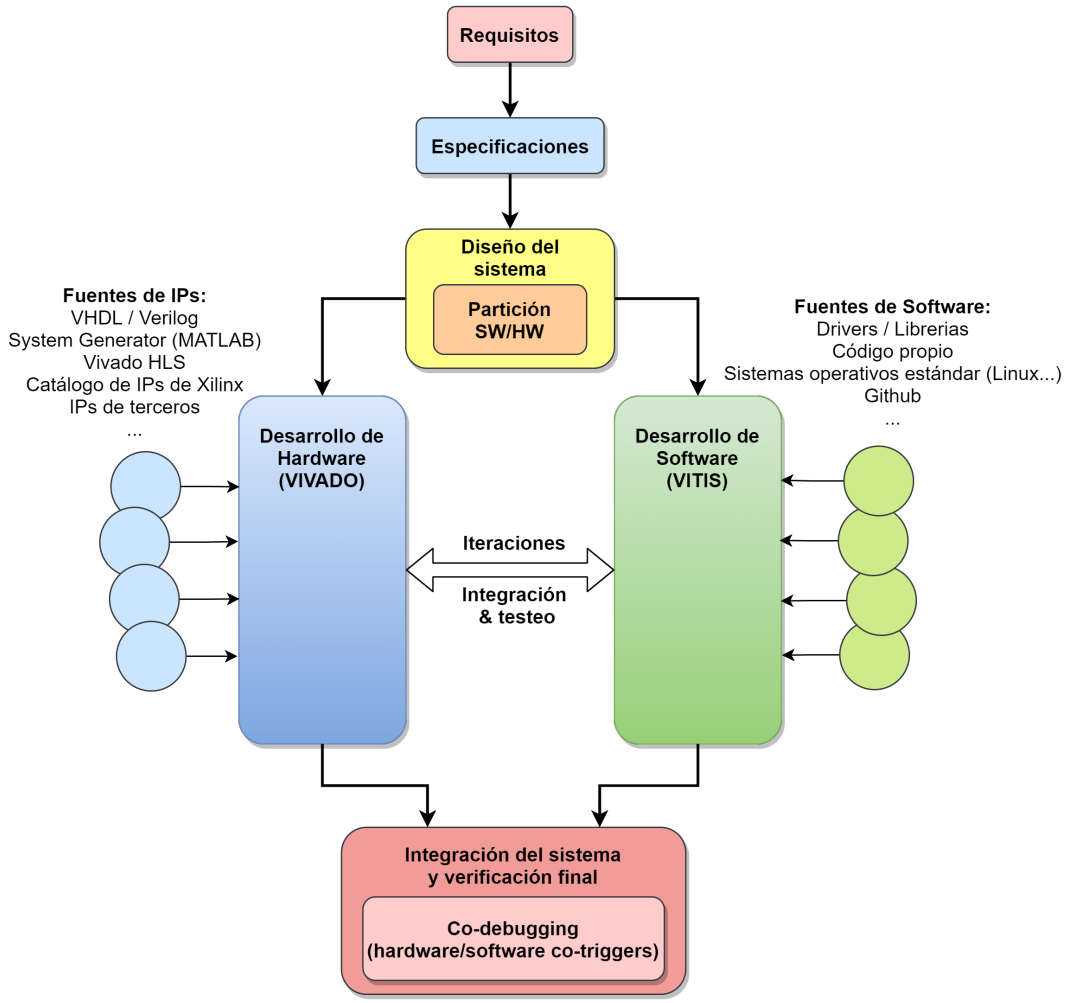

Figura 3.2 Flujo de diseño.

periféricos y otra lógica a implementar en el PL; crear las conexiones adecuadas entre estos bloques y el PS, y configurar adecuadamente el PS. Por ejemplo, una plataforma hardware puede incluir las interfaces a un bus USB y una UART para depuración, junto con un co-procesador que sirva de soporte al software ejecutándose en el ARM.

Para crear el diseño del sistema usaremos *IP Integrator*, donde iremos añadiendo y conexionando al diagrama de bloques IPs creadas por nosotros junto con IPs de terceros. Un bloque que no debe faltar es el de *ZYNQ7 Processing System*, el cual es una representación del PS. Lo usaremos para conectar el ARM con la lógica programable y para configurar muchos aspectos del PS como relojes y periféricos.

Una vez terminado nuestro diseño el siguiente paso será exportar la plataforma. Esta plataforma podremos exportarla como hardware o como emulación de hardware[4]. En cualquier caso el resultado es un archivo con extensión .xsa, que tiene las especificaciones

[4] De esta forma el código del PL es compilado en un modelo RTL conductual que puede ejecutarse en el simulador de Vivado, proporcionando un comportamiento de la lógica ciclo a ciclo. Esto nos permite desarrollar nuestra aplicación sin necesidad de tener el hardware a mano. Además, se consiguen iteraciones más rápidas en el flujo de diseño, ya que no es necesario pasar por todo el proceso de compilación de hardware, el cual es significativamente más lento que el de compilación de la emulación.

del hardware como propiedades de configuración del procesador, información de conexión de periféricos, mapa de direcciones y código de inicialización del dispositivo. La parte de creación de la plataforma hardware se corresponde con los pasos 1-6 del esquema de la figura 3.3.

3.2.2 Creación de dominio

El siguiente paso será crear un projecto de plataforma en Vitis e importar el archivo xsa generado en Vivado (paso nº 7 en la figura 3.3). En Vitis una plataforma es una combinación de componentes hardware (xsa) y componentes software (dominios/BSP, bootloaders...). Un dominio es un paquete de soporte de placa (BSP) o un sistema operativo con una colección de drivers sobre los cuales construir tu aplicación. Es posible crear múltiples aplicaciones para un dominio. Cada dominio está vinculado a un solo procesador.

A la hora de crear un dominio podemos elegir si queremos que el procesador ejecute un sistema operativo (como Linux o FreeRTOS) o si por el contrario va a ser *standalone*. Esto último significa que la aplicación que creemos para dicho procesador sera *bare-metal*, es decir, operará directamente sobre el hardware controlando los recursos de este sin ninguna capa de abstracción. Deberemos valorar si queremos usar un sistema operativo, cuando queramos correr múltiples aplicaciones en paralelo o usar código portable más cercano a la capa de usuario; o si por el contrario preferimos un dominio *standalone*, para sistemas más sencillos o donde queramos tener más control sobre los recursos del hardware (a menudo escribiendo nuestros propios drivers), a costa de trabajar a bajo nivel con aplicaciones poco portables y exclusivas a un dispositivo.

Trabajar con un dominio *standalone* puede ser un arma de doble filo ya que, al tener el control absoluto de los recursos, es más fácil cometer errores que, de haber usado un sistema operativo, no se hubieran cometido. Un ejemplo es el acceso a memoria: un sistema operativo asigna un espacio de memoria a cada tarea, impidiendo que pueda manipular la memoria fuera del espacio que le ha sido asignado. Sin un sistema operativo que actúe de salvaguarda debemos tener cuidado en el acceso a memoria, ya que podríamos corromperla sin querer (escribiendo, por ejemplo, encima de nuestro propio código ejecutable).

Una vez creados los dominios (paso nº 8) que queramos dentro del proyecto de plataforma podremos pasar al siguiente paso.

3.2.3 Creación de aplicación

Lo siguiente será crear un proyecto de sistema, donde agruparemos todas las aplicaciones que se ejecutarán de manera simultanea. Un proyecto de sistema puede tener varias aplicaciones solo si el dominio incluye un sistema operativo; si por el contrario el dominio es *standalone*, este solo podrá tener una aplicación asociada a él (que a su vez estará asociada a un procesador ya que recordemos que cada dominio esta asociado a único procesador).

Cada aplicación estará compuesta de uno o más archivos fuentes con sus correspondientes archivos de cabecera.

3.2.4 Creación de imagen boot

Una vez hayamos terminado de depurar el sistema, el último paso consistirá en grabar en la memoria no volátil (puede ser una memoria flash o en una tarjeta microSD) una imagen booteable. Esta imagen se compone del *First Stage Bootloader* (FSBL), de los archivos

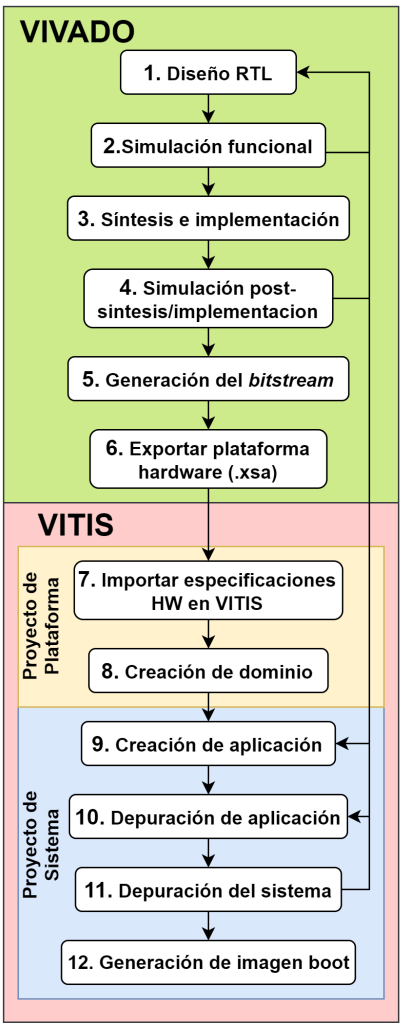

Figura 3.3 Pasos a seguir.

ELF[5] (códigos objeto del programa) y del *bitstream*, agrupados en particiones. El proceso de inicialización del sistema es el siguiente:

1. El código de inicio de fase-0, almacenado en una BootROM interna, se encarga de configurar el ARM y los periféricos necesarios para extraer el FSBL y almacenarlo en la memoria interna.
2. El FSBL inicializa el PS (relojes, I/O, etc.) con los datos de configuración proporcionados a través de las herramientas de Xilinx.
3. El FSBL programa el PL usando el *bitstream* (si es que ha sido proporcionado).
4. Por último, carga en memoria el bootloader de fase-2 (para el caso de sistemas operativos) o la aplicación *bare-metal* (para dominios *standalone*) y le concede el control del procesador.

[5] *Executable Linkable Format* es un formato estándar de archivos ejecutables y códigos objeto resultantes de la compilación.

Usaremos la herramienta *Bootgen*, proporcionada por Xilinx, para crear la imagen booteable. Esta se encargará de tomar los archivos fuente y crear las particiones necesarias de acuerdo al formato BIF (*Boot Image Format*). El layout del archivo resultante puede verse en [17].

3.3 Control de versiones y builds automáticas

Una de las mejores prácticas que podemos establecer en cualquier proyecto que involucre la creación de código es la de mantener un control de versiones. El control de versiones es un sistema que registra los cambios en un archivo o conjunto de archivos a lo largo del tiempo para que puedan recuperarse versiones específicas más adelante. Estas versiones se almacenan en un repositorio local u, opcionalmente, en un repositorio remoto que puede estar ubicado en la nube (Github, Bitbucket...). Las herramientas de Xilinx no presentan un método directo, por lo que existen distintas aproximaciones que puedes adoptar para establecer el control de versiones de un sistema basado en Zynq, a continuación se muestra el método adoptado a lo largo del proyecto.

3.3.1 Jerarquía de carpetas

Una de las partes más importantes es crear un espacio de trabajo en el explorador de archivos. En este espacio de trabajo se guardan, ordenados por carpetas, todos los archivos relacionados con el proyecto, desde documentos hasta código fuente. La carpeta del proyecto (que a su vez se descompone en subcarpetas) estará guardada dentro del repositorio principal, junto con otros proyectos. En la figura 3.4 aparece un ejemplo típico de subcarpetas que pueden existir en un proyecto de este tipo:

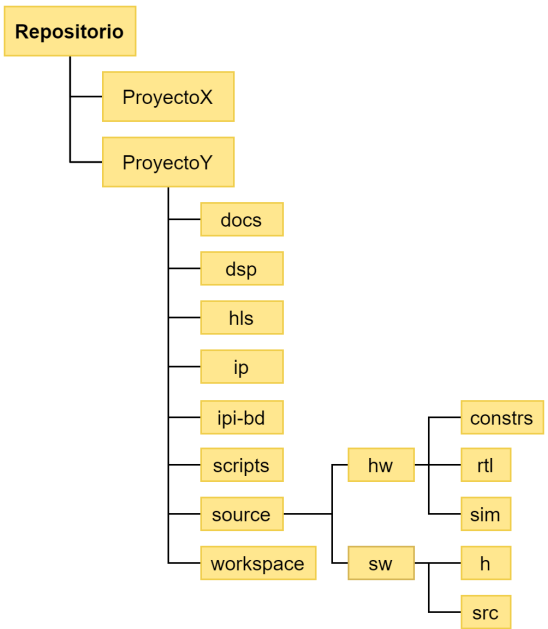

Figura 3.4 Jerarquía de carpetas del proyecto.

- **docs:** documentación del proyecto.

- **dsp:** archivos generados con *System Generator for DSP* de Matlab.
- **hls:** archivos generados por Vitis HLS para la síntesis de alto nivel de circuitos lógicos.
- **ip:** biblioteca de IPs que hayamos creado nosotros y que usemos en el proyecto.
- **ipi-bd:** diseños de bloques generados con *IP Integrator* de Vivado.
- **scripts:** aquí guardaremos todos los scripts (TCL, python...) que usemos para generar el proyecto, hacer las builds, etcétera.
- **source:** aquí ira todo el código fuente, separado en software y hardware.
  - **sw/src:** archivos .c/.cpp.
  - **sw/h:** cabeceras .h/.hpp.
  - **hw/constrs:** archivos .xdc (restricciones de timing y de pinout).
  - **hw/rtl:** código HDL, puede ser Verilog o VHDL.
  - **hw/sim:** archivos de simulación (testbenches).
- **workspace:** este será el workspace tanto de Vivado como de Vitis. Estos programas generarán de forma automática multitud de archivos, la mayoría de ellos serán basura o innecesarios.

La clave esta en que realizaremos el control de versiones de todas las carpetas menos de la de workspace[6]. Guardaremos en el repositorio solo los archivos fuente a partir de los cuales podamos reconstruir el proyecto. Para reconstruir el proyecto usaremos scripts TCL, que en este caso pueden ser autogenerados escribiendo el comando *write_project_tcl* en Vivado. Desde Vivado y Vitis también podremos autogenerar scripts que realicen builds automáticas.

Un aspecto importante para poder llevar a cabo este método es enlazar los archivos fuente en vez de copiarlos directamente en el workspace. De esta manera, para cualquier modificación del archivo que se realice dentro de Vitis o Vivado, el cambio se producirá en el archivo original (ubicado en source\).

[6] Con Git esto lo haremos añadiendo un archivo de texto llamado .gitignore donde incluiremos la rutas que no queramos incluir, en este caso workspace\.

4 Audio digital

Todos los sonidos que escuchamos con nuestros oídos son ondas de presión en el aire. A lo largo de la historia (empezando con la demostración de Thomas Edison del primer fonógrafo en 1877) han sido numerosos los intentos por capturar estas ondas de presión en un medio físico para poder reproducirlas más tarde, regenerando las mismas ondas de presión.

Aunque los intentos de digitalizar el sonido y la teoría vienen de décadas antes [18-19], el sonido digital llegó al público general por primera vez en 1982 con el Compact Disc (CD), el cual fue un gran éxito comercial desarrollado conjuntamente por Sony y Philips. El audio digital trajo consigo numerosas ventajas, algunas de ellas son:

- **Facilidad de manipulación y edición:** Puedes copiar el archivo tantas veces como se quiera sin perjudicar al original. Además, es relativamente sencillo editar archivos de audio con un ordenador.
- **No se deteriora:** Los sonidos grabados en un soporte digital no pierden calidad con el paso del tiempo ni por el uso.
- **Menor espacio de almacenamiento:** Es posible guardar miles de minutos en un disco duro.
- **Streaming:** Con la llegada de los servicios de streaming de audio es posible acceder a prácticamente cualquier canción, programa de radio, etc., en cuestión de segundos.

4.1 Representación digital del sonido

El audio digital se obtiene mediante la discretización (obtención de una secuencia de valores discretos) de una señal analógica mediante dos procesos: muestreo y cuantificación. Estos dos procesos se llevan a cabo en el conversor analógico-digital y están estrechamente ligados a la calidad del audio obtenido.

4.1.1 Muestreo

El muestreo consiste en fijar la amplitud de la señal eléctrica en intervalos regulares de tiempo (frecuencia de muestreo). Según el teorema de Nyquist, para recomponer una señal que ha sido previamente muestreada y que no se pierda información, es necesario que

la frecuencia de muestreo sea de al menos el doble de la frecuencia máxima de la señal. Como el espectro audible cubre frecuencias desde los 20Hz hasta los 20kHz, será necesario muestrear al menos a 40kHz. Por ejemplo en el formato CD se emplea una tasa de muestreo de 44,1 kHz. La Audio Engineering Society recomienda usar 48kHz para la mayoría de las aplicaciones [20]. También es común ver equipos de altas prestaciones que utilizan una frecuencia de muestreo de 96kHz. Esto suele hacerse para poder relajar las especificaciones de los filtros de anti-aliasing.

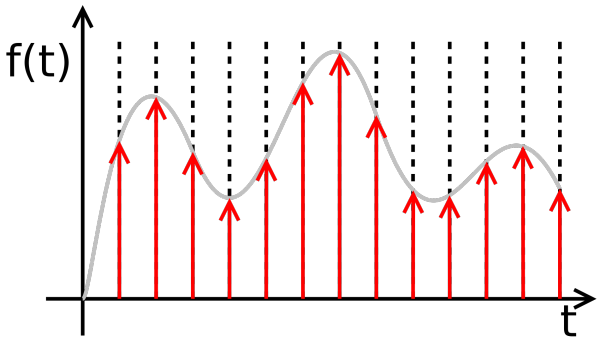

Figura 4.1 Proceso de muestreo de una señal continua (Fuente: wikipedia.org).

4.1.2 Cuantificación

La cuantificación consiste en asignar a cada muestra obtenida en el proceso de muestreo un valor entero, binario, de rango finito y predeterminado. La resolución o profundidad de bits es el número de bits empleados en el proceso de cuantificación para representar la señal. Por ejemplo, un conversor de 12 bits discriminará entre 4096 niveles de señal equidistantes.

El formato resultante se conoce como PCM (*Pulse-Code Modulation*), es el estándar en audio digital. En este proyecto se utilizará el formato PCM lineal, que quiere decir que se utilizó una cuantificación lineal.

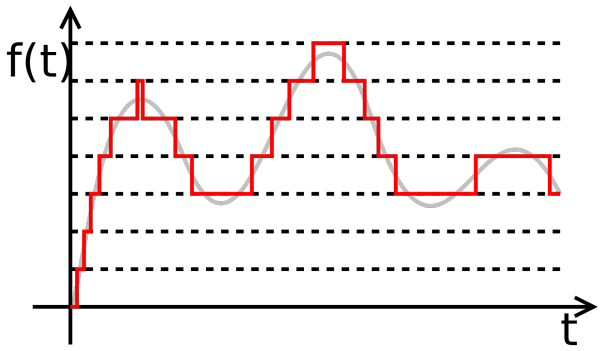

Figura 4.2 Proceso de cuantificación de una señal muestreada (Fuente: wikipedia.org).

4.2 Síntesis de audio digital

Aunque existen muchos métodos de síntesis [21], en este proyecto nos dedicaremos a la síntesis de formas fijas de onda, que consiste en generar señales periódicas mediante la continua repetición de una forma de onda. Esta forma de onda puede ser calculada muestra

a muestra en tiempo real mediante una formula matemática o puede estar contenida en una tabla de valores calculados de antemano. Esta última aproximación, conocida como *Wavetable Synthesis*, será la que utilicemos. Cuantos más valores hayan almacenados en la tabla, mejor será la aproximación y calidad de la onda.

Las principales ventajas de este método es que es fácil de implementar, presenta muy buena calidad y es eficiente en el tratamiento de señales periódicas. Como inconveniente, se requiere de espacio en la memoria para almacenar las formas de onda y el sonido no presenta ninguna variación temporal (es monótono). Para corregir esto último deberemos acudir a técnicas de modulación en amplitud o en frecuencia; de síntesis aditiva (sumar varias señales, generalmente armónicos, para crear señales nuevas); o al uso de envolventes acústicas (o ADSR) que modelen de forma dinámica la amplitud para imitar a un instrumento real.

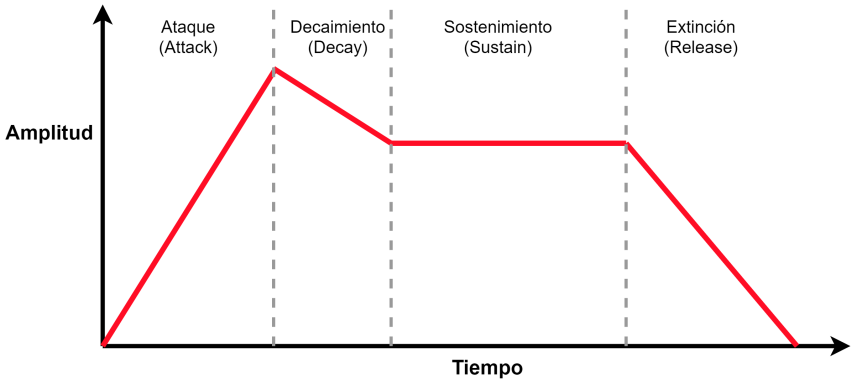

Figura 4.3 Envolvente acústica.

A continuación veremos el algoritmo DDS, usado para la síntesis de formas fijas de onda.

4.2.1 Síntesis digital directa

La síntesis digital directa, o DDS por sus siglas en inglés, es una técnica usada para generar formas de onda analógicas usando medios digitales [22]. Las formas de onda son reconstruidas a partir de una "plantilla" almacenada en la memoria. Esta plantilla contiene los valores de amplitud de la señal para cada fase.

Un sistema DDS se compone de tres bloques funcionales, un acumulador (que consta de un registro de fase y un sumador), un conversor de fase a amplitud de señal (este será el bloque de memoria donde almacenaremos la plantilla de la forma de onda) y un conversor digital-analógico. Se requiere además de una referencia de frecuencia que proviene generalmente de un oscilador de cristal.

El acumulador de fase genera un número correspondiente al ángulo de fase deseado de la onda, mientras que el conversor fase-amplitud lo que hace es generar la función de onda mostrando a su salida en cada momento la amplitud de la señal asociada a la fase de entrada. Por último el conversor D/A proporciona una salida analógica.

Se puede ver la operación del DDS como un contador incremental que recorre las entradas de memoria del conversor fase-amplitud. Para cambiar la frecuencia simplemente cambiamos la velocidad a la que se incrementa el contador. Dado que la frecuencia de

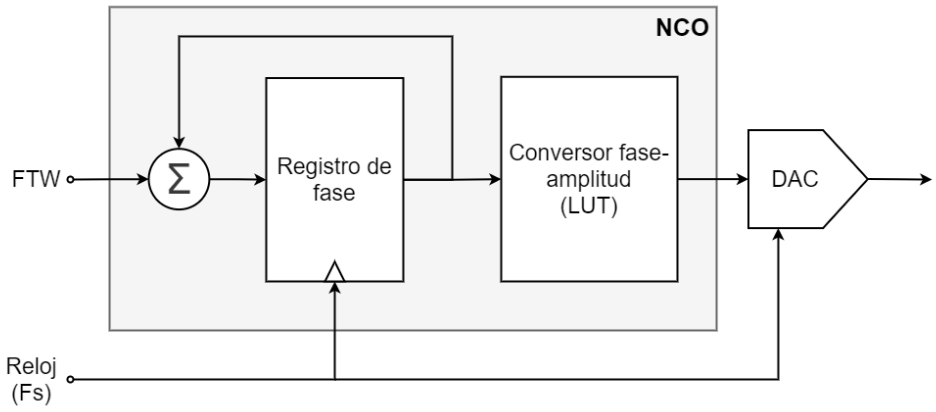

Figura 4.4 Algoritmo DDS.

reloj de referencia es fija, deberemos modificar el valor de entrada al sumador, también conocido como FTW (*Frequency Tuning Word*). De esta manera, dada una frecuencia fija de referencia Fs y un ancho de palabra del contador de n bits, la frecuencia de la señal de salida viene dada por la siguiente formula:

$$f_o = \frac{F_s \cdot FTW}{2^n}$$

Visto de otro modo, podemos calcular el valor de FTW para obtener una frecuencia deseada como:

$$FTW = round\left[\frac{f_o \cdot 2^n}{F_s}\right]$$

Donde el resultado se redondea ya que el valor de FTW solo puede ser un número entero. Usaremos esta última ecuación más adelante para calcular la FTW de todas las notas del sintetizador.

4.3 MIDI

El lenguaje MIDI (acrónimo para *Musical Instrument Digital Interface*) es un estándar que especifica una interfaz y un formato de datos para que los dispositivos que producen y controlan el sonido—como sintetizadores, sistemas de audio y ordenadores—puedan comunicarse entre sí. Este es utilizado para transmitir información en tiempo real[1] para la reproducción de una pieza musical. Una de las características mas importantes del lenguaje MIDI es que este no define el sonido en sí, sino solo la secuencia de instrucciones para crear el sonido en el sintetizador de destino.

Antes de la creación de MIDI no existía ningún estándar para la comunicación entre instrumentos musicales electrónicos. Visionarios como Ikutaru Kakehashi de Roland comenzaron a preocuparse de que esta falta de compatibilidad entre los fabricantes restringiría el uso de sintetizadores, provocando una disminución en las ventas, por lo que propuso la creación de un estándar. Esto llevo a que Dave Smith y Chet Wood, de Sequential Circuits,

[1] "Tiempo real" significa que cada mensaje se envía exactamente en el momento en que debe ser interpretado por el sintetizador de destino (el cual puede ser hardware o software).

presentaran un artículo en 1981 en AES proponiendo un concepto para una interfaz de sintetizadores universal (*Universal Synthesizer Interface* [23]). Finalmente y tras la colaboración de distintas empresas americanas y japonesas como Roland, Oberheim Electronics, Sequential Circuits, Yamaha, Korg y Kawai, la especificación MIDI fue creada y publicada en Agosto de 1983 [24].

4.3.1 Mensajes MIDI

Los mensajes MIDI se envían a través de una conexión serie asíncrona a 31250 baudios. El formato de los mensajes está compuesto por 10 bits: 1 bit de inicio, 8 bits de datos y 1 bit de parada. Los mensajes están compuestos por bytes de estado y de datos. El primer byte es siempre un byte de estado que determina el tipo de mensaje, al cual pueden seguirle uno o más bytes de datos con parámetros adicionales. El número de bytes de datos que le siguen dependerá del tipo de mensaje. Para diferenciar entre ambos tipos de bytes nos fijamos en el bit que ocupa la séptima posición, el cual será un 1 si se trata de un byte de estado y un 0 si se trata de uno de datos. A excepción de algunos mensajes MIDI, el byte de estado contiene el número de canal MIDI. Hay 16 canales MIDI posibles, numerados del 0 al 15.

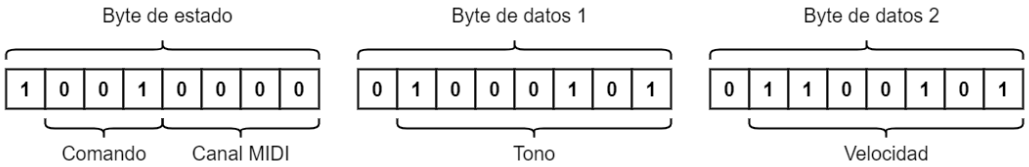

Figura 4.5 Bytes MIDI.

Las instrucciones pueden enviarse en tiempo real o almacenarse en un archivo MIDI (dependiendo del dispositivo). Estos archivos se conocen como *Standard MIDI Files* (SMF) y tienen la extension .mid.

4.3.2 Mensajes más importantes

- **NOTE ON:** El mensaje se envía cuando el intérprete pulsa una tecla del teclado musical. Contiene parámetros para especificar el tono de la nota así como la velocidad (intensidad de la nota cuando se golpea). Cuando un sintetizador recibe este mensaje, comienza a tocar esa nota con el tono y el nivel de fuerza correctos. La estructura del mensajes es la siguiente:
  - Byte de estado : 1001 CCCC
  - Byte de datos 1 : 0TTT TTTT
  - Byte de datos 2 : 0VVV VVVV

  Donde "C" es el número del canal (0-15), "T" es el tono de la nota (0-127) y "V" es la velocidad (0-127). El valor del tono indica la nota a interpretar, siendo el DO central el número 60. Este valor se mueve en semitonos, siendo de esta manera DO# el número 61, RE el número 62, etc. En la figura 4.5 aparece la correspondencia de cada nota con su valor de tono asociado.
- **NOTE OFF:** Este mensaje se envía cuando deja de pulsarse la tecla del teclado. El primer byte de datos deberá tener el valor de la nota que se ha dejado de pulsar,

mientras que el segundo byte puede tener el mismo valor que en el mensaje de NOTEON asociado a la nota que se ha dejado de pulsar, o puede ser diferente, dependiendo del instrumento (hay instrumentos que tienen en cuenta la velocidad con la que se libera la nota). La estructura del mensaje es la siguiente:

- Byte de estado : 1000 CCCC
- Byte de datos 1 : igual que *NOTE ON*.
- Byte de datos 2 : puede ser igual que *NOTE ON* o no, dependiendo del instrumento.

Cada mensaje de NOTE ON requiere de su correspondiente mensaje de NOTE OFF; de lo contrario, la nota se reproducirá para siempre.

Existen muchos más mensajes MIDI como mensajes para la selección de instrumento o mensajes que incluyen parámetros de control como volumen, ruleta de modulación (a menudo asignada a efectos de vibrato o tremolo), pedal de sustain... En este proyecto solo se hace uso de los dos tipos de mensaje arriba mencionados, por lo que si quiere observar una lista detallada con todos los comandos disponibles puede dirigirse a [25].

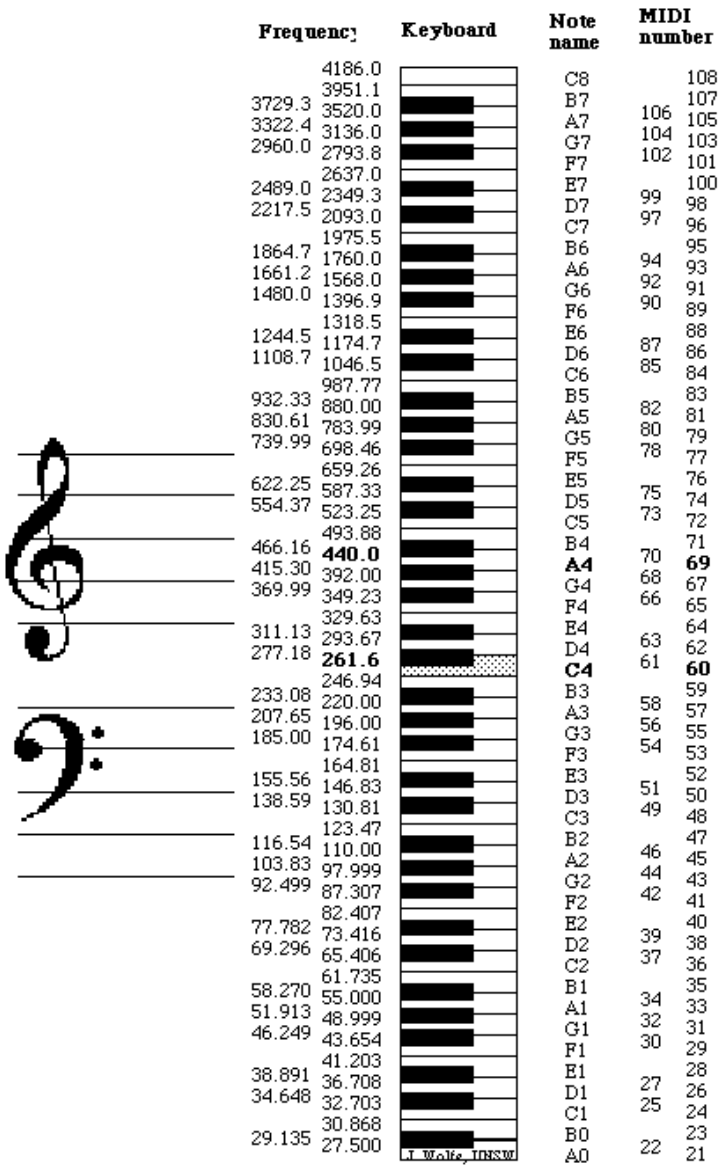

Figura 4.6 Notas MIDI (Fuente: newt.phys.unsw.edu.au/jw/notes.html).

5 Implementación

En este capítulo se detallará como se han ido realizando los distintos módulos de los que se compone el sistema. Para dar cierto orden a la consecución del capítulo, se han dividido los módulos (y por ende, el sistema completo) en lo que se conoce en el ámbito del diseño de procesadores como *datapath* y *control unit*. Nótese que estos términos no son empleados en sentido estricto, ya que no estamos diseñando ningún procesador; sin embargo, hacen una analogía perfecta a la hora de representar una división funcional del sistema en:

- **Datapath:** compuesto por los componentes hardware que realizan las operaciones de procesamiento de datos. En nuestro caso, los datos son de audio, por lo que el *datapath* sera el conjunto de componentes que atraviesa el flujo de audio desde que se genera hasta que es enviado a los auriculares. Todo este flujo de datos se produce exclusivamente en el PL.
- **Control Unit:** compuesto por los componentes hardware y software que le dicen al *datapath* como debe de comportarse. Aquí se agrupa toda la parte de configuración del sistema, comunicaciones, gestión de recursos, etc. Esta labor esta concentrada en su mayoría en el PS aunque también habrá parte de ella en el PL.

Siguiendo este esquema, empezaremos definiendo el flujo de datos primero y la unidad de control después.

5.1 Generación de notas

El corazón de Zynth será el módulo de generación de notas. Este está compuesto a su vez de tres osciladores controlados numéricamente (NCO), uno ira a la frecuencia fundamental de la nota y los otros dos irán al doble y al cuádruple[1] de frecuencia para generar el segundo y cuarto armónico respectivamente. Se podrá configurar tanto la amplitud como la forma de onda de salida de cada oscilador. Las ondas generadas en cada oscilador se sumarán a la salida. Los parámetros de configuración de los osciladores están contenidos en dos señales que se ramifican al entrar al módulo. Esas señales, que pueden verse en la figura 5.2, son:

[1] En electrónica digital, para multiplicar un vector por 2, por 4, o por cualquier número igual a 2^n siendo n un número entero, solo tienes que desplazar el vector a la izquierda n bits.

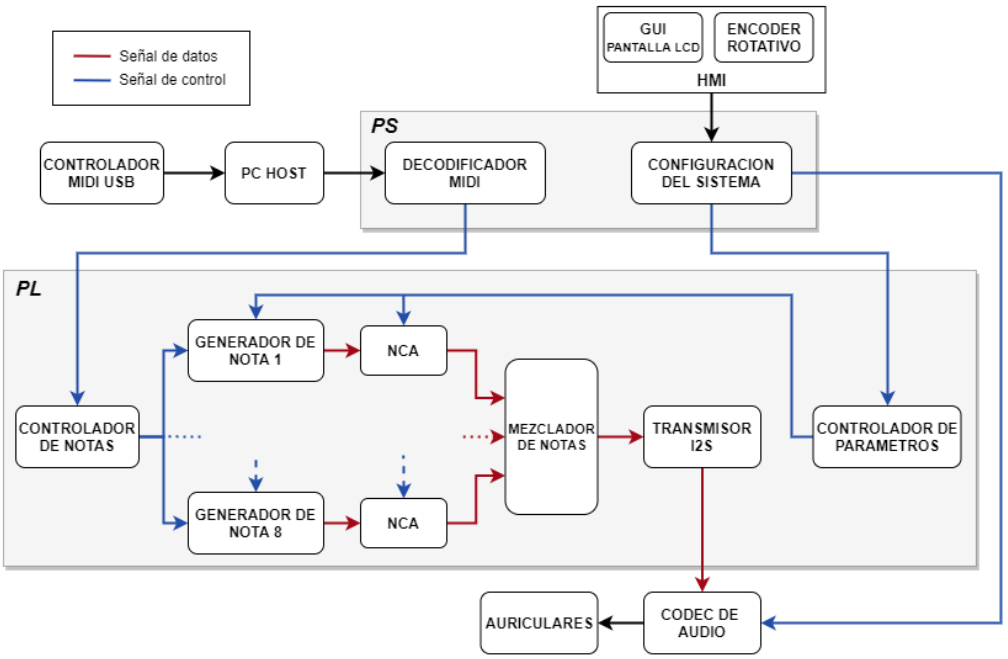

Figura 5.1 Esquema simplificado de Zynth.

- **Amp123:** vector de 24 bits que contiene las amplitudes de los tres osciladores. La amplitud de cada oscilador está codificada por un número de 8 bits, por lo que existen 256 valores disponibles para esta.
- **WaveSelect123:** vector de 6 bits que contiene la selección del tipo de onda de los osciladores. Cada oscilador puede generar cuatro tipos de onda distintas, por lo que la señal de selección de cada uno sera de 2 bits, siendo:
  - "00": Senoide
  - "01": Cuadrada
  - "10": Diente de sierra
  - "11": Triangular

Los NCOs han sido implementados como se explicó en la sección 4.2.1 de síntesis digital directa. Se componen de un circuito acumulador cuya salida es la entrada a un conversor fase-amplitud, el cual es una tabla de valores previamente calculados con la amplitud de la onda para cada fase. Estos valores se calcularon con la ayuda de un script de Matlab y se guardaron en una memoria BRAM dentro de la FPGA.

El circuito acumulador consiste en un contador que se incrementa cada ciclo de reloj un valor definido por FTW. Así, la cuenta que lleve el contador, que a su vez es la fase de la onda que queremos generar, se calculará como $fase = fase + FTW$. El contador, que es ascendente, tendrá integrado la gestión de *overflow*: imagina que el valor máximo es 10, la cuenta se incrementa en 2 y el valor actual es 9. En el próximo ciclo de reloj el valor del contador no será 0, sino 1 (se tendrá en cuenta la diferencia de la suma con el valor de cuenta máximo cuando se vuelva a empezar de cero). Esto último se hace para evitar *glitches* en la generación de las notas. El acumulador se activará con la señal de *Enable*, mientras tanto permanecerá en reposo y con su valor interno igual a cero.

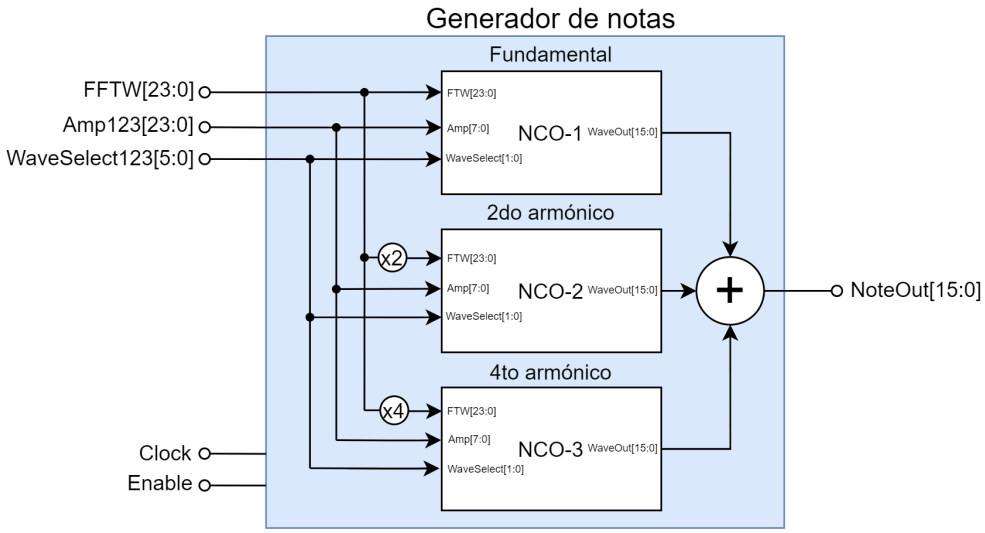

Figura 5.2 Esquema del generador de notas.

Solo necesitamos usar la técnica de *wavetable* para la señal sinusoidal ya que, para generar la señal de diente de sierra se toma directamente el valor del acumulador; para la señal triangular también se toma directamente de la salida del acumulador, pero con la peculiaridad de que el acumulador funcionará en este caso como un contador ascendente-descendente y la FTW será el doble (doble de frecuencia); y por último la señal cuadrada se obtendrá utilizando un comparador a la salida del acumulador que haga conmutar la salida entre dos valores. Estas señales, que solo adoptan valores positivos ($0 \leftrightarrow 2^{16}-1$), serán transformadas a tipo *signed* (es decir, podrán adoptar valores negativos y positivos desde $-2^{15} \leftrightarrow 2^{15}-1$) codificadas en complemento a dos.

En la figura 5.3 se puede ver una simulación de un NCO con distintas opciones de salida. En la figura 5.4 aparece la suma de dos señales a distinta frecuencia.

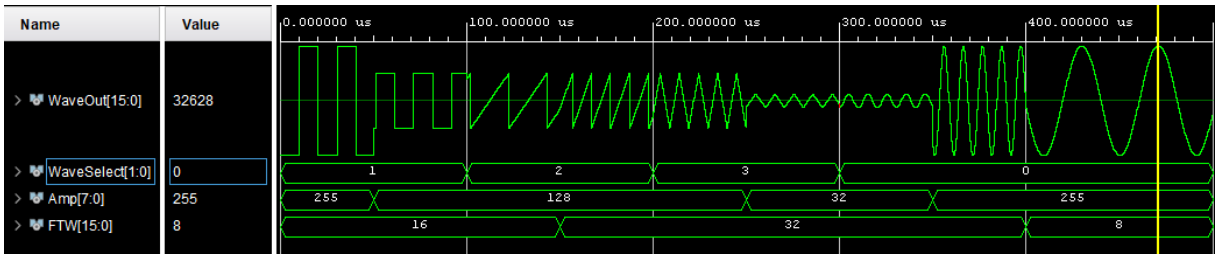

Figura 5.3 Simulación del NCO.

5.2 NCA

El NCA, o *Numeric Controlled Amplitude*, nos permite realizar un control dinámico de amplitud de manera programable. Este control de amplitud se utiliza tanto para implementar una modulación (mediante un LFO), como para generar una envolvente acústica en la nota generada.

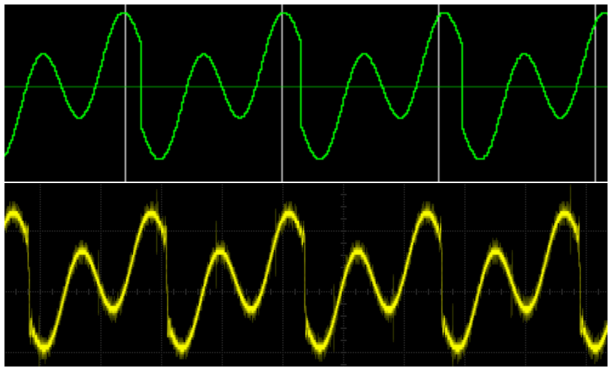

Figura 5.4 Simulación y captura de osciloscopio de una nota generada mediante la suma de una señal diente de sierra y una señal sinusoidal al doble de frecuencia (no están a la misma frecuencia arriba y abajo).

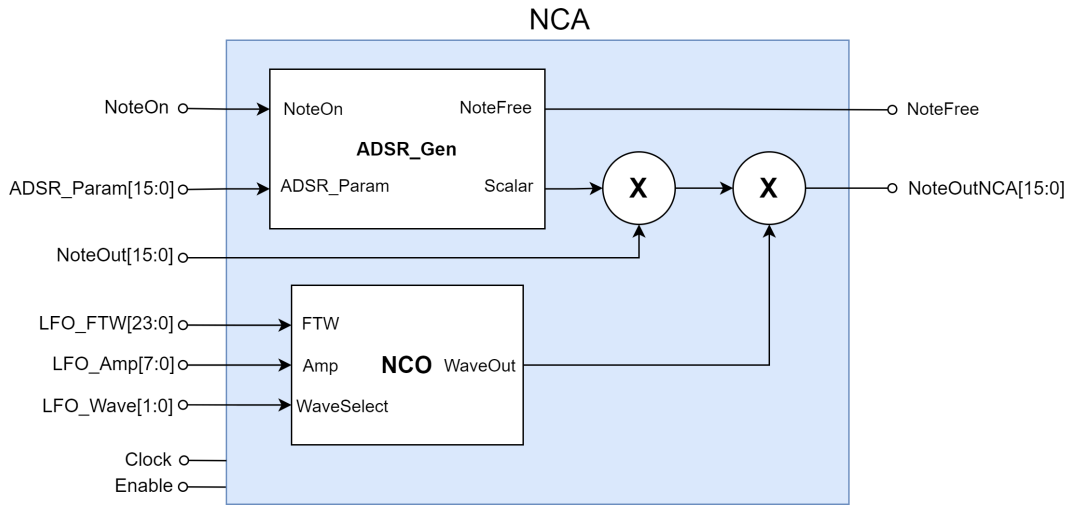

Figura 5.5 Esquema del NCA.

5.2.1 LFO

El LFO (*Low Frequency Oscillator*), como su propio nombre indica, no es más que un oscilador convencional que oscila a baja frecuencia. Este oscilador, cuya salida será de 16 bits, multiplicará a la nota, también de 16 bits, almacenando el resultado en un registro de 32 bits[2]. De esos 32 bits cogemos únicamente los 16 más significativos (es decir, [31:16]).

5.2.2 Envolvente acústica

La envolvente acústica, también conocida como envolvente ADSR, es una técnica que consiste en modificar en el tiempo la amplitud de un sonido para simular un instrumento de verdad. El módulo encargado de dicha envolvente (en el esquema aparece como ADSR_Gen) se encarga de generar un escalar de 16 bits con el valor de la envolvente en cada momento y que multiplicara a la señal de audio. Dicho número escalar es generado mediante una

[2] Para que no se produzca overflow, el registro donde se almacene el resultado de multiplicar dos señales, de n y m bits respectivamente, debe tener un ancho mínimo de $n+m$ bits

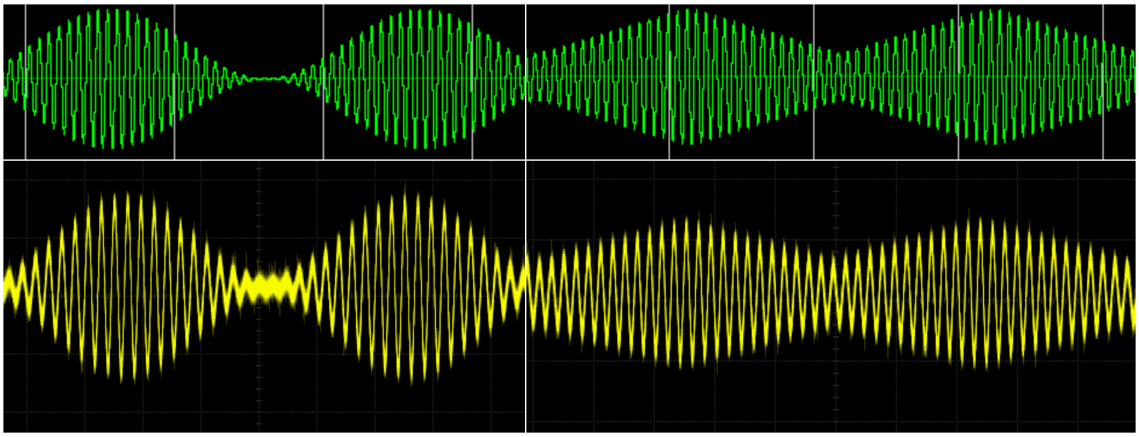

Figura 5.6 Simulaciones y capturas de osciloscopio del LFO en funcionamiento.

máquina de estados finita, que aparece en la figura 5.9. Los parámetros de configuración (*attack, decay, sustain* y *release*), de 4 bits cada uno, están contenidos dentro de la señal ADSR_Param.

Al comienzo, la máquina de estados está a la espera de que se le asigne una nota (la asignación de las notas pulsadas por el teclado está arbitrada por el controlador de notas como veremos más adelante), el valor del escalar será cero y la señal de *NoteFree* valdrá 1, indicando que tanto el generador de notas como el NCA asociado a él se encuentran libres. Cuando llegue su turno, al pulsar una nota en el teclado, el valor de *NoteOn* pasará a ser 1, marcando el comienzo de la fase de ataque. El valor del escalar comenzará a aumentar hasta llegar al valor máximo, donde pasara a la fase de decaimiento. La velocidad a la que sucede esto estará definida por el parámetro de *attack* que hayamos configurado. Si el parámetro de *attack* esta al máximo, se pasará directamente a la fase de decaimiento produciendo un ataque instantáneo. El funcionamiento en el resto de estados es similar y se dejará la interpretación al lector. Cabe señalar que, si en cualquiera de los estados *NoteOn* pasa a valer 0, indicando que se ha dejado de pulsar la tecla, se pasará automáticamente a la fase de extinción.

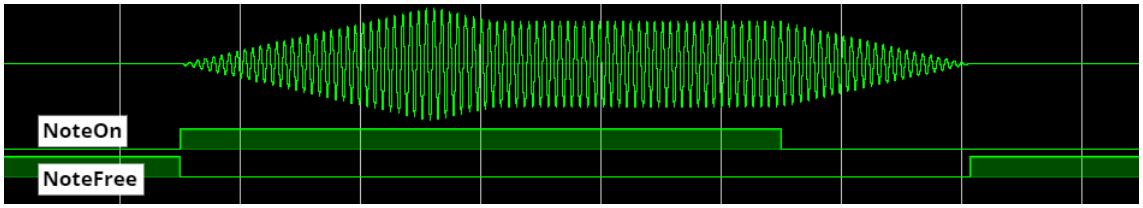

Figura 5.7 Simulación de la envolvente, junto con las señales de *NoteOn* y *NoteFree*.

5.3 Mezclador de notas

Como estamos trabajando en el diseño de sistemas digitales a bajo nivel en VHDL, no podemos sumar directamente las notas ya que, suponiendo que se encuentren al 100 %, se produciría un desbordamiento. Para solucionar este problema debemos habilitar bits extra en el registro donde se almacene la suma. Para conocer el número de bits extra que

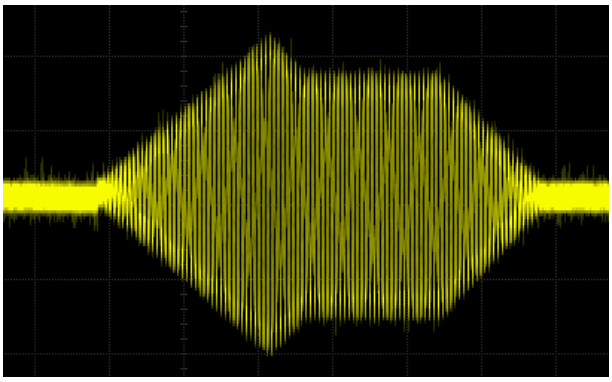

Figura 5.8 Envolvente acústica vista en el osciloscopio.

necesitamos usamos la función $log_2(m)$, siendo m el número de señales[3] que se quieren sumar entre sí [26]. De esta manera, como nuestro sintetizador tendrá una polifonía de 8 voces, el número de bits extra que necesitamos es $log_2(8) = 3$, así que reescalaremos todas las señales añadiendo 3 ceros a la izquierda del bit más significativo antes de sumarlas. Para la señal de salida, tomaremos los 16 bits más significativos de la señal de 19 bits procedente de la suma de las notas.

5.4 Transmisor I2S

El último módulo dentro del flujo de datos es el transmisor I2S. Este se encargará de tomar el flujo de datos de audio y enviarlo al codec mediante el protocolo I2S. Este protocolo es una variante de I2C adaptada a audio [27]. En él tendremos las dos señales que se encuentran en I2C (señal de datos en serie y señal de reloj) más una tercera exclusiva que servirá para indicar si los datos pertenecen al canal izquierdo o al derecho.

La selección del canal se hace mediante una señal (WS) con un *duty-cycle* del 50 % y que tiene la misma frecuencia que la frecuencia de muestreo (en este caso, $48kHz$). Para señales estereo, la especificación establece que el audio izquierdo se transmite en el ciclo bajo de WS y el canal derecho se transmite en el ciclo alto. Por lo general, esta señal se sincroniza con el flanco descendente del reloj serie (SCK), ya que los datos se muestrean en el flanco ascendente. Cabe destacar que el reloj de selección de canal va adelantado un pulso a la transmisión del bit más significativo de dicho canal. Esto permite que el receptor almacene la palabra anterior y borre su buffer de entrada para recibir la siguiente.

La frecuencia de la señal de reloj serie será de $1.536MHz$, ya que por cada ciclo de WS debemos transmitir 32 bits (16 por cada canal). Como la señal maestra de reloj (MCLK), que es la señal de reloj utilizada por todos los módulos del PL, es de $12.288MHz$[4], el primer paso será escalar dicha señal para obtener SCK. Esto se hace usando un contador que invierta la señal SCK cada cuatro pulsos de MCLK (lo que equivaldría a dividir $12.288MHz$ entre 8). Una vez obtenido SCK usaremos dicha señal como reloj interno del transmisor, es decir, actualizaremos todos los registros cuando se produzca su flanco de bajada.

[3] Suponemos que todas las señales que queremos sumar tienen el mismo número de bits, en este caso, 16 bits.

[4] El motivo de seleccionar dicha frecuencia esta relacionado con la configuración del codec de audio y será explicado en la sección dedicada a ello.

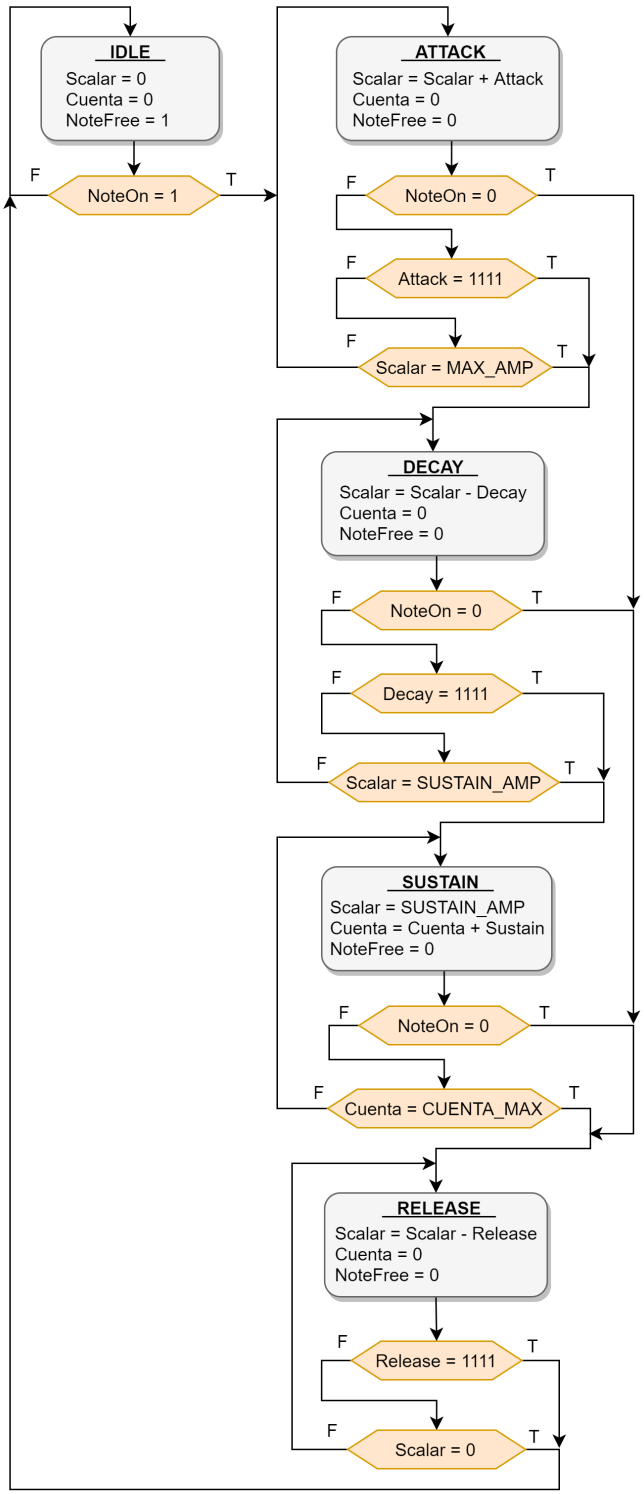

Figura 5.9 Algoritmo de generación del escalar que multiplica a la señal de audio para producir la envolvente acústica.

A continuación crearemos una máquina de estados que genere la señal de WS y que indique que canal transmitir en cada momento. El algoritmo empleado en dicha máquina aparece en la figura 5.11. Las señales de entrada son muestreadas cuando $SampleFlag = 1$. Esto se produce durante un ciclo de SCK, justo antes de comenzar a transmitir el canal

izquierdo. Las señales de entrada son almacenadas en los registros LDATA y RDATA. De esta forma, la señal de salida sera LDATA(BitCount) cuando estemos en el estado LEFT_TX y RDATA(BitCount) cuando estemos en el estado RIGHT_TX.

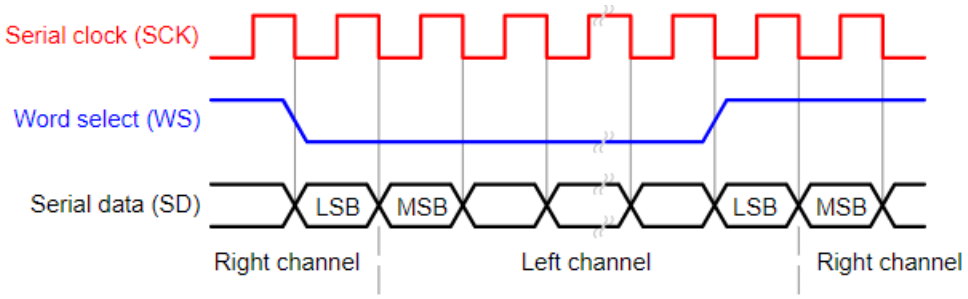

Figura 5.10 Diagrama temporal del protocolo I2S.

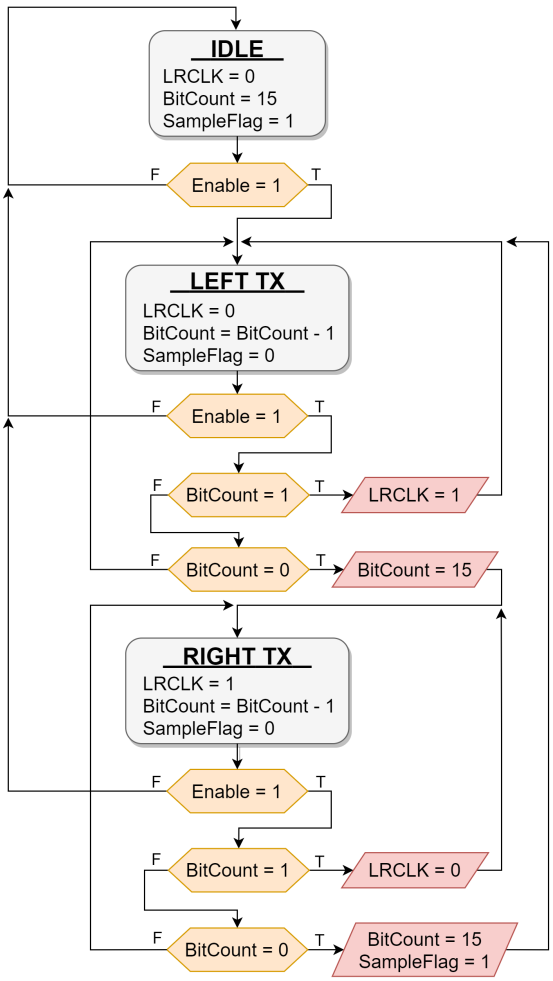

Figura 5.11 Algoritmo de generación de la señal de selección de canal WS.

5.5 Interfaz humano-máquina

La interfaz de usuario está compuesta de una pantalla LCD y de un encoder rotativo que usaremos para navegar por los menús. Desde la interfaz podremos configurar todos los parámetros del sintetizador.

5.5.1 Encoder rotativo

Los encoders rotativos *incrementales* (también llamados de cuadratura) emiten pulsos eléctricos en función de la rotación del eje. A diferencia de los *absolutos*, no disponen de ninguna información sobre la ubicación en el momento de la puesta en tensión y deben reinicializarse cada vez que se interrumpe el suministro de energía. Mientras que los *absolutos* poseen un contador interno, en los *incrementales* deberá ser el programa o software el que lleve la cuenta.

Figura 5.12 Encoder rotativo incremental (Fuente: amazon.es).

La interfaz de un encoder absoluto es sencilla, ya que este dispondrá de pines de salida donde este codificado en binario el valor de su posición. Debido a que llevan una lógica interna incorporada, este tipo de encoders son más caros. Por el contrario, la interfaz de un encoder incremental es más simple, solo tendrá dos pines de salida, pero requerirá de una lógica (esta vez interna a nuestro dispositivo) que permita decodificar los pulsos emitidos por el encoder. Para entender cómo debe ser este controlador echemos un vistazo a los pulsos emitidos por el encoder en cada rotación en la figura 5.13.

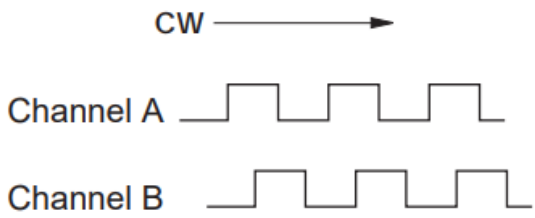

Figura 5.13 Pulsos emitidos por el encoder en una rotación CW (*Clock-wise*).

Los dos canales del encoder se encuentran a nivel alto. Cuando se produce una rotación, ambos canales entran en contacto con una lamina que va conectada a tierra. La posición de los canales es tal que, cuando se produce una rotación en el sentido de las agujas del reloj, el canal A pasará a valer 0 antes que el canal B; y cuando se produce una rotación en sentido contrario al de las agujas del reloj, el canal B pasará a valer 0 antes que el canal A.

El controlador que implementaremos en el PL monitorizará ambos canales y detectará cual de ellos se vuelve 0 primero. El controlador dispone de un temporizador para descartar *glitches* que se puedan producir por el rebote de las laminas o por cualquier otra cosa. El modelo de encoder empleado lleva incorporado ademas un botón. El controlador se encargará también de descartar los rebotes producidos al presionar el botón.

Las tres señales de salida de nuestro controlador, CW (*Clock-wise*), CCW (*Counter Clock-Wise*) y BTN (*Botón*) entrarán al PS a modo de interrupciones. A continuación veremos en detalle como funcionan las interrupciones en Zynq (más concretamente, en el Cortex-A).

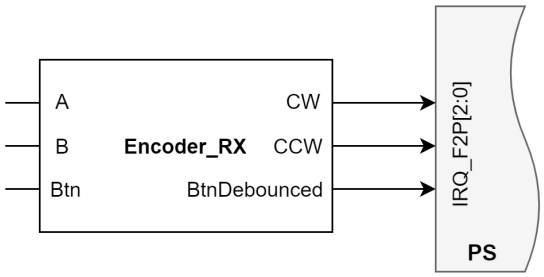

Figura 5.14 Bloque encargado de recibir las señales del encoder. Las señales de salida entran al PS mediante las lineas IRQ_F2P (*Fabric−to−processor*) para interrupciones PL−PS.

5.5.2 Interrupciones

La ejecución de un programa en el ARM es interrumpida automáticamente cuando ocurren ciertas excepciones, como la activación de una señal de interrupción externa o la ocurrencia de una falla interna (por ejemplo, si intentamos ejecutar una instrucción indefinida). Cuando ocurre una interrupción, el procesador interrumpe la ejecución normal del programa y obtiene su siguiente instrucción de una dirección especial, que variará en función del tipo de interrupción. Esta dirección se conoce tipicamente como "vector de interrupciones", y la instrucción contenida en dicho vector es un salto a una subrutina, conocida como *handler*, que se ocupará del evento que causó la interrupción.

Los dispositivos periféricos externos o integrados pueden generar una interrupción activando las señales de interrupción de hardware IRQ (*Interrupt Request*) o FIQ (*Fast Interrupt Request*) de ARM. Por lo general, estas dos señales no son activadas directamente desde dispositivos periféricos, sino por un circuito controlador de interrupciones que es en sí mismo un circuito periférico del procesador. El *Generic Interrupt Controller* (GIC) de ARM puede recibir y procesar más de 100 señales de interrupción de varios dispositivos periféricos integrados y externos, asignando a cada señal un número identificador que es fijo.

El GIC realiza un seguimiento de las fuentes de interrupción, por lo que cuando se activa IRQ, el *handler* puede consultar al GIC para saber qué causó la interrupción y pasar el control a una rutina de servicio (*Interrupt Service Routine*) escrita específicamente para esa interrupción. El GIC también permite que el software habilite, deshabilite, enmascare y priorice las fuentes de interrupción. Antes de que se puedan utilizar las interrupciones, el

GIC debe configurarse para asignar prioridades de interrupción y habilitar las interrupciones deseadas.

Para usar interrupciones en cualquier aplicación, deben llevarse a cabo los siguientes pasos:

- La(s) fuente(s) de interrupción (periféricos integrados y/o dispositivos externos) deben ser configuradas para generar una señal de interrupción cuando se cumplan las condiciones deseadas.
- El GIC debe ser propiamente configurado: hay que habilitar cada señal de interrupción deseada y asignarle una prioridad.
- El *handler* de la interrupción debe ser escrito para que consulte en el GIC que interrupcion está activa y bifurque a la ISR escrita para esa interrupción.
- Una ISR debe ser escrita para cada interrupción que deba ser atendida.
- Debe habilitarse el ARM para que pueda recibir y procesar interrupciones.

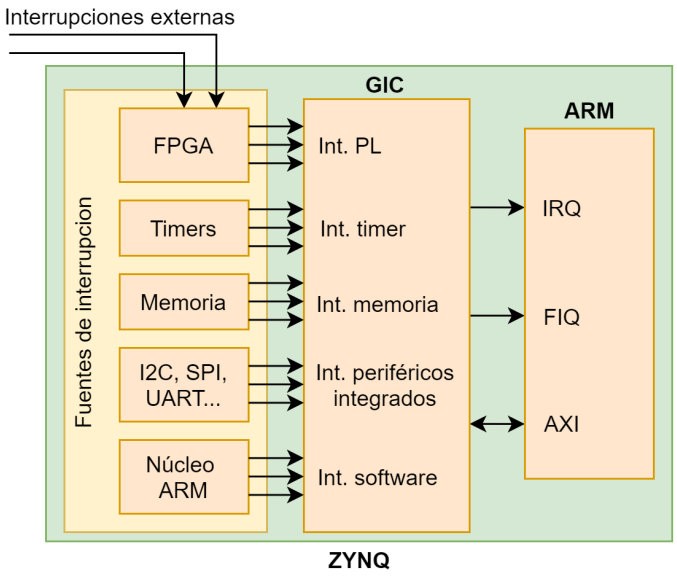

Figura 5.15 Generic Interrupt Controller.

5.5.3 GUI

La interfaz gráfica de usuario o GUI (*Graphical User Interface*) se ha realizado mediante un módulo LCD de 16x2 caracteres. El módulo consta de los siguientes pines de conexión:

- **Vo:** controla el contraste y el brillo de la pantalla LCD. Usando un simple divisor de voltaje con un potenciómetro podemos hacer ajustes finos del contraste.
- **RS (Register Select):** cuando el pin RS está configurado en BAJO, entonces estamos enviando comandos a la pantalla LCD (como colocar el cursor en una ubicación específica, borrar la pantalla...). Y cuando el pin RS se establece en ALTO, estamos enviando datos/caracteres a la pantalla.

- **R/W (Read/Write):** Con este pin seleccionamos si queremos leer datos del módulo (ALTO) o escribir datos en el módulo (BAJO). Dado que solo estamos usando esta pantalla LCD como un dispositivo de salida, vamos a fijar este pin a tierra. Esto lo obliga a entrar en el modo escritura.
- **E (Enable):** utilizado para habilitar la pantalla. Es decir, cuando este pin se establece en BAJO, a la pantalla LCD no le importa lo que esté sucediendo con R/W, RS y las líneas del bus de datos; cuando este pin se establece en ALTO, la pantalla LCD está procesando los datos entrantes.
- **D0-D7 (Data Bus):** pines que llevan los datos de 8 bits que enviamos a la pantalla.
- **A-K (Anode & Cathode):** se utiliza para controlar la luz de fondo de la pantalla LCD.

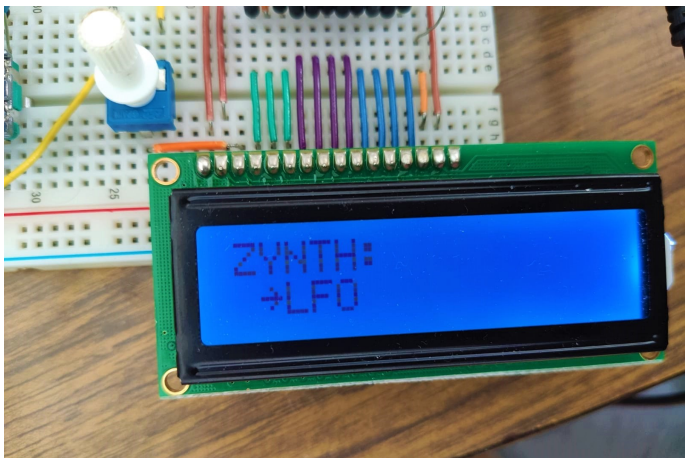

Figura 5.16 Módulo LCD. En la fila de arriba se muestra el nombre del menú actual. En la fila de abajo se van mostrando los submenús de los que se compone dicho menú actual. Zynth es el nombre del menú principal..

La UI está formada por un conjunto de menús y submenús. Con la rotación del encoder nos moveremos por los submenús y con el botón entraremos dentro del submenú que aparece en pantalla. El submenú seleccionado pasará a ser el menú actual y mostrará a su vez submenús internos, formando una estructura ramificada. Todos los menús muestran la opción "Atrás", donde se volverá al menú anterior. Llegará un momento donde la UI nos pida insertar el valor de un parámetro, en este caso seleccionaremos el valor deseado y pulsaremos el botón del encoder. En ese momento quedará reflejado instantáneamente el cambio tal y como veremos más adelante en la sección de configuración de notas.

Uno de los requisitos para que el contraste en las letras sea suficiente como para verlas, es que la diferencia de tensión entre el pin AK y el pin Vo debe ser de al menos 5V. Esto supone un problema en nuestro diseño ya que Zynq trabaja con voltajes de 3.3V, por lo que necesitaríamos que el pin Vo tuviera un valor de al menos -2V. Para conseguir esta tensión negativa recurriremos al circuito mostrado en la figura 5.17. El circuito toma como entrada una señal cuadrada generada por un Arduino (de 0-5V) y genera a su salida una tensión negativa igual a la diferencia de voltajes de la entrada. En este caso la señal de salida será, teóricamente, de -5V, aunque en la práctica sera menor debido a la caída de tensión resultante de la polarización de los diodos.

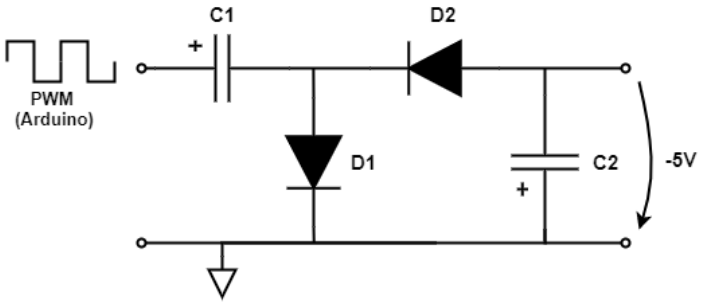

Figura 5.17 Circuito generador de voltaje negativo a partir de una señal PWM con un ciclo del 50 %.

Controlador LCD

El controlador que se encuentra en el módulo LCD está basado en el famoso chip Hitachi HD44780 [28], desarrollado en la década de 1980. En su interior hay una memoria ROM con la información (píxeles) de todos los caracteres disponibles y una memoria RAM donde se almacenan los caracteres que se muestran en la pantalla. Además posee dos registros:

- Data Register (DR)
- Instruction Register (IR)

Con el pin de entrada RS seleccionaremos qué registro queremos usar para leer o escribir en él. Existen dos tipos de interfaces con el controlador: bus de datos de 4 bits o de 8 bits. Un bus de 4 bits usa menos pines pero añade dificultad al diseño. En este trabajo se usa un bus de 8 bits, ya que tenemos pines GPIO de sobra en nuestra placa.

El registro de instrucciones (IR), como su propio nombre indica, servirá para enviar comandos al controlador como borrar la pantalla, mover el cursor, activar y desactivar el cursor. Cuando escribamos algún valor en los registros, deberemos mantenerlo durante un tiempo determinado para que le dé tiempo al controlador a leerlo y procesarlo (el tiempo mínimo necesario para cada instrucción aparece en el datasheet).

El registro de datos (DR) se utiliza como un lugar de almacenamiento temporal de datos para escribir o leer desde la memoria RAM. En dicha memoria escribiremos la dirección de la memoria ROM donde se halla el carácter que queramos representar. Con cada escritura la dirección de la memoria RAM (esto es, el cursor) aumenta.

El LCD esta controlado por el PS, por lo que se ha escrito una librería para manejar la realizar la configuración y la escritura en la pantalla. La implementación puede verse en los archivos *lcd.h* y *lcd.c* dentro del repositorio.

5.6 Configuración del sistema

La configuración del sistema, orquestada por el PS, se divide en dos partes:

1. *Configuración de las notas:* el PS se encargará de enviar al PL, mediante el bus AXI, las señales necesarias de configuración. Estas señales reflejarán los parámetros de configuración que hayamos seleccionado mediante la interfaz gráfica.

2. *Configuración del codec:* antes de poder generar las notas, durante la fase de inicialización del sistema (donde inicializaremos los drivers y configuraremos los recursos del hardware), se debe llevar a cabo la configuración del codec para poder utilizarlo.

Veamos en más detalle como están implementadas estas dos funcionalidades:

5.6.1 Configuración de las notas

Existen un total de 13 parámetros de configuración que se reparten en los distintos módulos del PL. Estos parámetros se hallan almacenados en unos registros que se encuentran en el interior de una IP que ha sido creada con ayuda de la funcionalidad *IP Packager* de Vivado. *IP Packager* nos permite crear IPs que se comuniquen mediante AXI sin que tengamos que preocuparnos por programar la interfaz AXI nosotros mismos. Para crearlas necesitaremos indicar que puerto AXI queremos usar y el número de registros que deseamos. En nuestro caso usaremos el puerto *M_AXI_GP0*, M quiere decir que el PS sera el maestro y GP que el puerto es de propósito general (*General Purpose*), a diferencia de los puertos HP de alto rendimiento (*High Performance*). El programa creará automáticamente una interfaz AXI dentro de la IP y un driver (un archivo .h) con las direcciones de los registros y las funciones necesarias para escribir o leer de ellos desde el PS. Los diseñadores pueden incluir su propia lógica dentro de la IP creada.

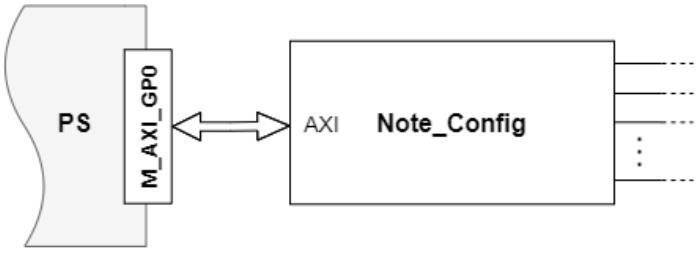

Figura 5.18 Módulo Note_Config, creado mediante *IP Packager*..

Cada vez que se introduzca un parámetro (esto es, justo cuando pulsemos el botón del encoder), se escribirá dicho valor en el registro adecuado dentro de la IP que acabamos de mencionar. Los registros internos de dicha IP están directamente conectados con las entradas de configuración de los distintos módulos del PL, por lo que el cambio será reflejado instantáneamente.

5.6.2 Configuración del codec

La configuración del codec se realiza a través del puerto de control del mismo mediante el protocolo I2C (tambíen puede configurarse para trabajar con SPI). Para el proyecto se ha escrito un driver de I2C desde cero, se trata de un driver sencillo pero que cumple con todas las funcionalidades que deseamos para el proyecto.

Dicho driver se compone, por un lado, de la definición de los registros: definimos punteros para cada uno de los registros de los que se compone el periférico, asegurándonos de que apunten a la dirección de memoria correcta.

Por otro lado se encuentran las funciones con las que interactuaremos: existe una función de inicialización, que lo que hará será poner a punto el módulo I2C, habilitándolo y

configurándolo para el tipo de conexión que deseamos; y una función de transmisión, donde le pasaremos una variable que contiene el mensaje y la dirección, y esta función se encargará de cargar la información en el buffer del periférico y enviarla. El tipo de conexión será la siguiente:

- Modo maestro: será el PS el encargado de iniciar la conexión y de controlar la señal de reloj. El codec será el esclavo, adoptando una actitud "pasiva" en la comunicación.
- Baudrate de $100kHz$: esto quiere decir que se enviará 1 bit cada 10 microsegundos. 100kHz suele ser la velocidad normal aunque el estándar de I2C soporta velocidades de hasta 5MHz. En nuestro caso, $100kHz$ es más que suficiente ya que solo se realizará una comunicación y será durante la etapa de inicialización del sistema.
- Direcciones de 7 bits. El módulo también soporta direcciones de 10 bits.

En la inicialización habilitaremos la interrupción de transferencia completa. Esta interrupción no llegará al GIC, solo se usará dentro de la función de transmisión para asegurar que se han enviado todos los bytes antes de salir de la función. Para ver la implementación del driver se puede acudir a los archivos *i2c.h* (para ver la definición de los registros y la definición de la variable de mensaje) e *i2c.c* (para ver la implementación de las funciones de inicialización y transmisión).

Una vez preparado el driver ya podremos comunicarnos con el codec. Los pasos a seguir para su configuración son los siguientes:

1. *Aplicar energía al codec.* Esto se hace automáticamente cuando conectamos la placa a la corriente. El chip dispone de un *Power-on-Reset*, que es un circuito que detecta la potencia aplicada al chip y genera una señal de reseteo que circula por todo el chip, colocando todos los registros en un estado conocido.
2. *Configurar el reloj interno.* Antes de poder realizar ninguna escritura o lectura de los registros internos del codec es necesario configurar su reloj. Para ello tenemos dos opciones: la primera consiste en usar su PLL interno, para conseguir la frecuencia deseada a partir de una señal de reloj cualquiera; la segunda opción consiste en proporcionar directamente al chip la señal de reloj que queramos que use internamente. La elección de la frecuencia de reloj es una decisión importante ya que está relacionada directamente con la frecuencia de muestreo de audio que queramos usar. En nuestro caso, se decidió no usar el PLL e introducir en el chip una señal de reloj generada desde Zynq[5]. Esta señal tendrá una frecuencia de $12.288MHz$, esto es así porque el codec divide dicha señal entre 256 para obtener la frecuencia de muestreo, consiguiendo de esta manera una frecuencia de muestreo de $48kHz$ que es la que buscábamos.
3. *Habilitar el reloj interno y cargar la configuración en los registros.* El ADAU1761 se compone de 66 registros de 8 bits, la dirección de memoria de cada registro es un número de 16 bits. Para poder escribir en cada registro deberemos, en la conexión I2C, escribir primero la dirección del registro (como la dirección se compone de 2 bytes, enviamos primero el byte superior y luego el inferior) y luego el valor que

[5] Lo que estamos haciendo aquí es usar el PLL de Zynq en vez de el PLL del codec. Generar la señal de reloj desde Zynq nos brinda una mayor flexibilidad y control, además de que usaremos dicha señal no solo para el codec, sino para todos los módulos del PL, obteniendo de esta manera todo un sistema sincronizado por el mismo reloj. Esto último es importante ya que, por ejemplo, el transmisor I2S necesita estar sincronizado con el codec para poder enviar datos de audio.

queramos introducir en el registro. Después de cada escritura el codec aumentará automáticamente la dirección de memoria, por lo que podremos escribir una serie de registros consecutivos, uno detrás de otro, sin que tengamos que cerrar la conexión y volver a abrirla inmediatamente después. La configuración de los registros dependerá obviamente del tipo de aplicación, para conocer la función de cada registro será necesario acudir al datasheet del chip.

No lo hemos mencionado antes, pero en las comunicaciones con I2C, el primer byte es siempre la dirección del dispositivo al que nos queremos dirigir. La dirección del codec es un número de 7 bits, cinco de los cuales son fijos y determinados, mientras que los otros dos están disponibles a través de dos de los pines de salida del codec. De esta forma, antes de poder dirigirnos al codec debemos realizar una lectura de dichos pines para poder formar la dirección completa.

5.7 Control MIDI

Las señales MIDI serán recibidas a través de un PC mediante protocolo UART, usándose como medio de comunicación el puerto JTAG encargado de la programación y depuración del dispositivo. El PC tiene varias opciones a la hora de generar dichas señales MIDI:

- *A través de un fichero MIDI.* Este tipo de ficheros llevan grabados, de acuerdo al formato estándar MIDI (SMF), una composición musical, como si de una partitura se tratara. En este caso el sintetizador actúa como un interprete de la obra musical.
- *A través de un teclado musical.* Aquí el interprete sería el humano que está tocando el teclado. Lo más normal es que este tipo de teclados se conecte al PC mediante puerto USB.
- *A través de un software ejecutándose en el PC.* Un software musical puede producir notas MIDI de muchas maneras, usando secuenciadores, loopers o incluso a través del teclado de escritura.

En cualquiera de los casos será necesario un "puente" entre el PC y el sintetizador. Esto es, necesitamos redireccionar las señales MIDI hacia el sintetizador. Para ello usaremos un programa ligero y de código libre llamado *SerialMidi*, donde seleccionaremos la entrada MIDI y la salida serie, y este se encargará de redireccionar la señal haciendo las transformaciones necesarias. Se ha elegido la mayor velocidad de transmisión permitida en el programa, 115200 bits por segundo, con la intención de reducir al máximo la latencia.

En el lado de la Zynq, deberemos configurar el módulo UART para que trabaje a la misma velocidad que el PC (en conexiones asíncronas es necesario que tanto emisor como receptor acuerden una velocidad fija) y para que genere una interrupción cada vez que el buffer de entrada tenga 3 bytes. El motivo de esto último es porque, como dijimos en la sección 4.3, los mensajes MIDI estándar se componen de 3 bytes (1 byte de estado y 2 bytes de datos).

Cuando se produce la interrupción, la rutina de servicio leerá los datos del buffer y los pasará a una función de decodificación. Esta función lo que hará sera transformar el valor de la nota a su valor de FTW correspondiente (recordemos que FTW es un número que usan los NCOs para variar la frecuencia de la señal generada). Estos valores serán calculados previamente y almacenados en una tabla en memoria, de manera que el valor

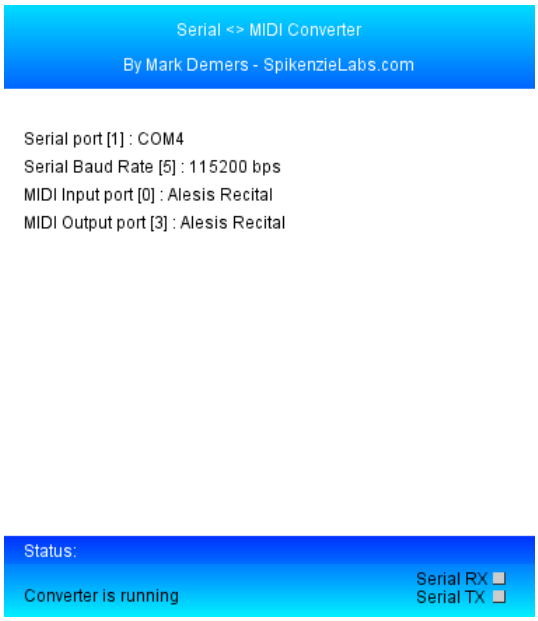

Figura 5.19 Vista del programa *SerialMidi*. COM4 es el puerto donde tenemos conectado el sintetizador y Alesis Recital es el nombre del teclado musical. El teclado aparece tanto en el puerto MIDI de entrada como el de salida, aunque el de salida no lo usaremos porque el sintetizador no genera ninguna señal MIDI..

de la nota, que será un número del 21 al 108 (ver figura 4.5), servirá como un índice para dicha tabla. Si el mensaje obtenido es de NOTEON, entonces el valor de FTW se asignará a un generador de notas (esto se hará de manera cíclica, de forma que cada nota nueva que se pulse se asigne a un nuevo generador de notas). Para ello escribirá, mediante AXI, en dos registros que se hallan en una IP implementada en el PL. Esta IP—que es el controlador de notas—contiene 9 registros internos, a los cuales podremos acceder desde el PS. 8 de ellos serán los valores de FTW de cada uno de los generadores de notas, mientras que el otro que queda será un registro donde cada bit indica si la nota se encuentra pulsada o no.

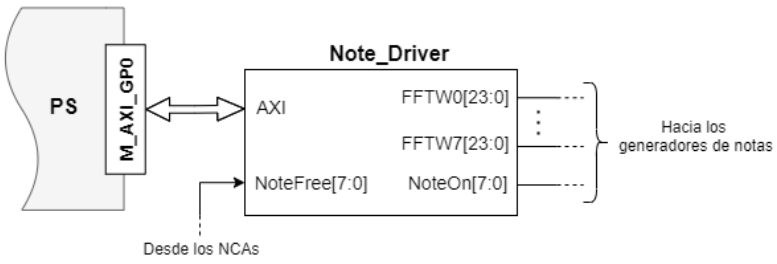

Figura 5.20 Módulo Note_Driver.

Veámoslo con un ejemplo: imaginemos que tocamos la nota DO en el teclado musical, generando una señal MIDI que viajará desde el teclado hasta el PC (mediante USB) y desde el PC hasta el sintetizador (mediante UART). Cuando la señal llegue se producirá una interrupción en el PS, la rutina de interrupción entonces leerá el valor de la nota (para el caso del DO central, este valor será de 60) y en función de si el mensaje es de NOTEON o de NOTEOFF se hará lo siguiente:

- **NOTEON:** esto querrá decir que acabamos de pulsar la nota. En este caso, lo primero que se hace es hallar el valor de FTW que produciría dicha nota. Estos valores han sido calculados de antemano por lo que solo necesitaremos consultar una tabla usando como índice el número 60. Una vez conocido FTW se escogerá un generador que produzca dicha nota. Hay 8 generadores y estos se van usando uno detrás de otro de forma cíclica, de manera que si el último en usarse fue el número 8, el siguiente será el 1. Una vez asignado un generador se escribirá dentro del controlador de notas el valor de FTW, en el registro asociado al generador escogido; también escribiremos un 1 en el bit correspondiente del registro de estado de las notas, indicando que la nota ha sido pulsada. El controlador de notas detectará si el generador que se quiere usar está libre o no y en función de esto aceptará o deshechará la petición. Por último guardaremos en memoria el valor de la nota junto con el generador asignado, para cuando se produzca la señal de NOTEOFF sepamos cual es el generador que debe parar.
- **NOTEOFF:** se recibirá este mensaje cuando dejemos de pulsar la nota. En este caso lo que haremos será buscar cual es el generador que estaba asociado a dicha nota y poner a cero el bit en el registro de estado de las notas correspondiente a dicho generador. Esto indicará al NCA que se ha dejado de pulsar la nota y que debe proceder a la fase de RELEASE. Cuando termine la fase de RELEASE, el NCA pondrá a 1 su salida NoteFree, indicando que la amplitud de la nota vale cero y que el par generador + NCA queda libre para poder ser utilizado por una nueva nota entrante.

6 Resultados

Dedicaremos este breve capítulo final a repasar los hitos logrados en el proyecto y a comentar como podría ampliarse el alcance del mismo y cuales serían las mejoras que se podrían incorporar.

Una de las razones que motivaron la elaboración de este proyecto era aprender a trabajar con sistemas basados en SoC, conocer las metodologías de diseño involucradas y manejar las herramientas necesarias. En lo que a esto se refiere, se ha realizado una gran labor (véase el capítulo 3) en aprender sobre una materia que puede ser desconocida para un alumno que termina sus estudios de GIERM.

A lo largo del desarrollo del proyecto se ha hecho especial hincapié en escribir código que fuera lo más eficiente posible. En este aspecto estoy muy contento con los resultados. Una de las pruebas de este esfuerzo ha sido que, en la parte de diseño digital, no se ha generado ningún *latch* en la síntesis (los *latches* son fruto de malas praxis de diseño y deben evitarse).

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	9632	0	53200	18.11
LUT as Logic	9566	0	53200	17.98
LUT as Memory	66	0	17400	0.38
LUT as Distributed RAM	0	0		
LUT as Shift Register	66	0		
Slice Registers	5498	0	106400	5.17
Register as Flip Flop	5498	0	106400	5.17
Register as Latch	0	0	106400	0.00
F7 Muxes	34	0	26600	0.13
F8 Muxes	0	0	13300	0.00

Figura 6.1 Reporte de utilización de la lógica programable.

En las figuras 6.2 y 6.3 se muestran la matrices de verificación de requisitos de acuerdo a [29], siendo los métodos de verificación:

- **Inspección (I):** Examinando de manera no destructiva, puede incluir medidas simples (por ejemplo observando en el osciloscopio).

- **Análisis (A):** Usando modelos o interpretando resultados (por ejemplo por simulación).
- **Demostración (D):** Observación del item en funcionamiento.
- **Test (T):** Evaluación o ejecución del item bajo condiciones, configuraciones, entradas, etc., controladas.

Req.	Nombre Req.	Verificación				Prueba
		I	A	D	T	
F.1	Generación de ondas periódicas	X	X		X	Observación en osciloscopio y en simulación (figura 5.3).
F.2	Ondas configurables en amplitud	X	X	X	X	Observación en osciloscopio y en simulación de notas a distintas amplitudes. Se han realizado pruebas de variación de amplitud (fig. 5.3) Además puede percibirse el efecto sonoro.
F.3	Ondas configurables en frecuencia	X	X	X	X	Observación en osciloscopio y en simulación (figura 5.3). Se ha comprobado mediante medidas en el osciloscopio que la frecuencia resultante es la deseada. Además es posible demostrar la frecuencia mediante referencias externas.
F.4	Generación de armónicos	X	X		X	Se han realizado tests de variación de los armónicos. Observación en osciloscopio y simulación (fig. 5.4).
F.5	Modulación en amplitud mediante LFO	X	X	X	X	Se han realizado tests variando los parametros de LFO y observando en osciloscopio y en simulación (fig. 5.6). El efecto sonoro producido puede demostrarse.
F.6	Generación de envolvente acústica	X	X	X	X	Se han realizado tests variando los parametros ADSR y observando en osciloscopio (fig. 5.7) y en simulación (fig. 5.8). El efecto sonoro producido puede demostrarse.
F.7	Mezcla de notas			X		Se han realizado pruebas de interpretación y el resultado armónico es el esperado.
F.8	Recepción de comandos MIDI				X	Se ha comprobado mediante debugging la correcta recepción y decodificación de los comandos recibidos.

Figura 6.2 Matriz de verificación de requisitos funcionales.

6.1 Conclusiones y futuro trabajo

Se ha visto como el uso de una FPGA puede resultar beneficioso en sistemas de audio polifónicos para la generación y procesamiento en paralelo del sonido. De esta manera conseguimos reducir al máximo la latencia, la cual es el principal enemigo en instrumentos digitales, ya que una latencia alta (mayor de 20ms) es percibida por el músico dificultando (o imposibilitando) la interpretación. En este tipo de diseños el procesador quedaría relegado a tareas de gestión, organización del sistema, comunicación con el exterior, etc. Incluso podría llegar a ejecutar un sistema operativo, aumentando las posibilidades de interfaz con el usuario y de configuración. Dicho de otro modo, tal y como vimos en el capítulo anterior, el procesador sería la unidad de control, mientras que la FPGA se encargaría del flujo de datos.

Req.	Nombre Req.	Verificación				Prueba
		I	A	D	T	
P.1.1	Profundidad de audio de 16 bits		X			Todo el sistema ha sido diseñado en 16 bits. En la fig. 5.3 puede verse como el valor del seno cuando está arriba es de 32628 ≈ (2^16)/2
P.1.2	Frecuencia de muestreo de 48kHz		X			El CODEC ha sido configurado de tal manera. No existe una prueba que pueda verificarlo.
P.2	256 niveles de amplitud de onda		X		X	El parámetro que controla la amplitud es de 8 bits. (ver Amp[7:0] en fig. 5.3).
P.3	Frecuencia de ondas en el rango audible	X	X	X	X	La frecuencia máxima esta limitada por el número de puntos que hayamos escogido para la plantilla del seno. Esta cantidad ha sido diseñada para tener un límite teórico de más de 20kHz. Se ha comprobado que el sintetizador responde adecuadamente a las 88 notas del teclado (desde 27.5Hz hasta 4186Hz)
P.4	Arm. con mismas prest. que ondas fund.		X	X	X	Igual que P.1-3
P.5	LFO con mismas prest. que ondas fund.		X	X	X	Igual que P.1-3
P.6	Parámetros de conf. de ADSR de 4 bits		X			Ha sido diseñado de tal manera.
P.7	Polifonía de 8 voces			X	X	Ha sido comprobado pulsando 8 notas en el teclado.
P.8.1	Comandos MIDI a través de UART			X		Ha sido diseñado de tal manera.
P.8.2	Comandos disponibles: NOTEON y NOTEOFF			X		Ha sido diseñado de tal manera.
O.1	Configuración mediante HMI (LCD + encoder)			X	X	Se han realizado pruebas en funcionamiento para validar el adecuado comportamiento.
O.2	Salida de audio por auriculares			X		Ha sido diseñado de tal manera.

Figura 6.3 Matriz de verificación de requisitos de prestaciones y operativos.

Las capacidades de las FPGAs las convierten en una plataforma muy atractiva para la implementación de algoritmos de procesamiento de la señal. Una clara ampliación del proyecto sería añadir filtros DSP para producir síntesis sustractiva. Además de filtros se pueden añadir otro tipo de efectos dentro de la cadena de sonido como reverbs, chorus, delays...

Por otro lado, sería sencillo añadir más sonidos mediante la técnica *wavetable*. Para ello solo tendríamos que coger un periodo de onda del sonido que queramos capturar (un piano, una guitarra, un saxo...) e introducirlo en una memoría ROM tal y como hicimos con la señal sinusoidal. La incorporación de nuevos sonidos y de filtros y efectos adicionales sería posible ya que aún quedan recursos disponibles dentro de la Zynq.

Zynth podría mejorarse añadiendo más funcionalidades, como la posibilidad de guardar presets personalizados en la memoria. También sería posible mejorarse la interfaz humano-máquina, diseñando una PCB dedicada a ello o usando pantallas más modernas para mejorar la interfaz gráfica.

Otro aspecto interesante que podría incluirse en el proyecto, siguiendo con la metodología de buenas prácticas, sería la implementación de un sistema de integración continua para automatizar tareas de compilado y validación.

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	12	0	140	8.57
RAMB36/FIFO*	0	0	140	0.00
RAMB18	24	0	280	8.57
RAMB18E1 only	24			

Figura 6.4 Reporte de utilización de los bloques de RAM.

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	40	0	220	18.18
DSP48E1 only	40			

Figura 6.5 Reporte de utilización de los slice DSP.

Referencias

[1] S. Simms (2018), *The Future of Electronic Music Performance* [Online]. Accedido el 24 de mayo de 2021, dirección: https://ask.audio/articles/the-future-of-electronic-music-performance

[2] C.M. Colonna, P.M. Kearns and J.E. Anderson, "Electronically produced music and its economic effects on the performing musician and music industry", J Cult Econ 17, pp. 69–75, 1993. DOI: https://doi.org/10.1007/BF02310583

[3] Mathews, M. V. "The Digital Computer as a Musical Instrument." Science, vol. 142, no. 3592, pp. 553–557, 1963.

[4] Sononews (2018), *El primer sintetizador* [Online].
Accedido el 24 de mayo de 2021, dirección: https://sononews.com/rca-mark-i

[5] Wikipedia, *Sintetizador Moog* [Online].
Accedido el 24 de mayo de 2021, dirección: https://es.wikipedia.org/wiki/Sintetizador_moog

[6] Wikipedia, *Minimoog* [Online].
Accedido el 24 de mayo de 2021, dirección: https://es.wikipedia.org/wiki/Minimoog

[7] Wikipedia, *EMS VCS3* [Online].
Accedido el 24 de mayo de 2021, dirección: https://en.wikipedia.org/wiki/EMS_VCS_3

[8] Vintage Synth, *Sequential Circuits Prophet 5* [Online].
Accedido el 24 de mayo de 2021, dirección: http://www.vintagesynth.com/sci/p5.php

[9] Wikipedia, *Roland Juno-60* [Online].
Accedido el 24 de mayo de 2021, dirección: https://en.wikipedia.org/wiki/Roland_Juno-60

[10] Wikipedia, *Yamaha DX7* [Online].
Accedido el 24 de mayo de 2021, dirección: https://en.wikipedia.org/wiki/Yamaha_DX7

[11] Wikipedia, *Frequency modulation synthesis* [Online].
Accedido el 24 de mayo de 2021, dirección: https://en.wikipedia.org/wiki/

Frequency_modulation_synthesis

[12] Xilinx, "Zynq-7000 SoC Technical Reference Manual", Aug. 2012 [Revised Apr. 2021].

[13] Xilinx, "Zynq-7000 SoC Product Selection Guide" [Online].
Accedido el 24 de mayo de 2021, dirección: https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf

[14] URL: http://www.pynq.io. Accedido el 24 de mayo de 2021.

[15] URL: https://www.tul.com.tw/ProductsPYNQ-Z2.html. Accedido el 24 de mayo de 2021.

[16] URL: https://www.tul.com.tw/ProductsPYNQ-Z2.html. Accedido el 24 de mayo de 2021.

[17] Analog Devices, "ADAU1761", Jan. 2009 [Revised Oct. 2018].

[18] Xilinx, "Boot Image Layout" in *Vitis Embedded Software Development Flow Documentation* [Online].
Accedido el 24 de mayo de 2021, dirección: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/bootimagelayout.html

[19] M. Joindot, L. Libois, G. Phelizon, J. Colin and P. Dupuis, "The predigital period (1937–1965) in Europe," 2008 IEEE History of Telecommunications Conference, 2008, pp. 60-65, doi: 10.1109/HISTELCON.2008.4668716.

[20] T. Fine (2008) "The Dawn of Commercial Digital Recording". ARSC Journal. Accedido el 24 de mayo de 2021, dirección: https://www.aes.org/aeshc/pdf/fine_dawn-of-digital.pdf)

[21] *AES recommended practice for professional digital audio - Preferred sampling frequencies for applications employing pulse-code modulation (revision of AES5-2003)*, AES standard AES5-2018, Dec. 2018.

[22] E. Miranda. *Computer sound design: synthesis techniques and programming*. Taylor and Francis, 2012.

[23] V. F. Kroupa, *Direct Digital Frequency Synthesizers*, IEEE Press, 1999, ISBN 0-7803-3438-8

[24] D. Smith, and C. Wood, "The ÚSI', or Universal Synthesizer Interface.", *Journal of The Audio Engineering Society*, Oct. 1981.

[25] *The Complete MIDI 1.0 Detailed Specification,* MIDI Association, Aug. 1983.

[26] ZipCPU (2017), *Bit growth in FPGA arithmetic* [Online].
Accedido el 24 de mayo de 2021, dirección: https://zipcpu.com/dsp/2017/07/21/bit-growth.html

[27] Philips Semiconductors, "Especificación del estandard I2S de Philips", NXP, Jun. 1996.

[28] Hitachi, "HD44780U: Dot Matrix Liquid Crystal Display Controller/Driver".
Accedido el 24 de mayo de 2021, dirección: https://www.sparkfun.com/datasheets/LCD/HD44780.pdf

[29] H. Guzmán, "Tema 4: Extracción de requisitos", *Proyectos Integrados*, Universidad de Sevilla [Online].
Accedido el 24 de mayo de 2021, dirección: http://woden.us.es/docs/docencia/PI3GIERM/04_Extraccion_de_Requisitos.pdf

Índice de Figuras