# Intérprete hardware de archivos MIDI (Piano Digital)

## MIDI File Hardware Player (Digital Piano)

Fernando Candelario Herrero

---

Trabajo de Fin de Grado

GRADO EN INGENIERÍA DE COMPUTADORES
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Madrid, junio de 2020

Directores:

José Manuel Mendías Cuadros
Carlos González Calvo

# Autorización de difusión

El abajo firmante, matriculado en el Grado de Ingeniería de Computadores de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "Intérprete hardware de archivos MIDI (Piano Digital)", realizado durante el curso académico 2019-2020 bajo la dirección de Carlos González Calvo y José Manuel Mendías Cuadros del Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fernando Candelario Herrero

Madrid, a 15 de junio de 2020

Esta obra está bajo una
Licencia Creative Commons
Atribución-NoComercial-CompartirIgual 4.0 Internacional.

# Agradecimientos

En primer lugar quiero agradecer a mis tutores Mendi y Carlos por darme la oportunidad de realizar este proyecto que tenía en mente desde hace unos años. Gracias a su constante apoyo he podido cumplir este hito personal.

También quiero agradecer a todos los compañeros y amigos con los que hemos formado alegres recuerdos durante estos locos años universitarios.

Por último quiero agradecer a mis padres por haberme dado su incondicional apoyo desde el día que nací, impidiendo que me ahogue en mis propios miedos sacando siempre lo feliz y alegre de cada día.

*Así, hoy vuela hacia las estrellas el héroe de esta edad. Vuela a través de regiones antes ignoradas.*
*Vuela hacia afuera de su mundo y, sin saberlo, va impulsado hasta el interno y luminoso centro. [1]*

# Índice general

**Índice general** I

**Índice de figuras** VI

**Índice de tablas** X

**Resumen** XI

**Abstract** XII

**1. Introducción** 2

1.1. Motivación . . . . . 2

1.2. Objetivos . . . . . 3

1.3. Antecedentes . . . . . 3

1.4. Plan de trabajo . . . . . 8

1.5. Organización de la memoria . . . . . 9

**2. Contexto tecnológico del proyecto** 12

2.1. Tecnología de la FPGA, placa Nexys 4 DDR y módulos Pmod . . . . . 12

2.2. El estándar MIDI . . . . . 16

2.2.1. Estructura de la cabecera . . . . . 18

2.2.2. Estructura de las pistas . . . . . 20

2.3. Síntesis de sonido mediante *Wavetable* . . . . . 26

2.3.1. Archivos .WAV, uso como una *Wavetable* . . . . . 26

2.3.2. Interpolación lineal para la generación de notas musicales . . . . . 29

2.4. Efecto Sustain de una nota . . . . . 33

2.5. Efecto *reverb* . . . . . 35
2.6. Aritmética en punto fijo . . . . . 36
2.7. Protocolo I2S . . . . . 38
2.8. Protocolo SPI . . . . . 39
2.9. Bluethooth y RS-232 . . . . . 41
2.10. Modulación PWM . . . . . 42
2.11. Herramientas de desarrollo . . . . . 43
2.11.1. Consola linux, sublime text 3 y C . . . . . 43
2.11.2. Vivado y VHDL . . . . . 44
2.11.3. Android Studio y Java . . . . . 45

**3. Aplicaciones auxiliares de consola** **49**
3.1. Midi Parser . . . . . 49
3.1.1. Características de los archivos MIDI soportados . . . . . 49
3.1.2. Interpretación correcta de los valores numéricos representados en formato de longitud variable . . . . . 51
3.1.3. Implementación del *Running Status* . . . . . 52
3.1.4. Conclusiones Midi Parser . . . . . 54
3.2. Note Conversor . . . . . 56
3.2.1. Detalles de implementación . . . . . 57
3.2.2. Conclusiones Note Conversor . . . . . 64

**4. Arquitectura *hardware* del piano digital, proyecto MIDI_Soc** **67**
4.1. Modos de funcionamiento, temporización y parametrización . . . . . 68
4.2. Visión general . . . . . 69
4.3. IP cores y su presencia en el diseño . . . . . 75
4.3.1. IP Clock Wizard y componente ClkGen . . . . . 75
4.3.2. IP MIG 7 y subcomponente ddr . . . . . 75

4.4. Mapeo de datos en memoria Flash y RAM . . . . . 80
4.5. Componente RamCntrl . . . . . 82
4.5.1. Visión general . . . . . 83
4.5.2. ¿Por qué un controlador de acceso a memoria? Naturaleza paralela del diseño . . . . . 86
4.5.3. Comandos de memoria, formato y funcionamiento . . . . . 87
4.5.4. Uso de memoria caché para la optimización de las peticiones de lectura de muestras y lógica de funcionamiento general . . . . . 91
4.5.5. Subcomponente Ram2Ddr . . . . . 93
4.6. Componente KeyboardCntrl . . . . . 94
4.6.1. Visión general . . . . . 94
4.6.2. Subcomponente UniversalNoteGen . . . . . 97
4.6.3. Subcomponente NotesGenerator . . . . . 103
4.6.4. Subcomponente Reverb . . . . . 106
4.6.5. Subcomponente CountGensOn . . . . . 108
4.6.6. Cálculo y mapeo de constantes en ROM, para la generación de audio 109
4.6.7. Mensajes MIDI soportados. Formato de los comandos de teclado . . . 111
4.7. Componente MidiParser . . . . . 115
4.7.1. Visión general . . . . . 115
4.7.2. Subcomponente MidiController . . . . . 117
4.7.3. Subcomponente ByteProvider . . . . . 121
4.7.4. Subcomponente ODBDProvider . . . . . 124
4.7.5. Subcomponente ReadHeaderChunk . . . . . 126
4.7.6. Subcomponente ReadTrackChunk y gestión del tempo . . . . . 130
4.7.7. Subcomponente TracksCmdSequencer . . . . . 135
4.8. Componente MainController . . . . . 137
4.9. Componentes de visualización . . . . . 140

4.9.1. Componente RgbLed . . . . . 140
4.9.2. Componente bin2segNexys4 . . . . . 141
4.10. Componentes para el procesamiento de señales externas . . . . . 143
4.10.1. Componente ButtonsSyncDebRiseEdge . . . . . 143
4.10.2. Componente resetSyncronizer . . . . . 144
4.11. Componente MySetup . . . . . 144
4.12. Componente ExternInterfaceCmdReceiver . . . . . 145
4.13. Componente CmdKeyboardSequencer . . . . . 148
4.14. Componente iisInterface_75Mhz . . . . . 149

**5. App Android MySoc, interfaz de uso del piano digital 153**
5.1. Funcionalidad y aspecto . . . . . 153
5.2. Aspectos destacables de la implementación . . . . . 156

**6. Conclusiones finales 160**
6.1. Resultados . . . . . 160
6.2. Consideraciones finales y conocimientos adquiridos . . . . . 162
6.3. Trabajo futuro . . . . . 165

**Bibliografía 172**

**A. Introduction 173**
A.1. Motivation . . . . . 173
A.2. Objetives . . . . . 174
A.3. Background . . . . . 174
A.4. Work schedule . . . . . 178
A.5. Document organization . . . . . 179

**B. Final Conclusions 181**
B.1. Results . . . . . 181

B.2. Final considerations and acquired knowledge . . . . . 183
B.3. Future work . . . . . 186

# Índice de figuras

1.1. Roland EP-10, primer piano electrónico comercializado por Roland . . . . . 4
1.2. Primer piano digital de la marca Casio, modelo CPS-201 . . . . . 4
1.3. Diseño de los primeros pianos digitales de la marca Roland, modelos HP-300 y HP-400 . . . . . 5
1.4. Modelo V3 Sound Sonority XXL de calidad profesional . . . . . 6
1.5. Modelo Miditech Pianobox Mini, gama de entrada . . . . . 6

2.1. Representación de un componente CLBs . . . . . 13
2.2. Representación de una matriz de interconexión en una FPGA . . . . . 14
2.3. Logotipo del protocolo MIDI . . . . . 16
2.4. Estructura general de un archivo MIDI . . . . . 18
2.5. Formato de los bits del campo division . . . . . 19
2.6. Algunos ejemplos de números representados en formato de longitud variable 21
2.7. Sintaxis de mensajes MIDI . . . . . 24
2.8. Representación del formato WAV . . . . . 27
2.9. Representación gráfica de la fórmula de interpolación . . . . . 30
2.10. Representación simplificada de una onda de sonido, diferenciando las etapas que la forman . . . . . 34
2.11. Diagrama de tiempo a seguir por las señales del protocolo I2S . . . . . 39
2.12. Diagrama de tiempo a seguir por las señales del protocolo SPIx2, ejemplificando la evolución de las señales debido al envío de un comando seguido de la dirección de inicio . . . . . 41

2.13. Diagrama de tiempo seguido por el protocolo RS-232, configurado a una velocidad de transmisión de 1200bps, 8 bits de datos, sin bit de paridad, y con un bit de stop . . . . . 42
2.14. Ejemplo de modulación PWM . . . . . 43
2.15. Logotipo de la aplicación Vivado . . . . . 45
2.16. Logotipo de SO Android . . . . . 46

3.1. Representación de organización de muestras en un archivo .wav . . . . . 57

4.1. Esquemático simplificado del diseño MIDI_Soc, muestra las principales interconexiones entre los componentes . . . . . 71
4.2. Organización de los bytes en memoria Ram, representando las direcciones en hexadecimal . . . . . 80
4.3. Esquemático simplificado del componente RamCntrl . . . . . 83
4.4. Formato de los Keyboard CMDs de solicitud de lectura, provenientes de KeyboardCntrl . . . . . 87
4.5. Formato de los Keyboard CMDs de respuesta a lectura, provenientes de KeyboardCntrl . . . . . 88
4.6. Formato de los MidiParser CMDs de solicitud de lectura . . . . . 88
4.7. Formato de los MidiParser CMDs de respuesta a lectura tipo A . . . . . 89
4.8. Formato de los MidiParser CMDs de respuesta a lectura tipo B . . . . . 90
4.9. Formato de los comandos de escritura . . . . . 90
4.10. Diagrama RTL que muestra el patrón de diseño seguido por las señales foundRow y rowIndex . . . . . 92
4.11. Esquemático simplificado del componente KeybordCntrl . . . . . 95
4.12. Diagrama RTL correspondiente a la ruta de datos utilizada para la síntesis de sonido mediante Wavetable y la aplicación de ajustes de intensidad . . . . 102

4.13. Diagrama RTL correspondiente a la topología utilizada para realizar las sumas de las muestras provenientes de los generadores de notas . . . . . 105
4.14. Esquemático simplificado de ReverbComponent . . . . . 106
4.15. Esquemático de la ruta de datos segmentada implementada en CountGensOn 109
4.16. Formato de los comandos de teclado recibidos por KeyboardCntrl . . . . . 112
4.17. Diagrama RTL que representa una posible implementación de las señales noteIndexOn y foundAviable . . . . . 114
4.18. Diagrama RTL que representa una posible implementación de las señales noteIndexOff y foundCode . . . . . 114
4.19. Esquemático del componente MidiParser, en el que se presentan las principales interconexiones entre los subcomponentes . . . . . 115
4.20. Diagrama RTL representando una posible implementación de las señales tmpFinish, tmpTracks y checkFinishTracksOk . . . . . 121
4.21. Esquemático simplificado del subcomponente ByteProvider . . . . . 124
4.22. Diagrama RTL correspondiente a la ruta de datos encargada de calcular el tiempo de espera en milisegundos por cada evento MTrk . . . . . 133
4.23. Esquemático simplificado del subcomponente TracksCmdSequencer . . . . . 137
4.24. Diagrama de tiempos asociado al patrón de onda a seguir para la iluminación de los *display* 7 segemntos de la placa Nexys 4 DDR . . . . . 142

5.1. Captura correspondiente a la ventana Connection, con la lista de dispositivos emparejados desplegada . . . . . 154
5.2. Captura correspondiente a la ventana File Transfer . . . . . 155
5.3. Captura correspondiente a la ventana Keyboard, mostrando las distintas opciones contenidas en el desplegable . . . . . 156

A.1. Roland EP-10, first electronic piano commercialized by Roland . . . . . 174
A.2. First digital piano from Casio, model CPS-201 . . . . . 175

A.3. First design of Roland digital pianos, models HP-300 y HP-400 . . . . . 175
A.4. Modelo V3 Sound Sonority XXL de calidad profesional . . . . . 176
A.5. Modelo Miditech Pianobox Mini, gama de entrada . . . . . 177

# Índice de tablas

2.1. Tabla de codificación entre notas y números que sigue el estándar MIDI . . . 26

2.2. Tabla de valores calculados por la función seno y la fórmula de interpolación, con el correspondiente cálculo del error cometido . . . . . 32

4.1. Tabla informativa sobre las variables genéricas del diseño . . . . . 69

4.2. Correlación entre etapas de lectura de la cabecera y estado encargado de realizar las acciones correspondientes a la etapa asociada . . . . . 129

4.3. Correlación entre acciones realizables por la interfaz y tramas de datos enviadas. Las letras $n$ y $v$ hacen referencia al código que representa una nota y a los 4 bits definen la intensidad de la nota respectivamente . . . . . 146

6.1. Tabla representando el grado de utilización de los distintos recursos de la FPGA C7A100T-1CSG324C . . . . . 162

B.1. Table which represents the different utilization degrees of the resources in the FPGA C7A100T-1CSG324C . . . . . 183

# Resumen

Este documento recoge el proceso de desarrollo de un intérprete de archivos MIDI implementado como una arquitectura puramente *hardware* sobre una FPGA. Para llevar a cabo dicha interpretación, el sistema es capaz de sintetizar y mezclar las notas correspondientes a cada una de las 88 teclas de un piano. El sonido se genera por una técnica basada en *Wavetables* que es capaz de reproducir con alta fidelidad el timbre propio de un piano, con una calidad de 48,8KHz de frecuencia de muestreo y 16 bits por muestra. El sistema también es capaz de aplicar un efecto *reverb* al sonido para dotarlo de mayor cuerpo. La arquitectura es altamente paralela y escalable tanto en el número máximo de pistas MIDI compatibles, como en el grado de polifonía (número de notas capaces de sonar al mismo tiempo).

Además, el sistema dispone de conectividad *Bluetooth* para que, desde una app Android también desarrollada dentro del proyecto, el intérprete MIDI pueda recibir los archivos a reproducir y que dicha reproducción pueda simultanearse con notas y acordes tocados por el usuario desde la misma app. La aplicación móvil también permite el inicio/detención de la reproducción de un archivo MIDI y la activación/desactivación del efecto *reverb*.

## Palabras clave

Estándar MIDI, Formato WAVE, Wavetable, Síntesis de sonido, FPGA, VHDL, C, Java, Bluetooth, Artimética en punto fijo.

# Abstract

This document explains the development process of a MIDI file interpreter which has been implemented as complete hardware architecture using an FPGA platform. In order to play MIDI files, this system can synthesis and mix the corresponding 88 notes of a piano. The sound synthesis is based on the use of Wavetables and thanks to it, it is possible to generate a high fidelity piano sound with quality of 48,8KHz and 16 bit per sample. This system is also able to apply a reverb effect to create a deeper sound. The system follows a high-parallel architecture and is also scalable both, in the maximum number of MIDI-compatible tracks, and in the polyphony degree (maximum number of notes that can be played at the same time).

Besides, the system has Bluetooth connectivity used to communicate with an Android app, which has also been developed in this project. Thanks to this app, the MIDI interpreter receives and plays MIDI files and at the same time, reproduces the notes and chords that the user is playing using the same app. The mobile application also allows the user to start/stop playing a MIDI file and start/stop de reverb effect.

## Keywords

MIDI Standard, WAVE Format, Wavetable, Sound Synthesis, FPGA, VHDL, C, Java, Bluethooth, Fixpoint arithmetic.

# Capítulo 1

# Introducción

## 1.1. Motivación

Desde los comienzos, la música en diferentes formas ha acompañado al ser humano a lo largo de la historia, sirviendo esta para su desarrollo tanto intelectual como espiritual. Gracias a la misma, el ser humano ha sido, y seguirá siendo, capaz de expresar y transmitir emociones imperecederas a través de las generaciones construyendo un rascacielos cultural que se extiende hasta nuestros días. La música, como todo en la vida, está en constante evolución, y los últimos cambios que ha sufrido dicho arte se deben en gran medida al desarrollo tecnológico de los últimos tiempos, la digitalización. Con ello la música se ha democratizado a todos los niveles. La digitalización ha supuesto un cambio de paradigma en el desarrollo de instrumentos musicales permitiendo una reducción tanto en costes como en espacio ocupado por el instrumento, el piano es un buen ejemplo de ello.

Desde que comencé a aprender piano, he sentido curiosidad de cómo es posible que los sistemas digitales que incluyen estos instrumentos son capaces de interpretar canciones y de generar un sonido tan nítido. Investigando el funcionamiento de estos instrumentos descubrí el estándar MIDI, estándar que siguen prácticamente la totalidad de pianos digitales que hay en el mercado. Así en el tercer año de universidad decidí que mi proyecto de fin de grado consistiría en desarrollar el diseño *hardware* de un piano digital capaz de reproducir archivos MIDI.

## 1.2. Objetivos

La intencionalidad de este proyecto consiste en crear un diseño *hardware* de código abierto para el desarrollo de un teclado polifónico digital capaz de reproducir archivos MIDI con alta fidelidad, pudiendo ser mejorado en el futuro. Con ello se pretende aprender sobre el estándar MIDI, la síntesis de sonido y la música.

Realizar una completa interpretación de un archivo MIDI no es trivial, debido principalmente a la gran cantidad de acciones que pueden ser representadas por ellos. Así este proyecto pretende servir como una base sólida para posteriormente ir mejorando la compatibilidad con estos archivos. A su vez, este proyecto pretende realizar una primera aproximación a la síntesis de sonido mediante *Wavetables*, técnica utilizada para generar sonido en los pianos digitales actuales.

El diseño *hardware* está pensado para su implementación en una FPGA, posibilitando la reconfiguración de los componentes del sistema. Este diseño se caracteriza por su modularidad y paralelismo, permitiendo la configuración del grado de polifonía y el número máximo de pistas a reproducir por el sistema. Estas características pretenden facilitar el desarrollo de futuras versiones por cualquiera que esté interesado.

## 1.3. Antecedentes

En el ámbito comercial, los primeros años de la década de los 70s supusieron el asentamiento definitivo de la tecnología de los pianos electrónicos, predecesores de los pianos digitales [2]. El modelo EP-10 por parte de la compañía japonesa Roland es un buen ejemplo de este hecho.

**Figura 1.1**: *Roland EP-10, primer piano electrónico comercializado por Roland* [2]

En la década posterior, con la reciente definición del estándar MIDI, comenzaron a aparecer los primeros pianos digitales. El uso del estándar MIDI, permitió la comunicación entre teclados desarrollados por distintos fabricantes lo que supuso una revolución en su momento. Algunos modelos pertenecientes a esta generación de pianos digitales son los Roland HP-300 y HP-400, y el Casio CPS-201.

**Figura 1.2**: *Primer piano digital de la marca Casio, modelo CPS-201* [3]

**Figura 1.3**: *Diseño de los primeros pianos digitales de la marca Roland, modelos HP-300 y HP-400*

[2]

Con el paso del tiempo, una gran variedad de marcas han ido incorporándose al mercado de los pianos digitales, algunas centrándose en ofrecer modelos con una buena relación calidad precio mientras otras se centran en ofrecer las mejores calidades tanto de sensación de pulsación como de sonido generado. Entre estos fabricantes encontramos a Yamaha, destacando su línea de productos Clavinova con los que se ganó una muy buena reputación. También encontramos a Kawai como otra de las compañías líderes en el desarrollo de pianos digitales.

Por otro lado, actualmente es posible encontrar sistemas que ofrecen una funcionalidad como caja de ritmos y como generador de sonido desligado del instrumento, llamados "Módulos de sonido GM (General MIDI)". El funcionamiento de estos módulos esta orientado a la recepción de mensajes MIDI y a la consecuente reproducción de las notas, pudiendo reproducir sonidos de distintos instrumentos según se configure. Estos mensajes MIDI pueden llegar al modulo a través de una conexión USB o MIDI, conectándose de esta forma a la interfaz instrumental deseada. La mayor ventaja de estos productos reside en su portabilidad

pudiendo transportar donde se requiera sonidos de muy buena calidad.

A continuación se muestran algunas imágenes de estos productos:

**Figura 1.4**: *Modelo V3 Sound Sonority XXL de calidad profesional* [4]

**Figura 1.5**: *Modelo Miditech Pianobox Mini, gama de entrada* [5]

A nivel académico, la mayoría de proyectos que podemos encontrar se centran exclusivamente en la síntesis de notas musicales, implementada en una plataforma *software* o *hardware*. A continuación, se describen algunos de ellos:

- Polyphonic FPGA Synthesizer [6], este proyecto consiste en la implementación de un teclado polifónico (hasta 4 notas) en una FPGA. La síntesis de sonido empleada en este proyecto es más simple que la llevada a cabo en este proyecto, consistiendo en la generación de una onda de sonido mediante modulación PWM.

- Digital VHDL Piano [7], este proyecto consiste en la generación de notas mediante el uso de un divisor de frecuencia y el posterior incremento de amplitud de dicha nota usando un circuito electrónico. El divisor de frecuencia y la interfaz de uso han sido implementado en una FPGA y el circuito electrónico ha sido desarrollado por el propio alumno.

- A-Simple-Electronic-Piano-on-FPGA [8], este proyecto implementa un teclado de piano utilizando un teclado PS2 como interfaz de uso del mismo y genera ondas cuadradas que estimulan un zumbador.

- Digital Keyboard [9], implementación de una aplicación python de escritorio en la que se muestra un piano digital, siguiendo principios universales de diseño.

No obstante se ha podido encontrar un par de proyectos de características similares. Uno de ellos perteneciente a la Universidad Politécnica de Madrid titulado como "Diseño e implementación de un sintetizador digital modular de audio" [10]. Dicho proyecto realiza un estudio y aplicación del estándar MIDI para la comunicación con la placa FPGA en la que se implementa el proyecto. La síntesis de sonido se realiza mediante la lectura de las muestras almacenadas en la FPGA (no usa memorias externas de tipo SDRAM o Flash), representando estas distintas formas de onda simples (triangulares, cuadradas, ...).

El segundo proyecto titulado como "Piano Digital" [11] pertenece a la Universitat Politécnica de Catalunya. Este proyecto consiste en la implementación de un sistema *hardware* en FPGA capaz de interpretar mensajes MIDI provenientes de un teclado, y consecuentemente realizar el envío de muestras de audio utilizando el protocolo S/PDIF. Las muestras de audio están almacenadas en la FPGA (tampoco hace uso de memorias externas) y provienen de formas de onda previamente calculadas.

## 1.4. Plan de trabajo

Para poder llevar a cabo este proyecto, se estructuró todo el trabajo a realizar en una serie de etapas con sus correspondientes fechas aproximadas de inicio y de finalización.

1. **Estudio y comprensión del estándar MIDI** (julio de 2019 a agosto de 2019). Desarrollo de una aplicación de consola que permita extraer mensajes de un archivo MIDI.

2. **Familiarizarse con la herramienta Xilinx Vivado 2018.2** (septiembre de 2019 a octubre de 2019.). Para ello se migró un diseño de cursos anteriores, originalmente realizado en la herramienta Xilinx ISE para la FPGA Spartan 3. Gracias a este ejercicio se pudo desarrollar con éxito el diseño del componente I2S encargado de enviar las muestras de audio, y el diseño del componente Ram2Ddr encargado de leer de manera eficiente una DDR2 SDRAM externa.

3. **Elaboración de una fórmula de interpolación para realizar síntesis de sonido mediante *Wavetable*** (octubre de 2019 a noviembre de 2019). Desarrollo de una aplicación de consola que permita contrastar la viabilidad y corrección del mecanismo de síntesis de sonido basado en interpolación de muestras de una *Wavetable*.

4. **Implementación en FPGA del piano digital, siguiendo un diseño modular** (diciembre de 2019 a enero de 2020). Se comenzó realizando el diseño del modulo que representa el teclado de un piano. Los momentos de mayor avance fueron, en las vacaciones de navidad con la realización de los primeros intentos de sintetizar sonido en tiempo real. Poco despues se terminó por decidir la arquitectura paralela del sistema mediante el envío de comandos a un controlador de memoria.

5. **Implementación en FPGA de un interprete de archivos MIDI, siguiendo un diseño modular** (enero de 2019 a marzo de 2020). Al piano digital se le añadió el modulo encargado de interpretar un archivo MIDI, habiéndose desarrollado con

ayuda de simulaciones. Al mismo tiempo se verifico el correcto funcionamiento de la arquitectura paralela.

6. **Desarrollo de una aplicación Android** (marzo de 2020 a abril de 2020). Esta aplicación realiza la función de interfaz externa para el uso del piano digital, mediante el establecimiento de una conexión *Bluetooth*. Gracias esta aplicación, se facilita la interacción con el piano digital por parte del usuario.

7. **Revisión y mejora tanto de la aplicación Android como del diseño *hardware* para el correcto funcionamiento de ambos** (abril de 2020 a mayo de 2020). En esta etapa se realizó limpieza de todo el código fuente y la parametrización del diseño *hardware*.

8. **Elaboración de la memoria de todo el proyecto** (mayo de 2020 a junio de 2020).

## 1.5. Organización de la memoria

A continuación, se procede a realizar una breve descripción de los capítulos que conforman esta memoria:

1. **Introducción**: Capítulo en el que se comentan las principales motivaciones y objetivos. A su vez se comentan brevemente algunos proyectos de similares características como el plan de trabajo seguido durante el desarrollo de este proyecto.

2. **Contexto tecnológico del proyecto**: Este capítulo aborda la explicación de las distintas herramientas y tecnologías utilizadas en este proyecto. La explicación sobre cómo se ha planteado la síntesis de sonido está incluida en la sección (§2.3) de este capítulo.

3. **Aplicaciones de consola**: Capítulo que aborda los detalles de implementación de las aplicaciones de consola Midi Parser y Note Conversors. También se comentan las conclusiones obtenidas gracias al desarrollo de ambos programas.

4. **Arquitectura *hardware* del piano digital, proyecto MIDI_Soc**: En este capítulo se lleva cabo la explicación de los distintos componentes y subcomponentes que forman el diseño *hardware* del piano digital.

5. **App Android MySoc, interfaz de uso del piano digital**: Capítulo en el que se explica tanto el modo de uso de la aplicación MySoc como los detalles a resaltar en su implementación.

6. **Conclusiones finales**: Capítulo final en el que se comentan los recursos finalmente utilizados por la FPGA usada definiendo un grado de polifonía de 16 notas y una compatibilidad con 6 pistas MIDI en el diseño final. También se realiza un comentario sobre los conocimientos obtenidos durante el desarrollo de este proyecto y el posible trabajo futuro a realizar.

# Capítulo 2

# Contexto tecnológico del proyecto

En este capítulo se explican el conjunto de tecnologías y herramientas utilizadas en el desarrollo del proyecto. Por otro lado, las referencias a las notas del piano utilizadas en este capítulo siguen el cifrado anglosajón [12].

## 2.1. Tecnología de la FPGA, placa Nexys 4 DDR y módulos Pmod

Las FPGAs (Field Programmable Gate Arrays) son dispositivos *hardware* programables y configurables que ofrecen un gran equilibro entre flexibilidad, eficiencia y consumo. Consisten en una matriz de bloques lógicos conocidos como CLBs (Configurable Logic Blocks) y una red de interconexión, ambos pueden ser configurados mediante el uso de de dispositivos anti-fuse o mediante el volcado de bits en celdas SRAM [13]. Estos bits de configuración vienen definidos en lo que se conoce como un archivo de configuración de la FPGA llamado *bitstream* resultado de un proceso de síntesis previo.

Los CLBs interconectados mediante dicha red contienen típicamente uno o varios circuitos combinacionales programables Look-Up Table (LUT), uno o varios biestables, lógica adicional y celdas de memoria SRAM requeridas para la configuración de los elementos anteriormente mencionados. Es en estos componentes donde se mapea el comportamiento secuencial o combinacional a realizar por nuestro circuito. Las tareas de entrada/salida se realizan en la periferia del dispositivo mediante bloques específicos denominados Input–Output–Blocks

(IOBs). Estos adaptan la señal eléctrica para la correcta compatibilidad con los dispositivos externos.

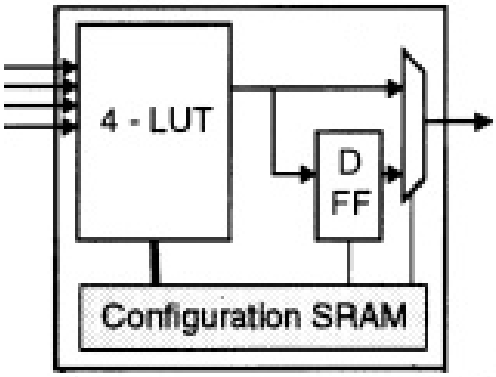


**Figura 2.1**: *Representación de un componente CLBs*
[13]

En el caso de la red de interconexión, se configura el enrutado de las señales mediante el bloqueo/apertura de conexiones entre los CLBs permitiendo así definir el comportamiento del *hardware* que queremos. Este bloqueo/apertura se realiza mediante la configuración de unos conmutadores, componentes básicos para la estructuración de redes.

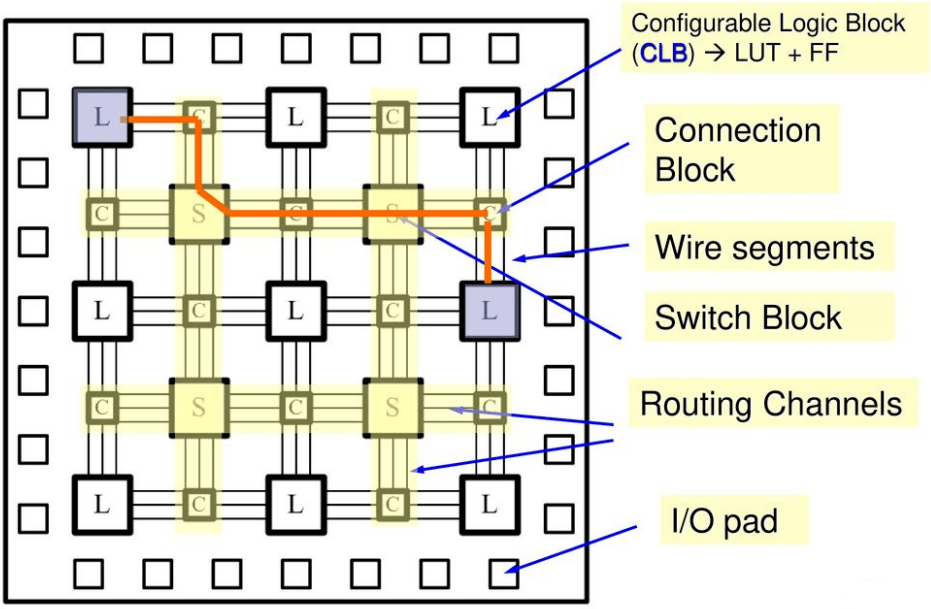


**Figura 2.2**: *Representación de una matriz de interconexión en una FPGA* [14]

Las FPGA actuales integran otros recursos heterogéneos aparte de las CLBs. Estos recursos son bloques de memoria RAM dedicada, multiplicadores, interfaces para periféricos, DSPs (Digital Signal Processor) así como los modelos de alta gama incluso microcontroladores o microprocesadores. La integración de todos estos elementos permite una mayor personalización de los sistema ajustándose mucho mejor a las necesidades de las compañías. Surge así el concepto de arquitecturas híbridas.

El diseño con FPGAs consiste en un proceso automatizado que se estructura típicamente en tres fases: síntesis, implementación y generación de *bitstream*.

- La fase de síntesis es la primera de todas. Consiste en el proceso de inferir un modelo del circuito a nivel de trasferencia entre registros a partir de la descripción dada en un lenguaje HDL (Hardware Description Language). Este modelo identifica los elementos funcionales que van a formar parte de nuestro circuito (módulos aritméticos, máquinas

de estado finito, memorias, etc) y cómo están interconectados.

- La fase que sigue a la síntesis es la implementación. Esta consiste en encontrar una red de elementos de la FPGA objetivo que satisfaga el comportamiento inferido en la anterior fase. Una vez encontrados, hay que ubicarlos e interconexionarlos en la placa, es decir, decidir qué componentes configurar e interconectar en nuestra matriz. Es un problema de optimización para el que existe un gran abanico de soluciones posibles pero hay que escoger las soluciones más óptimas. Es más, también hay que tener en cuenta que el enrutado cumpla como mínimo las restricciones de temporización de nuestro circuito para que la señal de reloj llegue correctamente a todos los componentes secuenciales. Esta fase se caracteriza por ser la más costosa computacionalmente y por tener la peculiaridad que un mismo diseño HDL no siempre da el mismo enrutado en un mismo chip de FPGA.

- Para finalizar, se lleva acabo la fase de generación de *bitstream*. Esto consiste en la generación de un archivo, típicamente en formato .bit, que almacena los bits de configuración tanto para la red de interconexión como para los componentes interconectados en dicha red. El chip de la FPGA lee dicho archivo y configura correspondientemente los componentes.

Para el desarrollo de este proyecto se ha usado la placa Nexys 4 DDR [15] de la distribuidora Digilent, haciendo uso del chip FPGA XC7A100T-1CSG324C [16] de la compañía Xilinx. Cabe destacar que esta placa se caracteriza por disponer de una memoria externa SDRAM DDR2 de 128 MiB [17] y de una memoria Flash también externa de 16 MiB [18], siendo ambas cruciales para el desarrollo de este proyecto ya que en ellas se almacenarán tanto el archivo MIDI como el audio muestreado. A su vez se han utilizado dispositivos "Pmod" siendo este un interfaz genérico de Digilent. Los dispositivos "Pmod" utilizados son el I2S2 [19] para conversión digital analógica del audio muestreado, y el BT2 [20] utilizado para el establecimiento de una conexión *Bluetooth* con la interfaz externa del sistema.

## 2.2. El estándar MIDI

El protocolo MIDI (Musical Instrument Digital Interface) [21] ha sido ampliamente aceptado y utilizado por músicos y compositores desde su concepción entre los años 1982 y 1983. El protocolo MIDI permite representar una interpretación musical en forma de datos digitalizados de forma eficiente, haciéndolo atractivo no solo para compositores e intérpretes musicales, sino también para aplicaciones computacionales que producen sonido, como aplicaciones multimedia o videojuegos. Gracias a la publicación del documento "General MIDI System specification", el protocolo ha sido ampliamente aceptado por la mayoría de interfaces PC/MIDI, soportado a su vez por el sistema operativo Microsoft Windows entre otros SOs.

**Figura 2.3**: *Logotipo del protocolo MIDI*
[22]

Originalmente fue desarrollado con el fin de intercomunicar sintetizadores entre sí. Actualmente el protocolo MIDI está ampliamente extendido como un método para sustituir o apoyar el audio muestreado en videojuegos y aplicaciones multimedia. La principal ventaja de uso de los archivos MIDI respecto al formato PCM de almacenamiento de audio muestreado (como los archivos .WAV), es el tamaño en bytes del archivo. Los datos MIDI son extremadamente pequeños si los comparamos con audio muestreado. Por ejemplo, un archivo que contenga audio muestreado estéreo de alta calidad requiere aproximadamente 10MiB/minuto, mientras que una secuencia MIDI ocupa aproximadamente 10Kib/minuto. Esta diferencia se debe a que un archivo MIDI no contiene audio muestreado, contiene solo

las instrucciones necesarias para que el sintetizador sepa que sonido generar. Estas instrucciones se conocen como mensajes MIDI, en estos mensajes viene codificada la información necesaria para que el sintetizador sepa que notas reproducir y con que volumen. Entre las ventajas de utilizar MIDI se incluye la facilidad para editar la interpretación musical, pudiendo cambiar el tempo o la tonalidad de la canción de forma sencilla.

Para el desarrollo de este proyecto, con el fin de lograr interpretar una canción representada según el protocolo MIDI, se ha puesto como objetivo poder entender un subconjunto de todos los mensajes MIDI soportados permitiendo saber las notas a tocar, el volumen de las mismas y su duración. Para esto último también será necesario saber el tempo de la canción.

A continuación, se procede a explicar el proceso de parseo de un archivo MIDI, pudiendo así extraer la secuencia de comandos contenida en dicho archivo que definen las notas que deben ser reproducidas y su duración para lograr así la interpretación de una pieza musical. En primer lugar se explica la estructura de un archivo MIDI.

En un archivo MIDI se distinguen dos partes diferenciadas con estructura propia: la cabecera y las pistas. La cabecera contiene información general acerca de la interpretación musical guardada. Las pistas contienen una secuencia de mensajes MIDI que contienen información relativa a los 16 canales MIDI. Tanto la cabecera como las pistas son reconocibles debido a que ambas empiezan con 4 caracteres ASCII, 'MThd' en el caso de la cabecera y 'MTrk' en el caso de las pistas. A ambas les siguen 32 bits de longitud, con el byte más significativo al principio comenzando por la izquierda (una longitud de 6 se almacenaría como 00000006, valores en hexadecimal). Esta longitud hace referencia al número de bytes de datos que siguen, por lo tanto los bytes pertenecientes los caracteres ASCII como los que componen la longitud, no están incluidos. Por ello, una cabecera o pista con longitud 6 ocuparía realmente 14 bytes.

Un archivo MIDI siempre empieza con una cabecera, seguida de una o varias pistas.

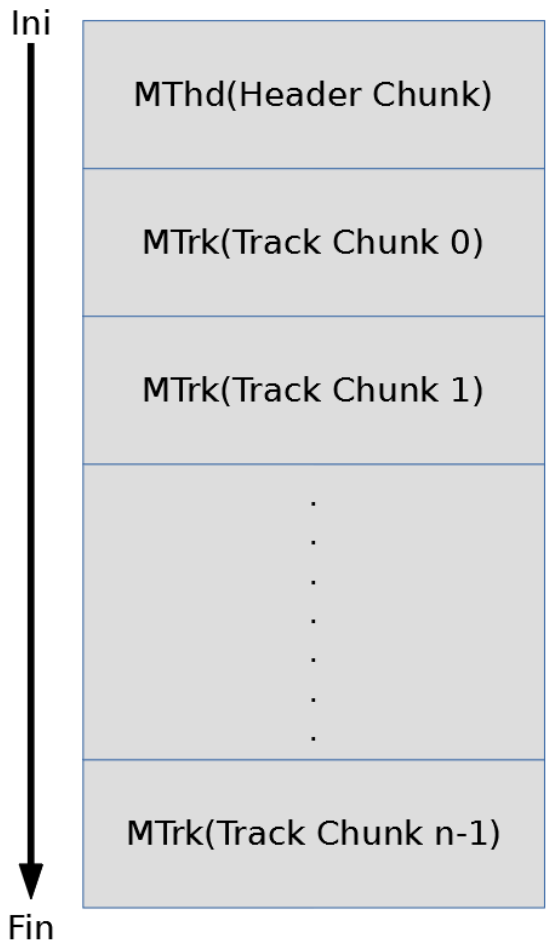


**Figura 2.4**: *Estructura general de un archivo MIDI*

### 2.2.1. Estructura de la cabecera

La información contenida en la cabecera sigue la sintaxis:

**<Cabecera>** = **<caracteres ASCII>** **<longitud>** **<formato>** **<ntrks>** **<division>**

La sección de datos formada por **<formato>, <ntrks> y <division>**, esta constituida por tres palabras de 16 bits, guardando el byte más significativo al principio.

- La primera sección **<formato>**, especifica la organización general del archivo. Solo 3 valores de formato están especificados, el 0, 1 y 2. El proyecto se ha centrado en

compatibilizar el desarrollo con los formatos 0 y 1.

- Formato 0: el archivo contiene una única pista con múltiples canales MIDI.
- Formato 1: el archivo contiene una o más pistas con múltiples canales MIDI que forman parte de la secuencia. No son independientes.
- Formato 2: el archivo contiene una o más pistas independientes con múltiples canales MIDI.

La sección <**ntrks**>, hace referencia al número de pistas presentes en el archivo. Ha de ser siempre 1 para el formato 0.

La tercera sección de datos <**division**>, especifica el significado del tiempo-delta. El tiempo-delta es la cantidad de tiempo que debe pasar para que la acción representada por el mensaje MIDI que le sigue se aplique. Tiene dos formatos, uno de tiempo métrico y otro que utiliza el tiempo del reloj interno de un dispositivo MIDI.

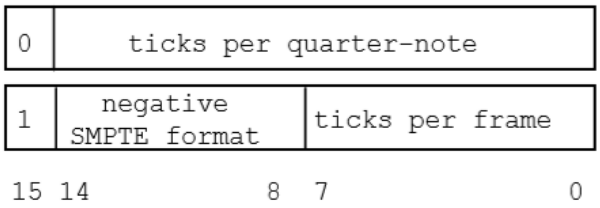


**Figura 2.5**: *Formato de los bits del campo division*
[21]

Este proyecto está centrado en el formato de tiempo métrico, este se indica representando el bit 15 con un valor lógico de 0. El resto de bits hacen referencia a un valor numérico que indica el número de tiempo-delta que forma lo que se conoce en música como una "Negra" (1/4 de tiempo de "Redonda"). Un ejemplo sería *division*, tomando el valor de 96, y siendo el intervalo de tiempo entre dos notas de una corchea, el valor del tiempo-delta del mensaje MIDI que activa la segunda nota sería de 48.

### 2.2.2. Estructura de las pistas

Es en las pistas donde los datos de la canción están ubicados. Cada pista no es más que una secuencia de eventos precedidos del tiempo-delta. El formato de las pistas es el mismo para los tres tipos de formatos generales anteriormente explicados en el apartado de cabecera (formatos 0, 1 y 2).

La sintaxis de una pista es la siguiente (el símbolo + indica uno o más, como mínimo debe haber un evento MTrk en una pista):

<**Pista**> = <**caracteres ASCII**> <**longitud**> <**evento MTrk**> +

La sintaxis de un evento MTrk es muy simple:

<**evento MTrk**> = <**tiempo-delta**> <**evento**>

Para continuar con la explicación de los archivos MIDI, debo aclarar que no todos los valores numéricos de un archivo MIDI se representan con el mismo formato. Hasta este punto, exceptuando el tiempo-delta, todos los valores numéricos se representaban en binario puro teniendo el byte más significativo en la izquierda (ídem con los bits). Algunos números en los archivos MIDI siguen otro tipo de representación denominada, **representación de longitud variable**. Este formato es conveniente ya que permite reducir en la medida de lo posible el tamaño de los archivos.

Los números que siguen la representación de longitud variable, están representados por 7 bits por byte, el más significativo primero. Todos los bytes excepto el último tienen el bit 7 con valor lógico 1, el resto tiene el valor lógico 0. Los bits dentro del byte que representan el número son los 7 bits menos significativos, mientras que el bit más significativo no se tiene en cuenta y sirve de centinela para saber si se ha llegado al último byte que forma el número. Si este bit es 1 no es el último byte y por el contrario si es 0, es el último byte que forma el número. Si el número a representar está comprendido en el intervalo [0,127],

entonces será representado exactamente con un solo byte.

| Number (hex) | Representation (hex) |
|---|---|
| 00000000 | 00 |
| 00000040 | 40 |
| 0000007F | 7F |
| 00000080 | 81 00 |
| 00002000 | C0 00 |
| 00003FFF | FF 7F |
| 00004000 | 81 80 00 |
| 00100000 | C0 80 00 |
| 001FFFFF | FF FF 7F |
| 00200000 | 81 80 80 00 |
| 08000000 | C0 80 80 00 |
| 0FFFFFFF | FF FF FF 7F |

**Figura 2.6**: *Algunos ejemplos de números representados en formato de longitud variable* [21]

El mayor número permitido es 0FFFFFFF así cualquiera de los valores que sigan esta representación no ocuparan más de 32 bits, es una precisión más que suficiente para representar la duración de las notas.

Esta forma de codificar valores numéricos es la que sigue el tiempo-delta. Como se mencionó con anterioridad, el tiempo-delta es la cantidad de tiempo que precede a un evento. Si un evento sucede al principio de la pista, o si dos eventos ocurren simultáneamente, el tiempo-delta correspondiente a ambos casos es de 0. El tiempo delta siempre debe estar presente.

La sintaxis de los eventos es la siguiente (el símbolo | hace referencia a un OR lógico):
**<evento>** = **<evento MIDI>** | **<evento SysEx>** | **<meta-evento>**

Un **<evento SysEx>** (SysEx se refiere a *System Exclusive*) es utilizado para especificar un mensaje exclusivo del sistema. Este tipo de eventos no son necesarios tenerlos en completa consideración para la realización del proyecto ya que no contienen información crucial para la interpretación de la canción almacenada. De estos eventos saber que se representan con los valores hexadecimales F0 o F7 y que a estos le sigue la longitud en bytes del evento

representada en formato de longitud variable. Los bytes de datos vienen a continuación de la longitud. Esta explicación viene representada en la sintaxis de abajo:

**<evento SysEx>** = < **F7** | **F0** > **<longitud>** **<bytes a ser transmitidos>**

En el caso del **<meta-evento>**, este sirve para especificar información no relacionada con el formato o con los secuenciadores. Siguen esta sintaxis:

**<meta-evento>** = < **FF** > **<tipo>** **<longitud>** **<bytes de datos>**

Todos los meta eventos comienzan con FF, prosiguen con el tipo de evento (que siempre tiene un valor inferior a 128), para luego continuar con la longitud en bytes del campo de datos que está representada en formato de longitud variable, y terminan con los datos en sí. Si no hay datos la longitud es 0.

Los meta eventos que se han tenido en cuenta para la aplicación Midi Parser han sido:

- **Armadura de tonalidad**, ejemplo en hex: **FF 59 02 sf mi**. Define la tonalidad de la pieza musical. El byte *sf* toma valores comprendidos por el intervalo [-7,7] (codificado en C2). El valor absoluto del número indica la cantidad de sostenidos en caso de que el número sea positivo, o bemoles en caso de que el número sea negativo de la armadura de tonalidad. El byte *mi* indica si es una tonalidad mayor (valor de 0) o menor (valor de 1).

- **Fin de Pista**, ejemplo en hex: **FF 2F 00**. Este tipo de meta-evento no es opcional, toda pista debe incluirlo. Sirve como referencia para indicar el punto exacto en el que finaliza la pista.

- **Establecer Tempo**, ejemplo en hex: **FF 51 03 tttttt**. Este evento indica un cambio

en el tempo. El valor numérico de los dígitos hexadecimales $t$, representa el número de microsegundos por tiempo de "Negra".

Para el diseño *hardware* del piano digital no se ha tenido en cuenta el meta-evento Armadura tonalidad ya que no es necesario para lograr la correcta interpretación de la pieza musical.

Para convertir el tiempo delta en milisegundos, basta con aplicar una sencilla operación algebraica:

$$Division = \frac{tiempoDelta}{qn}$$

$$Tiempo(en\ ms) = \frac{tiempoDelta \cdot \frac{(Tempo(\frac{\mu s}{qn}))}{Division}}{1000}$$

El símbolo $qn$ hace referencia al tiempo de "Negra", $Tempo(uS/qn)$ es el valor en microsegundos por tiempo de "Negra" establecido por un meta evento de establecer tempo. El valor de $Division$ (número de tiempos delta transcurridos por tiempo de "Negra") viene dado por la cabecera del archivo MIDI.

Cabe aclarar que todos los archivos MIDI deben especificar el tempo y el ritmo del compás. Si no está especificado, se asume que el compás es de 4/4 y el tempo es de 120 pulsos por minuto. Esta información es considerada como un meta evento. En el formato 0, estos eventos deben estar ubicados al principio de la pista. En el formato 1 deben estar en la primera pista. En el formato 2, cada pista contiene su propio tempo y ritmo.

Finalizando con la explicación de los tipos de eventos de un evento MTrk, los eventos MIDI son cualquier mensaje de canal MIDI. Un mensaje de canal MIDI consiste en una secuencia de uno o tres bytes, dependiendo del estado (tipo de mensaje), el primer byte representa el estado del mensaje MIDI. La diferencia entre un byte de estado y un byte de

datos reside en el bit más significativo, en el caso de tener el valor lógico 1 es un byte de estado y en caso contrario es un byte de datos. Los 4 bits más significativos del byte de estado hacen referencia a un mensaje de voz y los 4 bits restantes hacen referencia al canal MIDI al que va dirigido. Es en los mensajes de voz en los que se encuentran los comandos de *Note-Off* y *Note-On* indicando que notas activar o suspender, el valor en hexadecimal de los bytes de estado son 8n y 9n respectivamente, siendo "n" un valor hexadecimal comprendido entre 0 y F (haciendo referencia al canal MIDI). A estos mensajes les siguen dos bytes de datos, un byte especificando la nota a activar y el otro byte indica la velocidad de la nota (volumen con el que activar la nota), en el caso de ser un mensaje *Note-Off* este último valor no se tiene en cuenta.

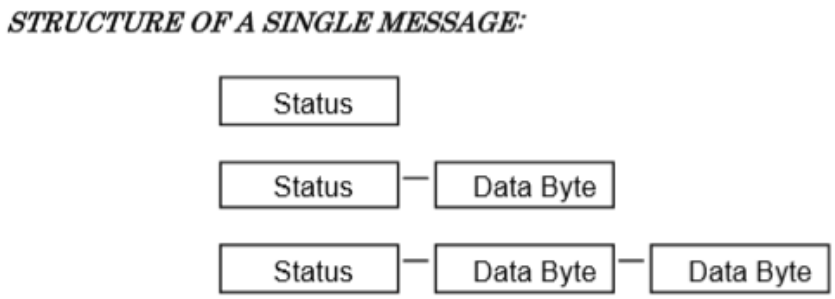


**Figura 2.7**: *Sintaxis de mensajes MIDI*
[21]

Los bytes que forman estos eventos se deben entender siguiendo lo que se conoce en el estándar MIDI como *Running Status*. El *Running Status* consiste en tener en cuenta el último byte de estado recibido de un mensaje de canal MIDI. De esta forma se pueden omitir los bytes de estado y transmitir solamente los bytes de datos si es requerido. Esto supone otro ahorro en el tamaño del archivo MIDI posibilitando el envío de un mensaje de canal MIDI solo con bytes de datos.

El *Running Status* es especialmente útil al enviar largas cadenas de mensajes *Note-On* y *Note-Off*, con la particularidad de que un mensaje *Note-On* con velocidad 0 tiene el mismo significado que un mensaje *Note-Off*.

A su vez, para el correcto reconocimiento de los mensajes *Note-On* y *Note-Off*, se debe poder reconocer parcialmente los mensajes MIDI *Program Change* y *Channel Key Pressure*, ambos mensajes tienen solo un byte de datos y el valor en hexadecimal de los bytes de estado son Cn y Dn respectivamente siendo "n", una vez más, un valor hexadecimal comprendido entre 0 y F (haciendo referencia al canal MIDI). El resto mensajes MIDI siempre tienen dos bytes de datos. En el caso de la aplicación Midi Parser, para poder reconocer la activación/desactivación del pedal sustain, se debe tener en consideración el tipo de mensaje MIDI *Control Change*, siendo el valor en hexadecimal del byte de estado del mismo Bn ("n" hace referencia al canal MIDI). Tras esto para saber si el mensaje se refiere para activar/-desactivar el pedal de sustain, solo hay que atender al primer byte de datos que su valor en hexadecimal es 40. El segundo byte de datos define su activación (valor 127 en decimal) o desactivación (0). Ejemplos en hexadecimal de los mensajes anteriormente mencionados sin incluir el tiempo-delta que les precede, aparecen a continuación:

- *Note-On*: **92 48 96**
- *Note-Off*: **82 48 96**
- *Program Change*: **C0 46**
- *Channel Key Pressure*: **DF 48**
- *Control Change, Sustain on*: **B0 40 7F**

Para terminar este subapartado adjunto una tabla que relaciona las notas musicales con los valores numéricos en decimal que se han tenido en cuenta, esta relación es la que sigue el primer byte de datos de los mensajes *Note-On* y *Note-Off*. Las notas a representar en el piano digital son las que abarcan desde A0 (La) hasta C8 (Do) ambas incluidas.

| Octava | C | C# | D | D# | E | F | F# | G | G# | A | A# | B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | - | - | - | - | - | - | - | - | - | 21 | 22 | 23 |
| 1 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
| 2 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 3 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| 4 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 |
| 5 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 |
| 6 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| 7 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
| 8 | 108 | - | - | - | - | - | - | - | - | - | - | - |

**Tabla 2.1**: *Tabla de codificación entre notas y números que sigue el estándar MIDI*

## 2.3. Síntesis de sonido mediante *Wavetable*

### 2.3.1. Archivos .WAV, uso como una *Wavetable*

Los archivos .WAV siguen el formato WAVE para almacenamiento de audio digital. Este formato ha sido conjuntamente desarrollado por IBM y Microsoft siendo un subconjunto de la especificación RIFF de Microsoft para el almacenamiento de archivos multimedia. Un archivo RIFF comienza por con una cabecera seguida de una secuencia de conjuntos de datos [23]. Típicamente un archivo WAVE sigue la estructura de un archivo RIFF conteniendo la cabecera y dos conjuntos de datos denominados “fmt” y “data”. El conjunto de datos “fmt” describe información acerca del sonido que está almacenado en el conjunto de datos, así encontramos la frecuencia de muestreo como el número de bits por muestra así como si el audio es mono o estéreo. Seguidamente del conjunto “fmt”, está el conjunto de datos que contiene el propio sonido.

El sonido es representado como una serie de que muestras representando numéricamente la forma de la onda sonora a intervalos equidistantes de tiempo. La codificación de estos valores numéricos siguen el formato PCM, que comprenden valores acotados por el intervalo (-1,1) codificados en punto fijo, teniendo un único bit de signo y el resto de bits corresponden a la parte decimal del número. El orden de los bytes que representan estas muestras es little endian, bytes menos significativos primero (empezando por la izquierda). En el formato

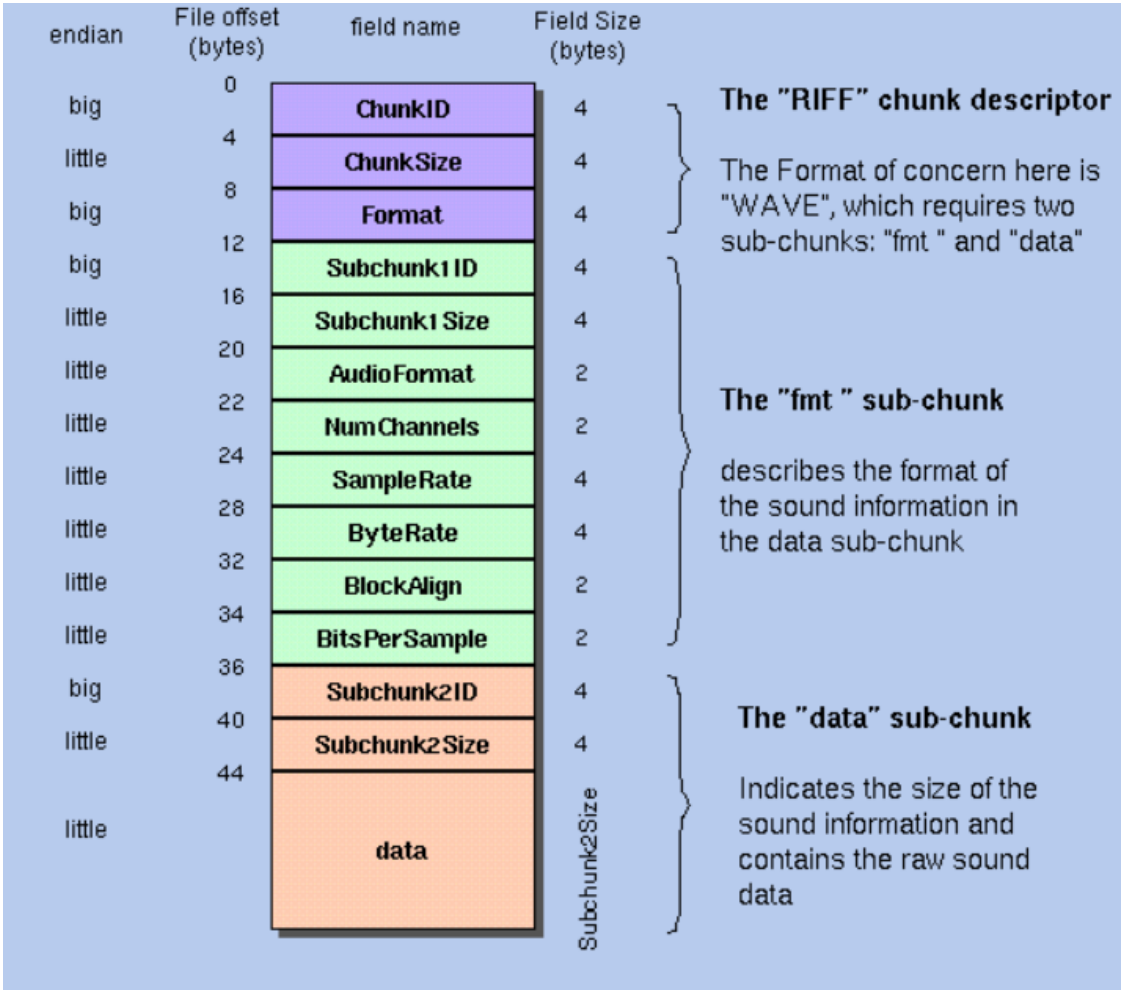


**Figura 2.8**: *Representación del formato WAV*
[23]

WAVE, el número de bits para codificar las muestras puede ser 8 bits, 16 bits o 24 bits dependiendo del grado de resolución necesitado. A su vez, la frecuencia de muestreo, es decir, el intervalo de tiempo que pasa entre la recepción de muestras, también viene definido en este formato (típicamente los valores más comunes son 44100Hz, 48000Hz o 96000Hz). Por ejemplo, si en un archivo .WAV la frecuencia de muestreo es de 44100Hz, el tiempo transcurrido entre la recepción de una muestra y otra es aproximadamente de $22,7\mu s$

Teniendo almacenado en la memoria de la FPGA archivos .WAV que representen la forma de onda de distintas notas, se pretende aplicar una técnica de síntesis de sonido basada en *Wavetables* [24]. Esta técnica consiste en la generación de una forma de onda periódica de una frecuencia dada mediante la interpolación de muestras procedentes de otra forma de onda periódica de menor frecuencia. Así teniendo almacenado, por ejemplo, la forma de onda correspondiente la nota Do se pretende conseguir la forma de onda que represente la nota Re sin necesidad de tenerla almacenada en memoria, manteniendo en gran medida la sonoridad característica del instrumento correspondiente que haya generado la nota, pero variando el período de la onda original.

Las notas de piano utilizadas tanto en la aplicación de consola Note Conversor como en el diseño *hardware* del intérprete de archivos MIDI, vienen almacenadas en archivos .WAV. Han sido descargados del sitio web oficial del proyecto FreePats [25], destinado a la creación de una colección libre y abierta de sonidos digitales provenientes tanto de instrumentos acústicos como sintetizados.

El autor original de los archivos .WAV es Alexander Holm. Estos archivos de audio son procedentes de un piano de cola Yamaha C5. Las notas han sido grabadas en intervalos menores de tercera (cada 3 notas contando la última e incluyendo notas alteradas) empezando por A0, a una frecuencia de 48Khz con una resolución de 24 bits por muestra.

### 2.3.2. Interpolación lineal para la generación de notas musicales

La tonalidad de una nota depende fundamentalmente de la frecuencia de la forma de onda generada en el aire. Uno de los métodos más económicos para realizar el cambio de tono (*pitch-shifting*) de una nota musical es mediante el remuestreo en el dominio del tiempo de la *Wavetable* que la contiene [24].

Suponiendo una frecuencia de muestreo común y que tenemos almacenada en una *Wavetable* $WTin$ las muestras correspondientes a una nota de frecuencia fundamental $fin$, es posible calcular las muestras de otra nota de frecuencia fundamental $fout$, con $fout > fin$ y almacenadas en $WTout$, usando la siguiente relación:

$$WTout[i] = WTin[ci] \quad \text{con} \quad ci = i \cdot (\frac{fout}{fin})$$

Esta fórmula describe un cribado no entero de muestras (con $ci > 1$). Obsérvese que es consistente con el caso de que $fout = fin$ ya que la expresión se transforma en la identidad $WTout[i] = WTin[i]$. No obstante, en esta expresión $ci$ es un valor real y los índices de las *Wavetables* deben ser enteros. En otras palabras, el valor de $WTin[ci]$ sería un valor no almacenado en $WTin$ pero intermedio entre WTin[floor(ci)] y la siguiente muestra almacenada, $WTin[floor(ci) + 1]$. Para calcularlo se interpolará linealmente entre ambos valores [26], resultando:

$$WTout[i] = WTin[ci] = WTin[floor(ci)]+$$
$$(WTin[floor(ci) + 1] - WTin[floor(ci)]) \cdot (ci - floor(ci))/(floor(ci) + 1 - floor(ci))$$

Simplificando y conociendo que $floor(ci) = integerPart(ci)$ y por tanto que $ci - floor(ci) = decimalPart(ci)$ la expresión queda:

$$WTout[i] = WTin[intCi] + (WTin[intCi+1] - WTin[intCi]) \cdot decCi \quad \text{con}$$
$$intCi = integerPart(ci) \quad decCi = decimalPart(ci) \quad ci = i \cdot (\frac{fout}{fin}) \tag{2.1}$$

Obsérvese que si $ci$ se representa en punto fijo, la obtención de la parte entera y decimal de un número es trivial. Asimismo, dado que la reproducción del sonido es secuencial no se requiere almacenar $WTout$, basta con recorrer $WTin$ y hacer en tiempo real unas sencillas operaciones. Por último, como el recorrido de $WTin$ es secuencial el cálculo de $ci$ no requiere multiplicación alguna sino un simple incremento en el término constante $fout/fin$.

En la imagen siguiente se muestra una representación gráfica de la fórmula de interpolación usando valores ejemplo, haciendo más intuitiva la comprensión de la misma.

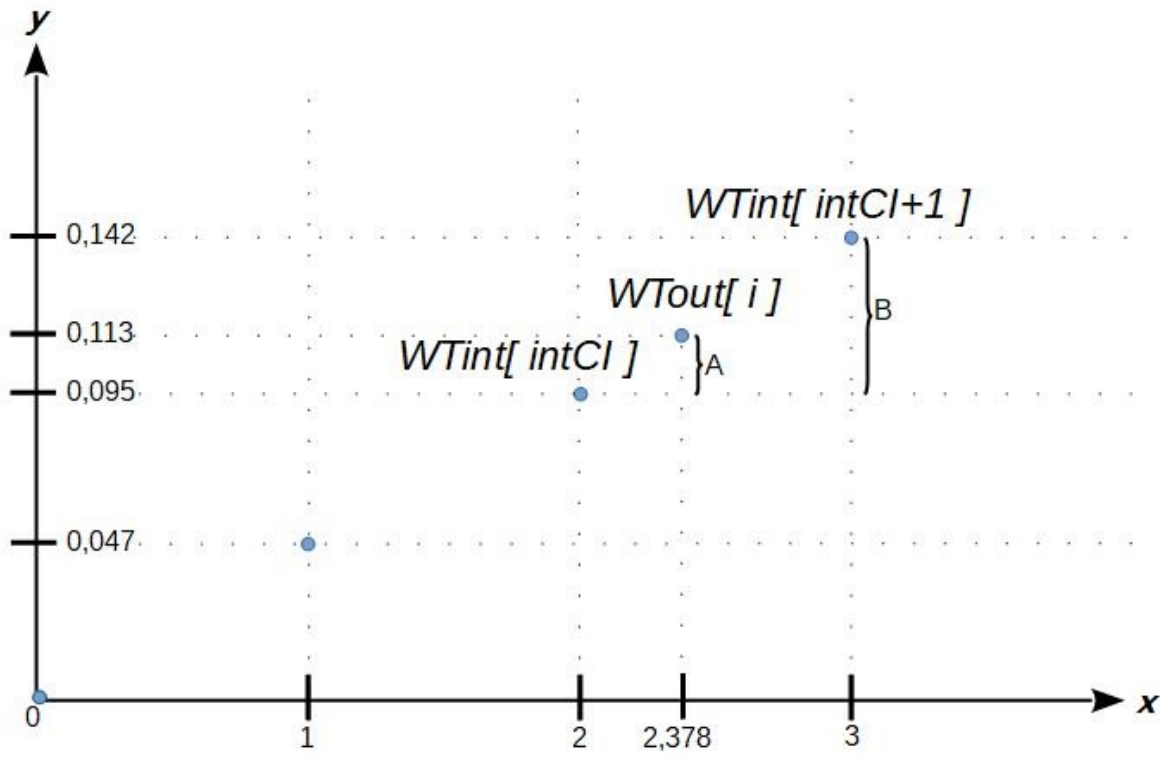


**Figura 2.9**: *Representación gráfica de la fórmula de interpolación*

En donde $A$ y $B$ se definen como:

$$A = (WTin[intCi + 1] - WTin[intCi]) \cdot decCi$$

$$B = WTin[intCi + 1] - WTin[intCi]$$

Para comprobar numéricamente la validez de la fórmula se han calculado los valores que toma una función seno de una frecuencia dada (según la nota musical deseada) muestreada a cierta frecuencia. Estas muestras serán comparadas con las muestras obtenidas al aplicar la fórmula de interpolación anteriormente definida.

A continuación, se presentan la definición de la fórmula para obtener una muestra de índice $i$ de la función seno, en la que tanto la frecuencia de muestreo ($fs$) y la frecuencia de la nota ($freq$) son variables, siendo esta última la que representa la frecuencia de la nota correspondiente.

$$Sample(i) = \sin(\frac{2 \cdot i \cdot \Pi}{(\frac{fs}{freq})})$$

Junto a la fórmula de interpolación (2.1), ambas son usadas para el cálculo de las muestras que figuran en la siguiente tabla:

| i | F4# | A4 | ci | A4' | Error A4' |
|---|---|---|---|---|---|
| 0 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 % |
| 1 | 0.047620 | 0.056621 | 1.189221 | 0.056610 | 0.020400 % |
| 2 | 0.095131 | 0.113061 | 2.378443 | 0.113030 | 0.027600 % |
| 3 | 0.142427 | 0.169138 | 3.567664 | 0.169092 | 0.027500 % |
| 4 | 0.189399 | 0.224672 | 4.756885 | 0.224627 | 0.020100 % |
| 5 | 0.235942 | 0.279486 | 5.946107 | 0.279470 | 0.005500 % |
| 6 | 0.281950 | 0.333402 | 7.135328 | 0.333357 | 0.013700 % |
| 7 | 0.327318 | 0.386249 | 8.324549 | 0.386152 | 0.025200 % |
| 8 | 0.371943 | 0.437857 | 9.513771 | 0.437733 | 0.028300 % |
| 9 | 0.415724 | 0.488059 | 10.702992 | 0.487945 | 0.023400 % |
| 10 | 0.458562 | 0.536696 | 11.892213 | 0.536639 | 0.010700 % |
| 11 | 0.500359 | 0.583611 | 13.081435 | 0.583560 | 0.008600 % |
| 12 | 0.541022 | 0.628653 | 14.270656 | 0.628511 | 0.022600 % |

**Tabla 2.2**: *Tabla de valores calculados por la función seno y la fórmula de interpolación, con el correspondiente cálculo del error cometido*

Para la obtención de los valores que figuran en la tabla, el valor que toma la variable $fs$ esta fijado a 48800, mientras que el valor de $freq$ depende de la nota a calcular. En este caso particular las frecuencias correspondientes a cada nota son: F4# (369,99Hz) y A4 (440Hz). La columna i hace referencia al índice, sirviendo este para numerarlas, así la primera muestra comienza con el índice 0.

Teniendo en cuenta esto, en las columnas F4# y A4 figuran los valores obtenidos al aplicar la fórmula de cálculo de muestra (§2.3.2), mientras que en la columna A4' hacen presencia las muestras obtenidas al aplicar la fórmula de interpolación (2.1).

En el caso de esta última fórmula, el valor de la variable $fin$ se fija a 369,99, utilizando así las muestras de la nota F4# para las variables $WTin[intCI]$ y $WTin[intCI + 1]$ al aplicar la fórmula de interpolación. La variable $fout$ depende de la frecuencia de la nota que queramos calcular, consecuentemente toma el valor de 440.

En la columna *ci* figura la evolución de dicha variable según el índice de muestra correspondiente. Al fijarse en las dos últimas filas, nos damos cuenta que la parte entera de *ci* deja de tener el mismo valor que el índice de muestra con lo que es necesario utilizar muestras

de la nota F4# que no figuran en la tabla presentada.

Por último en la columna Error A4', figura el error obtenido al comparar las muestras de las columnas A4 y A4'. Para el cálculo de este error se ha seguido la siguiente fórmula, dando el error directamente como un porcentaje:

$$Error(sample_1, sample_2) = 100 \cdot \frac{(sample_1 - sample_2)}{sample_1}$$

Como se puede observar en la tabla, el error en la gran mayoría de los casos es entorno a la centésima. Realizando el calculo de 48800 muestras, se ha calculando la media de los valores obtenidos en las columna Error A4', siendo así dicha media de 0,019 %. Redondeando nos da un error aproximado por muestra del 0,02 % en este caso.

Si tenemos en cuenta la media del error anteriormente calculado, podemos aventurar a decir que este comportamiento se mantiene independientemente del número de muestras calculadas, ya que el error medio esperado es de 0,02 % y un error de esa magnitud no afecta significativamente a la forma de onda que se pretende generar.

Teniendo en cuenta estos resultados, podemos decir que la fórmula se comporta justo como queremos bajo un contexto puramente teórico.

Tras realizar estas comprobaciones numéricas, se procedió a la realización de la aplicación de consola NoteConversor (§3.2), con el fin de comprobar la validez del método utilizando muestras de formas de onda reales (es decir, no puramente sinusoidales) de distintas notas, así como la viabilidad de su adaptación a aritmética de punto fijo.

## 2.4. Efecto Sustain de una nota

Típicamente en el espectro de una nota se pueden diferenciar las etapas *Attack*, *Decay*, *Sustain*, *Release* [21]. La siguiente imagen representa de forma simplificada la forma de onda de una nota musical, en donde se pueden diferenciar las estas etapas:

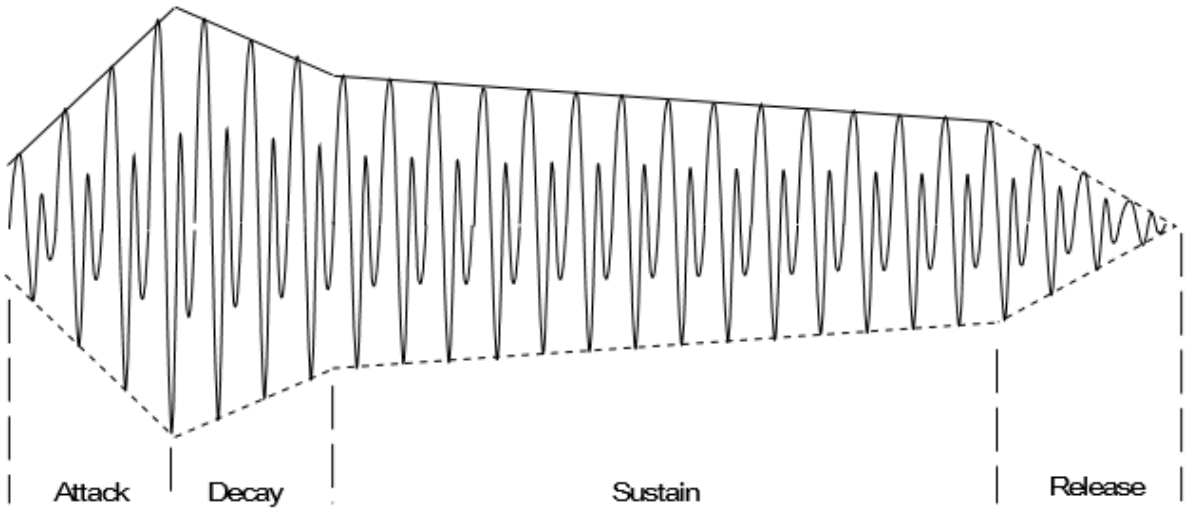


**Figura 2.10**: *Representación simplificada de una onda de sonido, diferenciando las etapas que la forman*

[21]

El efecto sustain consiste en prolongar de forma acotada o indefinidamente, la duración de la etapa *Sustain* de una nota. Esto se realiza reproduciendo en bucle un intervalo de muestras pertenecientes a dicha etapa.

Un importante detalle a tener en cuenta al establecer que par de muestras corresponden al inicio y al final del intervalo a repetir en bucle, es que ambas deben ser de las primeras muestras que inician un período en la onda. Esto es para evitar gliches en el sonido producido. Para poder calcular que muestra se corresponde a un inicio de período, se ha ideado la siguiente fórmula:

$$FirstSampleOfPeriod = floor((TotalNumPeriods - offset) \cdot SamplesPerPeriod)$$
$$\text{con} \quad SamplesPerPeriod = \frac{fs}{freq} \quad TotalNumPeriods = \frac{TotalNumOfSamples}{SamplesPerPeriod}$$

La lógica de la fórmula consiste en un primer momento en averiguar cuantas muestras forman un período en nuestro archivo de sonido muestreado, esto mismo es lo que representa $SamplesPerPeriod$. Tras obtener el número de muestras por período, se obtiene el total

de períodos que hay en nuestra forma de onda siendo $TotalNumPeriods$ la variable que referencia esto. Finalmente, averiguamos el número de muestra (la dirección que buscamos) que forma el inicio de un período en particular, en función de una variable *offset* que permite seleccionar un período a lo largo de la forma de onda.

El uso de la variable *offset* es necesario ya que dependiendo de la nota a generar, será más conveniente escoger un período, más cercano a la etapa *Decay* o más cercano a la etapa *Release*. Esta selección de período depende fundamentalmente de la frecuencia de la nota, siendo en notas agudas necesario un perído cercano a la etapa *Decay* (duración de la etapa *Sustain* más corto, en comparación con notas graves) y en las notas graves acercase más a la etapa *Release* (duración etapa *Sustain* más largo, en comparación con notas agudas).

En el piano digital, los generadores de notas (§4.6.2) son los reponsables de realizar dicho efecto.

## 2.5. Efecto *reverb*

Por propia naturaleza los músicos siempre han estado en busca de la generación de sonidos únicos [27]. Con la llegada de la electrónica y la era digital, ha sido posible aumentar el espectro de sonidos generados por el ser humano. Una forma de obtener estos sonidos únicos es mediante el uso de distintos efectos sonoros generados por sistemas de procesamiento analógico o digital, pudiendo clasificar los efectos en función del dominio de procesamiento, pudiendo ser dinámico, afectando a la frecuencia o usando retardos.

El efecto *reverb* en particular pertenece al grupo de los efectos que usan retardos, es decir, muestras de sonido anteriormente generadas. Este efecto emula la mezcla de una señal de audio con sus reflexiones acústicas atenuadas. Gracias a él, el sonido sintetizado tiene más cuerpo a costa de perder nitidez, lo cual dependiendo de la interpretación musical puede ser o no conveniente.

La siguiente ecuación describe el efecto *reverb* aplicado sobre muestras de audio llevado

a cabo en el piano digital:

$$y(n) = x(n) + G \cdot x(n - D) \quad \text{con} \quad n, D \in \mathbb{N} \quad y \quad 0 < G < 1{,}0$$

Nótese que esta ecuación emula una única reflexión acústica en cada instante la cual viene retardada un número de ciclos de muestreo determinado por la variable $D$. El grado de atenuación aplicado a dicha reflexión en este proyecto es de $G = 0,5$ que se implementa en *hardware* como un simple desplazamiento aritmético de un bit a la derecha.

En el diseño *hardware* final el valor del retardo es parametrizable, habiéndose aplicado un retardo de 100 ms (4880 muestras) en el diseño publicado en GitHub (§4).

## 2.6. Aritmética en punto fijo

Es de saber público que la representación digital de la información utiliza dígitos binarios (bits). En una colección de $n$ dígitos binarios presenta $2^n$ posibles estados. El significado de estos posibles depende enteramente de la interpretación que se le quiera dar, así esta colección de ceros y unos puede ser entendida como un número entero, un número natural, etc [28].

El formato en punto fijo hace referencia a la forma de interpretar dicha colección de dígitos binarios, definiendo así el número de dígitos que representan la parte entera y la parte decimal de un número. Típicamente se utiliza la sintaxis $Qn.m$ siendo $n$ el número de dígitos que representan la parte entera, comúnmente son los dígitos de mayor peso dentro de nuestro conjunto ordenado de dígitos binarios. El valor de $m$ hace referencia al número de dígitos que representan la parte decimal, siendo estos típicamente los dígitos de menor peso. En caso de utilizar un dígito para indicar el signo, este figura en el dígito más significativo y por ello debe ser contabilizado en la parte entera.

De esta forma es posible representar un subconjunto determinado de los números racionales, así los números racionales contenidos en dicho subconjunto se definen como $a/2^n$ en donde $a \in Z, n \in N$ siendo $n$ el cardinal de la colección de dígitos binarios con los que se

representa el número. Consecuentemente, para saber que número racional es representado por un número en binario puro, basta con aplicar la fórmula:

$$rational = \frac{binNumber}{2^m}$$

Por ejemplo, supongamos que $binNumber$ toma el valor 25 y el formato de representación escogido es Q4.4. Aplicando la fórmula, obtenemos que el número racional asociado al valor entero 25 en formato Q4.4 es 1.5625.

A la hora de operar con este formato hay que tener en cuenta la posibilidad de over/underflow al realizar la suma, por ello es necesario añadir un bit más de resolución a la parte entera. Al realizar esto el formato de los bits resultante de la operación queda alterado siguiendo la siguiente regla [29]:

$$Q(n+1).m = Qn.m + Qn.m$$

Nótese que para poder realizar dicha operación, es necesario que ambos operandos sigan el mismo formato.

A su vez, al realizar multiplicaciones hay que tener en cuenta que el número de bits necesarios para poder representar el resultado es igual al producto de multiplicar el número de bits de ambos operandos. Por ello el formato de los bits de salida cambia siguiendo la regla [29]:

$$Q(a+x).(b+y) = Qa.b \cdot Qx.y$$

Nótese que para poder realizar dicha operación, no es necesario que ambos operandos sigan el mismo formato.

El uso de este tipo de aritmética ha sido necesaria en el proyecto para poder operar con números racionales a un bajo coste computacional. Al operar en punto fijo, conlleva una

optimización tanto en tiempo de procesamiento como en recursos *hardware* utilizados, ya que las operaciones realizadas utilizan los mismos módulos aritméticos que se utilizan al operar con números enteros.

## 2.7. Protocolo I2S

El protocolo IIS (*Integrated Interchip Sound*) [30] desarrollado por la empresa Philips, es utilizado en este proyecto para transmitir muestras digitales de audio al códec CS4344 [31]. Las muestras están codificadas en C2. Este protocolo serie síncrono, define las siguientes señales y comportamientos asociados, para poder realizar el envío/recepción de los bits que componen una muestra.

- **MCLK**: correspondiente a *Master Clock*. Esta señal define el reloj maestro que permite mantener en todo momento la sincronización durante el envío/recepción de los bits.
- **LRCLK**: abreviatura asociada a *Left-Right Clock*, conocido también como reloj de muestra. Esta señal permite indicar el canal al que están siendo enviados/recibidos los bits correspondientes a una muestra.
- **SCLK**: abreviatura correspondiente a *Serial Clock*, también referida como *Bit Clock*. Esta señal indica el momento en que los bits deben ser volcados (flanco de subida) y capturados (flanco de bajada).
- **STDI**: señal en la que se se vuelcan los bits correspondientes a una muestra recibida.
- **STDO**: señal en la que se se vuelcan los bits correspondientes a una muestra enviada.

En la siguiente imagen se muestra el patrón de onda que se pretende generar por cada una de las señales. Las señales DIN y DOUT se corresponden con STDI y STDO respectivamente:

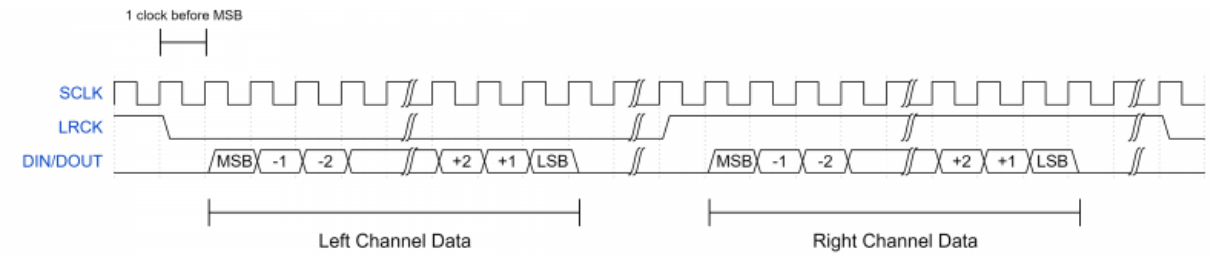


**Figura 2.11**: *Diagrama de tiempo a seguir por las señales del protocolo I2S* [30]

## 2.8. Protocolo SPI

El *Serial Peripheral Interface* [18] es un protocolo serie utilizado para la comunicación entre uno o varios *chips hardware*. En este proyecto se ha hecho uso de él para comunicar el sistema con una memoria Flash externa que almacena de manera no volátil las *Wavetables* (§4.11). Se trata de un protocolo serie síncrono, en el que se establece una comunicación maestro-esclavo, siendo el componente maestro el encargado de generar las señales CS y SCK para el resto de componentes esclavos. Puede ser configurado para que trabaje con una (SPIx1), dos (SPIx2) o cuatro (SPIx4) líneas para el intercambio de datos entre los componentes. En función de las variantes anteriormente mencionadas, el intercambio de datos se realiza con mayor o menor velocidad siendo la más veloz SPIx4 y la menor SPIx1.

Las líneas que define este protocolo son:

- **CS/ss_n**: línea de selección del esclavo activa a baja. Permite seleccionar al componente esclavo correspondiente, estando este preparado para recibir órdenes por parte del componente maestro.

- **SCK**: línea que sirve como referencia de sincronización para los componentes. Permite indicar el momento de volcar y capturar los bits por las líneas IO0 y IO1. Los bits se vuelcan en los flancos de bajada y se capturan en los flancos de subida.

- **IO0**: línea de entrada/salida que sirve tanto para el envío de los correspondientes bits que representan instrucciones/comandos y atributos asociados(por ejemplo, direccio-

nes a datos), como para la recepción de los bits correspondientes a los datos solicitados. En caso de utilizar la variante simple, esta línea solo sirve para el envío de datos.

- **IO1/IO2/IO3**: líneas de entrada/salida que sirven tanto para el envío como la recepción de los correspondientes bits de datos entre el componente esclavo seleccionado y el componente maestro.

El esquema de funcionamiento de este protocolo sigue una serie de etapas:

1. Selección del componente esclavo (activación de CS) con el que el componente maestro quiere comunicarse.

2. Envío por la línea IO0 del comando/instrucción a procesar por el componente esclavo.

3. En función del tipo del comando/instrucción enviado en la anterior etapa, se procederá al envío de datos por parte el componente esclavo (esclavo transmite datos, maestro recibe) o por parte del componente maestro (maestro transmite datos, esclavo recibe). En función de la variante del protocolo el envío de datos se realiza el número de líneas utilizadas para la realización de esta etapa varía. Así, en caso de la variante SPIx1 solo se define una sola línea (IO1), para SPIx2 se definen dos líneas (IO0/IO1), y para SPIx4 cuatro líneas (IO0/IO1/IO2/IO3) para el envío de datos.

4. Para finalizar/interrumpir la comunicación entre el esclavo y el maestro, basta que el maestro deseleccione al esclavo mediante la desactivación de la señal CS.

A continuación se adjunta un diagrama de tiempo en el que el componente esclavo requiere de una espera de 8 ciclos antes de realizar el envío de datos al componente maestro:

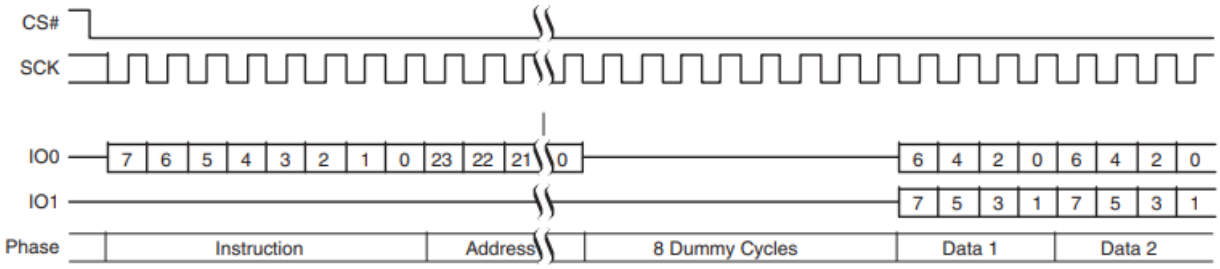


**Figura 2.12**: *Diagrama de tiempo a seguir por las señales del protocolo SPIx2, ejemplificando la evolución de las señales debido al envío de un comando seguido de la dirección de inicio*

[18]

## 2.9. Bluethooth y RS-232

En este proyecto la tecnología *Bluetooth* ha sido utilizada para la comunicación con la interfaz externa y el piano digital (§4.12). El uso de dicha tecnología inalámbrica se ha realizado mediante la interacción con el chip RN-42 de la compañía Roving Networks [32], integrado en el "Pmod" BT2. Este chip provee una interfaz de uso basada en el protocolo RS-232, así mediante este protocolo se reciben los bits de datos que a su vez han sido recibidos por el integrado RN-42 mediante una conexión *Bluetooth*.

El protocolo RS-232 [33] es un protocolo asíncrono serie de transmisión de bytes. Por ello mediante el uso de unos bits de marcaje ubicados al inicio, bit de start, y al final, uno o más bits de stop, de la trama de bits de datos enviada, permitiendo así mantener la sincronización durante la comunicación. También el protocolo ha sido concebido para añadir un bit de paridad ubicado justo antes del/los bit/s de stop, característica no presente en el uso que se le ha dado a este protocolo en este proyecto.

Este protocolo define las líneas Tx y Rx utilizadas para el envío y recepción respectivamente. A su vez, es necesaria la definición de un par más de líneas para notificar el envío y la recepción de las tramas de bits de datos transmitidos.

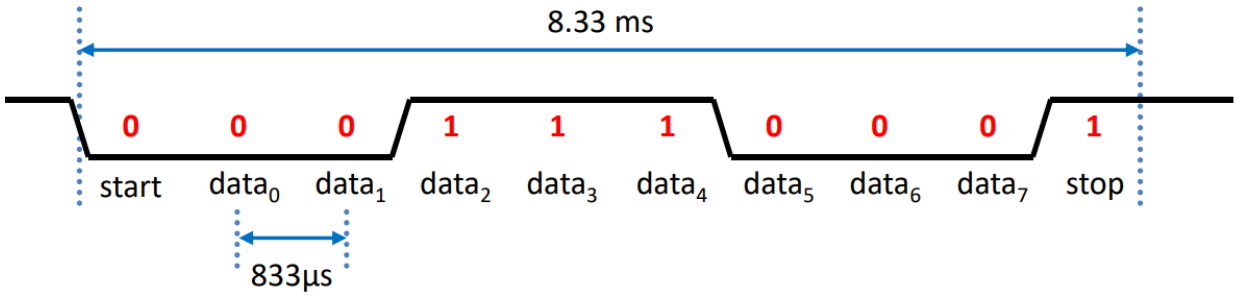


**Figura 2.13**: *Diagrama de tiempo seguido por el protocolo RS-232, configurado a una velocidad de transmisión de 1200bps, 8 bits de datos, sin bit de paridad, y con un bit de stop* [33]

## 2.10. Modulación PWM

La modulación por ancho de pulsos PWM [34], consiste en la generación de pulsos que varían en su ciclo útil en función de una señal analógica de entrada o de un valor digital determinado. El uso de PWM en el control ha sido ampliamente estudiado en los últimos años, y su implementación tradicionalmente se ha basado en soluciones analógicas, con microcontroladores o basadas en máquinas de estados, siendo esta última la que se ha utilizado en este proyecto.

En la modulación por ancho de pulsos PWM, se transforma una señal analógica en una señal digital con ancho de pulso proporcional al valor de la señal analógica previamente muestreada. El muestreo de la señal debe ser por lo menos el doble de su ancho de banda, para garantizar la correcta modulación de la señal. A continuación se muestra una imagen que ejemplifica la aplicación de la modulación PWM partiendo de una señal analógica muestreada:

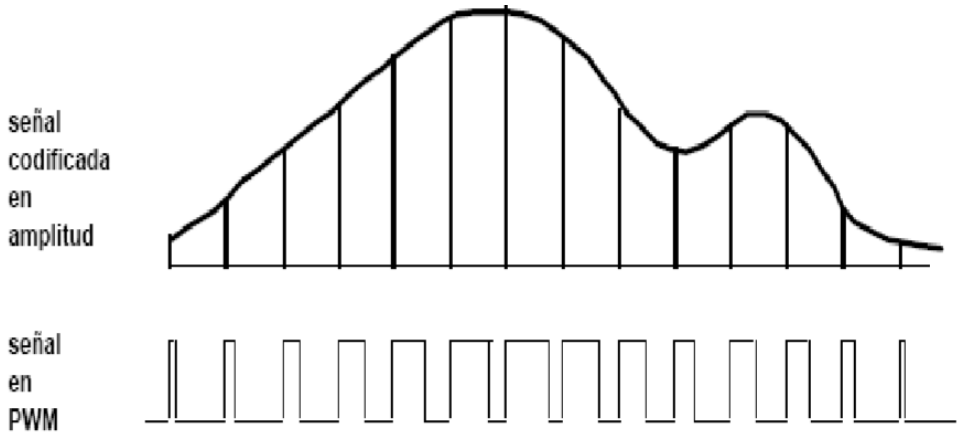


**Figura 2.14**: *Ejemplo de modulación PWM*
[34]

En el desarrollo del piano digital, el uso de esta técnica de modulación ha sido utilizada para la regulación de la intensidad de iluminación de los leds RGB que dispone la placa (§4.9.1).

## 2.11. Herramientas de desarrollo

A continuación, se procede a describir las herramientas y lenguajes de programación que se han utilizado para el desarrollo del proyecto. Cada uno tiene su razón de uso que se procede a explicar.

### 2.11.1. Consola linux, sublime text 3 y C

Se han desarrollado dos aplicaciones de consola previas al desarrollo *hardware*:

- **Midi Parser**. Este programa ha permitido concretar el algoritmo y conjunto de operaciones a seguir para poder interpretar correctamente los bytes de un archivo MIDI y así poder reproducir una canción.

- **Note Conversor**. Este programa pretende comprobar la viabilidad de la técnica de síntesis de sonido mediante *Wavetables* con aritmética en punto fijo. Este programa

lee y genera un archivo .WAV con las muestras resultantes de aplicar el algoritmo de interpolación explicado anteriormente (§2.3).

Ambos programas a realizar se caracterizan por trabajar a nivel de bytes con lo que es conveniente el uso de un lenguaje de programación que esté diseñado específicamente para trabajar a ese nivel. Por ello he decidido utilizar el lenguaje de programación C.

El lenguaje de programación C fue creado por Brian Kernighan y DennisRitchie a mediados de los años 70 [35]. El lenguaje C es un lenguaje para programadores en el sentido de que proporciona una gran flexibilidad de programación y una muy baja comprobación de incorrecciones, de forma que el lenguaje deja bajo la responsabilidad del programador acciones que otros lenguajes realizan por sí mismos. Esto permite una gran adaptación de los programas para una plataforma *hardware* en concreto, siendo esta adaptación tan buena como bueno sea el programador. Este lenguaje es el utilizado para el desarrollo de aplicaciones embebidas, sistemas operativos y demás aplicaciones que requieran un trabajar a un nivel de abstracción medio-bajo. Al tener que trabajar directamente con los bytes de los archivos, es unos de los lenguajes que mejor se adapta a las necesidades de ambas aplicaciones.

Bajo mi perspectiva, lo más adecuado para el desarrollo en C es el uso de alguna distribución Linux ya que el desarrollo de este sistema operativo basado en UNIX, esta profundamente ligado a este lenguaje de programación. La distribución finalmente utilizada es Lubuntu, versión ligera de Ubuntu basada en Debian. El uso de la consola de Linux para la compilación y ejecución del programa mediante el uso del archivo Makefile es común en el proceso de desarrollo de ambos programas.

El código fuente de ambos programas se ha escrito usando la aplicación sublime text 3 [36], un editor de texto con el que estoy muy familiarizado.

### 2.11.2. Vivado y VHDL

Para el desarrollo del diseño *hardware* del piano digital, se ha utilizado una aplicación de diseño HDL proporcionada por Xilinx, Vivado versión de 2018. Esta aplicación es necesaria

debido a la compatibilidad con el chip FPGA que se utiliza en la placa Nexys 4 DDR. Esta aplicación soporta los lenguajes Verilog y VHDL.

**Figura 2.15**: *Logotipo de la aplicación Vivado*
[37]

El lenguaje HDL utilizado es VHDL. VHDL surge a principios de los 80 de un proyecto DARPA (Departamento de Defensa de los EE.UU.) llamado VHSIC – Very High Speed Integrated Circuits [38]. VHDL no es un lenguaje de programación, por ello conocer su sintaxis no implica necesariamente saber diseñar con él. VHDL es un lenguaje de descripción de *hardware*, que permite describir circuitos síncronos y asíncronos. En particular VHDL permite tanto una descripción de la estructura del circuito (descripción a partir de subcircuitos más sencillos), como la especificación de la funcionalidad de un circuito utilizando formas familiares a los lenguajes de programación. La misión más importante de un lenguaje de descripción HW es que sea capaz de simular perfectamente el comportamiento lógico de un circuito sin que el programador necesite imponer restricciones [39].

La interacción entre la FPGA y Vivado se lleva a cabo mediante una conexión USB pudiendo así cargar el archivo *bitstream* para configurar la matriz de interconexión. A su vez permite la escritura en la memoria Flash de la placa, característica esencial para poder así volcar el audio muestreado.

### 2.11.3. Android Studio y Java

Por último para el desarrollo de una interfaz externa que permita la interacción del usuario con el piano digital, se ha optado por realizar una aplicación Android para dispositivos

móviles. La razón principal de esta decisión es debido a la gran flexibilidad que ofrece el programar para este sistema operativo, pudiendo utilizar los servicios del mismo con relativa facilidad. Esto será necesario para establecer la comunicación *Bluetooth* con el piano digital pudiendo así accionar las notas y volcar archivos MIDI en la placa para su reproducción. Una de las herramientas más populares para el desarrollo de aplicaciones Android, es Android Studio. Android Studio permite ejecutar y depurar la aplicación desde cualquier smartphone pudiendo así tener una visión más acertada del comportamiento y aspecto final de la aplicación. La aplicación en desarrollo se instala en el dispositivo, previamente habiendo hecho una configuración del mismo, mediante una conexión USB. Bajo esta herramienta se suele desarrollar en Kotlin o Java. En este caso se desarrollará la aplicación en Java debido a la previa experiencia de programación que tengo sobre este lenguaje.

**Figura 2.16**: *Logotipo de SO Android*
[40]

Java comienza a desarrollarse en 1991. En 1995 la primera versión de Java es publicada por Sun Microsystems [41]. Java es un lenguaje de programación orientado a objetos y fuertemente tipado. Es lo que se conoce como un lenguaje interpretado, es decir, el código escrito por el programador es interpretado por una máquina virtual. A partir de este código fuente, la máquina virtual genera un código en formato bytecode que acaba traduciéndose a las instrucciones máquina de la plataforma *hardware* objetivo. Java es famoso por ser un lenguaje multisistema independiente de la arquitectura *hardware* subyacente, esto permite

una gran reuso de código. Para que un programa funcione solo es necesaria la instalación de la máquina virtual de Java. Otra de las ventajas es la facilidad para depurar el código ya que la máquina virtual se encarga gestionar este proceso, lo que acelera el desarrollo de aplicaciones. El inconveniente principal de este lenguaje de programación es que requiere un equipo con mayores capacidades para su correcto funcionamiento debido a que la gestión de memoria del sistema no siempre es la más óptima y el código resultante no se ejecuta tan rápido como el código de otros lenguajes compilados. Se pierde rendimiento para ganar en flexibilidad.

# Capítulo 3

# Aplicaciones auxiliares de consola

Las aplicaciones de consola que se proceden a explicar en este capítulo, han permitido validar conceptos previamente a su implementación en *hardware* puro. Han sido necesarias ya que el desarrollo en *hardware* implica una gran complejidad y de esta forma se ha podido tener una referencia del resultado que se espera obtener al leer un archivo MIDI y al sintetizar sonido. No se describirán en profundidad ya que son desarrollos software auxiliares al objeto principal de este proyecto que es el diseño de una arquitectura *hardware*.

## 3.1. Midi Parser

Tras definir las características básicas de un archivo MIDI, se procede a explicar la implementación del programa Midi Parser, aplicación de consola desarrollada en C, que vuelca por la salida estándar una descripción de los mensajes contenidos en un archivo MIDI. Este programa se encuentra en un repositorio de Github bajo una licencia GPLv3.0, en la siguiente URL: https://github.com/fernandoka/MidiParser.git

### 3.1.1. Características de los archivos MIDI soportados

Los archivos MIDI soportados en este proyecto deben cumplir los requisitos correspondientes al formato 0 o 1, quedando el formato 2 excluido. Por esta razón, todas las pistas siguen el mismo tempo/ritmo, no son pistas independientes. Es necesario que el valor del

campo *division* este comprendido en el intervalo [1, 32767]. Los canales MIDI no se soportan, sencillamente son ignorados.

A continuación, se procede a una recapitulación de la sintaxis de los mensajes MIDI de los que se ha sido capaz de reconocer. Las letras en minúscula hacen referencia a la adquisición de cualquier valor de ese dígito hexadecimal :

- Meta-eventos
  - *Fin de Pista*: **FF 2F 00**
  - *Establecer tempo*: **FF 51 03 tttttt**
  - *Armadura de tonalidad*: **FF 59 02 sf mi**
- Eventos SysEx: < **F7** | **F0** > <**longitud**> <**bytes a ser transmitidos**>
- Eventos MIDI. La letra "q" hace referencia a los dígitos hexadecimales que forman el primer byte de datos, y la letra w representa lo mismo para el segundo byte de datos.
  - *Note-On*: **9n qq ww**
  - *Note-Off*: **8n qq ww**
  - *Program Change*: **Cn qq**
  - *Channel Key Pressure*: **Dn qq**
  - *Control Change Suntain on/off*: **Bn qq ww**

Los mensajes MIDI anteriores son reconocibles por la aplicación de consola Midi Parser (§3.1). En el caso del intérprete *hardware* de archivos MIDI habría que descartar los mensajes de "Armadura de tonalidad" y "Control Change Suntain on/off".

Para reconocer con éxito dichos mensajes hay que resolver dos problemáticas principales: interpretación correcta de los valores numéricos representados en formato de longitud variable e implementar correctamente el *Running Status*.

### 3.1.2. Interpretación correcta de los valores numéricos representados en formato de longitud variable

La función en el código fuente de la aplicación encargada de devolver un entero de 32 bits es la siguiente:

```
static long int readVarLength(int *numBytesReaded){
        long int val;
        unsigned char c;

        *(numBytesReaded) = 1;

        if ((val = getNextByte()) & 0x80){
                val &= 0x7f;
                do{
                        val = (val << 7) +
                                ((c = getNextByte()) & 0x7f);
                        *(numBytesReaded)+=1;
                }while(c & 0x80);
        }

        return val;
}
```

Esta función está completamente basada en el pseudo-código proporcionado por la especificación MIDI. Como se puede observar el algoritmo a seguir es relativamente sencillo. Después de la lectura del primer byte, se ejecuta un bucle que continúa leyendo bytes hasta que el bit más significativo del byte que se ha leído sea 0. En cada iteración hay que poner el bit más significativo del byte leído a 0 y desplazar 7 bits a la izquierda los bits de una variable utilizada como acumulador, esta contendrá al final de la última iteración el valor numérico buscado. La variable *numBytesReaded* no forma parte del algoritmo como tal, almacena el número de bytes leídos.

Esta función necesita ser utilizada cada vez que se deba leer un tiempo-delta, es decir cada vez que se inicie la lectura de un evento MTrk. A su vez también debe ser utilizada para la lectura correcta del campo longitud tanto en un evento SysEx como en un Meta-evento.

### 3.1.3. Implementación del *Running Status*

El *Running Status* hace referencia a una manera de interpretar los bytes, propia del estándar MIDI. Sirviendo como breve recapitulación, decir que el *Running Status* tiene en cuenta el byte de estado del último mensaje MIDI recibido, permitiendo reducir el número de bytes a transmitir y como consecuencia reduciendo el tamaño del archivo. La función en el código fuente que implementa esta característica se muestra a continuación:

```
static bool_t readEvents(int length, mtrk_event_t *eventsList){

        int n,bytesReaded;
        unsigned char status,c;

        n = 0;
        while( length > 0 && n < MAX_EVENTS ){

                eventsList[n].deltaTime =
                        readVarLength(&bytesReaded);
                c = getNextByte();
                length-=bytesReaded+1;

                if( c & 0x80 ){
                        status = c;
                }
                else{
                        seekNbytes(-1);
                        length++;
                }

                parseEvent( &length,status,&(eventsList[n]) );
                printEvent(eventsList[n]);
                n++;
        }

        file.track[file.numTracks].numEvents = n;

        return eventsList[n-1].type == META_EVENT &&
                    eventsList[n-1].meta.type == END_OF_TRACK &&
                    eventsList[n-1].meta.length == 0 &&
                    n<= MAX_EVENTS;
}
```

Es en esta función en donde se define el bucle principal para la lectura de un archivo MIDI. Tiene como argumento de entrada un entero *length* en el que está almacenado el número de bytes que se esperan leer, y como argumento de salida se sirve de una lista de eventos denominada *eventsList*. Para esta lista se ha definido el tipo específico *mtrk_ event* que nos sirve para guardar los eventos de una pista (este tipo viene definido en la cabecera del código fuente, archivo midi_parser.h). Así esta función almacena los eventos que va leyendo a partir de la dirección proporcionada por el puntero de la variable *eventsList*. Siguiendo la estructura de un evento MTrk, se procede a la lectura del tiempo-delta haciendo una llamada a *readVarLength*. Seguidamente, el *Running Status* se implementa sencillamente comprobando si el último byte leído es un byte de estado o no, para ello se comprueba el bit más significativo con el uso de una máscara (con valor 80 en hexadecimal) y la operación lógica AND. En caso de que el byte leído sea un byte de estado, en la variable *status* se copia el valor de dicho byte, en caso contrario el byte leído era un byte de datos por lo que debemos retroceder el descriptor de fichero una posición e incrementar el valor de *length* para que la función *parseEvent* tenga en cuenta dicho byte. Es en la variable *status* donde se almacena el último byte de estado leído, define así el estado del *Running Status* en un tiempo determinado. Decir que el bucle principal itera hasta que no quede ningún byte por leer (comprobando si el valor de *length* es mayor a 0) o se llene la lista de eventos MTrk, esta lista es de tamaño fijo y se define con una constante en tiempo de compilación (no se ha visto necesario el uso de memoria dinámica). Finalmente, la función retorna un valor booleano que depende si la lectura ha finalizado correctamente, es decir si el último evento guardado se corresponde a un Meta-evento *Fin de Pista*, no se espera ningún byte por leer (argumento *length* es igual a 0) y no se ha sobrepasado el número máximo posible de eventos a guardar por el programa.

### 3.1.4. Conclusiones Midi Parser

Tras explicar estas dos problemáticas, el resto de funciones del código fuente del programa se dedica principalmente a comprobar de forma ordenada los tipos de bytes que se van leyendo del archivo, comprobando si estos bytes coinciden con los bytes esperados por el tipo de mensajes MIDI anteriormente mencionados. Con esto se ha logrado tener un programa que reconoce los mensajes MIDI de mayor relevancia para la reproducción de una pieza musical. A continuación, se muestra parte de la salida obtenida del parseo de un archivo MIDI de ejemplo:

```
Reading midiFiles/Papermoon.mid
Reading Header Chunk
        Length: 6
        Format: 1
        ntrks: 2
        division: 480

 ------------------- Starting Reading Track Chunk -------------------
Length of Track Chunk 1800

Delta Time value: 0
 Event type hex: ff
        Meta Event
        Time Signature Event, b1: 4 b2:2 , time Signature is: 4/4


Delta Time value: 0
 Event type hex: ff
        Meta Event
        Key Signature Event, b1: fe b2:0 , key signature is:
                Sib Major


Delta Time value: 0
 Event type hex: ff
        Meta Event
        Set Tempo Event, length value: 3 , us: 444444

Delta Time value: 0
 Event type hex: b0
```

```
        Control Change Event
        B1 code 79 , B1 val 121 , B2 code 0 , B2 val 0


Delta Time value: 0
 Event type hex: c0
        Program Change

Delta Time value: 0
 Event type hex: b0
        Control Change Event
        B1 code 7 , B1 val 7 , B2 code 64 , B2 val 100


Delta Time value: 0
 Event type hex: b0
        Control Change Event
        B1 code a , B1 val 10 , B2 code 40 , B2 val 64


Delta Time value: 0
 Event type hex: b0
        Control Change Event
        B1 code 5b , B1 val 91 , B2 code 0 , B2 val 0


Delta Time value: 0
 Event type hex: b0
        Control Change Event
        B1 code 5d , B1 val 93 , B2 code 0 , B2 val 0


Delta Time value: 0
 Event type hex: ff
        Meta Event


Delta Time value: 0
 Event type hex: 90
        Note ON event
        Key code 43 , Key val 67 , vel: 64 note: G4
```

```
Delta Time value: 455
 Event type hex: 90
        Note ON event
        Key code 43 , Key val 67 , vel: 0 note: G4
```

Como conclusión, decir que con el desarrollo de esta aplicación se ha demostrado la viabilidad de poder extraer mensajes MIDI, habiendo así tenido éxito en este objetivo parcial podemos extrapolar este proceso a una plataforma puramente *hardware* para así realizar el piano digital.

## 3.2. Note Conversor

Este programa se encuentra en un repositorio de Github bajo una licencia GPLv3.0, en la siguiente URL: https://github.com/fernandoka/NoteConversors.git

La aplicación de consola Note Conversor se hizo con el propósito de comprobar empíricamente la viabilidad de la fórmula de interpolación, teniendo así una muy buena aproximación del resultado a obtener en la implementación *hardware* que se realizará en el diseño del piano digital. Por ello se ha realizado la aplicación con dos distintas implementaciones de la fórmula de interpolación, una opera con aritmética en punto flotante (carpeta NoteConversor_double_3) y otra en punto fijo (NoteConversor_fix_3). La implementación de esta última supuso al mismo tiempo un ejercicio de ensayo ya que la implementación *hardware* se realiza también en punto fijo.

Ambas versiones del programa comprueban si la cabecera del archivo leído sigue el formato WAVE (archivos con extensión .wav). Seguidamente leen las muestras contenidas en dicho archivo ubicándolas en memoria y operando con ellas. Tras realizar la interpolación de todas las muestras, se procede a copiar la cabecera del archivo de entrada actualizando el tamaño en bytes según el número de muestras que hayan sido interpoladas. Finalmente, vuelca tanto la cabecera actualizada y las muestras interpoladas en un archivo de salida nombrado de la misma forma que el archivo de entrada pero concatenando al final la cadena de caracteres *_cool*.

### 3.2.1. Detalles de implementación

Un detalle de implementación consiste en la organización de las muestras en memoria siendo tratadas como una matriz. Esto se realiza así debido a que facilita la indexación de las muestras al aplicar la fórmula de interpolación. El sonido puede estar en mono o en estéreo, en caso de que el sonido sea estéreo las muestras de cada canal se almacenan intercaladas, viendo esta organización como una matriz permite una fácil identificación de muestra. Una posible ejemplificación de cómo se almacenan las muestras de audio estéreo aparece a continuación, el primer índice sirve para identificar el canal y el segundo índice sirve para identificar la muestra por canal:

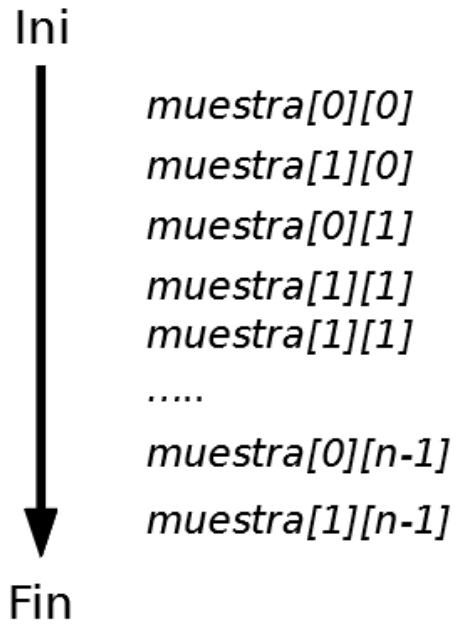


**Figura 3.1**: *Representación de organización de muestras en un archivo .wav*

En caso de audio mono solo hay un canal de audio, por lo que todas las muestras van dirigidas a ambos canales. Parte del código fuente que realiza la organización de las muestras en memoria como una matriz, llama a la función de interpolación y organiza las muestras resultado de la interpolación para su posterior escritura en el archivo de salida, se muestra

a continuación:

```
// The samples are seen like if they are stored in a matrix
samples = (int*) malloc(sizeof(int)*totalSamplesInFile);
outSamples = (int*) malloc(totalSamplesInFile*sizeof(int));

wavDataIndex = 0;
for(int j = 0; j < width; j++)
        for(int i = 0; i < height; i++){
                samples[i*width+j] = rawDataToInt(wavData,
                        wavDataIndex, bytesPerSample);
                wavDataIndex += bytesPerSample;
        }


//Interpolate samples
numOfInterpolatedSamplesPerChannel = interpolateSamples(height,
        width, samples, outSamples);

wavDataIndex = 0;
for(int j = 0; j < width; j++)
        for(int i = 0; i < height; i++){
                intToRawData(wavData, outSamples[i*width+j],
                        wavDataIndex, bytesPerSample);
                wavDataIndex += bytesPerSample;
        }
```

La variable *height* (alto/filas de la matriz) hace referencia al canal, y *width* (ancho/-columnas de la matriz) hace referencia al número de muestras por canal. El primer par de bucles *for*, realiza la conversión de los bytes de datos a muestras indexadas como una matriz, y en el segundo par de bucles *for* se encargan de hacer la operación inversa con las muestras resultantes de la interpolación (pasar de indexar muestras a indexar bytes) para su posterior escritura en el archivo de salida.

El cometido de la función *rawDataToInt*() consiste en organizar los bytes de datos para poder indexarlos en un array de muestras (las muestras son valores enteros, recordemos que están codificados en punto fijo siguiendo el formato PCM). En el caso de la función *intToRawData*() se encarga de hacer la operación inversa, es decir organizar los bytes de las muestras para poder indexarlos en un array de bytes.

Esto es necesario ya que al leer los datos del fichero estos vienen indexados como bytes y se deben representar estos bytes como muestras para poder operar cómodamente con ellas. La cantidad de bytes que ocupa cada muestra viene definido previamente en la cabecera del archivo.

La implementación de las dos funciones anteriormente mencionadas es la siguiente:

```
static int rawDataToInt(unsigned char *c, int index, int size){
        int n,j,i;

        j = n = 0;
        for (i = index; i < index+size-1; i++){
                n =  c[i]<< 8*j | n;
                j++;
        }

        n |=  ((signed char)c[i]<< 8*j);

        return n;
}

static void intToRawData(unsigned char *c, int n, int index,
                int size){
        int i;

        for(i = index; i < index+size;i++){
                c[i] = (unsigned char)(n & 0xFF);
                n >>= 8;
        }

}
```

Como se puede observar la función *rawDataToInt*() tiene tres argumentos de entrada: *c* es el puntero a la primera dirección del array de bytes leídos del archivo de entrada, *index* almacena el índice del primer byte que se tiene que tener en cuenta para realizar la conversión, y *size* que indica el número de bytes por muestra. Así sumando el valor de *index* y *size* se obtiene el índice del último byte que representa una misma muestra. Este último byte hay que tratarlo de forma distinta respecto a los otros, ya que este byte es el más significativo (codificación del valor en little endian) y por consiguiente se debe tratar

como un valor con signo (casting con *signed char*). Esto explica el por qué el bucle itera hasta el penúltimo byte incluido. Los bytes se vuelcan en la variable entera $n$ (inicializada previamente a 0) mediante la operación lógica OR. Estos bytes se ubican correctamente en la variable entera gracias al desplazamiento de bits realizado por $c[i] << 8 * j$.

Los argumentos de la función $intToRawData()$ son: $c$ argumento de salida que guarda el puntero a la primera dirección del array de bytes que serán escritos en el archivo de salida, $n$ argumento de entrada que representa la muestra a descomponer en bytes, los argumentos $index$ y $size$ representan lo mismo que en la función $rawDataToInt$. El bucle $for$ en esta función va guardando los bytes menos significativos de la variable $n$ en el array representado por $c$. Esto se hace mediante el uso de una máscara ($0xFF$) y la operación lógica AND. A su vez por cada iteración se desplazan 8 bits a la derecha para que estos ocupen la posición del último byte dentro de la muestra, así en la próxima iteración del bucle se tendrá en cuenta este byte.

Cabe destacar la implementación de la fórmula en punto fijo. Parte del código fuente asociado a esta es el que precede:

```
do{

        if( decimalPart == 0 ){

                for (int i = 0; i < height; ++i)
                        outSamples[i*width+j] =
                                samples[i*width+integerPart];
        }
        else{

        for (int i = 0; i < height; ++i){
                outSamples[i*width+j] = roundCheckOverUnderFlow(
                        ( ((long long int )
                        samples[i*width+integerPart])<<QM_ARITH )
                        +
                        ((long long int)decimalPart
                        *
                        (long long int)
```

```
                        (samples[i*width+integerPart+1]
                        -
                        samples[i*width+integerPart]))
                        );
                }//for
        }//else

        ci += fix_step;
        // Take the integer part
        integerPart = (unsigned int)(ci>>QM_ARITH);
        // Take the decimal part
        decimalPart = (unsigned int)(ci & 0xFFFFFFFF);
        j++;

// Until I can do the interpolation.
//Width is the number of samples per channel
}while( integerPart+1 < width );
```

Como se puede apreciar en el código, la evolución de *ci* en cada iteración consiste en realizar la suma entre el propio valor de *ci* y de *fix_step*. La variable *ci* se inicializa a 0, y *fix_step* se inicializa con el valor correspondiente de dividir la frecuencia objetivo y la frecuencia base, frecuencias correspondientes a la sinusoide que queremos generar y de la sinusoide que usamos sus muestras para interpolar. Definiendo así la variable *ci*, su evolución es equivalente a multiplicar *fix_step* por el índice de muestra por el que estamos, lo que se corresponde con la definición dada para esta variable en un primer momento. Mediante el uso de un desplazamiento y de una máscara ($0xFFFFFFFF$), se obtiene por separado los bits que representan la parte entera y la parte decimal respectivamente.

El comportamiento dentro del bucle depende de *decimalPart*, que a su vez depende directamente de *ci*. Si el valor de la parte decimal de *ci* (almacenado en la variable *decimalPart*) es igual a 0, se procede simplemente a copiar el valor de la muestra indexada por el valor de la parte entera de *ci* (almacenado en la variable *integerPart*) en el array *outSamples*. Esto es debido a que la multiplicación por 0 anula toda la parte a la derecha de la suma en la fórmula. El array *outSamples* usa la variable $j$ como índice de muestra interpolada. En caso de que *decimalPart* sea distinto de 0, se procede a aplicar todas las operaciones

de la fórmula. Estas operaciones aritméticas están realizadas teniendo una representación en punto fijo de Q32.32, la constante $QM_ARITH$ se inicializa con valor 32. En la gran mayoría de los sistemas actuales el tipo de *long long int* nos asegura una representación numérica mínima de 64 bits necesaria para operar en Q32.32.

Comenzando con la resta, esta se realiza sin ninguna peculiaridad, el resultado de la resta estará en este caso en formato Q2.QM siendo QM el numero de bits que representan la parte decimal. QM depende de cuantos bits representen cada muestra, esto viene definido en la cabecera del archivo. Al realizar la multiplicación hay que tener en cuenta que la representación debe ser con el mismo número de bits tanto de los valores de $decimalPart$ como del resultado de la resta para obtener así precisión al realizar la multiplicación. Por ello se realiza la conversión al tipo *long long int* tanto del resultado de la resta como de $decimalPart$. La variable $decimalPart$ sigue un formato en punto fijo de Q32.32, esto implica que el resultado de la multiplicación estará en formato Q34.QM+32. Debido a que el resultado de la multiplicación sigue el formato Q34.QM+32, para realizar correctamente la suma es necesario desplazar 32 bits a la izquierda el valor indexado por la parte entera de $ci$ ($integerPart$) en el array de muestras. Este desplazamiento obliga a realizar la conversión a *long long int* para poder tener resolución. De esta forma están alineadas la parte entera y la parte decimal al realizar la suma (ambos operandos tienen el mismo número de bits representando su parte decimal). Lo último a realizar es redondear el valor obtenido, para ello se utiliza la función $roundCheckOverUnderFlow()$.

El código fuente de dicha función es:

```
static int roundCheckOverUnderFlow(long long int n){
        int aux, max, min;

        max = maxFormatValues[file.numBitsPerSample/8-1];
        min = minFormatValues[file.numBitsPerSample/8-1];

        n += (long long int)1<<(QM_ARITH-1);
        aux = (int)(n >> (QM_ARITH));

        if (aux > max)
                aux = max;
        else if(aux < min)
                aux = min;

        return aux;
}
```

Esta función solo consta de un argumento $n$ que representa el valor a redondear. La técnica de redondeo utilizada consiste en sumar 0.5 al valor a redondear y luego truncar. El valor de 0.5 se define desplazando un 1 a la posición del bit más significativo que forma parte de los bits que representan la parte decimal de un número en punto fijo. Esto mismo representa 1 << *(QM_ARITH-1)*, colocando así un 1 en el bit 31. Para obtener resolución, la conversión de tipos es necesaria. Después para truncar se desplazan los bits a la derecha quedando así QM bits decimales. Solo queda saturar, esto es necesario ya que se puede dar el caso que el valor obtenido tras truncar no pueda ser representable con el formato en el que se beben guardar las muestras produciéndose over/under-flow (recordemos que este formato es Q1.QM, siendo QM el número de bits de la parte decimal que es obtenido al leer la información de la cabecera del archivo). Al saturar comparas si el valor final está dentro del intervalo representable por el formato de salida, en caso de que el valor no perteneciese a este intervalo, el número tomará el valor máximo o mínimo representable en dicho formato dependiendo si se ha producido overflow (valor obtenido mayor al máximo representable) o underflow (valor obtenido menor al mínimo representable). La obtención de los valores de las cotas se realiza mediante la indexación en dos arrarys de constantes ($maxFormatValue$ y $minFormatValue$) obteniendo el valor máximo y mínimo representable dada una reso-

lución de bits por muestra. Ejemplificando, en caso de que la resolución del audio fuera de 24 bits, las correspondientes cotas mínima y máxima se definen respectivamente en hexadecimal como *0xFF800000* y *0x007FFFFF*, almacenando dicho valor en la variable $aux$, esto solo ocurriría en el caso de que este produjese en primera instancia underflow o overflow respectivamente.

### 3.2.2. Conclusiones Note Conversor

Gracias al desarrollo de la aplicación de consola NoteConversor, se ha podido comprobar empíricamente la viabilidad de la fórmula tomando valores de las formas de onda correspondientes a las notas del piano.

Es necesario aclarar que esta técnica de síntesis no carece de limitaciones. Una es la de no poder sintetizar notas de frecuencias inferiores a la frecuencia de la nota base, ya que no se puede realizar una interpolación sin tener un valor que haga de cota inferior al valor que queremos aproximar. Ejemplificando, en caso de que queramos obtener la nota G4# (415,30Hz) a partir de la nota A4(440Hz), el valor de $ci$ sería de 0,943875, este valor quiere decir que la muestra que intentamos aproximar esta a una distancia de 0,056125 por debajo de la primera muestra de nuestra nota base teniendo solamente una variable de referencia, imposibilitando así la aplicación de la fórmula. Esta misma problemática es extrapolable a la muestra que hace de cota superior.

Esta limitación, acota los valores que puede tomar $ci$ al intervalo $[1, \infty]$.

También hay que tener en cuenta la limitación sobre el número de muestras generadas, siendo este siempre menor o igual al número de muestras que contenga la Wavetable base. Esto es debido a la evolución de la variable $ci$ que en caso de ser mayor a uno, la evolución de su parte entera no siempre se incrementará de uno en uno, provocando que muestras de la Wavetable sean ignoradas.

Sabiendo el número de muestras de la sinusoide base ($WavetableNumSamples$), se puede calcular el número de muestras que serán generadas al interpolar mediante la siguiente

fórmula:

$$NumInterpolatedSamples = WavetableNumSamples \cdot (targetFreq/baseFreq)$$

La última limitación a tener en cuenta, consiste en el incremento del error al sintetizar notas a una distancia mayor a un tono, la aproximación realizada pierde mucha precisión. Esto obliga a tener 4 notas por octava (siendo estas C, D#, F#, A) si se quiere poder generar todas las notas del piano. Esta limitación se ha comprobado empíricamente mediante el desarrollo de la aplicación NoteConversor. Dicha limitación, bajo un contexto puramente teórico, no existe ya que se trabaja con la forma de onda seno, que carece de la complejidad que tienen las formas de onda generadas por los instrumentos musicales. Dicha complejidad es la causante del error cometido al interpolar que se incrementa al realizar el cálculo para notas superiores. Esto es debido que la forma de onda real no solo tiene en cuenta la frecuencia sino también el timbre, la intensidad o la duración. Estos factores también afectan a la forma de onda final.

Esta última limitación hay que tenerla muy presente al diseñar el piano digital ya que impone una restricción al número de notas ha almacenar en la memoria del sistema, siendo necesario almacenar al menos 30 notas (4 notas por octava, exceptuando la octava 0 y la octava 8 que solo es necesario guardar una nota ya que delimitan el teclado). Esto supone la necesidad de disponer de una memoria con capacidad suficiente para almacenar estas notas en el piano digital.

Para reducir el tamaño de los archivos de las notas ha almacenar en la memoria se han modificado: la duración del audio de todas las notas se reducido a 3s provocando que todos los audios tengan el mismo número de muestras, se ha pasado de audio estéreo a mono y los bits de resolución por muestras se han reducido de 24 a 16. Estos cambios, junto a la eliminación de la cabecera de los archivos de audio incluyendo así solo las muestras de audio, han permitido almacenar 30 notas en un archivo binario de 11MB de tamaño.

# Capítulo 4

# Arquitectura *hardware* del piano digital, proyecto MIDI_Soc

En este capítulo se aborda la explicación de las principales características de la arquitectura *hardware* del piano digital, dicho proyecto se ha nombrado MIDI_Soc. Tanto el diseño final como el bloque de datos a cargar en Flash y las simulaciones de algunos componentes, se pueden encontrar en el siguiente repositorio de GitHub bajo una licencia GPLv3.0: https://github.com/fernandoka/Digital-Piano-VHDL-MIDI-Player.git

Para facilitar la explicación, la sintaxis de la que se va ha hacer uso para referenciar el número de líneas por señal viene indicado por un número limitado por corchetes ubicado tras nombrar la señal. También la estructuración llevada acabo en este capítulo se basa en los componentes y subcomponentes que forman el diseño. Algunos componentes y subcomponentes vienen representados inicialmente con un esquemático simplificado y una enumeración de las señales que definen la interfaz de entrada y de salida. Progresivamente, se procede a comentar algunos detalles de la implementación y/o funcionamiento en el componente o subcomponente en cuestión.

Hay que aclarar el significado de la sintaxis "log2(a)". Esto hace referencia a una función que devuelve el valor del logaritmo en base 2 del parámetro $a$. Esta función se utiliza para definir el número de líneas de algunas señales que son parametrizables.

Comencemos pues, con la explicación de las principales variables parametrizables del diseño y una distinción de los posibles modos de funcionamientos. Para luego dar pie, a una visón general de todo el sistema e ir progresivamente detallando cada componente.

## 4.1. Modos de funcionamiento, temporización y parametrización

El sistema diseñado presenta 3 modos distintos de funcionamiento. En el modo **Setup**, el sistema está inactivo a la espera de que se ordenarne la inicialización de la memoria DDR2 SDRAM (de alta velocidad) con el bloque de datos (*Wavetables* y constantes) almacenado en la memoria Flash no volátil (de baja velocidad). Esto se realiza mediante la pulsación del botón BTNC de la placa. El modo **Reproducción MIDI** permite al usuario iniciar/-detener la reproducción de un archivo MIDI (botón BTNU), activación/desactivación del efecto *reverb* (botón BTNL), y la activación/desactivación de la conexión con la interfaz externa (botón BTND). Y el modo de **Teclado interactivo**, permite el completo uso de las funcionalidades que otorga la interfaz externa de las cuales destacan la subida de archivos MIDI, y la reproducción de notas y a acordes interactuando con un teclado.

El sistema, además, debe cumplir un requisito de temporización estricto impuesto por la frecuencia de muestreo a la que opera el audio códec. Esta frecuencia al ser de 48,8KHz implica que este dispositivo debe recibir una nueva muestra aproximadamente cada $0,2049\mu s$. Dada la frecuencia del sistema (75MHz) obtenemos que los ciclos de los que disponemos para realizar todas las operaciones/movimientos-de-datos son de 1536 (75MHz/48,8KHz) para audio estéreo y de 3072 (el doble) para audio mono.

Por último, algunos componentes y subcomponentes presentes en este diseño son parametrizables mediante la asignación de valores a variables genéricas en el código fuente VHDL. A continuación se adjunta una tabla en donde se correlacionan las principales variables genéricas con su significado, los valores permitidos, y el valor final que toma en el diseño publicado en GitHub:

| Variable | Significado | Valores posibles | Valor final |
|---|---|---|---|
| MAX_NUM_TRACKS | Número máximo de pistas compatibles con el sistema. | $n > 1 \quad con \quad n \in \mathbb{N}$ | 6 |
| NUM_NOTES_GEN | Número de generadores de notas (grado de polifonía). | $2^n \quad con \quad n \in \mathbb{N}$ | 16 |
| FIFO_DEPTH_BL | Profundidad del buffer en donde se almacenan los mensajes provenientes de la interfaz externa. | $n > 1 \quad con \quad n \in \mathbb{N}$ | 8 |
| REVERB_TIME | Tiempo en ms del retardo aplicado para obtener el efecto *reverb*. | $n \geq 0 \quad con \quad n \in \mathbb{R}^+$ | 100.0 |
| BAUDRATE_BL | Velocidad de transmisión representada en baudios, establecida para recibir los bits provenientes de la interfaz externa. | $n > 1 \quad con \quad n \in \mathbb{N}$ | 115200 |

**Tabla 4.1**: *Tabla informativa sobre las variables genéricas del diseño*

## 4.2. Visión general

MIDI_Soc es el componente principal que engloba el resto de componentes del sistema siendo así el de mayor nivel de abstracción en el diseño, contiene los siguientes 13 componentes:

- **ClkGen**: componente encargado de generar una señal de reloj a una frecuencia de 200MHz requerida por RamCntrl.
- **ButtonsSyncDebRiseEdge**: componente encargado de adaptar las señales externas provenientes de los botones de la placa.
- **RgbLed**: componente encargado de generar el patrón de iluminación a seguir por los

leds RGB de la placa según el estado actual del sistema.

- **Bin2segNexys4**: componente encargado de generar las formas de onda necesarias para la visualización de dígitos hexadecimales en los *displays* 7 segmentos de la placa.
- **ResetSyncronizer**: componente encargado de adaptar la señal proveniente del botón de reset de la placa, haciendo que tenga una activación asíncrona y desactivación síncrona.
- **issInterface_75Mhz**: componente encargado de realizar el envío serializado de las muestras de audio al códec de audio CS4344 [31], que incluye el "Pmod" I2S2 [19].
- **MySetup**: componente encargado de inicializar la memoria RAM con el bloque de datos guardado en memoria Flash.
- **MainController**: componente encargado de orquestas la correcta activación del resto de componentes según su estado actual.
- **CmdKeyboardSequencer**: componente encargado de secuencializar los comandos de teclado (§4.16), siendo estos los que activan o desactivan las notas.
- **ExternInterfaceCmdReceiver**: componente encargado de interpretar los mensajes provenientes de la interfaz externa.
- **RamCntrl**: componente encargado de secuencializar las solicitudes de lectura de datos por parte de KeyboardCntrl y MidiParser.
- **KeyboardCntrl**: componente encargado de generar las muestras de audio teniendo en cuenta las notas activas y la activación/desactivación del efecto *reverb*. Las notas cambian de estado mediante la recepción de comandos de teclado (§4.16).
- **MidiParser**: componente encargado de leer un archivo MIDI y de generar la correspondiente secuencia de comandos de teclado que representa la interpretación musical contenida en el dicho archivo.

Los de mayor complejidad son RamCntrl, KeyboardCntrl y MidiParser por ser estos los que implementan la mayor parte de la funcionalidad del sistema.

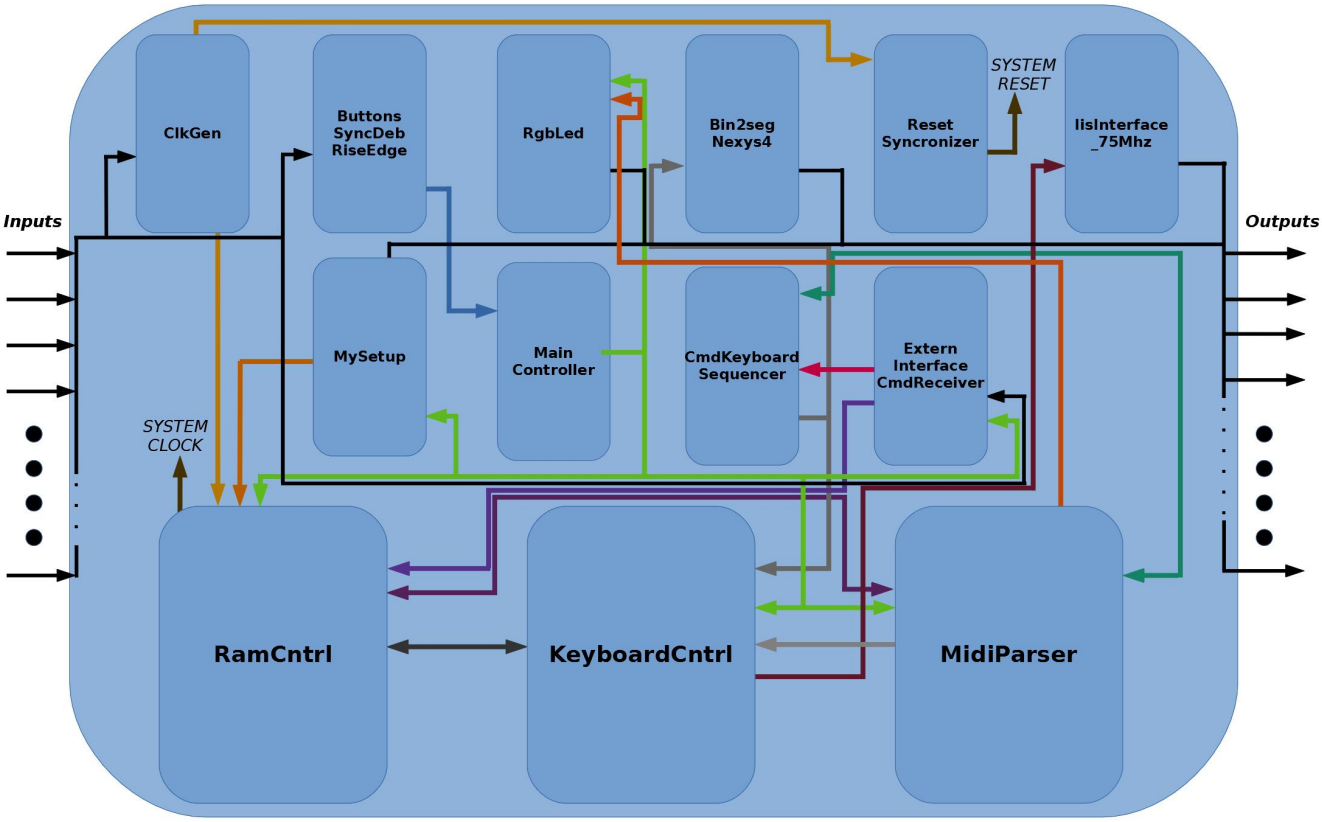


**Figura 4.1**: *Esquemático simplificado del diseño MIDI_Soc, muestra las principales interconexiones entre los componentes*

Cabe destacar la lógica a seguir por los *displays* de 7 segmentos. Estos pueden mostrar el número actual de notas activas y el último comando de teclado (§4.16) enviado a KeyboardCntrl o, los primeros 4 bytes que forman un comando de escritura en memoria (§4.9).

Para finalizar se enumeran las principales señales que forman la interfaz de uso de este componente:

**Señales de entrada, (*Inputs*):**

- **clk_i [1]**: señal del reloj del sistema, proporcionada directamente por la placa. En este caso particular la frecuencia de la señal de reloj es de 100Mhz.

- **resetn_i [1]**: señal de reset a baja del sistema. Al pulsar el botón de reset se pone a 0, y se inicia el reseteo de todos los componentes del sistema, inicializándolos a un estado seguro.

- **Botonera**, señales provenientes de los botones de la placa, activas a alta al presionar el correspondiente botón.

  - **btnc_i [1]**: asociada al botón btnc de la placa. Inicializa la memoria RAM del sistema con el bloque de datos (§4.4) almacenado en la memoria Flash.

  - **btnu_i [1]**: asociada al botón btnu de la placa. Esta señal inicia o detiene la reproducción de un archivo MIDI. Si el sistema esta reproduciendo detiene la reproducción, y en caso contrario inicia la reproducción.

  - **btnl_i [1]**: asociada al botón btnl de la placa. Señal que activa o desactiva el efecto *reverb* del sonido proveniente de la placa. Lógica de comportamiento igual a btnu_i.

  - **btnd_i [1]**: asociada al botón btnd de la placa. Dicha señal activa o desactiva la conexión con la interfaz externa, permitiendo a esta la transferencia de archivos a la placa, el inicio y detenimiento de la reproducción de un archivo MIDI, y la activación/desactivación del efecto *reverb*.

- **btRxD [1]**: línea de transferencia de datos serie (RxD) proveniente del dispositivo Pmod BT2 [20].

- **io1 [1]**: línea de transferencia de datos con la memoria Flash mediante el protocolo SPI, configurada solo como entrada, limitada así a la realización de lecturas. Esta línea junto a io0 permiten realizar lecturas en paralelo de dos bits.

**Señales de salida, (*Outputs*):**

- **Displays 7 Segmentos**
  - **disp_seg_o [8]**: conjunto de señales que permiten indicar los leds de un *display* de 7 segmentos que deben iluminarse.
  - **disp_an_o [8]**: conjunto de señales que permiten elegir el *display* de 7 segmentos a iluminar, utilizando una codificación *one hot* para realizar esta selección.
- **led_0 [16]**: señales activas a alta para iluminar los 16 leds genéricos de la placa. El led 14 muestra el estado del efecto *reverb* y el led 15 indica si la memoria RAM esta llena. El resto de señales han sido utilizadas para depurar distintos componentes.
- **Leds RGB**, señales que indican las intensidades de los colores rojo, azul, verde mediante PWM.
  - Las señales correspondientes al led rgb derecho de la placa (LD16) son: **rgb1_blue_o, rgb1_green_o, rgb1_red_o**.
  - Las señales correspondientes al led rgb izquiedo de la placa (LD17) son: **rgb2_blue_o, rgb2_green_o, rgb2_red_o**.
- **btRst_n [1]**: señal de reset a baja del dispositovo Pmod BT2 [20]. Esta señal es generada por el sistema para ordenar el reseteo del periférico, esto ocurre al pulsar la señal de entrada resetn_i.
- **Señales del Pmod I2S2 [19]**, definidas según el protocolo I2S (§2.7).
  - Las señales diriguidas al códec de audio CS5343/44 [42] son: **mclkAD, sclkAD, lrclkAD**.
  - Las señales diriguidas al códec de audio CS4344 [31] son: **mclkDA, sclkDA, lrclkDA**.

- **sdti [1]**: línea de datos serie por la que se transmiten los bits a ser recibidos por el códec de audio CS4344 [31] incluido en el dispositivo Pmod I2S2.

- **cs_n**: siguiendo el protocolo SPI (§2.8), esta señal selecciona el dispositivo esclavo para que este habilite la lectura de comandos recibidos por la línea io0. Activa a baja.

- **Interfaz física DDR2** . Las señales de salida que se muestran a continuación, permiten la navegación por el diagrama de estados implementado en el controlador interno de la memoria DDR2 que incorpora la placa Nexys 4 DDR que se ha utilizado durante el desarrollo de este proyecto [17]. Son generadas directamente por el subcomponente ddr (IP Core Mig 7) (§4.3.2). Dichas señales son: **ddr2_addr, ddr2_ba, ddr2_ras_n, ddr2_ras_n, ddr2_cas_n, ddr2_we_n, ddr2_ck_p, ddr2_ck_n, ddr2_cke, ddr2_cs_n, ddr2_dm, ddr2_odt**.

**Señales de entrada y salida, (*InOut*)**. Estas señales están presentes tanto en el lado *Inputs* como en el de *Outputs*, el uso como entrada o salida depende de la lógica del sistema:

- **io0 [1]**: Línea de transferencia de datos con la memoria Flash mediante el protocolo SPI, configurada como entrada/salida. Esta línea al configurarse como salida, es utilizada para el envío de comandos al controlador SPI de la memoria Flash. Por el contrario al configurarse como entrada, permite la lectura de los bits enviados por el controlador SPI del otro extremo. Esta línea junto a io1 permiten realizar lecturas en paralelo de dos bits.

- **Interfaz física DDR2** .Las señales de entrada/salida que se muestran a continuación, permiten la navegación por el diagrama de estados implementado en el controlador interno de la memoria DDR2 que incorpora la placa Nexys 4 DDR que se ha utilizado durante el desarrollo de este proyecto [17]. Son generadas directamente por el subcomponente ddr (IP Core Mig 7) (§4.3.2). Dichas señales son: **ddr2_dq, ddr2_dqs_p, ddr2_dqs_n**.

## 4.3. IP cores y su presencia en el diseño

Xilinx y sus socios tienen una rica librería de diseños completamente testeados y configurables, a disposición de todos aquellos que estén envueltos en algún desarrollo que involucre a alguno de sus productos compatibles con dicha librería. Esto se hace con la finalidad de acelerar el proceso de desarrollo y permitir desarrollar una demo técnica del producto cuanto antes [43]. Los diseños pertenecientes a esta librería se denominan IP cores (iniciales IP hacen referencia a *Intellectual Property*). A su vez, Vivado provee una interfaz gráfica para la correcta configuración e implementación en el proyecto llamada *IP Integrator*.

Este proyecto hace uso de dos IP cores que se proceden a explicar a continuación.

### 4.3.1. IP Clock Wizard y componente ClkGen

El componente ClkGen esta basado en el IP core ClockWizard. Así mediante el uso del *IP Integrator*, se ha procedido a la configuración del componente. Este componente es el encargado de generar una frecuencia de reloj de 200Mhz a partir de la frecuencia de 100Mhz que provee la placa por defecto. Internamente, la conversión se realiza mediante el uso de unos componentes electrónicos integrados en la FPGA llamados MMCM. Estos componentes mediante electrónica analógica cambian la fase y la frecuencia de la señal de entrada para generar las distintas frecuencias requeridas [44]. El uso de este componente es necesario ya que el subcomponente ddr utiliza una frecuencia de 200Mhz siendo esta mayor a la proporcionada nativamente por la placa (100Mhz).

### 4.3.2. IP MIG 7 y subcomponente ddr

El subcomponente ddr esta basado en el IP core MIG 7 *Memory Interface generator 7* [45]. Así mediante el uso del *IP Integrator*, se ha procedido a la configuración del componente. Los parámetros de configuración a destacar son:

- **Período de la señal de reloj de funcionamiento de la interfaz física**. Este período ha sido configurado a un valor de 3333ps, resultando en una frecuencia aproximada

de 300 MHz.

- **Ratio de frecuencia de la interfaz física con el controlador**. El ratio escogido ha sido de 4:1, es decir, 1/4 de la frecuencia a la que trabaja la interfaz física resultando en una frecuencia efectiva de 75MHz.

- **Ancho de palabra y uso de máscara**. El número de bytes a tener en cuenta por cada transacción es de 16. También se ha habilitado el uso de una máscara de 2 bytes para la seleccionar el par concreto de bytes a tener en cuenta al realizar una transacción.

- **Número de bancos máquina**. Se han establecido 4 bancos máquina. Un banco máquina maneja un único banco DRAM al mismo tiempo.

- **Ordenación de los comandos**. Permite al controlador reordenar los comandos recibidos para optimizar las transacciones. La configuración establecida sigue el orden estricto de llegada, no se reordenan en el controlador.

- **Frecuencia de reloj de entrada**. La frecuencia de reloj de entrada establecida es de 200MHz. Esta señal se utiliza para la generación de la señal de reloj de la interfaz física mediante el uso de un PLL o MMCM [44].

- **Polaridad del reset del sistema**. Polaridad establecida a baja.

- **Instanciación del bloque XADC**. La configuración llevada a cabo instancia el componente XADC (*Xilinx Analog-to-Digital Converter*)[46]. Este componente es utilizado par conocer la temperatura del sistema y así poder ajustar adecuadamente los valores eléctricos de las señales que se comunican con la interfaz física.

A continuación se describen las señales que forman la interfaz de uso del subcomponente que nos ocupa. Entre estas, también debe incluirse las contenidas en el grupo "Interfaz física

DDR2" descrito en la interfaz de MIDI_Soc, tanto las señales de salida (§4.2) como las de entrada/salida (§4.2):

**Señales de entrada, (*Inputs*):**

- **sys_clk_i [1]**: señal del reloj del sistema a 200MHz, proporcionada por el componente ClkGen.
- **sys_rst [1]**: señal de reset a baja del subcomponente. Inicializa a un estado seguro todos los elementos dentro del subcomponente.
- **app_addr [27]**: señales que en su conjunto forman los bits que representan la dirección de memoria de la solicitud actual.
- **app_cmd [3]**: señales que en su conjunto forman los bits que representan el comando de la solicitud actual.
- **app_en [1]**: señal activa a alta habilitando el registro por parte del subcomponente ddr de los conjuntos de señales app_addr y app_cmd.
- **app_wdf_data [128]**: señales que en su conjunto forman los bits a ser escritos por un comando de escritura.
- **app_wdf_end [1]**: señal activa a alta indicando que los datos en el conjunto de señales app_wdf_data en ciclo actual, se corresponden a los últimos datos enviados.
- **app_wdf_mask [16]**: conjunto de señales que representa la máscara utiliza al realizar una escritura.
- **app_wdf_wren [1]**: señal activa a alta indicando que los bits de datos correspondientes al conjunto de señales app_wdf_data.
- **app_sr_req [1]**: entrada reservada, debe tener un valor de 0 lógico.

- **app_ref_req [1]**: señal activa a alta ordenando una operación de refresco de la memoria DRAM. En el disño esta señal toma un valor de 0 lógico.

- **app_zq_req [1]**: señal activa a alta ordenando una operación de calibración de la memoria DRAM. En el disño esta señal toma un valor de 0 lógifco.

**Señales de salida [45], (*Outputs*):**

- **app_rd_data [128]**: señales que en su conjunto forman los bits que han sido leídos por el envío del último comando de lectura.

- **app_rd_data_end [1]**: señal activa a alta indicando que los datos en el conjunto de señales app_rd_data en ciclo actual, se corresponden a los últimos datos solicitados.

- **app_rd_data_valid [1]**: señal activa a alta indicando que los valores tomandos por el conjunto de señales app_rd_data es valido.

- **app_rdy [1]**: señal activa a alta indicando que el subcomponente ddr puede aceptar el envío de nuevos comandos.

- **app_wdf_rdy [1]**: señal activa a alta indicando que el buffer FIFO encargado de almacenar comandos, acepta escrituras.

- **app_sr_active [1]**: entrada reservada, en el diseño esta conectada a tierra.

- **app_ref_ack [1]**: señal activa a alta indicando que el controlador de memoria ha enviado un comando de refresco a la interfaz PHY. En el diseño esta conectada a tierra.

- **app_zq_ack [1]**: señal activa a alta indicando que el controlador de memoria ha enviado un comando de calibración a la interfaz PHY. En el diseño esta conectada a tierra.

- **ui_clk [1]**: señal de reloj a la que trabaja las solicitudes de la memoria. En el diseño su frecuencia es de 75 Mhz.

- **ui_clk_sync_rst [1]**: señal activa a alta indicando un reseteo sincronizado de los elementos externos al subcomponente ddr.

- **init_calib_complete [1]**: señal activa a alta indicando la realización tanto de la inicialización como la calibración de la memoria, estando el subcomponente ddr preparado para su uso.

## 4.4. Mapeo de datos en memoria Flash y RAM

El mapeo de los bytes en memoria RAM se ha organizado en 3 secciones de datos: *A0-C8-16bitsMono*, *ODBD_ ConstsReducedSet*, y *Papermoon*.

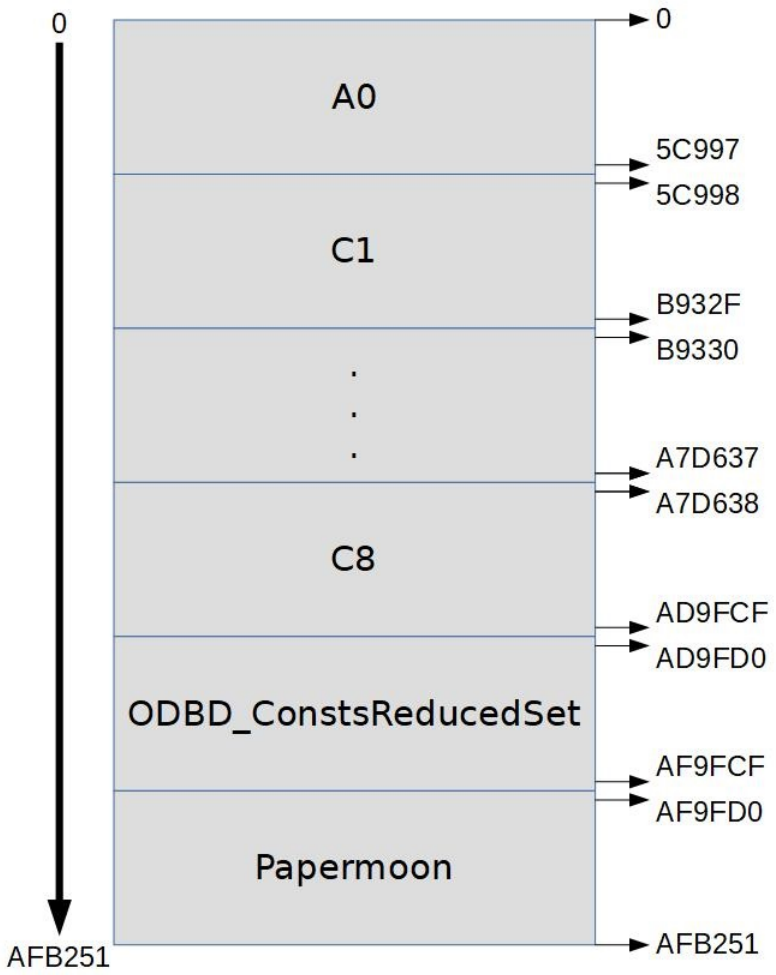


**Figura 4.2**: *Organización de los bytes en memoria Ram, representando las direcciones en hexadecimal*

En la primera sección *A0-C8-16bitsMono*, es en donde se almacenan las *Wavetables* utilizadas en la síntesis de sonido. Las notas almacenadas son aquellas que se sitúan a una distancia de 3/2 de tono, siendo la secuencia: A0, C1, D1#, F1#, A1,..., C8. Todas

las notas están en mono, y tienen el mismo número de muestras que son 189644. Para facilitar el direccionamiento de muestras, se ha decidido tener una resolución de 16 bits por muestras, esta facilidad es debido a que 16 es potencia de 2. Haber direccionado muestras de 24 bits de resolución habría complicado el diseño del componente RamCntrl para poder realizar lecturas de 24 bits, si se hubiese querido aprovechar al máximo la memoria al no dejar direcciones de bytes con datos no relevantes. De todas formas, cargar muestras de 24 bits no ha sido posible debido al espacio insuficiente en la memoria Flash de la placa siendo esta de 16MiB y habiendo necesitado 16,28 MiB (17067960 bytes correspondientes a muestras). Se podría haber intentando recortar más la duración de algunas notas pero ello hubiera supuesto que no todas las notas tienen el mismo número de muestras lo que hubiese añadido otra complicación, y en su conjunto hubiera sido bastante más engorroso trabajar con muestras de 24 bits.

Por ello se decidió usar una resolución de 16 bits por muestra, resultando el tamaño en bytes de cada nota de 379288, multiplicando este número por la cantidad de notas a almacenar en memoria (30) obtenemos 11378640 bytes, que son unos 10,85 MiB siendo este el tamaño final del espacio ocupado por esta sección.

La sección contigua es *ODBD_ ConstsReducedSet*, en donde vienen una serie de bytes que codifican en punto fijo (formato Q4.20) una serie de valores constantes necesarios para realizar el cálculo del tiempo de duración de los mensajes MIDI. Estos valores almacenados representan todos los resultados posibles al realizar la división de $1/division$, recordemos que *division* es una constante que viene definida en la cabecera de un archivo MIDI. Por razones de eficiencia se ha preferido tabular la división en lugar de implementar un divisor completo. El valor de *division* viene definido por 16 bits, en el caso de este proyecto solo se tiene en cuenta el archivo MIDI si el formato *division* tiene el bit 16 a 0. Por ello los valores a almacenar son 32768 (15 bits), cada uno de esos valores tiene 4 bytes de resolución (formato Q4.20). El valor viene representado por los últimos 3 bytes mientras que el byte

más significativo se limita a extender el bit de signo. El uso de 4 bytes se realiza con la finalidad de facilitar el direccionamiento de la constante, la razón es la misma que en el caso de usar 16 bits por muestra. El tamaño total de esta sección se obtiene de multiplicar 32768 (número de constantes) por 4 (número de bytes por constante) siendo el de 131072 bytes, que son unos 128KiB.

La sección final *Papermoon* consiste simplemente en un archivo MIDI que contiene una canción de piano titulada de la misma forma que la sección. Contiene 4738 bytes (4,62KiB) que representan una cabecera MIDI dos pistas con mensajes MIDI. Al cargar un nuevo archivo MIDI en la placa, se usara la dirección de inicio de esta sección sobreescribiendo los bytes que hubieren en ella. El resto del espacio de la memoria RAM (en total son 128MiB ) sirven para almacenar el archivo MIDI transferido.

Este bloque de datos ocupa finalmente unos 11 MiB Dichas secciones se ubican en memoria RAM tal y como se muestra en el diagrama inicial (§4.2). En memoria Flash, previo al bloque de datos, viene el archivo *bitstream* (.bit) necesario para programar la FPGA sin necesidad de transferir el archivo del PC a la placa, por ello el bloque de datos se ubicará a partir de la última dirección ocupada por el archivo *bitstream* (en memoria Flash). En caso de no utilizar esta funcionalidad de la FPGA, el bloque de datos en memoria Flash si direcciona igual que en memoria RAM.

## 4.5. Componente RamCntrl

Este componente se encarga de secuencializar las solicitudes de lectura de memoria RAM que realizan los componentes KeyboardCntrl y MidiParser. También se encarga de gestionar las solicitudes de escritura que realizan los componentes MySetup y ExternInterfaceCmdReceiver. Gracias a la gestión que realiza RamCntrl, los componentes KeyboardCntrl y MidiParser realizan solicitudes de lectura concurrentemente, por ello estamos ante una ar-

quitectura paralela.

En el siguiente esquemático se muestra la estructura general del componente:

### 4.5.1. Visión general

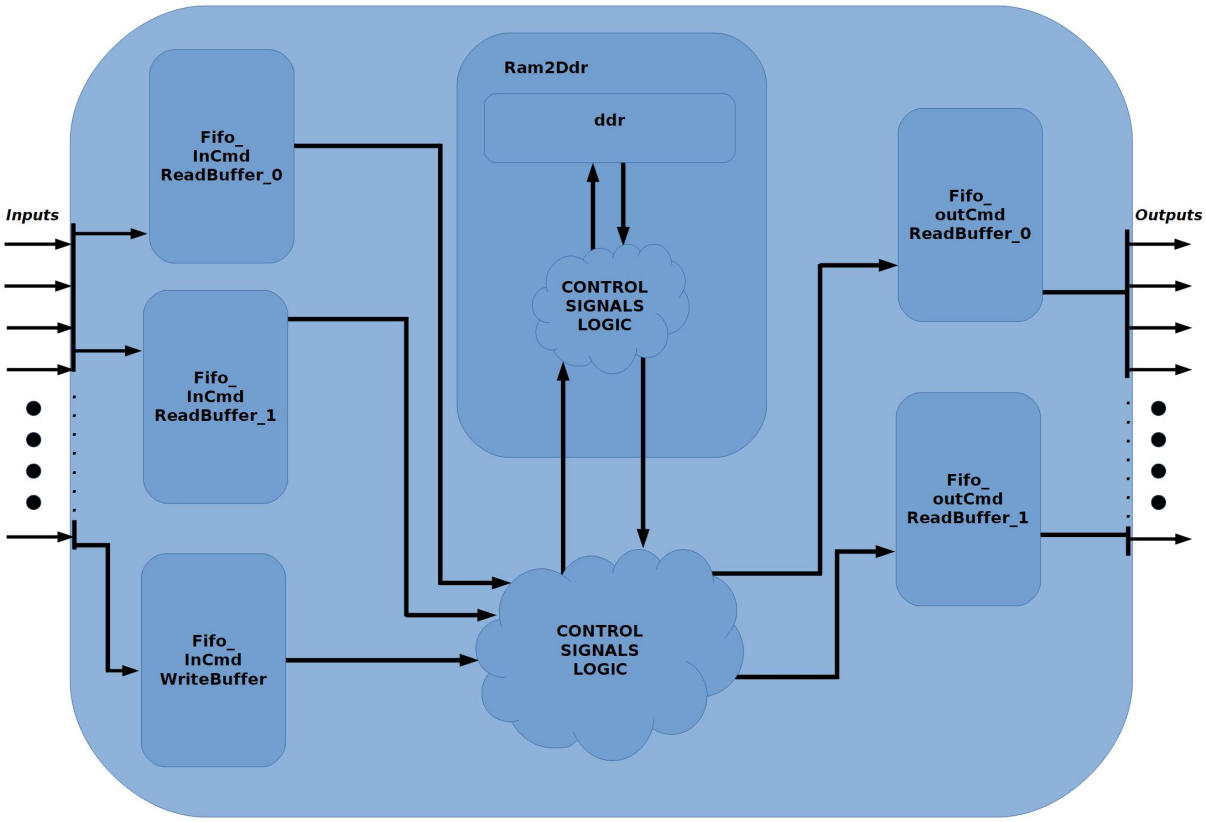


**Figura 4.3**: *Esquemático simplificado del componente RamCntrl*

Las principales señales que definen la interfaz de uso de RamCntrl son:

**Señales de entrada, (*Inputs*):**

- **clk_200Mhz_i [1]**: señal del reloj a 200Mhz procedente del componente ClkGen. Necesaria para el funcionamiento del componente ddr (IP core MIG 7).
- **rdWr[1]**: señal de selección de modo de funcionamiento de RamCntrl. Consume comandos de escritura si esta a baja, o consume comandos de lectura si esta a alta.

- **inCmdReadBuffer_0 [log2(MAX_NUM_TRACKS)+26]**:
  señales que en su conjunto forman los bits que representan un comando de lectura de memoria proveniente del componente MidiParser.

- **wrRqtReadBuffer_0 [1]**: señal que permite la escritura del comando proveniente de las líneas de inCmdReadBuffer_0 en el buffer Fifo_inCmdReadBuffer_0. A alta la escritura se realizara al comienzo del siguiente ciclo, a baja no.

- **inCmdReadBuffer_1 [log2(NUM_NOTES_GEN)+26]**:
  señales que en su conjunto forman los bits que representan un comando de lectura de memoria proveniente del teclado.

- **wrRqtReadBuffer_1 [1]**:
  señal que permite la escritura del comando proveniente de las líneas de inCmdReadBuffer_1 en el buffer Fifo_inCmdReadBuffer_1. A alta la escritura se realizara al comienzo del siguiente ciclo, a baja no.

- **rdRqtReadBuffer_0 [1]**:
  señal que en alta ordena la lectura del buffer Fifo_outCmdReadBuffer_0. En baja no realiza ninguna acción.

- **rdRqtReadBuffer_1 [1]**:
  señal que en alta ordena la lectura del buffer Fifo_outCmdReadBuffer_1. En baja no realiza ninguna acción.

- **inCmdWriteBuffer [42]**: señales que en su conjunto forman los bits que representan un comando de escritura en memoria. Puede provenir de MySetup o de ExternInterfaceCmdReceiver.

- **wrRqtWriteBuffer [1]**: señal que permite la escritura del comando proveniente de las líneas de inCmdWriteBuffer en el buffer Fifo_inCmdWriteBuffer. A alta la escritura se realizara al comienzo del siguiente ciclo, a baja no.

**Señales de salida, (*Outputs*):**

- **ui_clk_o [1]**: señal de reloj a 75MHz, proveniente del componente ddr (IP core MIG 7). Esta es la señal de reloj que utilizan todos los elementos síncronos del sistema exceptuando los elementos de los componentes de ddr, resetSyncronizer. La frecuencia de esta señal define la frecuencia de trabajo de las operaciones de lectura/escritura de memoria desde una perspectiva exterior al componente ddr.

- **fullCmdReadBuffer_0 [1]**: esta señal indica si el buffer Fifo_inCmdReadBuffer_0 está lleno o no. A alta indica que esta lleno, a baja en caso contrario.

- **fullCmdReadBuffer_1 [1]**: esta señal indica si el buffer Fifo_inCmdReadBuffer_1 está lleno o no. A alta indica que esta lleno, a baja en caso contrario.

- **outCmdReadBuffer_0 [log2((MAX_NUM_TRACKS))+129]**:
  señales que en su conjunto forman los bits que representan un comando de respuesta diriguido al componente MidiParser. Provienen del buffer Fifo_outCmdReadBuffer_0.

- **emptyResponseRdBuffer_0 [1]**:
  esta señal indica si el buffer Fifo_outCmdReadBuffer_0 está vacío o no. A alta indica que esta vacío, a baja en caso contrario.

- **outCmdReadBuffer_1 [log2((NUM_NOTES_GEN))+16]**: señales que en su conjunto forman los bits que representan un comando de respuesta dirigido al componente KeyboardCntrl. Provienen del buffer Fifo_outCmdReadBuffer_1.

- **emptyResponseRdBuffer_1 [1]**:
  esta señal indica si el buffer Fifo_outCmdReadBuffer_1 está vacío o no. A alta indica que esta vacío, a baja en caso contrario.

- **fullCmdWriteBuffer [1]**: esta señal indica si el buffer Fifo_inCmdWriteBuffer está lleno o no. A alta indica que esta lleno, a baja en caso contrario.

- **writeWorking [1]**: señal que se pone a alta si se está produciendo una escritura en memoria o a baja en caso contrario.

A demás de estas señales hay que añadir las contenidas en el grupo "Interfaz física DDR2" descrito en la interfaz de MIDI_Soc, tanto las señales de salida (§4.2) como las de entrada/salida (§4.2).

### 4.5.2. ¿Por qué un controlador de acceso a memoria? Naturaleza paralela del diseño

Desde un principio la naturaleza altamente paralela del sistema a desarrollar planteaba un problema: era necesario poder leer de una única memoria varias muestras para poder generar sonido polifónico en tiempo real, y leer al mismo tiempo un archivo MIDI e interpretar sus bytes.

En el desarrollo del proyecto, se planteo en primera instancia gestionar las lecturas y escrituras mediante una lógica similar a la seguida en los semáforos y mutex utilizados en sistemas operativos. Esto consistía en llevar un orden secuencial sobre a que componente servir los datos de memoria y a cuales no resultando en un controlador de memoria que no aprovechaba las posibilidades de concurrencia real que ofrece el *hardware*. La idea principal para poder aprovechar dicha concurrencia consiste en que los componente no tengan que interrumpir su actividad tanto al enviar una solicitud como al recibir una respuesta, es decir, que puedan enviar y recibir al mismo tiempo solicitudes y respuestas respectivamente.

Esto se ha logrado mediante el uso de buffers FIFO (*First In First Out*) y a la definición de una serie de comandos que sean totalmente autocontenidos, es decir, una vez que el comando es enviado el componente de origen se despreocupa completamente de dicho comando ya que este contiene toda la información necesaria para poder desencadenar las acciones que ejecutan una lectura/escritura completa.

### 4.5.3. Comandos de memoria, formato y funcionamiento

Existen dos tipos principales de comandos: los de lectura y los de escritura. A su vez, los comandos de lectura se diferencian dos tipos los provenientes de MidiParser y los provenientes de KeyboardCntrl, llamemoslos *MidiParser CMDs* y *Keyboard CMDs* respectivamente. Comenzando por los comandos de lectura, estos vienen acompañados de dos formatos, uno destinado para manejar la solicitud de lectura a memoria, y otro para devolver dicho dato de forma que el componentes que halla solicitado la lectura sepa interpretar la respuesta.

En el caso de los *Keyboard CMDs* de solicitud de lectura, siguen el siguiente formato:

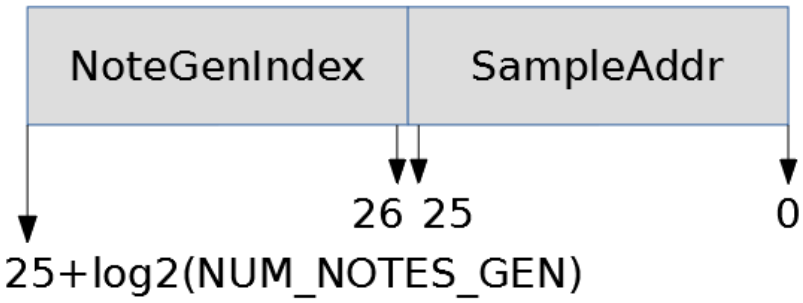


**Figura 4.4**: *Formato de los Keyboard CMDs de solicitud de lectura, provenientes de KeyboardCntrl*

Como describe la anterior imagen, los bits desde el 0 hasta el 25 representan la dirección de la muestra que se solicita leer (*SampleAddr*), representando el resto de bits el índice del generador de nota al que va asociada esa petición. Estos comandos se almacenan en el buffer de entrada Fifo_inCmdReadBuffer_1, ordenandose una escritura en este mediante la señal wrRqtReadBuffer_1. Cabe aclarar que la dirección de muestra proviene de un generador de nota en particular, quedando *NoteGenIndex* como una referencia al generador de notas (§4.6.2) que ha solicitado dicha muestra.

Tras la explicación del formato de los *Keyboard CMDs* de solicitud de lectura, se procede a definir el formato de las respuestas asociadas es estas peticiones:

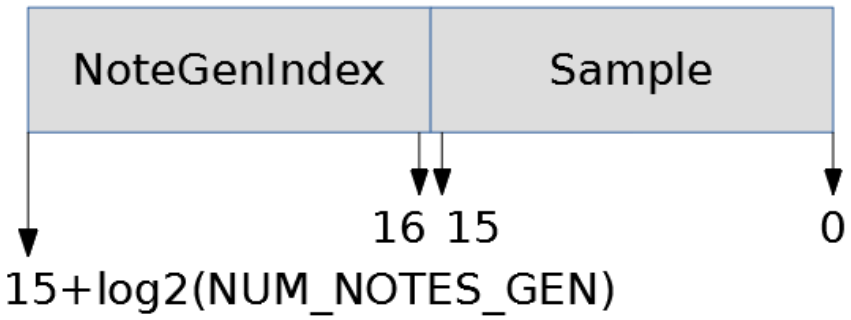


**Figura 4.5**: *Formato de los Keyboard CMDs de respuesta a lectura, provenientes de KeyboardCntrl*

Este formato es muy similar al anterior. Se diferencia solo en ser un comando de menor tamaño debido a que los bits del 0 al 15 representan la muestra de audio a enviar como respuesta. *NoteGenIndex* se mantiene con el mismo valor para que el componente KeyboardCntrl internamente sepa a que generador de nota debe entregar esta muestra. Estos Keyboard CMDs de respuesta a lectura se almacenan en el buffer de salida Fifo_outCmdReadBuffer_1

Por otro lado los *MidiParser CMDs* de solicitud de lectura, siguen el siguiente formato:

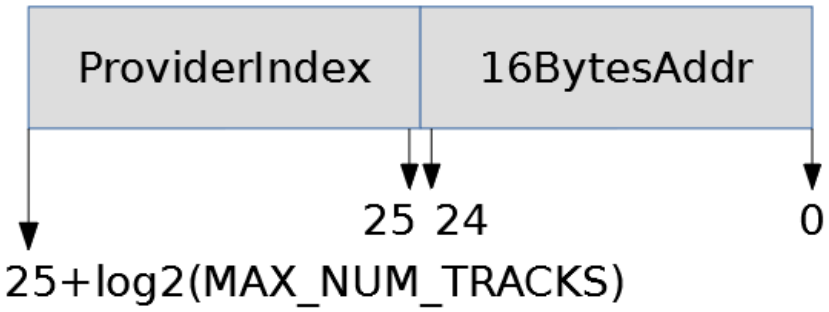


**Figura 4.6**: *Formato de los MidiParser CMDs de solicitud de lectura*

Este formato sigue conservando un cierto grado de semejanza con los anteriores. En este caso particular, los bits del 0 al 24 representan una dirección de 16 bytes (128 bits), solicitada

por un subcomponente contenido en MidiParser. El campo *ProviderIndex* se encarga de la identificación de dicho subcomponente que en caso de valer 0 se corresponde a una petición proveniente de ODBDProvider (§4.7.4), y en caso contrario de algún ByteProvider (§4.7.3) en particular.

Según el valor del campo *ProviderIndex* se genera como respuesta un *MidiParser CMD* de respuesta a lectura tipo A o tipo B. A continuación se muestra una imagen en la que se distinguen los campos del tipo A, comenzando así con su explicación:

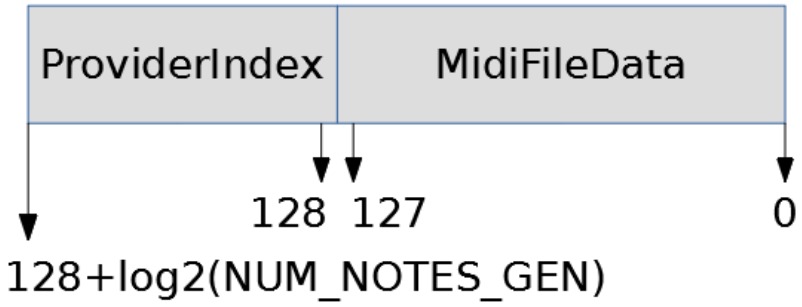


**Figura 4.7**: *Formato de los MidiParser CMDs de respuesta a lectura tipo A*

El tipo A, se corresponde al formato de respuesta al ser el valor de *ProviderIndex* distinto a 0. Bajo esta premisa, sabemos que la solicitud proviene proviene de un subcomponente ByteProvider por lo que se envía como respuesta los 16 bytes asociados a la dirección *16BytesAddr*. Esta respuesta figura en el campo *MidiFileData* correspondiéndose con los bytes de datos de un archivo MIDI almacenado en memoria.

En caso de que el valor de *ProviderIndex* sea 0, se genera como respuesta un *MidiParser CMD* de respuesta a lectura tipo B. La siguiente imagen muestra los campos que lo componen:

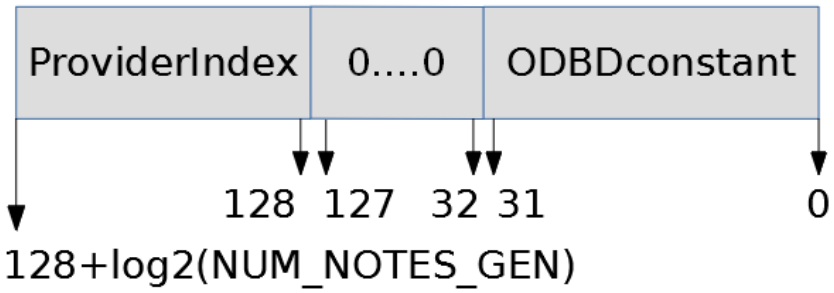


**Figura 4.8**: *Formato de los MidiParser CMDs de respuesta a lectura tipo B*

El tipo B se caracteriza por tener gran parte de sus bits a 0. Esta respuesta contiene los 4 bytes de datos que solicita el subcomponente ODBDProvider (§4.7.4), representando estos bytes una constante en particular de la sección ODBD_ConstsReducedSet.

Ambos tipos de MidiParser CMDs de respuesta a lectura se almacenan en el buffer de salida Fifo_outCmdReadBuffer_0. A su vez ambos no modifican el valor del campo de *ProviderIndex* al tratar ambas respuestas, manteniéndose este igual que en el *MidiParser CMD* de solicitud de lectura.

Tras abordar la explicación de los comandos de lectura, se procede a explicar el formato del comando de escritura, utilizado por dos componentes, ExternInterfaceCmdReceiver y MySetup. Este formato es sencillo ya que solo requiere de dos campos, uno para indicar la dirección de escritura y otro que contenga los datos a escribir, esto mismo es lo que representan los campos *2BytesAddr* y *Data* en la siguiente imagen:

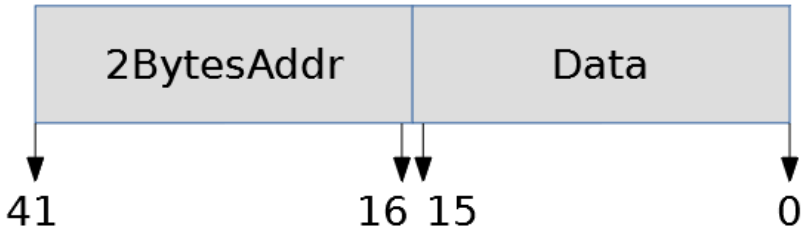


**Figura 4.9**: *Formato de los comandos de escritura*

Estos comandos de escritura se almacenan en el buffer Fifo_inCmdWriteBuffer, a la espera de ser procesados.

La escritura se realiza direccionando pares de bytes debido a que no se ha podido lograr el uso del subcomponente ddr para realizar escrituras de más de 2 bytes. Al intentar utilizar este subcomponente para realizar escrituras de más de 2 bytes ocurría que se escribían los datos en otras direcciones distintas a las asociadas a estos datos.

Gracias al uso de los buffers y a los formatos de los comandos de lectura y escritura, es posible paralelizar las peticiones de lectura a memoria. Nótese que el comportamiento paralelo del sistema ocurre siempre y cuando los buffers no estén llenos, en caso de que se llenaran el comportamiento sería secuencial ya que produciría una interrupción en el componente que envía la solicitud de lectura. Estos componentes antes de enviar un comando de lectura comprueban si el buffer en donde se va ha escribir el comando esta lleno o no, en caso de que este lleno esperan hasta que deje de estarlo.

### 4.5.4. Uso de memoria caché para la optimización de las peticiones de lectura de muestras y lógica de funcionamiento general

Con la finalidad de optimizar el tiempo de las peticiones de muestras, se ha implementado un banco de registros haciendo desempeñando la función como de una pequeña memoria caché.

La estructura del banco de registros esta formada por dos pares de registros por generador de nota, uno de 128 bits (16 bytes) para guardar la trama de datos, y otro de 23 bits para guardar la dirección correspondiente a esa trama de 16 bytes. Por ello el tamaño de la caché depende directamente de la variable genérica NUM_NOTES_GEN. Así en caso de tener 16 generadores de notas, el tamaño sería de 302 bytes (256 de datos y 46 de direcciones). El uso de esta caché permite guardar la última trama leída de 16 bytes de un *Keyboard CMDs* de solicitud de lectura. Pudiendo reutilizar dicha trama para acelerar la entrega de las siguientes peticiones de muestras evitando la interacción con el subcomponente Ram2Ddr. La

política de sustitución de bloques en la memoria caché es FIFO.

En lo que respecta a la lógica de funcionamiento general, cabe destacar que siempre se procesan los *Keyboard CMD* de solicitud de lectura en pares (de dos en dos). Esto es debido a la naturaleza de funcionamiento de los generadores de notas, ya que estos siempre generan 2 peticiones de lectura para poder realizar la interpolación.

Terminando con esta subsección, es interesante aclarar cual ha sido la implementación llevada a cabo para las señales internas *foundRow* y *rowIndex* gracias a las cuales permiten saber si la caché contiene la trama solicitada y el índice de dicha entrada en la caché respectivamente. El siguiente diagrama RTL muestra una posible implementación de estas señales, ambas señales están parametrizadas por NUM_NOTES_GEN.

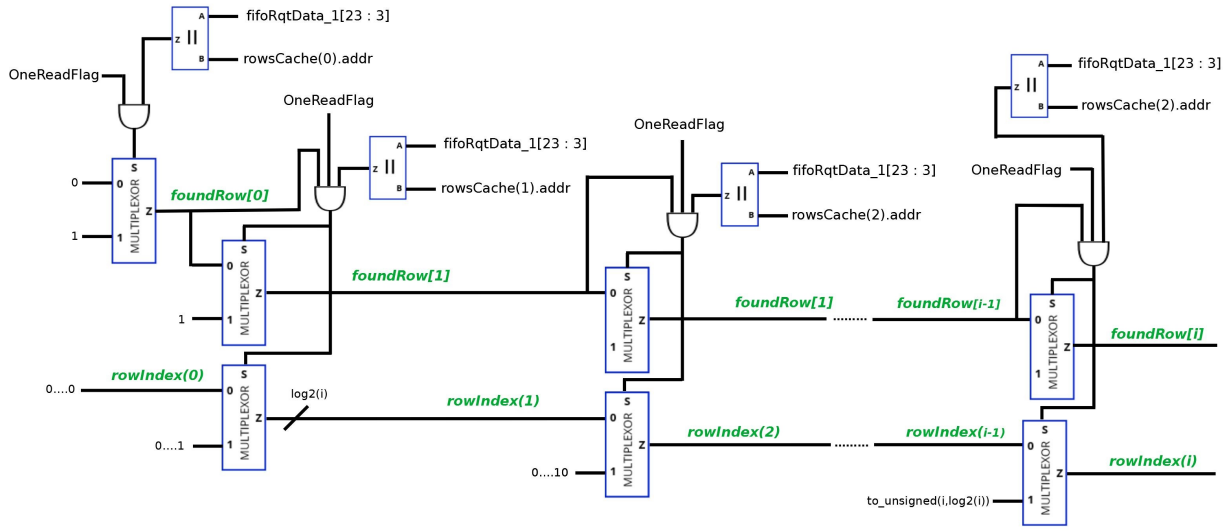


**Figura 4.10**: *Diagrama RTL que muestra el patrón de diseño seguido por las señales foundRow y rowIndex*

### 4.5.5. Subcomponente Ram2Ddr

Ram2Ddr permite simplificar la interacción con la memoria DDR2 realizada a través del subcomponente ddr. Así este subcomponente se encarga de generar las señales necesarias para interactuar con el subcomponente ddr al realizar una lectura o escritura.

A continuación se presentan las principales señales definidas en su interfaz de uso, a las que hay que incluir las señales contenidas en el grupo "Interfaz física DDR2" descrito en la interfaz de MIDI_Soc, tanto las señales de salida (§4.2) como las de entrada/salida (§4.2):

**Señales de entrada, (*Inputs*):**

- **clk_200Mhz_i [1]**: señal del reloj a 200MHz procedente del componente ClkGen. Necesaria para el funcionamiento del subcomponente ddr (IP core MIG 7).
- **ram_a [26]**: conjunto de señales que permiten indicar la dirección a leer o escribir.
- **ram_dq_i [16]**: conjunto de señales que permiten indicar la trama de 2 bytes a escribir.
- **ram_cen [1]**: señal de habilitación del componente, activa a baja.
- **ram_oen [1]**: señal que inicia una lectura, activa a baja.
- **ram_wen [1]**: señal que inicia una escritura, activa a baja.

**Señales de salida, (*Outputs*):**

- **ui_clk_o [1]**: señal del reloj a 75MHz procedente del subcomponente ddr. Es la señal de reloj que utilizan todos los componentes y subcomponente excepto resetSyncronizer y ddr.
- **ram_dq_o [128]**: conjunto de señales que permiten indicar la última trama de 16 bytes leída.
- **ram_ack [1]**: señal que notifica la finalización de una lectura o escritura, estando un cyclo a alta.

Para la realización de una escritura se requiere una dirección de 26 bits, es decir, direccionamos 2 bytes al escribir. Las lecturas, se hacen en tramas de 16 bytes en una sola orden. Por ello, al realizar lecturas se tienen en cuenta solo los 23 primeros bits de *ram_a* direccionando así 16 bytes. Los bytes leídos se vuelcan en la señal de salida *ram_dq_o*.

Un detalle de implementación a resaltar consiste en el uso de una máscara de 16 bits al realizar las escrituras de 2 bytes. Esto es necesario ya que el subcomponente ddr utiliza tanto en las lecturas como en las escrituras tramas de 16 bytes (que ha sido leída o va a escribirse), y para poder realizar una escritura con éxito es necesario poder escribir solo 2 bytes.

Finalmente hay que recalcar que tanto la máscara como la trama de 16 bytes a escribir son usadas por el subcomponente ddr.

## 4.6. Componente KeyboardCntrl

### 4.6.1. Visión general

Este componente se encarga de realizar la funcionalidad de un teclado de piano de 88 notas. La activación y desactivación de dichas notas se realiza mediante el envío de unos comandos de teclado, estos mismos incluyen la intensidad (volumen) con la que debe sonar cada nota. Estos comandos pueden provenir de los componentes MidiParser o ExternInterfaceCmdReceiver. Los comandos se tratan individualmente por lo que es necesario secuencializar los comandos provenientes de MidiParser o ExternInterfaceCmdReceiver, siendo el componente encargado de ello CmdKeyboardSequencer.

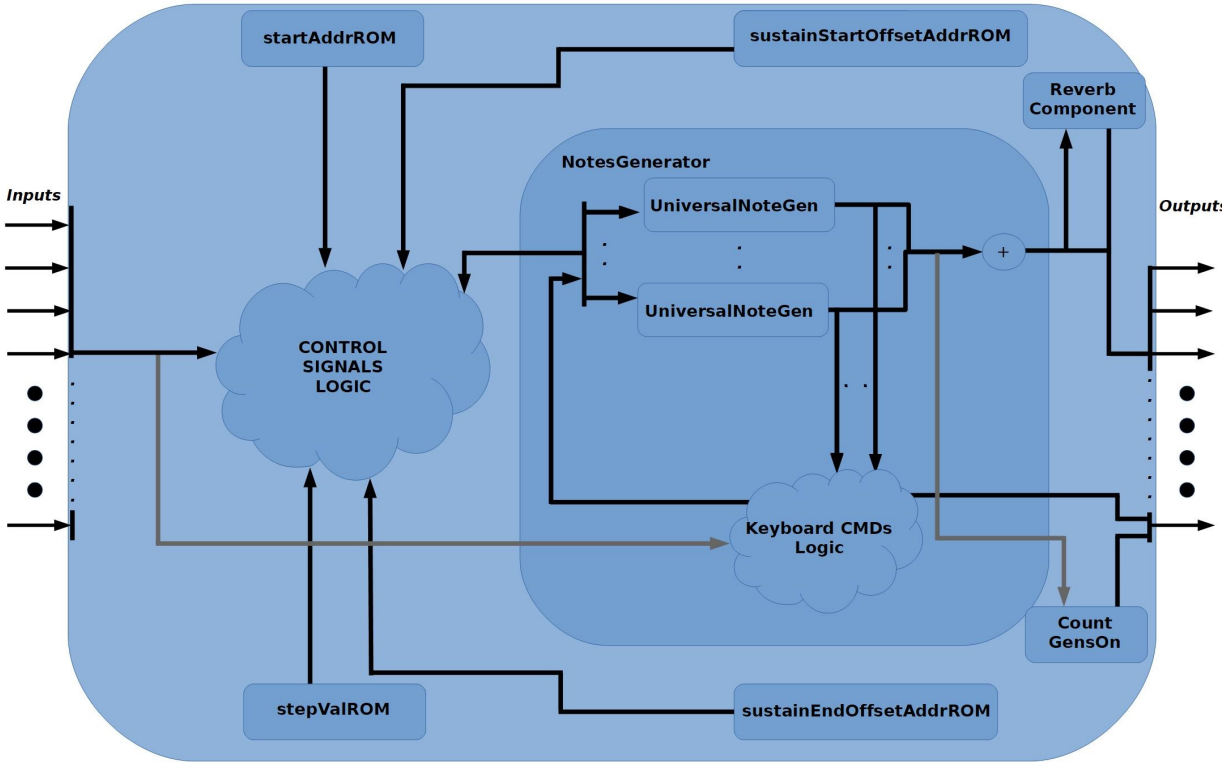


**Figura 4.11**: *Esquemático simplificado del componente KeybordCntrl*

La interfaz de uso de este componente incluye las siguientes señales:

**Señales de entrada, (*Inputs*):**

- **cen [1]**: señal de habilitación del componente, activa a baja.
- **midiParserOnOff [1]**: señal que permite indicar el inicio de lectura de un archivo MIDI por parte del componente MidiParser. Si esta a alta, el componente ha iniciado el proceso de parseo, a baja no.
- **externInterfaceStatus [1]**: señal que permite indicar si el sistema tiene activado (1 lógico) o desactivado (0 lógico) la conexión con la interfaz externa, esta señal proviene de MainController.
- **aviableCmd [1]**: señal que indica la llegada de un nuevo comando de teclado por el

conjunto de señales **cmdKeyboard**. Esto se indica con la puesta en alta durante un ciclo de dicha señal.

- **cmdKeyboard [15]**: señales que en su conjunto forman los bits que representan un comando de teclado.
- **reverbStatus [1]**: señal que permite indicar si el sistema tiene habilitado o no el modo *reverb*, esta señal proviene de MainController. Si esta a alta indica que el modo *reverb* debe activarse, esto se lleva a cabo mediante el subcomponente Reverb. Si esta a baja, el subcomponente Reverb no aplica ningún cambio a la muestra de sonido proveniente del subcomponente NotesGenerator.
- **sampleRqt [1]**: señal que indica la solicitud de una nueva muestra con la puesta en alta durante un ciclo. La frecuencia de activación de esta señal depende directamente de la frecuencia de muestreo a la que este configurado el componente iisInterface_75Mhz.
- **mem_emptyBuffer [1]**: señal activa a alta, indicando si el buffer encargado de almacenar los *Keyboard CMDs* de respuesta a lectura (Fifo_outCmdReadBuffer_1) esta vacío o no.
- **mem_CmdReadResponse [log2(NUM_NOTES_GEN)+16]**:
  señales que en su conjunto forman los bits que representan un *Keyboard CMDs* de respuesta a lectura. Provenientes directamente de RamCntrl (concretamente del buffer Fifo_outCmdReadBuffer_1).
- **mem_fullBuffer [1]**: señal activa a alta, indicando que el buffer encargado de almacenar los *Keyboard CMDs* de solicitud de lectura (Fifo_inCmdReadBuffer_1) esta lleno.

**Señales de salida, (*Outputs*):**

- **keyboard_ack [1]**: señal que permite indicar la finalización al proceso de aplicación de un comando de teclado. Activo a alta durante un ciclo.

- **sampleOut [24]**: señales que en su conjunto forman los bits que representan la muestra de sonido a ser enviada a través del componente iisInterface_75Mhz.
- **numGensOn [log2(NUM_NOTES_GEN)+1]**: señales que en su conjunto forman los bits que representan el número de generadores de notas activas, codificado en binario puro.
- **mem_CmdReadRequest [log2(NUM_NOTES_GEN)+26]**: señales que en su conjunto forman los bits que representan un *Keyboard CMDs* de solicitud de lectura.
- **mem_readResponseBuffer [1]**: señal activa durante un ciclo, realizando una orden de lectura al buffer Fifo_outCmdReadBuffer_1.
- **mem_writeReciveBuffer [1]**: señal activa durante un ciclo, realizando una orden de escritura al buffer Fifo_inCmdReadBuffer_1.

En primer lugar, se procede a explicar el funcionamiento de los subcomponentes generadores de notas, llamados en el diseño como UniversalNoteGen.

### 4.6.2. Subcomponente UniversalNoteGen

Este subcomponente se encarga de generar las muestras de audio correspondientes a una nota en particular. Para ello, UniversalNoteGen necesita dos valores numéricos. La dirección de memoria correspondiente a la primera muestra de la nota utilizada como nota base, codificada en binario puro en el conjunto de señales startAddr_In. Y, en función de la nota objetivo que se quiera generar, se fija un valor inicial a la variable *ci* de nuestra fórmula de interpolación, representado por las señales stepVal_In.

En caso de que la nota objetivo este guardada en memoria, el valor de *ci* es fijado a 1,0. Este subcomponente siempre aplica la fórmula de interpolación por lo que siempre solicita un par de muestras.

Adicionalmente, UniversalNoteGen se encarga de reproducir en bucle la etapa de sustain de la onda de sonido, aplicando según la iteración en dicho bucle, una disminución

en la intensidad de la nota para lograr el efecto sustain deseado (§2.4). Las muestras que delimitan el intervalo de muestras a reproducir en bucle, vienen direccionadas por el valor de las señales de entrada sustainStartOffsetAddr_In (primera muestra del intervalo) y sustainEndOffsetAddr_In (última muestra del intervalo).

Adicionalmente, al solicitar el apagado del subcomponente se reproduce la etapa *Release* de la nota antes de desactivar la señal de salida working.

La interfaz de uso de UniversalNoteGen incluye las siguientes señales:

**Señales de entrada, (*Inputs*):**

- **noteOnOff [1]**: señal de activación/desactivación del generador de nota. En caso de estar a alta, el subcomponente genera sonido (envío de muestras). Si esta a baja y en caso de estar reproduciendo una nota, se procede el apagado de la misma.
- **sampleRqt [1]**: señal que indica una solicitud de una nueva muestra a enviar al Pmod I2S2 con la puesta en alta durante un ciclo. La frecuencia de activación de esta señal depende directamente de la frecuencia de muestreo a la que este configurada el componente iisInterface_75Mhz.
- **startAddr_In [26]**: señales que en su conjunto forman los bits que representan la dirección de la primera muestra de la nota a generar.
- **sustainStartOffsetAddr_In [26]**: señales que en su conjunto forman los bits que representan la dirección de la primera muestra perteneciente al intervalo de muestras que se utiliza para alargar la duración de la nota (alargar la etapa de *Sustain* de la nota).
- **sustainEndOffsetAddr_In [26]**: señales que en su conjunto forman los bits que representan la dirección de la úlitma muestra perteneciente al intervalo de muestras

que se utiliza para alargar la duración de la nota (alargar la etapa de *Sustain* de la nota).

- **stepVal_In [64]**: señales que en su conjunto forman los bits que representan la variable *ci* de la fórmula de interpolación. Esta variable cambia en función de la nota a generar. En caso de que la nota a generar no requiera aplicar la fórmula de interpolación (esta almacenada en memoria), el valor se fija a 1,0. El número representado sigue un formato de punto fijo Q32.32 (§2.6).

- **noteVelocity [4]**: señales que en su conjunto forman los bits que representan la intensidad con la que debe sonar la nota, modificando así su volumen. Sigue una codificación *one hot*.

- **samples_in [15]**: señales que en su conjunto forman los bits que representan una muestra de sonido. Las muestras proceden de la memoria RAM.

- **memAckSend [1]**: señal activa a alta durante un ciclo para indicar la confirmación de la última solicitud de dos muestra realizada por este subcomponente. Esta confirmación es realizada por lógica implementada en NotesGenerator para el envío de Keyboard CMDs de solicitud de lectura.

- **memAckResponse [1]**: señal activa a alta durante un ciclo indicando la llegada de una nueva muestra proveniente de memoria RAM. Esta señal es producida por la lógica implementada en NotesGenerator para gestionar la recepción de los Keyboard CMDs de respuesta a lectura.

**Señales de salida, (*Outputs*):**

- **working [1]**: señal activa a alta indicando que el generador de notas esta generando sonido.

- **sampleOut [24]**: señales que en su conjunto forman los bits que representan la muestra de sonido generada tras aplicar la interpolación. Esta muestra se sumara junto a

las demás muestras provenientes del resto de los generadores de notas. La muestra resultante de esta suma, sera la que recibira ReverbComponent para aplicar el efecto *reverb* si procede. El subcomponente encargado de realizar la suma de muestras es NotesGenerator.

- **addr_out [26]**: señales que en su conjunto forman los bits que representan la dirección de la primera muestra (*WTin[intCi]*) para realizar la interpolación.

- **memSamplesSendRqt [1]**: señal activa a alta, realizando una solicitud de dos muestras de audio.

Para modificar la intensidad del sonido en función del conjunto de señales noteVelocity, el subcomponente se limita a desplazar a la izquierda o derecha los bits que representan la muestra de sonido interpolada. Previamente a este desplazamiento se concatenan 4 bits con el valor del bit de signo, tanto por la derecha como por la izquierda al resultado de la interpolación, este último representado con 16 bits. Por ello este desplazamiento se realiza sobre una trama de 24 bits evitando así tener que saturar el resultado. Las intensidades representadas por el conjunto de señales noteVelocity son 5, y siguen una codificación *one hot* que se describe a continuación:

- **Intensidad muy alta**, codificado con 4 bits como 1000. Factor multiplicativo por 4 lo que conlleva un desplazamiento de los bits dos posiciones a la izquierda.

- **Intensidad alta**, codificado con 4 bits como 0100. Factor multiplicativo por 2 lo que conlleva un desplazamiento de los bits una posición a la izquierda.

- **Intensidad normal**, codificado con 4 bits como 0000. Factor multiplicativo por 1, por lo que no es necesario realizar ningún desplazamiento.

- **Intensidad baja**, codificado con 4 bits como 0010. Factor multiplicativo por 0,5 lo que conlleva un desplazamiento de los bits una posición a la derecha.

- **Intensidad muy baja**, codificado con 4 bits como 0001. Factor multiplicativo por 0,25 lo que conlleva un desplazamiento de los bits dos posiciones a la derecha.

A continuación se muestra un diagrama RTL de una posible implementación *hardware* de la ruta de datos necesaria para poder realizar todas las operaciones aritméticas que implican la aplicación de la fórmula de interpolación y los correspondientes ajustes de intensidad, para así obtener la muestra de salida. En este caso, la ruta de datos esta segmentada en casi todas las operaciones necesarias excepto, las operaciones de saturación y el desplazamiento final llevado a cabo en función de la intensidad de la nota. La segmentación no supone ningún problema en la sincronización del sistema, ya que las peticiones de muestras por parte del componente iisInterface_75Mhz (señal sampleRqt activa durante un ciclo) se realizan a una frecuencia de 48800 Hz (correspondiente a la frecuencia de muestreo de las notas guardadas en memoria) siendo esta muy inferior a la frecuencia de trabajo del sistema que son 75MHz. Consecuentemente, siempre que se solicite una nueva muestra, la señal finalVal siempre tendrá el resultado correcto.

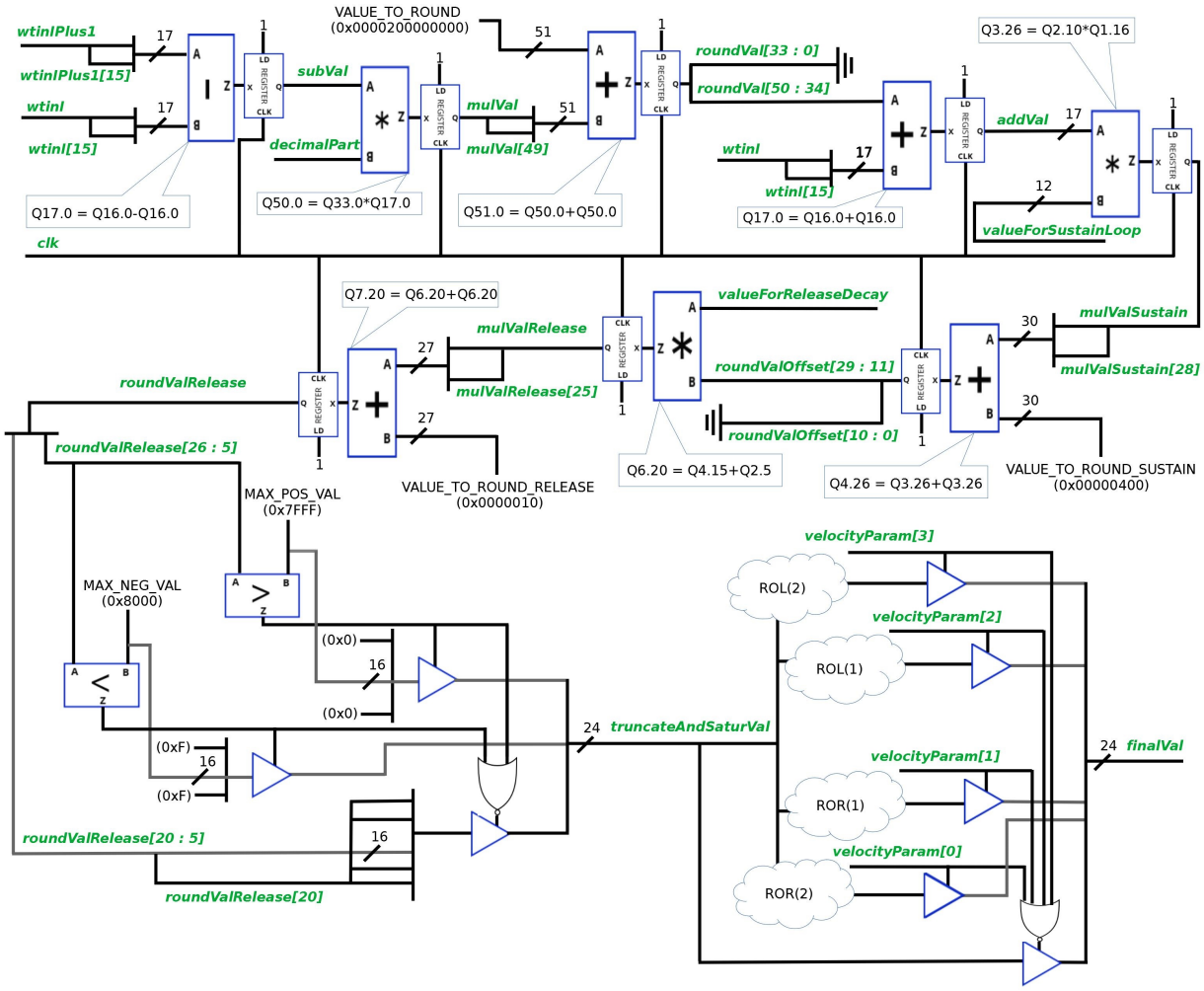


**Figura 4.12**: *Diagrama RTL correspondiente a la ruta de datos utilizada para la síntesis de sonido mediante Wavetable y la aplicación de ajustes de intensidad*

Las nubes de lógica combinacional realizan un desplazamiento rotacional a izquierda o derecha de los 24 bits de la señal truncateAndSaturVal. Esta operación se realiza según la intensidad de sonido y permite realizar una multiplicación por una potencia de dos cómodamente, por ejemplo en caso de ser la intensidad muy alta acabaría multiplicando por 4 (ROL(2)), o en caso de tener una intensidad muy baja 0,25 (ROR(2)). Es necesario la realización de rotaciones (o la concatenación equivalente) ya que se debe de conservar los bits de signo de la muestra. En todo momento se trabaja en C2.

### 4.6.3. Subcomponente NotesGenerator

La funcionalidad desarrollada por NotesGenerator es otorgar polifonía al teclado de piano que se esta implementando. Para ello se realiza la suma total de las muestras provenientes de todos los generadores de notas (UniversalNoteGen) implementados en el diseño. El número de generadores de notas presentes en este subcomponente depende directamente de la variable genérica NUM_NOTES_GEN (§4.1). A su vez, este subcomponente se encarga de gestionar el envío de los *Keyboard CMDs* de solicitud de lectura y de recibir los *Keyboard CMDs* de respuesta a lectura.

Las señales a destacar en su interfaz de uso son:

**Señales de entrada, (*Inputs*):**

- **note_on [NUM_NOTES_GEN]**: conjunto de señales activas a alta, permiten indicar individualmente la generación de sonido para cada uno de los subcomponentes UniversalNoteGen.
- **mem_emptyResponseBuffer [1]**:
  señal activa a alta, indicando si el buffer encargado de almacenar los *Keyboard CMDs* de respuesta a lectura (Fifo_outCmdReadBuffer_1) esta vacío o no.
- **mem_CmdReadResponse [log2(NUM_NOTES_GEN)+16]**:
  señales que en su conjunto forman los bits que representan un *Keyboard CMDs* de respuesta a lectura. Provenientes directamente de RamCntrl (concretamente del buffer Fifo_outCmdReadBuffer_1).
- **mem_fullReciveBuffer [1]**: señal activa a alta, indicando que el buffer que almacena los *Keyboard CMDs* de solicitud de lectura (Fifo_inCmdReadBuffer_1) esta lleno.

**Señales de salida, (*Outputs*):**

- **working [NUM_NOTES_GEN]**: conjunto de señales activas a alta, indicando individualmente que subcomponente UniversalNoteGen están generando sonido.

- **sampleOut [24]**: señales que en su conjunto forman los bits que representan la muestra de sonido resultante de realizar la suma conjunta de todas las muestras de sonido generadas por cada uno de los subcomponentes UniversalNoteGen. Esta muestra se le aplicara un efector *reverb* si procede, siendo ReverbComponent el subcomponente encargado de aplicar dicho efecto.

- **mem_CmdReadRequest [log2(NUM_NOTES_GEN)+26]**: señales que en su conjunto forman los bits que representan un *Keyboard CMDs* de solicitud de lectura.

- **mem_readResponseBuffer [1]**: señal activa durante un ciclo, realizando una orden de lectura al buffer Fifo_outCmdReadBuffer_1.

- **mem_writeReciveBuffer [1]**: señal activa durante un ciclo, realizando una orden de escritura al buffer Fifo_inCmdReadBuffer_1.

La suma total de todas las muestras provenientes de los generadores de notas, se calcula mediante sumas parciales implementadas en una ruta de datos multiciclo, siguiendo esta una topología de árbol binario balanceado. Las sumas parciales son realizadas por los subcomponentetes MyFiexedSum. Estos permiten realizar una suma saturando el valor final en caso de haber over/underflow y almacenar dicho valor en un registro resultando. Gracias a esto el total es representado con el mismo número de bits que los sumandos.

El siguiente fragmento de código fuente del diseño y el esquemático consecutivo, representan la topología de la suma que se pretende lograr:

```
--------------------------------------------------------------------
-- PIPELINED SUM
--        Manage the sums of all notes, is organized as a balanced
--        tree
--------------------------------------------------------------------
genTreeLevels:
for i in 0 to log2(NUM_NOTES_GEN)-1 generate
        genFixedSumsPerTreeLevel:
        for j in 0 to ( (NUM_NOTES_GEN/2**(i+1)) - 1) generate
                sum: MyFiexedSum
                generic map(WL=>24)
                port map( rst_n =>rst_n, clk=>clk,
                a_in=>notesGen_samplesOut(i)(j*2),
                b_in=>notesGen_samplesOut(i)(j*2+1),
                c_out=>notesGen_samplesOut(i+1)(j));
        end generate;
end generate;

sampleOut <= notesGen_samplesOut(log2(NUM_NOTES_GEN))(0);
```

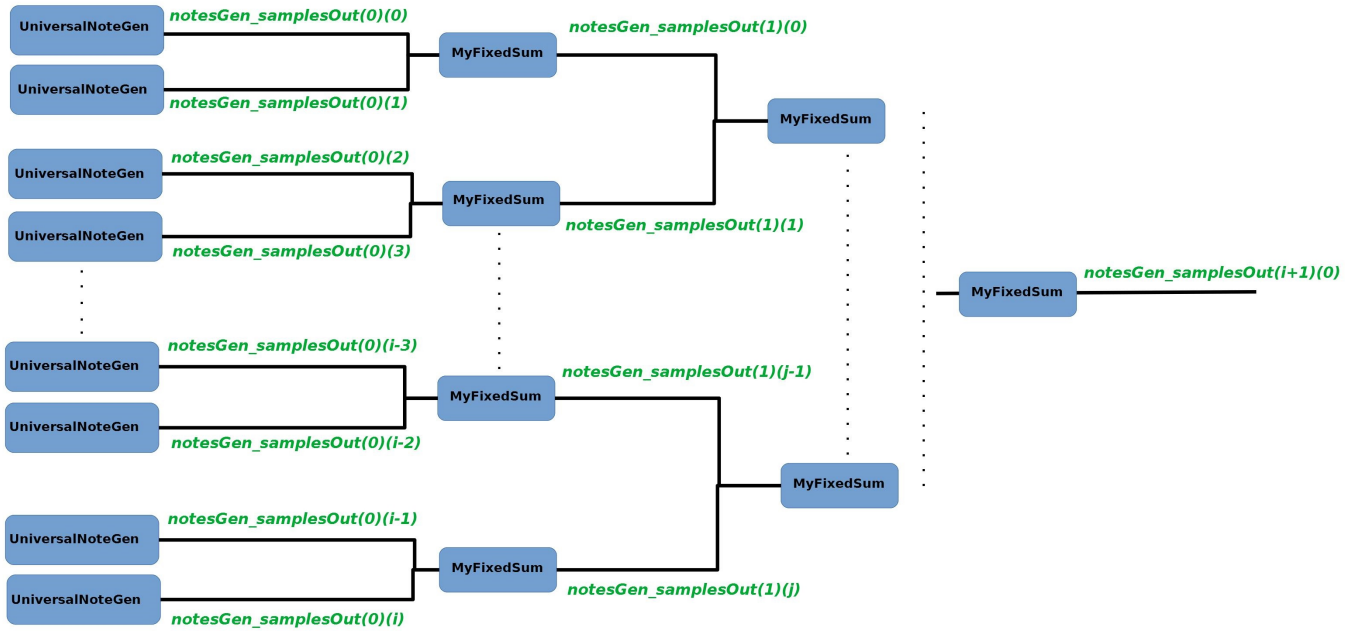


**Figura 4.13**: *Diagrama RTL correspondiente a la topología utilizada para realizar las sumas de las muestras provenientes de los generadores de notas*

### 4.6.4. Subcomponente Reverb

Subcomponente contenido en KeyboardCntrl encargado de aplicar el efecto *reverb* (§2.5) a la muestra de sonido proveniente de NotesGenerator, en caso de que el modo *reverb* este activado. El diseño subido a GitHub (§4) cuenta con un efecto *reverb* de 100 ms, resultando en un buffer de 4880 entradas. Para la obtención del número de entradas que debe tener el buffer se sigue la siguiente ecuación:

$$FIFO_DEPTH = floor((fs/1000) * REVERB_TIME)$$

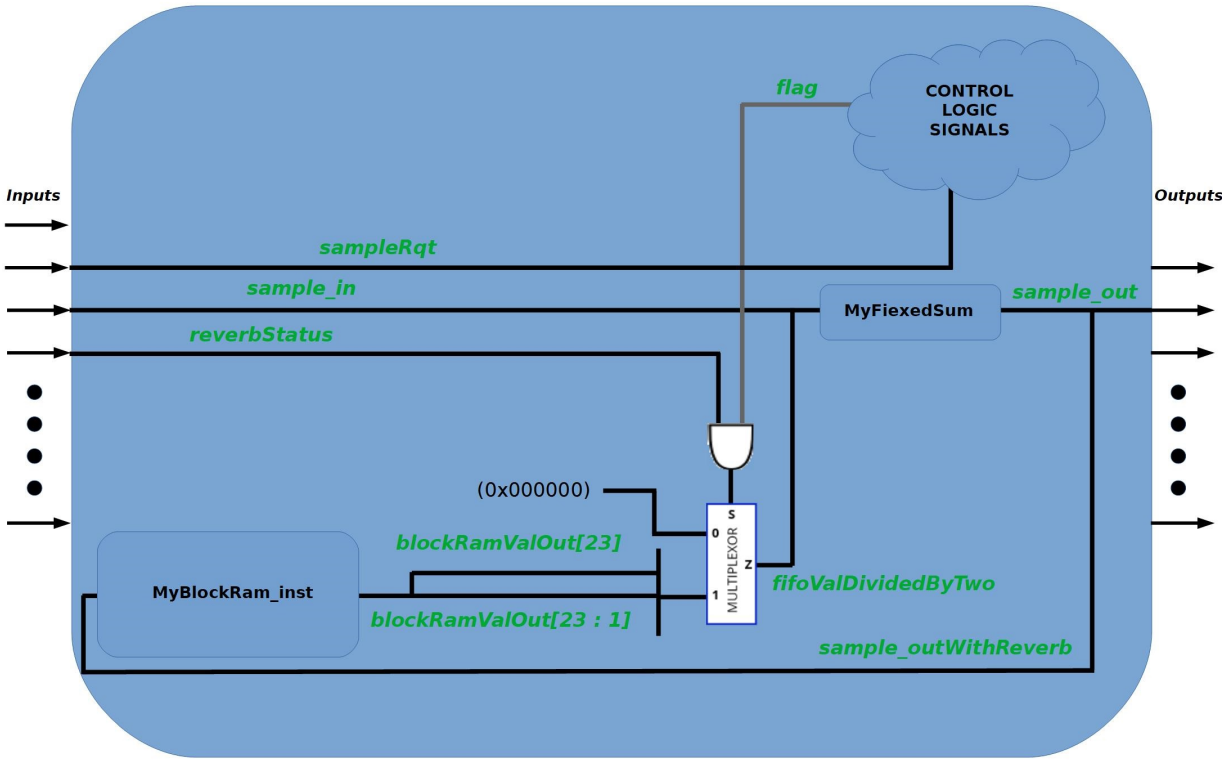


**Figura 4.14**: *Esquemático simplificado de ReverbComponent*

Las principales señales presentes en su interfaz de uso son las siguientes:

**Señales de entrada, (*Inputs*):**

- **reverbStatus [1]**: señal activa a alta, indicando la habilitación del efecto *reverb*. Proveniente del componente MainController.
- **sampleRqt [1]**: señal que indica la solicitud de una nueva muestra con la puesta en alta durante un ciclo. La frecuencia de activación de esta señal depende directamente de la frecuencia de muestreo a la que este configurada el componente iisInterface_75Mhz.
- **sample_in [24]**: señales que en su conjunto forman los bits que representan una muestra de sonido. Las muestra procede de NotesGenerator.

**Señales de salida, (*Outputs*):**

- **sampleOut [24]**: señales que en su conjunto forman los bits que representan la muestra de sonido resultante de aplicar el efecto *reverb* si procede. En caso de no ser necesaria la aplicación del efecto *reverb*, las señales son idénticas a la señales de samples_in.

Como muestra el esquemático al inicio de este subcomponente (§4.14), el efecto *reverb* se logra sumando el valor de la muestra actual (sample_in) y el valor dividido entre 2 (mediante un desplazamiento de la trama de bits de una posición a la derecha, y una extensión de signo por la izquierda) de la muestra resultante de hacer dicha suma hace $x$ milisegundos. Así desde la activación del efecto *reverb*, el buffer de muestras comienza a llenarse, y una vez que esta lleno, comienza aplicarse este efecto. Durante la fase de llenado se suma 0 a la muestra entrante. A partir del momento en que el buffer este lleno, se comienza a leer del buffer y se continua escribiendo en él (un ciclo después, debido a la realización de la suma) por cada muestra nueva recibida. La suma se implementa rehusando el subcomponente MyFiexedSum.

Un detalle de implementación sobre el diseño del buffer a resaltar, consiste en tener en cuenta un elemento *hardware* propio de las FPGAS, los módulos BlockRam. Al contrario que los buffer usados en RamCntrl, este buffer requiere de más bytes para almacenar las muestras (del orden de KiB). Para este propósito las FPGAS vienen dotadas de unos bloques de memoria síncronos llamados BlockRam, que son capaces de almacenar una mayor cantidad de bits. Así mediante el uso de estos módulos se facilita el mapeo del circuito en la fase de implementación y se ahorran componentes de la FPGA para mapear otros componentes tanto secuenciales (como los registros usados en los buffers de RamCntrl) como combinacionales.

### 4.6.5. Subcomponente CountGensOn

Este sencillo subcomponente sirve para conocer el número de generadores activos dado un instante de tiempo. Para ello solo se necesita saber que generadores de notas están actualmente activos.

Las señales a destacar en su interfaz de uso son:

**Señales de entrada, (*Inputs*):**

- **notesOnOff [NUM_NOTES_GEN]**: conjunto de señales que indican los generadores de notas que están activos.

**Señales de salida, (*Outputs*):**

- **numGensOn [log2(NUM_NOTES_GEN)+1]**: señales que en su conjunto forman los bits que representan el número de generadores de notas que están activos.

La implementación llevada a cabo en el diseño consiste en la concatenación de multiplexores en los que su señal de selección de entrada esta conectada a las distintos bits de la señal notesOnOff. Así si el bit i-ésimo de notesOnOff tiene un valor 0, se propaga un valor

numérico 0 a la salida del multiplexor, y en caso contrario se propaga un valor numérico 1. Estos valores propagados se van sumando. Esta ruta de datos esta segmentada añadiendo un registro a la salida de cada multiplexor, tardando así NUM_NOTES_GEN ciclos para que la señal numGensOn tanga el valor buscado en la señal de salida .

El siguiente diagrama RTL termina de aclararlo:

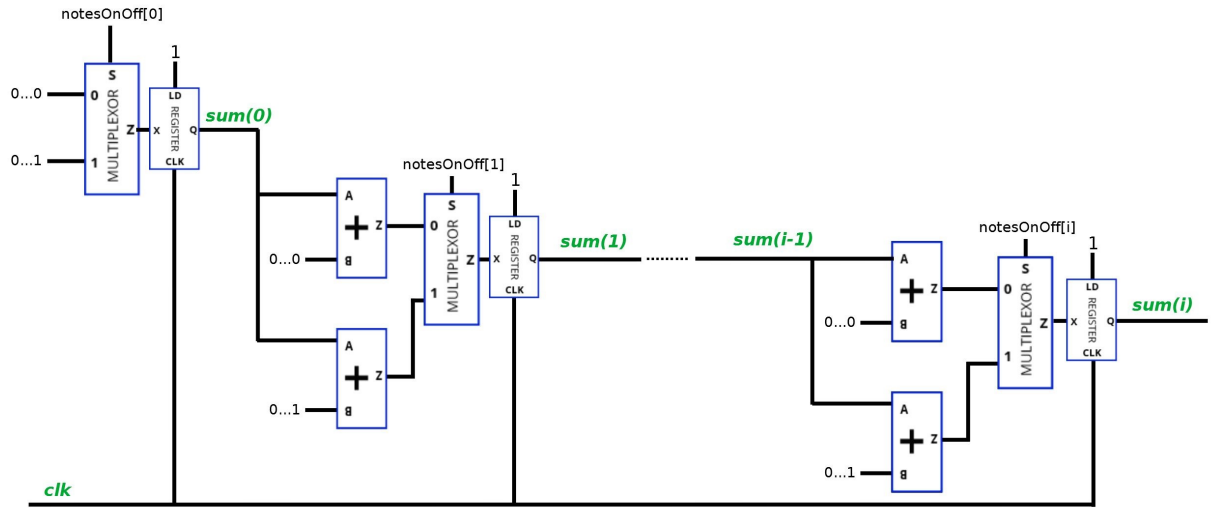


**Figura 4.15**: *Esquemático de la ruta de datos segmentada implementada en CountGensOn*

### 4.6.6. Cálculo y mapeo de constantes en ROM, para la generación de audio

En la explicación del subcomponente Universal note se ha comentado superficialmente el uso de distintas constantes para la generación de sonido, según el diseño estas constantes estarían almacenadas en una memoria ROM contenidas en el componente KeyboardCntrl. Estas constantes son: la dirección de la primera muestra de la nota base utilizada para interpolar (en UniversalNoteGen, el conjunto de señales startAddr_In), la dirección de la primera muestra perteneciente al intervalo a repetir en el bucle de sustain (en UniversalNoteGen, el conjunto de señales sustainStartOffsetAddr_In), la dirección de la última muestra pertenciente al mencionado intervalo, y el valor de la variable *ci* de la fórmula de interpola-

ción.

Los valores de las direcciones de las notas almacenadas en memoria son simplemente los 30 primeros múltiplos de 189644 incluyendo el 0. Esto es debido a que las solicitudes de lectura a memoria de los UinversalNoteGen se realizan direccionando muestras (necesitándose 26 bits para representar la dirección de una muestra, una muestra ocupa 2 bytes). Así pues, sabiendo que cada nota es representada por 189644 muestras, para direccionar las notas solo hay que direccionar los múltiplos de este número. En el código fuente el cálculo de estos valores se realiza durante la etapa de síntesis mediante la aplicación de dicha multiplicación por cada definición de entrada de la ROM.

La fórmula para el cálculo de muestras correspondientes al inicio de un período (§2.4) ha sido utilizada para definir los distintos intervalos para realizar el efecto sustain en función de la nota a reproducir. Para la aplicación de esta fórmula en el diseño, se ha programado una sencilla subrutina denominada en el código fuente como getSustainAddr(). Esta subrutina se aplica a cada entrada de las memorias ROM que contienen los valores para las señales sustainStartOffsetAddr_In (primera muestra del intervalo), y sustainEndOffsetAddr_In (última muestra del intervalo). Dicha subrutina es ejecutada durante el proceso de síntesis del diseño automatizando el cálculo de estas constantes. Obsérvese que al establecer la dirección de la muestra de inicio, el valor de la variable *offset* en la fórmula debe ser mayor que el valor de esa misma variable usada para calcular la dirección de la muestra final.

Por último el valor de la variable *ci* de la fórmula de interpolación, se obtiene dividiendo la frecuencia de la nota base utilizada entre la frecuencia de la nota objetivo. Este valor finalmente es representado en el formato de punto fijo Q32.32.

La indexación de estos valores se realiza mediante los dos bytes que representan cada nota (§2.1).

Terminando con esta subsección, aclarar que las señales que representan estas constantes se propagan a todos los subcomponentes UniversalNoteGen (generadores de notas) contenidos en NotesGenerator.

### 4.6.7. Mensajes MIDI soportados. Formato de los comandos de teclado

La lógica para la activación de los generadores de notas (subcomponentes UniversalNoteGen) se realiza a nivel del componente KeybaordCntrl. Esta lógica se puede entender como un intérprete de los comandos de teclado, pudiendo entender estos como pseudo-mensajes MIDI. Estos comandos de teclado siguen un formato en particular para poder soportar 2 tipos de funcionalidades básicas, la activación de una nota (*NoteOn*) y el apagado de una nota (*NoteOff*).

El formato de los comandos de teclado consiste en 15 bits en los que se distinguen los siguientes campos:

- **Velocity [3 : 0]**. Este campo indica la intensidad con la que debe sonar la nota representada en el campo **NoteCode** en caso que el campo CMD representa la activación de una nota. Si no representa el apagado de la nota, no se tiene en cuenta este campo. Puede tomar los siguientes valores que representan las distintas intensisdades, descritas en más detalle en el subcomponentes UniversalNoteGen (§4.6.2):
    - **Intensidad muy alta**, codificado como 1000.
    - **Intensidad alta**, codificado como 0100
    - **Intensidad normal**, codificado como 0000.
    - **Intensidad baja**, codificado como 0010.
    - **Intensidad muy baja**, codificado como 0001.

- **NoteCode [11 : 4]**. Campo que indica la nota que debe sonar o dejar de sonar en función del valor representado en el campo **CMD**. La codificación de las notas se corresponde a la que define el estándar MIDI (§2.1).

- **CMD [13 : 12]**. Campo que representa la acción a realizar por el intérprete. Si los bits tienen el valor de 10, se debe activar la nota correspondiente al campo **NoteCode** con la intensidad del campo **Velocity** (*Note-On*). En caso de tener el valor de 01, se debe desactivar la nota correspondiente al campo **NoteCode** (*Note-Off*). Si los bits toman valores diferentes el mensaje se ignora y el estado de los generadores de nota no varía.

- **SourceIndex [14]**. Este campo indica si el comando de teclado proviene del componente MidiParser (valor del bit a 0) o del componente ExternInterfaceCmdReceiver (valor del bit a 1). Esto permite saber que generadores de notas han sido activados por MidiParser o ExternInterfaceCmdReceiver, así si se ordena el fin de reproducción de la canción o si se sale del modo conexión con la interfaz externa, se sabe que generadores de notas deben ser desactivados.

La siguiente imagen representa los campos anteriormente descritos:

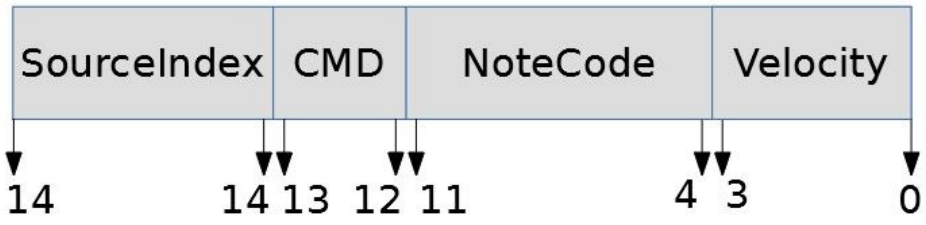


**Figura 4.16**: *Formato de los comandos de teclado recibidos por KeyboardCntrl*

Este intérprete implementa por cada generador de nota dos pares de registros de un solo bit, uno para el almacenamiento del flag que permite el encendido/apagado del generador

de nota correspondiente, y el otro para el marcaje llevado acabo para saber que componente ha sido responsable de la activación del generador de nota correspondiente. A su vez se almacena en un registro de 8 bits, la nota que esta sonando en un generador determinado, así en caso de que llegue una orden de activación de una nota que ya este sonando, el mensaje se ignorara. Consecuentemente solo puede haber una misma nota activa, por ejemplo si ya esta sonando C4 se ignoraran todas las solicitudes de activación de la nota C4. Esto se realiza mediante el uso de la señales internas noteIndexOff y foundCode.

Terminando con la explicación de KeyboardCntrl, en las siguientes páginas se adjuntan un par de diagramas RTL que muestran una posible implementación de las señales:

- **foundAviable**, señal interna activa a alta que indica si hay algún generador de nota disponible. Consecuentemente, esta señal estará a baja solamente cuando todos los generados de sonido estén activos.

- **noteIndexOn**, señales internas que en su conjunto representan el índice del generador de nota que debe ser activado.

- **foundCode**, señal interna activa a alta que indica si hay algún generador de nota activo con la misma nota que la representada en el campo NoteCode del comando de teclado recibido (representado con el conjunto de señales cmdKeyboard).

- **noteIndexOff**, señales internas que en su conjunto representan el índice del generador de nota que debe ser desactivado.

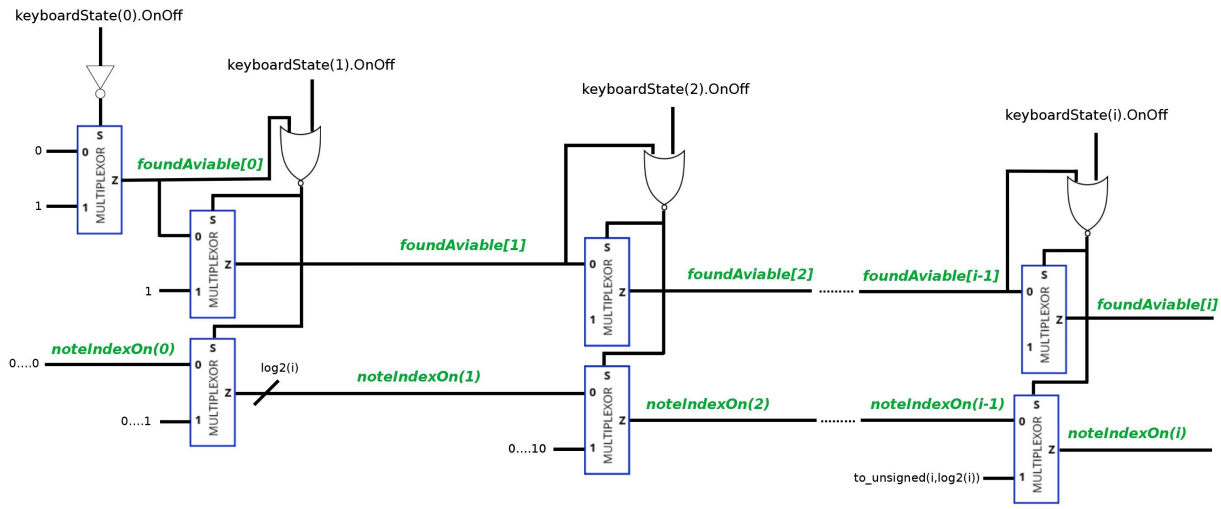


**Figura 4.17**: *Diagrama RTL que representa una posible implementación de las señales noteIndexOn y foundAviable*

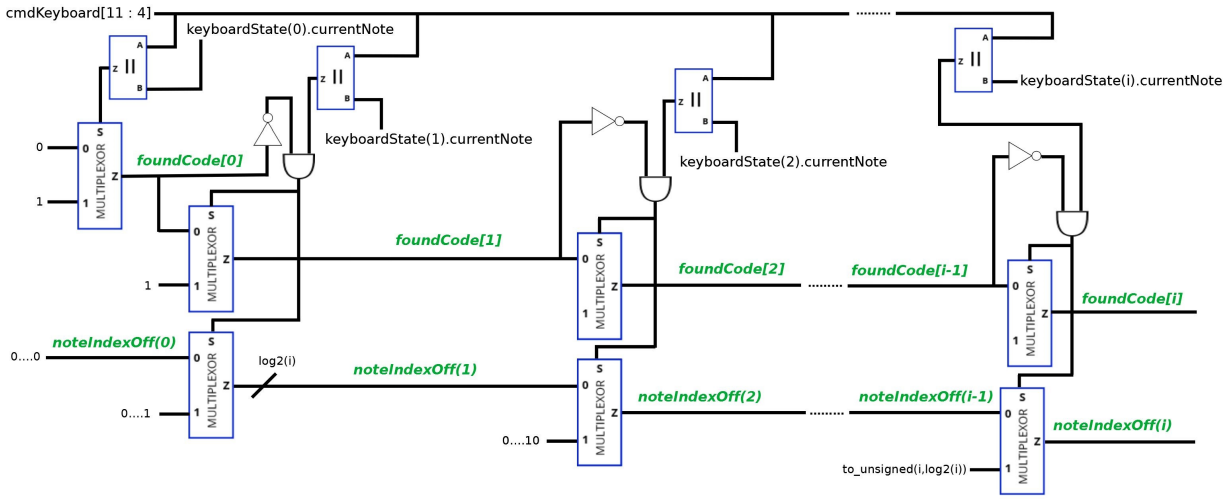


**Figura 4.18**: *Diagrama RTL que representa una posible implementación de las señales noteIndexOff y foundCode*

## 4.7. Componente MidiParser

### 4.7.1. Visión general

El componente MidiParser es el encargado de reproducir el archivo MIDI almacenado en memoria mediante el envío de los comandos de teclado al componente CmdKeyboardSequencer, para posteriormente ser finalmente enviados a KeyboardCntrl. Antes de iniciar la reproducción del archivo MIDI, se realiza un parseo del mismo para comprobar si la cabecera y las pistas cumplen con los requisitos requeridos (§3.1.1). La lectura del archivo MIDI se realiza con el envío y recepción de los *MidiParser CMDs* de solicitud de lectura (§4.6) y *MidiParser CMDs* de respuesta a lectura (§4.7) respectivamente, siendo en MidiParser donde se define la lógica de gestión de estos comandos de memoria. También es en este componente en donde se define la lógica de gestión de tempo (§4.7.6).

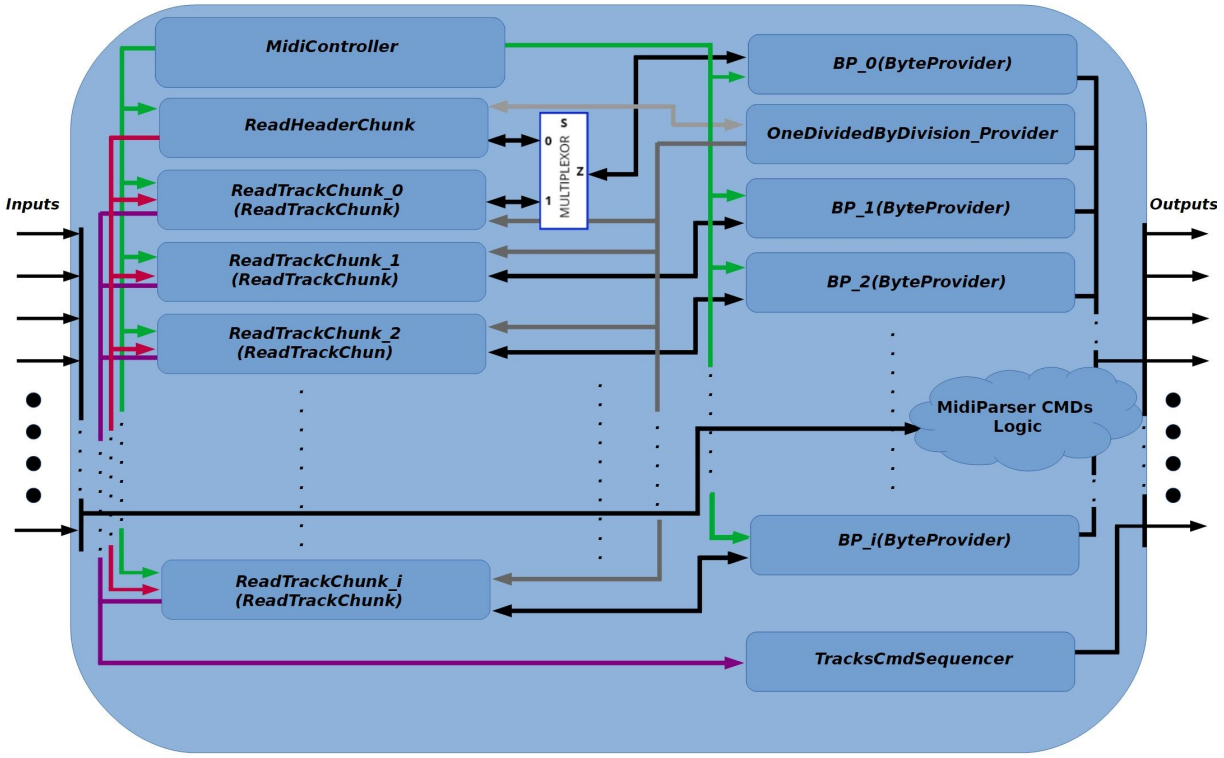


**Figura 4.19**: *Esquemático del componente MidiParser, en el que se presentan las principales interconexiones entre los subcomponentes*

Las señales a destacar en la interfaz de uso de MidiParser se enumeran a continuación:

**Señales de entrada, (*Inputs*):**

- **cen [1]**: señal de habilitación del componente y de los distintos subcomponentes que contiene. Activa a baja.
- **readMidifileRqt [1]**: señal activa a alta durante un ciclo iniciando la reproducción de un archivo MIDI.
- **keyboard_ack [1]**: señal activa a alta durante un ciclo indicando la confirmación de aplicación del último comando de teclado enviado.
- **mem_emptyBuffer [1]**: señal activa a alta, indicando si el buffer encargado de almacenar los *MidiParser CMDs* de respuesta a lectura (Fifo_outCmdReadBuffer_0) esta vacío o no.
- **mem_CmdReadResponse [log2(MAX_NUM_TRACKS)+129]**: señales que en su conjunto forman los bits que representan un *MidiParser CMDs* de respuesta a lectura. Provenientes directamente de RamCntrl (concretamente del buffer Fifo_outCmdReadBuffer_0).
- **mem_fullBuffer [1]**: señal activa a alta, indicando que el buffer que almacena los *MidiParser CMDs* de solicitud de lectura (Fifo_inCmdReadBuffer_0) esta lleno.

**Señales de salida, (*Outputs*):**

- **fileOk [1]**: señal activa a alta indicando si el archivo MIDI cumple los requisitos de formatos exigidos por el diseño (más detalles en las secciones ReadHeaderChunk, y ReadTrackChunk).
- **OnOff [1]**: señal activa a alta indicando si el componente MidiParser a iniciado el proceso de lectura del archivo MIDI.

- **statesOut_MidiCntrl [5]**: conjunto de señales utilizadas para conocer el estado actual del subcomponente MidiController. Los estados siguen una codificación *one hot*.

- **aviableCmd [1]**: señal activa a alta indicando la llegada de un nuevo comando de teclado por el conjunto de señales **cmdKeyboard**.

- **cmdKeyboard [15]**: señales que en su conjunto forman los bits que representan un comando de teclado.

- **mem_CmdReadRequest [log2(MAX_NUM_TRACKS)+26]**: señales que en su conjunto forman los bits que representan un *MidiParser CMDs* de solicitud de lectura.

- **mem_readResponseBuffer [1]**: señal activa a alta durante un ciclo, realizando una orden de lectura al buffer Fifo_outCmdReadBuffer_0.

- **mem_writeReciveBuffer [1]**: señal activa a alta durante un ciclo, realizando una orden de escritura al buffer Fifo_inCmdReadBuffer_0.

A continuación se procede a la explicación de los distintos subcomponentes que intervienen en la reproducción de un archivo MIDI. Dicha reproducción es iniciada mediante la activación de la señal de entrada readMidifileRqt, señal directamente conectada al subcomponente MidiController.

### 4.7.2. Subcomponente MidiController

MidiController se encarga de ordenar la activación de los subcomponentes ReadHeaderChunk (encargado de leer la cabecera) y de los ReadTrackChunk (encargados de leer las pistas). También se encarga de la multiplexación de las señales provenientes del subcomponente BP_0(ByteProvider), tal y como se muestra en el esquemático al comienzo

de esta sección (§4.19). Así MidiController gestiona quien es el subcomponente que realiza las peticiones de bytes al subcomponente BP_0(ByteProvider), siendo este usado por ReadHeaderChunk o ReadTrackChunk_0 según convenga. El número de subcomponentes ReadTrackChunk a activar depende directamente del número de pistas que tenga el archivo MIDI.

Las principales señales presentes en la interfaz de uso de MidiController se enumeran a continuación:

**Señales de entrada, (*Inputs*):**

- **readMidifileRqt [1]**: señal activa a alta durante un ciclo iniciando la reproducción de un archivo MIDI.
- **finishHeaderRead [1]**: señal activa a alta durante un ciclo indicando la finalización de lectura de la cabecera del archivo MIDI.
- **headerOK [1]**: señal activa a alta, indicando si la cabecera cumple el formato exigido.
- **finishTracksRead [MAX_NUM_TRACKS]**: señales activas a alta durante un ciclo, notificando la finalización de una lectura en modo comprobación o en modo reproducción. Cada línea corresponde a uno de los subcomponentes ReadTrackChunk.
- **tracksOK [MAX_NUM_TRACKS]**: señales activas a alta, indicando si la última lectura en modo de comprobación fue exitosa. Cada línea corresponde a uno de los subcomponentes ReadTrackChunk.
- **ODBD_ValReady [1]**: señal activas a alta, indicando si el registro interno del subcomponente OneDividedByDivision_Provider contiene el valor de la última constante solicitada. La solicitud de esta constante la realiza el subcomponente ReadHeaderChunk.
- **numTracksToRead [16]**: señales que en su conjunto forman los bits que representan el número de pistas que tiene el último archivo MIDI leído por ReadHeaderChunk.

**Señales de salida, (*Outputs*):**

- **readHeaderRqt [1]**: señal activa a alta durante un ciclo indicando a ReadHeaderChunk que debe iniciar la comprobación de la cabecera del archivo MIDI cargado en memoria.

- **muxBP_0 [1]**: señal correspondiente a las entradas de selección de los multiplexores encargados de multiplexar las señales del subcomponente BP_0 (ByteProvider) para que este pueda ser usado por ReadHeaderChunk (valor 0 lógico) o por ReadTrackChunk_0 (valor 1 lógico).

- **goFirstRead [1]**: señal activa a alta durante un ciclo ordenando el "reseteo" al estado inicial de todos los BytesProvider.

- **readTracksRqt [2*MAX_NUM_TRACKS]**: señales directamente conectadas al par de entradas de cada ReadHeaderChunk que con su activación a alta durante un ciclo, permiten iniciar una lectura en modo comprobación o en modo reproducción.

- **parseOnOff [1]**: señal activa a alta, indicando el inicio del proceso completo de lectura del archivo MIDI llevado por el componente MidiParser.

- **fileOk [1]**: señal activa a alta indicando si el archivo MIDI cumple los requisitos de formatos exigidos por el diseño (más detalles en las secciones ReadHeaderChunk, y ReadTrackChunk).

- **statesOut_MidiCntrl [5]**: conjunto de señales utilizadas para conocer el estado actual del subcomponente MidiController. Los estados siguen una codificación *one hot*.

El diseño de este subcomponente da a entender que se hace uso de una memoria ROM asíncrona. La ROM es utilizada para el almacenamiento de las tramas de bits que tiene que ser volcada en el conjunto de señales readTracksRqt para la activación del número exacto de subcomponentes ReadTrackChunk.

En el código fuente tanto el valor de las entradas de la ROM, como la multiplexación de entradas de la misma en función del número de pistas del archivo MIDI a reproducirse, se define de la siguiente forma:

```
romVals:
process(numTracksToRead)
        type    rom_t    is array( 0 to MAX_NUM_TRACKS) of
                 std_logic_vector(MAX_NUM_TRACKS*2-1 downto 0);

    variable romValues   :    rom_t;
begin

    readTracksRom <=romValues(to_integer(
        unsigned(numTracksToRead)));

    romValues :=(others=>(others=>'0'));
    for i in 1 to MAX_NUM_TRACKS loop
        for j in 1 to i loop
            romValues(i)(2*(j-1)) :='1';
        end loop;
    end loop;
end process;
```

Como se puede observar, la ROM se implementa según la variable genérica MAX_NUM-_TRACKS, flexibilizando el uso de este subcomponente.

Cabe destacar la implementación de las señales internas tmpFinish, tmpTracks y check-FinishTracksOk utilizadas para detectar la finalización de reproducción de los ReadTrack-Chunk. El siguiente diagrama RTL se corresponde a la implementación que da a entender el código fuente del diseño:

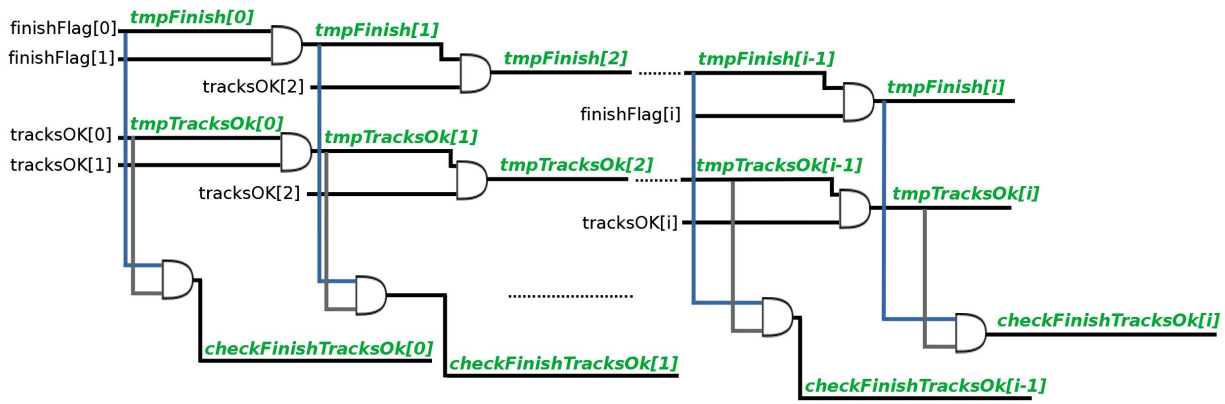


**Figura 4.20**: *Diagrama RTL representando una posible implementación de las señales tmpFinish, tmpTracks y checkFinishTracksOk*

Por último decir que los subcomponentes ByteProviders sufren un "reseteo" siempre que finaliza una orden de reproducción, mediante la activación de la señal de salida goFirstRead. Esto es necesario ya que los ByteProviders cachean en un registro de 128 bits la última trama de datos obtenida de un MidiParser CMD de respuesta a lectura. Esto provoca que en caso de cargar un nuevo archivo MIDI en memoria, pueda haber una falta de coherencia con la trama de 128 bits de datos contenida en el byte provider.

Ejemplificando, suponiendo que no existe la funcionalidad de relacionada con la señal goFirstRead y el último intento de lectura de un archivo MIDI ha acabado en un error de formato en la cabecera, independientemente del archivo MIDI contenido en memoria los siguientes intentos de lectura fallaran. Esto es debido a que los datos están cacheados en BP_0 y por ello este no realiza ninguna petición a memoria impidiendo así la actualización de estos datos. Así la señal goFirstRead obliga a todos los subcomponentes ByteProviders a realizar una petición a memoria en que caso de que se les haga una nueva solicitud.

### 4.7.3. Subcomponente ByteProvider

Este subcomponente se encargan de servir los bytes de datos correspondientes al archivo MIDI. Para servir estos datos, ByteProvider solicita el envío de un *MidiParser CMD* de solicitud de lectura (§4.6) y espera hasta la llegada del dato recibido por el *MidiParser CMD*

de respuesta a lectura tipo A (§4.7) correspondiente. En el diseño del sistema hay tantos ByteProviders como subcomponentes ReadTrackChunk.

La presencia de estos esta justificada debido a dos principales razones:

1. La lectura de un archivo MIDI esta dirigida por bytes, es decir, la unidad mínima de interpretación de los datos del archivo son los bytes por lo que necesitamos poder hacer llegar dichos bytes de datos desde memoria al componente MidiParser de la forma más simple posible. Estos bytes de datos deben llegar de la forma más paralela posible a todos los ReadTrackChunk encargados de leer las pistas del archivo que estuviese siendo procesado.

2. Debido a la interfaz otorgada por el IP core MIG 7 (nombrado en el sistema como el subcomponente ddr), esta siempre realiza lecturas de 16 bytes. Es posible aprovechar este hecho y cachear esa lectura con el fin optimizar el número de MidiParser CMDs de solicitud de lectura enviados. Al cachear la trama de 16 bytes, se logra que las próximas solicitudes de los 15 siguientes bytes se obtengan directamente de la trama cacheada, sin necesidad de tener que generar un MidiParser CMDs de solicitud de lectura.

Las señales a destacar en la interfaz de uso de este subcomponente son:

**Señales de entrada, (*Inputs*):**

- **addrInVal [27]**: señales que en su conjunto forman los bits que representan la dirección del byte solicitado.

- **byteRqt [1]**: señal activa a alta durante un ciclo solicitando el byte en la dirección de memoria indicada por addrInVal. Este byte es devuelto a través de las señales nextByte.

- **goFirstRead [1]**: señal activa a alta durante un ciclo ordenando el "reseteo" al estado incial del subcomponente.

- **dataIn [128]**: conjunto de señales formadas por la trama de datos correspondiente al campo MidiFileData del *MidiParser CMD* de respuesta a lectura tipo A. Esta señales es compartida por todos los subcomponentes ByteProvider.

- **memAckResponse [1]**: señal activa a alta durante un ciclo indicando la recepción de una nueva trama de datos a través de las señales dataIn.

- **memAckSend [1]**: señal activa a alta durante un ciclo indicando la confirmación por parte de la lógica de MidiParser, el envío de un MidiParser CMDs de solicitud de lectura.

**Señales de salida, (*Outputs*):**

- **byteAck [1]**: señal activa a alta durante un ciclo indicando la actualización de las señales nextByte representando estas el último byte de datos solicitado.

- **nextByte [8]**: señales que en su conjunto forman los bits que representan el último o penúlitmo byte de datos solicitado.

- **addr_out [23]**: señales que en su conjunto forman los bits que representan la dirección de la última trama de datos solicitada. La dirección representada direcciona 16 bytes, por ello el uso de 23 bits.

- **memSamplesSendRqt [1]**: señal activa a alta, solicitando a parte de la lógica de MidiParser el envío de un MidiParserCMD de solicitud de lectura, utilizando addr_out y tomando el campo ProviderIndex el valor correspondiente según el ByteProvider solicitante.

La arquitectura de este subcomponente consiste principalmente en el uso de un registro de 16 bytes en el que se almacena la respuesta de la última solicitud de datos realizada por el mismo. A su vez, se hace uso de otro registro para almacenar la dirección del último byte de datos que le han solicitado a ByteProvider. De esta forma la trama de 16 bytes es

cacheada y mediante la decodificación de la dirección que acompaña una solicitud de un byte de datos realizada a ByteProvider, se reutilizan los bytes de datos cacheados.

Finalizando con la explicación de este subcomponente, se presenta un esquemático simplificado en el que figuran los principales elementos de *hardware* que forman un ByteProvider:

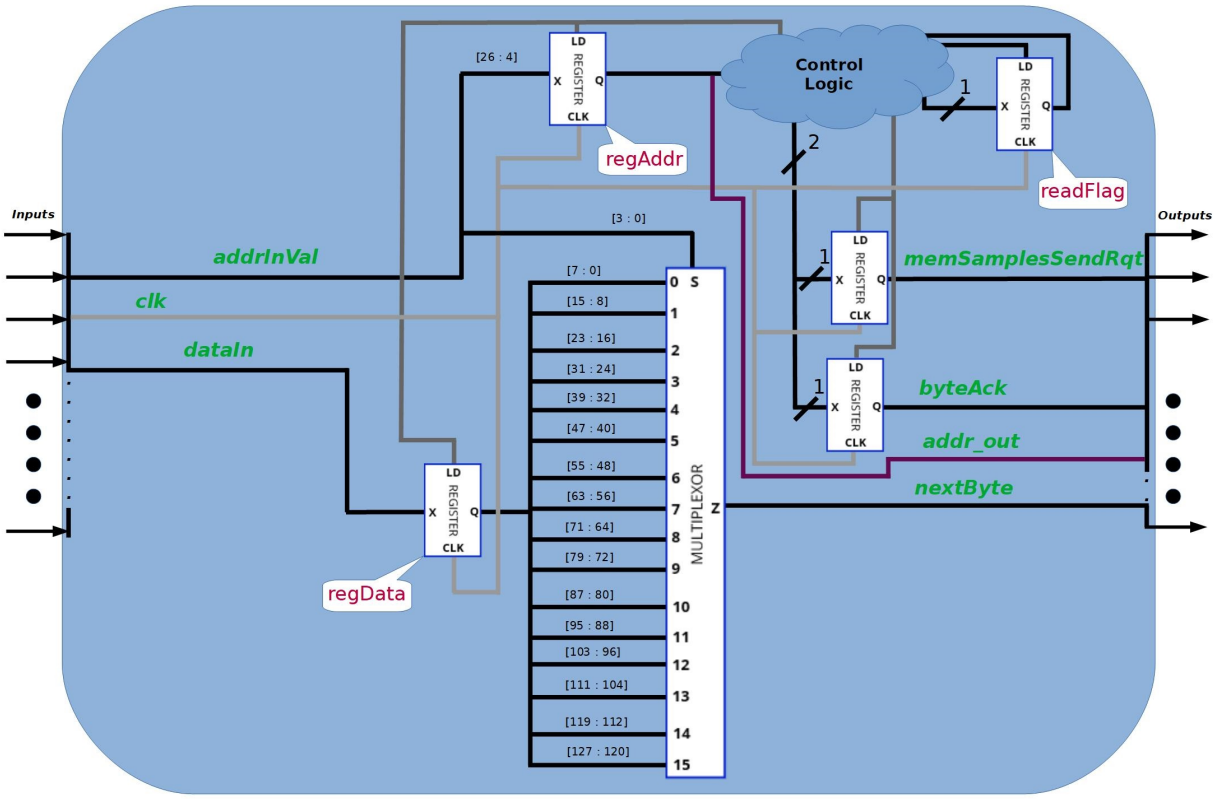


**Figura 4.21**: *Esquemático simplificado del subcomponente ByteProvider*

### 4.7.4. Subcomponente ODBDProvider

Este subcomponente se encarga de almacenar el valor adecuado de la constante ODBD requerida según el archivo MIDI que se este leyendo. Al recibir una solicitud de lectura por parte de ReadHeaderChunk, este subcomponente es el encargado solicitar el envío de un *MidiParser CMD* de solicitud de lectura (§4.6) y esperar hasta la llegada del dato recibido por el *MidiParser CMD* de respuesta a lectura tipo B (§4.8). Este dato se corresponde con la constante ODBD de memoria, siendo esta el valor de dividir uno entre la variable *division*

contenida en la cabecera del archivo MIDI que esta siendo procesado.

Las señales a destacar en la interfaz de uso de ODBDProvider son:

**Señales de entrada, (*Inputs*):**

- **readRqt [1]**: señal activa a alta durante un ciclo iniciando la solicitud de la constante ODBD en la dirección resultante de sumar START_ADDR y el valor representado por las señales de entrada **division** menos uno. Tras recibir la respuesta el valor es volcado en las señales de salida OneDividedByDivision.

- **division [15]**: señales que en su conjunto forman los bits que representan el valor de la última constante división leída por el subcomponente ReadHeaderChunk. Se utiliza para direccionar las constantes de 32 bits almacenadas en la memoria.

- **dataIn [24]**: conjunto de señales formadas por la trama de datos correspondiente a los últimos 24 bits del campo ODBDconstant del *MidiParser CMD* de respuesta a lectura tipo B.

- **memAckResponse [1]**: señal activa a alta durante un ciclo indicando la recepción de una nueva constante ODBD a través de las señales dataIn.

- **memAckSend [1]**: señal activa a alta durante un ciclo indicando la confirmación por parte de la lógica de MidiParser, el envío de un MidiParser CMDs de la solicitud de lectura tipo B.

**Señales de salida, (*Outputs*):**

- **readyValue [1]**: señal activa a alta indicando que la última solicitud de lectura de una constante ODBD ha sido realizada con éxito, actualizando el valor de las señales OneDividedByDivision en función del valor de la constante leída.

- **OneDividedByDivision [24]**: señales que en su conjunto forman los bits que representan la constante ODBD.

- **addr_out [25]**: señales que en su conjunto forman los bits que representan la dirección de la constante ODBD de datos solicitada. La direeción representada direcciona 4 bytes, por ello el uso de 25 bits.

- **memSamplesSendRqt [1]**: señal activa a alta, solicitando a parte de la lógica de MidiParser el envío de un MidiParserCMD de solicitud de lectura, utilizando addr_out y tomando el valor 0 el campo ProviderIndex.

Para obtener la dirección de la constante solicitada (representada por las señales de entrada division), es necesario sumar la dirección de comienzo de la sección ODBDConstsReducedSet (§4.2) y el valor del campo division menos uno. La resta de menos uno es necesaria ya que las señales de entrada division nunca va a tener el valor 0, acotando los posibles valores al intervalo [1,32767]. El valor de la constante ODBD es "publicada" a todos los subcomponentes ReadTrackChunk a través de las señales de salida OneDividedByDivision. Así estos son capaces de calcular el tiempo a esperar por cada mensaje MIDI que este siendo interpretado.

### 4.7.5. Subcomponente ReadHeaderChunk

Este subcomponente, a petición de MidiController, se encarga de leer la cabecera del archivo MIDI almacenado en memoria y paralelamente comprueba si la cabecera cumple el formato exigido por el sistema. También se encarga de solicitar una lectura al subcomponente ODBDProvider en caso de que el valor del campo *division* cumpla con los requisitos. El resultado de esta comprobación es transmitido a MidiController.

La comprobación del formato se realiza ordenadamente según las siguientes etapas:

1. Los primeros 4 bytes deben de tener el valor en hexadecimal 4d546864. Estos valores se corresponden a los caracteres ASCII MThd ubicados al inicio de la cabecera de cualquier archivo MIDI.

2. Los siguientes 4 bytes deben de tener el valor en hexadecimal 00000006. Estos bytes se corresponden al campo longitud de la cabecera, indicando la cantidad de bytes que faltan por leer en la misma. Este valor es común a todos los archivos MIDI

3. Los sucesivos 2 bytes deben tener el valor en hexadecimal de 0000 o 0001. Estos bytes se corresponden al campo formato de la cabecera, indicando así la organización de la información contenida en el archivo MIDI. El sistema es compatible con el formato 0 (valor 0x0000) y formato 1 (valor 0x0001), en caso de tener otro valor el proceso de lectura de la cabecera termina sin éxito.

4. Los próximos 2 bytes son los correspondientes al campo *ntrks* de la cabecera, indicando el número de pistas que contiene el archivo MIDI que esta siendo procesado. En caso de que el formato del archivo MIDI sea 0, este campo tiene que tener el valor 1 en caso contrario la lectura de la cabecera termina sin éxito. Por otro lado, si el formato es 1, el valor del campo *ntrks* debe ser menor o igual al valor especificado en la variable genérica MAX_NUM_TRACKS, en caso contrario el proceso de lectura de la cabecera termina sin éxito.

5. Los consecuentes 2 bytes, se corresponden al campo *division* de la cabecera, indicando la forma de entender el tiempo-delta que acompaña a cada mensaje MIDI. Para poder finalizar la lectura de la cabecera con éxito, el bit más significativo de este campo debe tener un valor lógico 0 y el valor numérico representado por el resto de bits debe ser superior a 0. En caso contrario el proceso de lectura de la cabecera termina sin éxito.

6. La última etapa realizada en este subcomponente para comprobar el correcto formato de un archivo MIDI consiste en contrastar las direcciones de comienzo de las pistas contenidas en el mismo. Al calcular las direcciones de comienzo de cada pista se comprueba que no se produzca overflow, (esto se realiza comprobando el bit de índice 27 del conjunto de bits que representan la dirección de comienzo de una pista). Esta comprobación es necesaria ya que puede darse el hipotético caso de que el archivo

almacenado no cupiese en la memoria, de esta forma nos aseguramos parcialmente que el archivo entero esta cargado en memoria. Esto último se termina de asegurar al realizar una lectura de comprobación por los subcomponentes ReadTrackChunk cada uno de ellos asociado a una pista del archivo MIDI.

La interfaz de uso de ReadHeaderChunk esta dotada de las siguientes señales:

**Señales de entrada, (*Inputs*):**

- **readRqt [1]**: señal activa a alta durante un ciclo iniciando la lectura de la cabecera del archivo MIDI almacenado en memoria.
- **byteAck [1]**: señal activa a alta durante un ciclo indicando la actualización de las señales nextByte representando estas el último byte de datos solicitado.
- **nextByte [8]**: señales que en su conjunto forman los bits que representan el último o penúlitmo byte de datos solicitado.

**Señales de salida, (*Outputs*):**

- **finishRead [1]**: señal activa a alta durante un ciclo indicando la finalización del proceso de lectura de la cabecera del archivo MIDI.
- **headerOk [1]**: señal activa a alta indicando que la última cabecera leída cumple los requisitos de formato de cabecera exigidos por el sistema.
- **numTracksToRead [25]**: señales que en su conjunto forman los bits que representan el número de pistas contenidas en el último archivo MIDI leído.
- **ODBD_ReadRqt [1]**: señal activa a alta durante un ciclo, solicitando al subcomponente OneDividedByDivision_Provider la actualización de la constante ODBD almacenada en dicho subcomponente.
- **division [16]**: señales que en su conjunto forman los bits que representan el valor del campo división de la última cabecera de un archivo MIDI leída.

- **tracksAddrStart [MAX_NUM_TRACKS*27]**: en caso de que el último archivo MIDI halla sido leído con éxito, estas señales en su conjunto forman los bits que representan las direcciones de comienzo de cada una de las pistas contenidas en el último archivo MIDI leído.

- **byteAddr [27]**: señales que en su conjunto forman los bits que representan la dirección del último byte solicitado a BP_0 (ByteProvider).

- **byteRqt [1]**: señal activa a alta, realizando una solicitud de un byte de datos al ByteProvider correspondiente (en esta caso BP_0).

Las acciones realizadas en cada una de las anteriores etapas se corresponden directamente a cada uno de los estados presentes en la máquina algorítmica de estados del subcomponente, exceptuando al estado inicial tras el reseteo **s0**, encargado de detectar la activación de la señal de entrada readRqt.

A continuación se expone la correlación entre etapas y estados:

| Estados | Etapa |
|---|---|
| s1 | Etapa 1 |
| s2 | Etapa 2 |
| s3 | Etapa 3 |
| s4 | Etapa 4 |
| s5 | Etapa 5 |
| s6 | Etapa 6 |
| s7 | Etapa 6 |

**Tabla 4.2**: *Correlación entre etapas de lectura de la cabecera y estado encargado de realizar las acciones correspondientes a la etapa asociada*

### 4.7.6. Subcomponente ReadTrackChunk y gestión del tempo

Este subcomponente, a petición de MidiController, es el encargado de leer una pista de un archivo MIDI. Esta lectura se puede realizar en modo comprobación o en modo reproducción. Solo al realizar una lectura en modo comprobación se actualiza la señal de salida trackOK, modificando apropiadamente el estado de finalización de la última lectura en modo comprobación. Por otro lado, al realizar una lectura en modo de reproducción, se envían comandos de teclados al subcomponente TracksCmdSequencer en función de los mensajes MIDI interpretados a lo largo del proceso de lectura y realizando la espera de tiempo correspondiente.

La interfaz de uso de este subcomponente esta dotada de las señales:

**Señales de entrada, (*Inputs*):**

- **readRqt [2]**: señales activa a alta durante un ciclo iniciando la lectura de la pista en modo comprobación (activa la línea de índice 0), o iniciando la lectura de la pista en modo reproducción. En caso de que ambas estén activas, el modo comprobación es prioritario.

- **trackAddrStart [27]**: señales que en su conjunto forman los bits que representan el primer byte perteneciente al inicio de una pista del archivo MIDI que esta siendo procesado.

- **OneDividedByDivision [24]**: señales que en su conjunto forman los bits que representan la última constante ODBD solicitada por ReadHeaderChunk.

- **currentTempo [24]**: señales que en su conjunto forman los bits que representan el valor numérico de los microsegundos por tiempo de "Negra", del último meta-evento de actualización de tempo recibido por cualquier pista. Siguiendo el estándar MIDI [21], el valor por defecto al iniciar la reproducción de un archivo MIDI es 500000.

- **updateTempoAck [1]**: señal activa por un ciclo indicando la confirmación de la última solicitud de cambio de tempo realizada por el subcomponente ReadTrackChunk en cuestión.

- **sequencerAck [1]**: señal activa por un ciclo indicando la confirmación de recepción del último comando de teclado envíado por el subcomponente ReadTrackChunk en cuestión.

- **nextByte [8]**: señales que en su conjunto forman los bits que representan el último o penúltimo byte de datos solicitado.

- **byteAck [1]**: señal activa a alta durante un ciclo indicando la actualización de las señales nextByte.

**Señales de salida, (*Outputs*):**

- **updateTempoRqt [1]**: señal activa a alta indicando una solicitud de cambio de tempo a la lógica de gestión de tempo contenida en el componente MidiParser.

- **updateTempoVal [24]**: señales que en su conjunto forman los bits que representan el valor de los datos recibidos por el último meta-evento de cambio de tempo leído por el subcomponente ReadTrackChunk en cuestión.

- **finishRead [1]**: señal activa durante un ciclo, indicando la finalización de la última orden de lectura recibida.

- **trackOK [1]**: señal que permite indicar el estado de finalización de la última orden de lectura recibida en modo comprobación. En caso de tener un valor de 1 lógico la lectura finalizo con éxito, y en caso contrario los datos no pueden ser interpretados correctamente por el subcomponente ReadTrackChunk en cuestión.

- **wrCmdRqt [1]**: señal activa a alta indicando el envío de un comando de teclado. Diriguida al subcomponente TracksCmdSequencer.

- **cmd [13]**: señales que en su conjunto forman los bits que representan el comando de teclado en proceso de envío o el último que ya halla sido enviado.

- **byteAddr [27]**: señales que en su conjunto forman los bits que representan la dirección del último byte solicitado a BP_i (ByteProvider) acorde al subcomponente ReadTrackChunk en cuestión.

- **byteRqt [1]**: señal activa a alta, realizando una solicitud de un byte de datos al ByteProvider correspondiente (en esta caso BP_i) acorde al subcomponente ReadTrackChunk en cuestión.

Antes de realizar el envío del comando de teclado, este subcomponente realiza la espera de tiempo necesaria que se indica en el tiempo-delta que acompaña a cada mensaje MIDI que esta siendo interpretado. Así se realiza la conversión de tiempo-delta a milisegundos siguiendo la consecuente fórmula (§2.2.2). Para la realización de esta conversión, en el diseño se ha decidido implementar una ruta de datos segmentada usando aritmética en punto fijo. El uso de este tipo de aritmética se debe al valor decimal de la constante ODBD (recordemos que esta se define como $1/division$).

En el siguiente diagrama RTL se muestra una posible implementación de dicha ruta de datos:

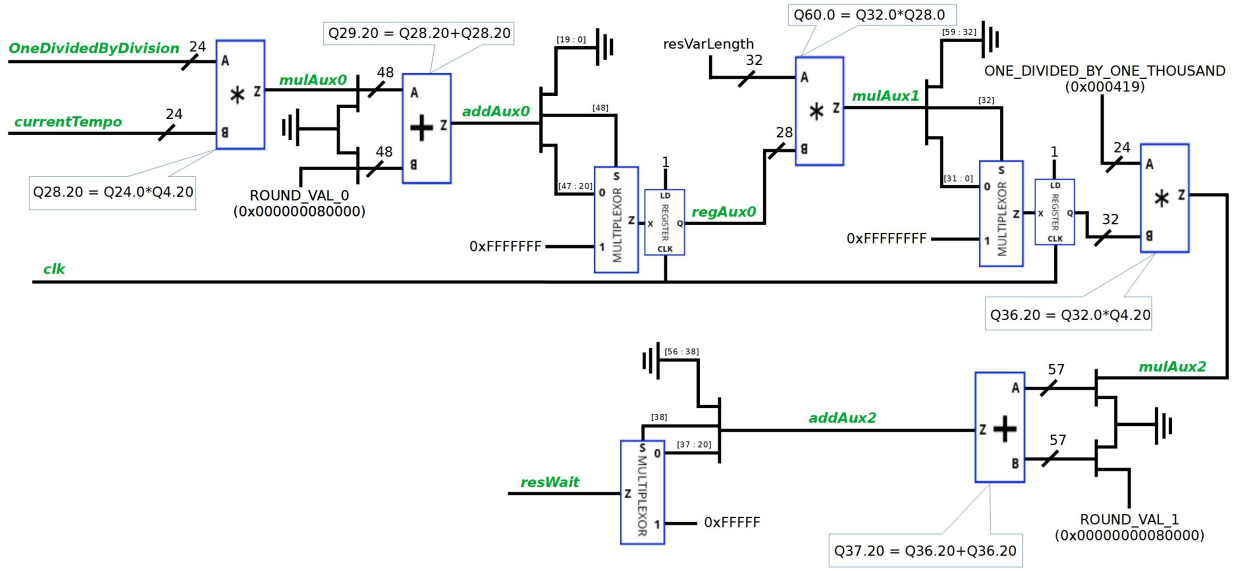


**Figura 4.22**: *Diagrama RTL correspondiente a la ruta de datos encargada de calcular el tiempo de espera en milisegundos por cada evento MTrk*

Cabe aclarar, que la segmentación aplicada en esta ruta de datos pretende facilitar el trabajo a las herramientas de síntesis e implementación reduciendo así los requisitos mínimos de *timing* del circuito. Este riesgo de tener una posible limitación en la frecuencia del reloj del sistema al intentar realizar estas operaciones en un solo ciclo, es causado principalmente por el gran número de líneas utilizadas al realizar la segunda multiplicación, y al hecho de tener que aplicar dos multiplicaciones seguidas (la lógica combinacional de la multiplicación es típicamente más costosa que la suma). Este hecho resulta en la posibilidad de tener un camino crítico, imponiendo así una restricción posiblemente más severa sobre los requisitos de *timing* del sistema si los comparamos con la misma ruta de datos segmentada.

Concluyendo con este párrafo, la ruta de datos final realiza el cálculo tras 2 ciclos de reloj y todos los valores con los que se opera son siempre positivos.

ReadTrackChunk contiene otros dos subcomponentes, MilisecondDivisor y ReadVarLength:

Empezando por el más sencillo de explicar, el subcomponente MilisecondDivisor se encarga de generar una señal periódica cuadrada de una frecuencia de 1KHz (período de 1 ms). Gracias a la señal producida por este subcomponente, es posible implementar la espera a realizar por cada evento MTrk (que contiene un mensaje MIDI) con un contador descendente que reste 1 tras transcurrir un milisegundo. Así este contador descendente previamente a iniciar la espera, sera inicializado al valor resultante de la ruta de datos segmentada que se ha mencionado con anterioridad.

Por otro lado, el subcomponente ReadVarLength presenta una mayor complejidad. Este subcomponente se encarga de interpretar todos aquellos bytes que codifiquen un número siguiendo el formato de longitud variable (§2.6). Este subcomponente realiza por si mismo la solicitud de los bytes, para ello es necesario implementar lógica a nivel de ReadTrackChunk que multiplexe las señales de salida byteAddr y byteRqt. Esto permite que tanto el subcomponente ReadVarLength y la lógica del propio ReadTrackChunk realicen solicitudes al subcomponente ByteProvider correspondiente.

Para confirmar la viabilidad de la correspondiente pista que este siendo leída en modo de comprobación se procede a realizar las siguientes acciones:

En primer lugar se coteja el valor de los primeros 4 bytes que debe ser ser igual al siguiente número en hexadecimal 4d54726b, estos bytes en ASCII forman la cadena de caracteres MTrk indicando el comienzo de una pista.

Tras comprobar la viabilidad de los primeros 4 bytes, se procede a leer los bytes correspondientes al campo longitud con el fin de verificar al final del proceso de lectura si este campo contiene un valor correcto o no. Los bytes correspondientes a dicho campo indican el número de bytes que preceden a partir del campo longitud, este valor es guardado en un registro. Durante el proceso de lectura en modo comprobación, al reconocer un meta-evento de fin de pista, se comprueba si el número de bytes leídos es igual al valor contenido en dicho campo.

Paralelamenta, la lógica de ReadTrackChunk comprueba al comienzo de la recepción de un nuevo evento MTrk si el número de bytes leídos hasta el momento no han superado el valor del campo longitud. Gracias a esto el número de bytes que pueden ser solicitados durante todo el proceso de lectura esta acotado, evitando así un posible bucle infinito ya que sin esta condición la lectura no finalizaría hasta reconocer un meta-evento de fin de pista.

En lo que respecta a la gestión de tempo, como se ha mencionado anteriormente su lógica esta ubicada al mismo nivel que la lógica encargada de gestionar los *MidiParser CMDs*.

Esta lógica consiste fundamentalmente en el uso de un contador para ir gestionando el turno de las señales updateTempoRqt procedentes de todos los subcomponentes ReadTrack-Chunk. Mediante la activación de dicha señal se procede a actualizar el registro interno que distribuye los valores de los bits contenidos en él a todos los ReadTrackChunk. De esta forma estos los ReadTrackChunk usan directamente el valor de las señales provenientes de dicho registro, para realizar el cálculo del tiempo a esperar por cada evento MTrk. A su vez, se envía una señal de confirmación de actualización al ReadTrackChunk correspondiente al turno (el mismo que ha enviado la solicitud). Concurrentemente, se procede a cambiar de turno.

Todas estas acciones son realizadas el ciclo siguiente al ciclo en que se halla detectado una solicitud de actualización de tempo (activación de la señal updateTempoRqt).

### 4.7.7. Subcomponente TracksCmdSequencer

Este subcomponente se encarga de secuencializar los envíos de los comandos de teclado provenientes de los ReadTrackChunk. Esta secuencialización es necesaria ya que Keyboard-Cntrl solo puede procesar un comando de teclado a la vez.

La interfaz de uso de este subcomponente esta contiene las señales:

**Señales de entrada, (*Inputs*):**

- **tracksCmd [MAX_NUM_TRACKS*14]**: conjunto de señales que forman los bits que representan los comandos de teclado generados por cada subcomponente ReadTrackChunk.

- **sendCmdRqt [MAX_NUM_TRACKS]**: conjunto de señales activas a alta, correspondientes a la señal de salida de cada subcomponente ReadTrackChunk.

- **keyboard_ack [1]**: señal activa a alta durante un ciclo indicando la confirmación de procesamiento del último comando de teclado enviado por este subcomponente. Proveniente del subcomponente CmdKeyboardSequencer.

**Señales de salida, (*Outputs*):**

- **seq_ack [MAX_NUM_TRACKS]**: señal activa a alta durante un ciclo indicando la confirmación de procesamiento del último comando de teclado enviado por este subcomponente. Dirigida al subcomponente ReadTrackChunk asociado al turno actual.

- **aviableCmdRqt [1]**: señal activa a alta realizando una solicitud de envío de un comando de teclado al subcomponente CmdKeyboardSequencer.

- **cmdKeyboard [14]**: señales que en su conjunto forman los bits que representan el último comando de teclado envíado a CmdKeyboardSequencer.

Este subcomponente se encarga básicamente de multiplexar los comandos de teclado provenientes en función del turno actual, estando turno implementado con el uso de un contador. A su vez en función del turno el resto de señales de la interfaz toman su correspondiente valor.

TracksCmdSequencer también usa el subcomponente reducedOr, que realiza una reducción de la operación OR dado un conjunto de señales de entrada.

En el siguiente esquemático simplificado, se pueden observar algunos módulos secuenciales básicos de los que hace uso este subcomponente:

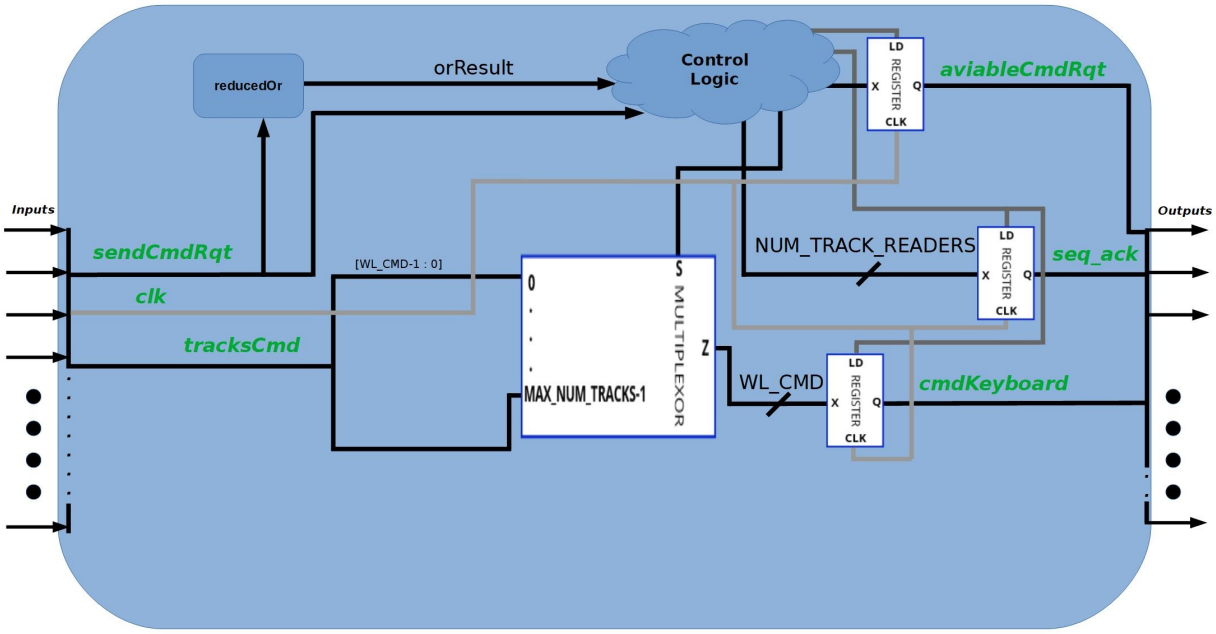


**Figura 4.23**: *Esquemático simplificado del subcomponente TracksCmdSequencer*

## 4.8. Componente MainController

Este componente es el controlador principal del sistema. Consecuentemente, orquesta al resto de componentes según las acciones que realice el usuario en la placa o en la interfaz externa.

En la interfaz de uso de este componente podemos encontrar las siguientes señales:

**Señales de entrada, (*Inputs*):**

- **sysStart [1]**: señal activa a alta durante un ciclo, indicando una solicitud de inicialización del sistema. Esta acción de inicialización es desarrollada por el componente MySetup, encargado de copiar los datos almacenados en la memoria Flash a la memoria RAM. Proveniente del componente ButtonsSyncDebRiseEdge (botón btnc en la placa).

- **playSong [1]**: señal activa a alta durante un ciclo, indicando una solicitud de reproducción de un archivo MIDI. Proveniente del componente ButtonsSyncDebRiseEdge

(botón btnu en la placa), o del componente ExternInterfaceCmdReceiver.

- **loadMidiFile [1]**: señal activa a alta durante un ciclo, indicando una solicitud de escritura en la memoria RAM de un nuevo archivo MIDI. Proveniente del componente ExternInterfaceCmdReceiver.

- **reverbOnOff [1]**: señal activa a alta durante un ciclo, indicando un cambio en el estado del modo *reverb*, si esta activo se desactiva y viceversa. Proveniente del componente ButtonsSyncDebRiseEdge (botón btnl en la placa), o del componente ExternInterfaceCmdReceiver.

- **externInterface [1]**: señal activa a alta durante un ciclo, indicando una solicitud de habilitación/deshabilitación de la conexión con la interfaz externa. Lógica equivalente a la seguida en la señal de entrada **reverbOnOff**. Proveniente del componente ButtonsSyncDebRiseEdge (botón btnd en la placa).

- **OnOffSong [1]**: señal activa a alta, indicando que el archivo MIDI almacenado en memoria esta siendo procesado por el componente MidiParser.

- **finSetup [1]**: señal activa a alta, indicando el fin del proceso de inicialización de la memoria RAM llevado a cabo por el componente MySetup.

- **finishFileReception [1]**: señal activa a alta durante un ciclo, indicando la finalización del proceso de escritura de un archivo MIDI en memoria, realizado por el componente ExternInterfaceCmdReceiver.

**Señales de salida, (*Outputs*):**

- **readMidifileRqt [1]**: señal activa a alta durante un ciclo, realizando una orden de reproducción al componente MidiParser para que este inicie la lectura del archivo MIDI almacenado en memoria.

- **reverbStatus [1]**: señal que indica el estado actual del modo *reverb*. En caso de que el modo *reverb* este activado, el valor lógico tomado por esta señal sera 1. En caso contrario el valor lógico sera 0.

- **iniSetup [1]**: señal activa a alta, realizando una solicitud de inicialización de la memoria RAM al componente MidiParser.

- **externInterfaceStatus [1]**: señal que indica la habilitación (valor lógico 1) o deshabilitación (valor lógico 0) de la conexión con la interfaz externa.

- **cenComponents [2]**: par de señales activas a baja que habilitan la lógica de procesamiento de los archivos MIDI y de los comandos de teclado por parte de los componentes MidiParser y KeyboardCntrl respectivamente.

- **muxControlSignals [1]**: señal que permite multiplexar las entradas del componente bin2segNexys4 que representan cada uno de los dígitos hexadecimales a mostrar en los *displays* de 7 segmentos de la placa. Las señales a multiplexar su salida son las que representan el último comando de teclado enviado a KeyboardCntrl, y los 32 bits menos significativos que representan un comando de escritura.

- **memRdWr [1]**: señal que permite seleccionar el modo de funcionamiento del componente RamCntrl. Consecuentemente, en caso de tener un valor lógico 0 procesara los comandos de escritura almacenados en el buffer Fifo_inCmdWriteBuffer. En caso contrario (valor lógico a 1), procesara los comandos de lectura almacenados en los buffers Fifo_inCmdReadBuffer_0 y Fifo_inCmdReadBuffer_1.

Parte de la lógica de este componente esta basada en el almacenamiento de un par de flags que representan el estado de los modos *reverb* y habilitación/deshabilitación de la conexión con la interfaz externa. A su vez, implementa una máquina algorítmica de 5 estados con sus correspondientes acciones a realizar según el estado y a su vez, implementación de salidas tipo Moore.

## 4.9. Componentes de visualización

### 4.9.1. Componente RgbLed

Este componente se encarga de iluminar adecuadamente los leds RGB que dispone la placa Nexys 4 DDR.

El led LD16 se encarga de indicar el estado de finalización de la última orden de reproducción de un archivo MIDI, así si muestra un color rojo el archivo MIDI contenido en la placa no cumple con los requisitos de formato exigidos. En caso de mostrar un color verde, el archivo MIDI es reproducible.

Por otro lado el led LD17, se encarga de indicar si el sistema no tiene cargadas las muestras en la memoria DDR en cuyo caso se iluminara de color rojo, y en caso contrario se iluminara en color verde. En caso de habilitar la interacción con la interfaz externa, la iluminación se tornara de un color morado. Durante la reproducción de un archivo MIDI, este led empieza a parpadear independientemente del color con el que se este iluminando. Por último, en caso de iniciar la subida de un archivo MIDI a la placa, el led se tornara de un color azul y comenzara a parpadear.

En la interfaz de uso de este componente se deben incluir como señales de salida las del grupo "LedsRGB" definidas en las interfaz de salida de MIDI_Soc (§4.2). También en la interfaz de uso de RgbLed podemos encontrar las siguientes señales:

**Señales de entrada, (*Inputs*):**

- **fileOk [1]**: señal activa a alta indicando si el archivo MIDI cumple los requisitos de formatos exigidos por el sistema.

- **externInterfaceStatus [1]**: señal que indica la habilitación (valor lógico 1) o deshabilitación (valor lógico 0) de la conexión con la interfaz externa.

- **playSong [1]**: señal activa a alta durante un ciclo, indicando una solicitud de reproducción de un archivo MIDI. Proveniente del componente ButtonsSyncDebRiseEdge (botón btnu en la placa), o del componente ExternInterfaceCmdReceiver.

- **mainControllerStatus [5]**: conjunto de señales que forman los bits que codifican el estado actual del componente MainController siguiendo un patrón *one hot*. El valor de estas señales esta reflejado en los leds del 0 al 4 de la placa.

Respecto a la implementación decir que principalmente hace uso de múltiples contadores para llevar a cabo tanto la funcionalidad de parpadeo, como la modulación PWM de salida. Esta modulación PWM, es realizada por 5 subcomponentes individuales, con una resolución de modulación de 8 bits.

### 4.9.2. Componente bin2segNexys4

Este componente permite la visualización de dígitos hexadecimales en el par de bloques de 4 *displays* de 7 segmentos que incluye la placa Nexys 4 DDR. Esta visualización es realizada iluminando los distintos leds que forman cada uno de los *displays* 7, a una frecuencia superior a la que el ojo humano puede percibir el parpadeo ( aprox 45Hz)[15].

En la interfaz de uso de bin2segNexys4 podemos encontrar las siguientes señales:

**Señales de entrada, (*Inputs*):**

- **Dígitos hexadecimales**. El siguiente grupo de señales tienen en común que todas representan un dígito en alguno de los dos bloques de 4 *displays* de 7 segmentos. La correlación de señales y *display* asociado se enumera a continuación:

  - **segRight_n0 [6]**: representan el dígito hexadecimal menos significativo del bloque derecho.
  - **segRight_n1 [6]**: representan el dígito hexadecimal ubicado a la izquierda del dígito menos significativo del bloque derecho.
  - **segRight_n2 [6]**: representan el dígito hexadecimal ubicado a la derecha del dígito más significativo del bloque derecho.
  - **segRight_n3 [6]**: representan el dígito hexadecimal más significativo del bloque derecho.

- **segLeft_n0 [6]**: representan el dígito hexadecimal menos significativo del bloque izquierdo.
- **segLeft_n1 [6]**: representan el dígito hexadecimal ubicado a la izquierda del dígito menos significativo del bloque izquierdo.
- **segLeft_n2 [6]**: representan el dígito hexadecimal ubicado a la derecha del dígito más significativo del bloque izquierdo.
- **segLeft_n3 [6]**: representan el dígito hexadecimal más significativo del bloque izquierdo.

Además de estas señales hay que añadir las contenidas en el grupo "Displays 7 Segmentos" descrito en la interfaz de MIDI_Soc como señales de salida (§4.2).

Las formas de onda que se pretende generar con este componente, vienen representadas en el diagrama de tiempos que se muestra al final de este párrafo. Las señales AN0, AN1, AN2 y AN3 se corresponderían a las 4 primeras líneas del conjunto de señales de salida disp_an_o, mientras que el conjunto de señales "Cathodes" se corresponde con disp_seg_o:

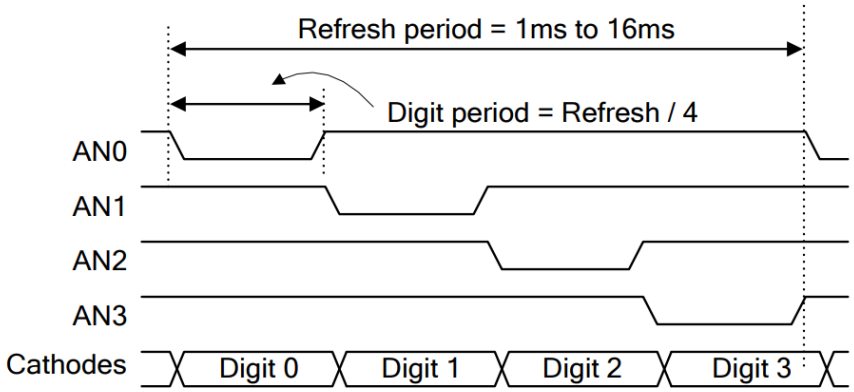


**Figura 4.24**: *Diagrama de tiempos asociado al patrón de onda a seguir para la iluminación de los display 7 segemntos de la placa Nexys 4 DDR*
[15]

## 4.10. Componentes para el procesamiento de señales externas

### 4.10.1. Componente ButtonsSyncDebRiseEdge

Este componente se encarga de procesar las señales provenientes de los botones externos de la placa con el fin de filtrar los "rebotes" producidos al pulsar y despulsar los botones. [47].

La interfaz de uso de este subcomponente esta compuesta por el grupo de señales de entrada "Botonera" presente en la interfaz de MIDI_Soc (§4.2). A dicha interfaz de uso, habrías que incluir también las siguientes señales:

**Señales de salida, (*Outputs*):**

- **Activación botones**. Grupo de señales activas a alta durante un ciclo al detectarse un flanco de subida en la señal de entrada correspondiente. La correlación con las señales de entrada se muestra a continuación:
    - **xRise_btnc [1]**: correspondiente con la señal de entrada btnc_i.
    - **xRise_btnu [1]**: correspondiente con la señal de entrada btnu_i.
    - **xRise_btnl [1]**: correspondiente con la señal de entrada btnl_i
    - **xRise_btnd [1]**: correspondiente con la señal de entrada btnd_i

Para ello este componente incluye un sincronizador de 2 etapas por cada botón con el fin de asegurar la captura de un valor lógico estable. A su vez la señal proveniente de cada uno de estos sincronizadores es filtrada por unos subcomponentes que encargan de realizar una espera de 50 ms con el fin de esperar que la señal proveniente de los botones se termine de estabilizar.

Por último a la salida de estos subcomponentes de espera, se conectan unos detectores de flanco de subida (uno por cada botón) correspondiéndose a la salida de estos a las señales de salida de este componente.

### 4.10.2. Componente resetSyncronizer

Este componente se encarga de adaptar la señal proveniente del botón reset del sistema, adaptando un sincronizador para que tenga una activación asíncrona y desactivación síncrona. De esta forma se evita la posible metaesbilidad y fallo de sincronización que puede provocar la señal de reset al desactivarse [48].

Las señales más relevantes de la interfaz de uso, se exponen a continuación: **Señales de entrada, (*Inputs*):**

- **x [1]**: señal activa a alta cuando el botón btnu de la placa es pulsado. En el diseño se inicializa a un valor lógico de 1.

**Señales de salida, (*Outputs*):**

- **xSync [1]**: salida sincronizada, correspondiente a la señal de entrada **x**.

## 4.11. Componente MySetup

Este componente se encarga de copiar todo el bloque de datos almacenado en la Flash en la memoria RAM de la placa, mediante el envío de comandos de escritura al componente (§4.9) RamCntrl.

Las señales a destacar en su interfaz de uso, se exponen a continuación:

**Señales de entrada, (*Inputs*):**

- **ini [1]**: señal activa a alta indicando el inicio de lectura de los datos almacenados en la memoria Flash de la placa, y el envío de los mismo a la memoria RAM.

- **memWrWorking [1]**: señal activa a alta indicando que el controlador de memoria esta actualmente realizando una escritura.

- **fullFifo [1]**: señal activa a alta indicando que el buffer Fifo_inCmdWriteBuffer en donde se escriben los comandos de escritura de memoria esta lleno.

**Señales de salida, (*Outputs*):**

- **fin [1]**: señal activa a alta indicando la finalización del proceso de transferencia datos entre la Flash (origen) y la memoria RAM (destino).
- **wrMemCMD [1]**: señal activa a alta ordenando la escritura de un comando en el buffer Fifo_inCmdWriteBuffer perteneciente al componente RamCntrl.
- **memCmd [42]**: conjuntos de señales que forman los bits que representan un comando de escritura de memoria.

Además de estas señales, hay que incluir la señal de entrada **io1** (§4.2), la señal de salida **cs_n** (§4.2), y la señal de entrada/salida **io0** (§4.2) estando estas tres definidas en la interfaz de uso de MIDI_Soc.

En este componente se instancia el subcomponente STARTUPE2 propiedad de Xilinx, permitiendo así la conexión con la señal sck para el uso del protocolo SPI para leer la memoria Flash. También se incluye el subcomponente fastSpiMaster_Dual sirviendo este como interfaz para el uso del protocolo SPI, siendo este el encargado de obtener los bytes de datos provenientes de la Flash. Estos bytes son capturados por una máquina algorítmica de 7 estados contenida en el componente MySetup, siendo esta la que gestiona el envío de los correspondientes comandos de escritura. La dirección de envío de estos bytes es calculada por esta misma máquina algorítmica de estados.

## 4.12. Componente ExternInterfaceCmdReceiver

Este componente se encarga de recibir los mensajes provenientes de la interfaz externa, transmitidos vía *Bluetooth*. En función de la acción ordenada por el usuario, se procede a enviar una trama de solo un byte o de 3 bytes para identificar la acción a realizar por el componente. A continuación se muestra una tabla con la correlación entre acciones y trama de bytes enviada:

| Acción | Trama |
|---|---|
| Envío de un archivo MIDI | 0x67 |
| Envío de un comando de teclado | 0x01\|02 nn 0v |
| Inicio/detención de la reproducción | 0x7e |
| Activación/desactivación efecto reverb | 0x5F |

**Tabla 4.3**: *Correlación entre acciones realizables por la interfaz y tramas de datos enviadas. Las letras n y v hacen referencia al código que representa una nota y a los 4 bits definen la intensidad de la nota respectivamente*

Exceptuando al envío de un comando de teclado, estas acciones provocan finalmente un cambio de estado en el componente MainController, siempre y cuando este habilitada la conexión con la interfaz externa. En caso que este deshabilitada la conexión con la interfaz externa, los bytes de datos recibidos se ignoran.

En el caso de ordenarse el envío de un archivo MIDI, este componente se encarga de generar y enviar los comandos de escritura (§4.9) a RamCntrl, empezando por la dirección de comienzo de la sección "Parpermoon" del bloque de datos inicialmente cargado en la RAM (§4.2). Tras recibir el primer byte que identifica esta acción, los siguiente bytes se corresponden con los datos del archivo MIDI seleccionado por el usuario.

Al un comando de teclado, este componente se limita a realiza una solicitud de envío al CmdKeyboardSequencer para que este termine enviándolo a KeyboardCntrl.

Estas acciones son llevadas cabo gracias a las señales que provee la siguiente interfaz de uso:

**Señales de entrada, (*Inputs*):**

- **externInterfaceStatus [1]**: señal que indica la habilitación (valor lógico 1) o deshabilitación (valor lógico 0) de la conexión con la interfaz externa.
- **sequencerAck [1]**: señal activa a alta indicando la confirmación de recepción del último comando de teclado enviado a CmdKeyboardSequencer.
- **btRxD [1]**: línea por la que se transmiten los bits de datos recibidos por el chip

RN42 [32] que incluye el dispositivo Pmod BT2 [20]. Esta transmisión de datos sigue el protocolo RS-232.

- **memRdWr [1]**: señal que indica si el componente RamCntrl esta en modo escritura (0 lógico) o en modo lectura (1 lógico).
- **memWrWorking [1]**: señal que se pone a alta si se está produciendo una escritura en memoria o a baja en caso contrario.

**Señales de salida, (*Outputs*):**

- **loadMidiFile [1]**: señal activa a alta durante un ciclo ordenando una transición al estado waitLoadMidiFile en el componente MainController.
- **finishFileReception [1]**: señal activa a alta indicando la finalización del proceso de carga en memoria RAM de un nuevo archivo MIDI.
- **memIsFull [1]**: señal activa a alta indicando que la memoria RAM se ha llenado por completo en la última carga de un archivo MIDI.
- **aviableCmd [1]**: señal activa a alta realizando el envío de un comando de teclado al componente CmdKeyboardSequencer.
- **keyboardCmd [15]**: conjunto de señales que forman los bits que representan un comando de teclado.
- **playSong [1]**: señal activa a alta durante un ciclo ordenando una transición al estado ReadMidiFile (en caso de que el estado actual sea FinishedSetup) o FinishedSetup (en caso de que el estado actual sea ReadMidiFile) en el componente MainController.
- **reverbOnOff [1]**: señal activa a alta durante un ciclo ordenando la activación/desactivación del modo *reverb*.

- **wrMemCMD [1]**: señal activa a alta ordenando la escritura de un comando en el buffer Fifo_inCmdWriteBuffer perteneciente al componente RamCntrl.

- **memCmd [42]**: conjuntos de señales que forman los bits que representan un comando de escritura de memoria.

Este componente contiene el subcomponente rs232receiver sirviendo este como interfaz con el protocolo rs232(§2.9) utilizado para la recepción de los bits de datos transmitidos el chip RN-42 [32].

A su vez incluye un buffer FIFO para el almacenamiento de los bytes de datos recibidos. El uso de este buffer es necesario principalmente para evitar la perdida de los comandos de teclado enviados por la interfaz externa, ya que la aplicación de una orden de *Note-off* en KeyboardCntrl requiere de tiempo para reproducir la etapa de *Release* de la nota que este siendo reproducida.

El resto de la lógica contenida en este componente consiste en una máquina algorítmica de 7 estados que gestiona las acciones a realizar según los mensajes provenientes de la interfaz externa.

## 4.13. Componente CmdKeyboardSequencer

Este componente se encarga de secuencializar los comandos de teclado provenientes de ExternInterfaceCmdReceiver y de MidiParser. Esta secuencialización es necesaria ya que el componente KeyboardCntrl solo es capaz de procesar un comando de teclado a la vez.

Las señales a destacar en su interfaz de uso son:

**Señales de entrada, (*Inputs*):**

- **cmdIn_0 [15]**: señales que en su conjunto forman los bits que representan un comando de teclado proveniente del componente ExternInterfaceCmdReceiver.

- **cmdIn_1 [15]**: señales que en su conjunto forman los bits que representan un comando de teclado proveniente del componente MidiParser.

- **sendCmdRqt [2]**: par de señales activas a alta que permiten indicar una orden de envío por parte de MidiParser (línea 1) y de ExternInterfaceCmdReceiver (línea 0).

- **keyboard_ack [1]**: señal activa a alta durante un ciclo indicando la confirmación de procesamiento del último comando de teclado enviado por este componente. Esta señal proviene del componente KeyboardCntrl.

**Señales de salida, (*Outputs*):**

- **seq_ack [2]**: par de señales activas a alta durante un ciclo indicando la confirmación de procesamiento del último comando de teclado enviado por este componente. Dirigida a MidiParser (línea 1) y a ExternInterfaceCmdReceiver (línea 0).

- **aviableCmd [1]**: señal activa a alta realizando el envío de un comando de teclado al componente KeyboardCntrl.

- **cmdKeyboard [15]**: señales que en su conjunto forman los bits que representan el último comando de teclado enviado al componente KeyboardCntrl.

Tanto la lógica interna este componente es completamente análoga a la del subcomponente TracksCmdSequencer (§4.7.7), exceptuando que la activación de la señal de salida aviableCmd solo dura un ciclo y solo tiene que secuencializar como mucho 2 comandos de teclado.

## 4.14. Componente iisInterface_75Mhz

Este componente se encarga de serializar el envío de las muestras de audio recibidas por el conjunto de señales de entrada outSample, siguiendo el protocolo I2S. Este componente se puede considerar como el componente maestro en el contexto de la comunicación, ya que se encarga de generar la correcta formas de onda de las señales MCLK, SCLK, LRCLK que requiere el protocolo. Consecuentemente el Pmod I2S2 [19] utilizado se configura en modo esclavo mediante un jumper.

La frecuencia de transmisión de muestras es aproximadamente de 48,8 KHz, coincidiendo con la frecuencia de muestreo de las notas de piano almacenadas en la memoria.

Las principales señales en su interfaz de uso son:

**Señales de entrada, (*Inputs*):**

- **outSample [24]**: conjunto de señales que forma los bits que representan una muestra a ser envíada al códec de audio CS4344 [31] incluido en el Pmod I2S2 utilizado.
- **sdto [1]**: línea de recepción de los bits que componen una muestra de audio recibida por el códec de audio CS5343/44 [42]incluido en el Pmod I2S2 utilizado. Esta línea no es utilizada en este proyecto y por ello esta conectada a tierra.

**Señales de salida, (*Outputs*):**

- **leftChannel [1]**: señal activa a alta indicando que el canal de audio actual es el izquierdo. En caso contrario el canal de audio se corresponde al derecho.
- **outSampleRqt [1]**: señal activa a alta durante un ciclo indicando la solicitud de una nueva muestra. Se activa cada vez que en leftChannel se produce un flanco de bajada o de subida.
- **inSample [24]**: conjunto de señales que forma los bits que representan una muestra recibida por el códec de audio CS5343/44 [42]incluido en el Pmod I2S2 utilizado. En el diseño no se utiliza y esta conectado directamente a tierra.
- **inSampleRdy [1]**: señal activa a alta durante un ciclo indicando la recepción de una nueva muestra. No se utiliza, conectada a tierra.
- **mclk [1]**: línea correspondiente a la señal MCLK que define el protocolo I2S.
- **sclk [1]**: línea correspondiente a la señal SCLK que define el protocolo I2S.
- **lrck [1]**: línea correspondiente a la señal LRCLK que define el protocolo I2S.

- **sdti [1]**: línea utilizada para el envío de los bits que componen la muestra de audio representada por outSample. Estos bits serán recibidos por el códec de audio CS4344 [31] .

# Capítulo 5

# App Android MySoc, interfaz de uso del piano digital

En este capítulo se comentan las principales características y modos de la app MySoc, desarrollada como interfaz externa del piano digital. La comunicación entre ambos se lleva a cabo usando la tecnología *Bluetooth*.

No se describirá en profundidad la implementación ya que es un desarrollo software complementario al objeto principal de este proyecto que es el diseño de una arquitectura *hardware*. Para conocer más detalles, tanto la apk instalable como el código fuente están disponibles en el siguiente repositorio de GitHub bajo una licencia GPLv3.0:
https://github.com/fernandoka/MySoc-Android-app

## 5.1. Funcionalidad y aspecto

En la app MySoc podemos diferenciar tres tipos de ventanas, cada una de ellas concebidas para cumplir una funcionalidad en particular. La navegación entre ventanas se realiza mediante el uso de un menú de navegación situado en la parte inferior de la pantalla del dispositivo. A continuación se procede a explicar el cometido de cada ventana, estando acompañada cada una de la captura de pantalla correspondiente.

**Ventana Connection**, esta ventana ha sido concebida para permitir al usuario esta-

blecer una conexión *Bluetooth* con el piano digital, y a su vez permite la desconexión con el mismo. Consta de dos botones, un botón es utilizado para proceder a la desconexión con el piano digital. El otro botón sirve para desplegar una lista con los dispositivos *Bluetooth* que previamente han sido emparejados. Seleccionando uno de estos dispositivos de la lista se procede a realizar la conexión.

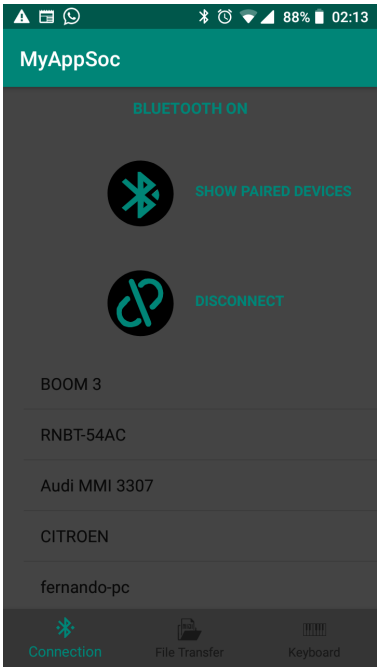


**Figura 5.1**: *Captura correspondiente a la ventana Connection, con la lista de dispositivos emparejados desplegada*

**Ventana File Transfer**, una vez establecida una conexión, esta ventana permite al usuario seleccionar y enviar un archivo MIDI al piano digital. El usuario también puede definir el número máximo de pistas que puede tener el archivo MIDI a ser enviado, con ello se asegura compatibilidad con el número de pistas máximas del piano digital (número que es parametrizable en el diseño). Estas acciones son realizadas mediante el uso de un par de botones y una entrada de texto que solo admite números, para así poder definir el máximo número de pistas. A su vez, un par de botones permiten tanto iniciar/detener la reproducción del archivo MIDI que actualmente esté almacenado en el piano digital, como

activar/desactivar el efecto *reverb*.

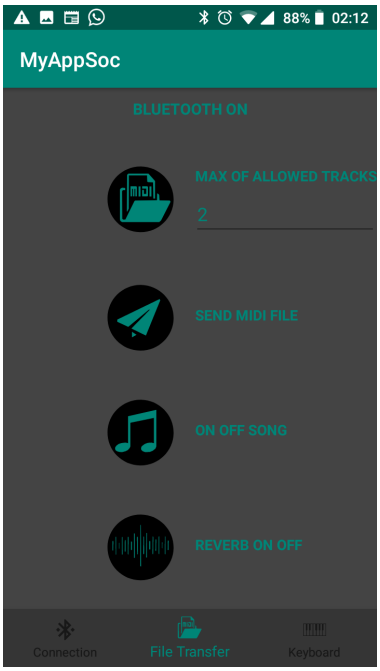


**Figura 5.2**: *Captura correspondiente a la ventana File Transfer*

**Ventana Keyboard**, esta ventana permite seleccionar una octava de piano, permitiendo al usuario tocar sus notas como si de un teclado digital se tratase. Para ello es necesario haber establecido una conexión y haber habilitado la interacción con la interfaz externa en el piano digital. También incluye un desplegable para configurar la intensidad de las notas. No ha sido posible la visualización de más de una octava debido a las dimensiones de la pantalla del dispositivo móvil.

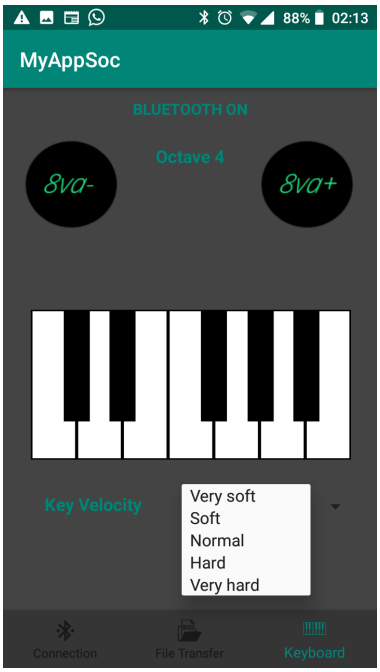


**Figura 5.3**: *Captura correspondiente a la ventana Keyboard, mostrando las distintas opciones contenidas en el desplegable*

## 5.2. Aspectos destacables de la implementación

La app MySoc ha sido desarrollada con la herramienta Android Studio, usando el lenguaje Java y siguiendo un paradigma de programación orientada a objetos. Las clases definidas han sido las siguientes:

- **Componentes de la Vista** : Conjuntos de clases encargadas de gestionar los eventos de activación provenientes de las componentes visuales. Para ello es necesario que dichas clases hereden de la clase *Fragment* para poder referenciar las componentes visuales definidas en el archivo XML correspondiente. Estas clases se encargan de gestionar la lógica relacionada con la captura de los eventos producidos por los distintos elementos visuales contenidos en cada ventana. Dichos eventos deben desencadenar acciones concretas en el contexto del proceso principal, que arranca con la clase MainActivity. Para ello es necesaria la declaración de unas interfaces definidas en las propias clases pertenecientes a la Vista, permitiendo así la comunicación con

el proceso principal mediante el uso de los métodos predefinidos en dichas interfaces. Consecuentemente MainActivity debe implementar todos los métodos predefinidos en estas interfaces.

A continuación se listan las clases pertenecientes a la Vista, cada una de ellas asociadas al corresponde archivo XML en el que se definen las componentes visuales de cada ventana. A su vez, vienen acompañadas con el nombre de la interfaz definida en cada clase para poder comunicarse con MainActivity:

- **BlConnectionFragment**: Nombre de la interfaz para comunicarse con MainActivity, *OnBlConnectionListener*.

  Nombre del archivo XML asociado *fragment_bl_connection*, correspondiente a la ventana Connection.

- **MidiInteractionFragment**: Nombre de la interfaz para comunicarse con MainActivity, *OnMidiInteractionListener*.

  Nombre del archivo XML asociado *fragment_midi_interaction*, correspondiente a la ventana File Transfer.

- **KeyboardFragment**:Nombre de la interfaz para comunicarse con MainActivity, *OnKeyboardListener*.

  Nombre del archivo XML asociado *fragment_keyboard*, correspondiente a la ventana Keyboard.

- **Componentes del Modelo-Controlador** : Estas clases se encargan de toda la lógica ajena a la captura de los eventos generados por los elementos visuales. Las clases que se incluyen son:

  - **MyConstants**: Clase estática, utilizada para albergar distintas constantes utilizadas en la clase MainActivity.

- **MidiUtils**: Clase que alberga la definición de los métodos encargados de realizar la validación de la cabecera de un archivo MIDI dado un número máximo de pistas. Estos métodos son llamados por la clase MainActivity.
- **MainActivity**: Clase que implementa todo el resto de la lógica. La conexión *Bluetooth* y el proceso de actualización del estado actual se ejecutan en 2 hilos independientes dentro del contexto del proceso principal. Al realizar la búsqueda de un archivo MIDI, esta se lleva a cabo mediante una petición a un servicio del sistema operativo.

Por último cabe destacar la implementación del método onAttachFragment definido en la clase MainActivity, el código fuente del mismo se muestra seguidamente:

```
// Register this instance like listener of OnBlConnectionListener,
// OnKeyboardListener and OnMidiInteractionListener
@Override
public  void onAttachFragment(Fragment fragment){
    if(fragment instanceof BlConnectionFragment)
        ((BlConnectionFragment) fragment).
                setOnBlConnectionListener(this);
    else if(fragment instanceof KeyboardFragment)
        ((KeyboardFragment) fragment).setOnKeyboardListener(this);
    else if(fragment instanceof MidiInteractionFragment)
        ((MidiInteractionFragment) fragment)
            .setOnMidiInteractionListener(this);
}
```

Este método se encarga de subscribir la instancia actual de la clase MainActivity en un atributo privado contenido en la instancia del parámetro *fragment*. Así al realizar las llamadas a los métodos definidos en las correspondientes interfaces, los métodos que finalmente se ejecutaran serán los que estén definidos en la instancia actual de MainActivity.

# Capítulo 6

# Conclusiones finales

Este capítulo aborda la explicación de las conclusiones finales obtenidas tras finalizar este proyecto. Se incluye un resumen de las características principales del sistema, así como consideraciones sobre el proyecto realizado y conocimientos adquiridos. Por último, se enumeran una serie de propuestas de mejora del piano digital.

## 6.1. Resultados

Recapitulando, el resultado final de este proyecto ha sido la realización con éxito de un piano digital capaz de interpretar archivos MIDI, diseñado como un sistema puramente *hardware*. La técnica de síntesis de sonido implementada en este sistema realiza una interpolación lineal utilizando dos muestras de sonido provenientes de una *Wavetable* en la que se almacena una nota real de piano (§2.3). La arquitectura diseñada destaca por su alto grado de paralelismo, su alta escalabilidad y por la aplicación de técnicas de diseño avanzadas tales como: uso de aritmética en punto fijo para reducir costes, rutas de datos segmentadas para aumentar el rendimiento, uso de caches de muestras de sonido para optimizar los accesos a las DDR2 SDRAM externa, etc. A su vez este diseño es altamente parametrizable pudiendo configurar fácilmente el número máximo de pistas MIDI compatibles, como el grado de polifonía (número de notas capaces de sonar al mismo tiempo). Este sistema ha sido implementado en una plataforma FPGA, dando lugar a un producto con las siguientes especificaciones:

| Especificaciones técnicas | |
|---|---|
| **Sistema** | Frecuencia de reloj: **75 MHz**<br>Alimentación: **5 V**<br>(*)Potencia disipada FPGA: **1,282 W**<br>(*)Potencia disipada DDR2 SDRAM: **300 mW**<br>(**)Potencia disipada SPI Flash: **despreciable**<br>Tiempo de cómputo máx. por muestra: **3072 ciclos** |
| **Audio** | Tipo de audio: **mono**<br>Frecuencia de muestreo: **48,8KHz**<br>Resolución muestras en memoria: **16 bits**<br>Resolución muestras audibles: **24 bits**<br>(*)Potencia disipada Códec: **53 mW** |
| **Interprete archivos MIDI** | Formatos archivos MIDI soportados: **0 y 1**<br>Tipo de tiempo-delta soportado: **tiempo métrico**<br>Máximo número de pistas MIDI soportadas: **6**<br>Mínimo número de pistas MIDI soportadas: **1**<br>Tamaño máximo de archivo MIDI soportado: **117 MiB** |
| **Piano Digital** | Grado de polifonía: **16**<br>Número de notas: **88**<br>Sonoridad/timbre: **Piano**<br>Efecto sonoro aplicable: ***Reverb*, 100 ms** |
| **Bluetooth** | Versión: **2.1**<br>(*)Potencia disipada: **100 mW** |

(*) Denota un valor estimado por la herramienta Vivado u obtenido de las hojas de características de los componentes. Debido al reciente confinamiento por la COVID-19, no ha sido posible realizar una medición real de los consumos por la imposibilidad de acceder al equipamiento necesario.

(**) Solo se accede a la memoria Flash durante el *setup* del sistema.

Esta implementación del piano digital utiliza aproximadamente un 22 % sobre el total de los distintos recursos interconexionables en la FPGA utilizada [16]. A continuación se adjunta una tabla con el número de los principales recursos utilizados por esta implementación en la placa Nexys 4 DDR [15]:

| Recurso | Utilización | Disponibles | Grado de utilización |
|---|---|---|---|
| LUT | 17685 | 63400 | 27.89 % |
| LUTRAM | 432 | 19000 | 2.27 % |
| FF | 18640 | 126800 | 14.70 % |
| BRAM | 8 | 135 | 5.93 % |
| DSP | 94 | 240 | 39.17 % |
| IO | 102 | 210 | 48.57 % |
| BUFG | 3 | 32 | 9.38 % |
| MMCM | 2 | 6 | 33.33 % |
| PLL | 1 | 6 | 16.67 % |

**Tabla 6.1**: *Tabla representando el grado de utilización de los distintos recursos de la FPGA C7A100T-1CSG324C*

[16]

En caso de quererse implementar una versión incrementando el número de pistas MIDI compatibles y/o el grado de polifonía habría que tener en cuenta los recursos de la plataforma objetivo ya que es posible que esta carezca de los necesarios, siendo los DSP y los IO los recursos más solicitados. Este diseño también requiere de una plataforma *hardware* que disponga tanto de una memoria RAM como de una memoria Flash de mínimo 12MiB, dejando aproximadamente 1 MiB libre para el almacenamiento de archivos MIDI.

Paralelamente este proyecto también incluye la app Android MySoc, dotando de una interfaz externa al piano digital con el fin de facilitar la interacción del usuario. Esta app mediante el establecimiento de una conexión *Bluetooth* permite al usuario subir un archivo MIDI a la memoria RAM de la placa, iniciar/detener la reproducción de un archivo MIDI, y activar/desactivar el efecto *reverb*. A su vez permite tocar en tiempo real las notas de una octava concreta del piano, que sonaran tanto si se esta reproduciendo un archivo MIDI o no.

## 6.2. Consideraciones finales y conocimientos adquiridos

Este proyecto ha demostrado la viabilidad de concebir un interprete MIDI como un sistema puramente *hardware*. Al compararlo con una implementación *System On Chip* que

haga uso de un microcontrolador, logra una mayor eficiencia energética sacrificando en escalabilidad del sistema y en amortización de los recursos *hardware*. En definitiva una implementación con microcontrolador, estaría ejecutando un *software* que realizase tanto la interpretación de un archivo MIDI como la gestión del resto de componentes *hardware* del sistema. Cada tipo de implementación presenta sus pros y contras, dependiendo del uso de cada cual principalmente de los objetivos a realizar. Este proyecto se planteó como un sistema enteramente *hardware* con el fin de tener un mayor control en el proceso de desarrollo evitando así el uso de complejas herramientas de terceros para utilizar microcontroladores, centrándose enteramente en el diseño de una buena arquitectura haciendo más meritorio este proyecto.

El mecanismo de síntesis por *Wavetables* utilizado en este proyecto ha resultado ser muy adecuado, obteniendo un sonido de bastante claridad y que sin lugar a dudas conserva gran parte de la sonoridad característica de un piano. La relación coste/fidelidad de esta técnica es muy buena ya que permite ahorrar casi el doble de memoria que la utilizada para el almacenamiento de las *Wavetables*, pudiendo generar 58 notas adicionales a partir de las 30 almacenadas obtenido finalmente las 88 notas del piano. En este proyecto esto ha supuesto un ahorro aproximado de 21MiB. Sin embargo la principal limitación de este mecanismo, reside en la necesidad de tener suficiente memoria para poder almacenar las *Wavetables* de todas las notas base. Este necesidad viene impuesta debido al error cometido al generar una nota a una distancia superior a un tono a partir de la nota base (la almacenada en la *Wavetables* utilizada), resultando en una muy mala sonoridad. Por ello es necesario almacenar como mínimo 4 notas por octava de piano, para así poder generar el resto de notas que forman una octava.

La interpretación de archivos MIDI llevada a cabo en el piano digital, es limitada ya que no se ha logrado capturar de forma generalizada la gestión del tempo. Esto provoca que al

reproducir algunos archivo MIDI estos tengan una desincronización rítmica entre las pistas que están siendo reproducidas.

Respecto a los conocimientos obtenidos, este proyecto me ha ayudado enormemente a asentar las bases de diseño *hardware* aprendidas a lo largo de la carrera logrando entender mucho mejor como es posible que los distintos componentes que forman cualquier sistema electrónico/digital, pueden llegar a comunicarse correctamente entre sí. Gracias a este proyecto he aprendido el funcionamiento de los protocolos SPI e I2S, ampliamente utilizados en los sistemas actuales.

Al mismo tiempo he logrado entender mejor las ondas de sonido y los cambios que sufren al aplicar sencillas operaciones aritméticas a las muestras que en su conjunto componen dichas ondas. Toda la teoría de ondas de sonido explicada en este documento era desconocida por mi parte al principio del desarrollo de este proyecto.

Este proyecto también me ha brindado la oportunidad de aprender sobre el estándar MIDI y la lectura de ficheros.

A causa de que el desarrollo *hardware* siempre se ha caracterizado por presentar una mayor complejidad si lo comparamos con el desarrollo *software*, me ha llevado a tener que profundizar bastante en el aprendizaje sobre herramientas EDA y HDL [49]. Conocimientos como el uso de componentes para *debuggear* (uso del componente ILA [50]) y así poder saber las formas de onda que están siendo generadas realmente por las distintas señales ha sido indispensable. También ha sido de obligado uso y adquisición de conocimientos medios para la realización de simulaciones de distintos componentes (§4). Sin estas simulaciones hubiese sido prácticamente imposible la realización de gran parte de los módulos presentes en el diseño final (componente MidiParser y subcomponentes contenidos (§4.7)). Mis conocimientos sobre el lenguaje VHDL también han sido ampliados, gracias a la comprensión de nueva sintaxis en la que cabe destacar *attribute*. Gracias a esta etiqueta, he podido personalizar el

proceso de síntesis de los diseños evitando las optimizaciones que se realizan en esta etapa. Esto ha supuesto un enorme ahorro de tiempo al generar el gran número de archivos *bitstream* para poder configurar la FPGA y consecuentemente probar las distintas versiones del diseño. A su vez, esto ha facilitado la identificación de las señales a *debuggear* simplificando el esquemático obtenido tras el proceso de síntesis. Tampoco ha sido un proceso trivial la adquisición de conocimientos relacionados con la aritmética en punto fijo, siendo una aritmética utilizada principalmente en sistemas *hardware* y en la optimización de aplicaciones.

También este proyecto me ha servido como una buena introducción a la programación de dispositivos móviles Android, gracias al desarrollo de la app MySoc. Gracias a esta app he asentado más mis conocimientos en el lenguaje Java y he podido aplicar satisfactoriamente los conocimientos aprendidos en las asignaturas de programación impartidas en mi grado.

Sin lugar a dudas todos estos conocimientos me serán de gran utilidad tanto en mi futuro laboral como académico.

## 6.3. Trabajo futuro

A continuación enumero algunas propuestas de mejoras a realizar sobre este proyecto:

- **Mejorar la calidad de sonido**. La calidad de sonido se puede mejorar con relativa facilidad, solo es necesaria más memoria no volátil, y modificar ligeramente el componente iisInterface_75Mhz y el subcomponente UniversalNoteGen. Al mejorar la calidad de sonido, primaría el aumento en la frecuencia de muestreo frente a la resolución por muestra, intentando así optimizar la memoria. Esta decisión se debe a el hecho de que al aumentar la frecuencia de muestreo esta a la vez aumentando la granularidad al realizar la síntesis de sonido, lo que conllevara en una reducción del error cometido al interpolar (§2.1).

  En caso de querer reproducir sonido estéreo, seria necesario cambiar el comportamiento de los subcomponentes UniversalNoteGen, y del subcomponente Reverb.

- **Añadir nuevos instrumento y/o mejorar el ajuste de intensidad de nota**. La inclusión de nuevos sonidos reproducibles a parte del piano es relativamente fácil, bastaría con tener almacenadas en memoria externa las *Wavetables* de las notas de los instrumentos que queramos añadir. Por otro lado para realizar un mejor ajuste en la intensidad de la nota, lo conveniente sería tener almacenada las *Wavetables* de una misma nota a distintas intensidades (fuerza al pulsar una nota).

  En ambos habría que mappear dichas *Wavetables* en memoria haciendo llegar las correspondientes direcciones a los subcomponentes UniversalNoteGen, los cuales habría que modificar en caso de incluir un nuevo ajuste de intensidad.

- **Mejorar el efecto sustain de las notas**. Esta es quizás la mejora más complicada. Como se ha comentado en el capítulo 2 (§2.4), el efecto sustain permite repetir indefinidamente la etapa *Sustain* de una nota mediante la repetición de las muestras definidas en un intervalo concreto, resultando así en un bucle. Para que el sonido generado carezca de *glitches*, es necesario que tanto la muestra de inicio y final del intervalo se correspondan a la muestra que inicia un período de la onda de sonido.

  El resultado obtenido en este proyecto no carece de *glitches* sonoros ya que no se ha realizado un estudio en profundidad de las ondas de sonido cargadas en la placa, para poder así identificar exactamente las muestras correspondientes al inicio de un período. Este es el estudio que habría que realizar para poder determinar el intervalo de muestras a reproducir en bucle sin la aparición de *glitches* sonoros.

- **Mejorar de la gestión del tempo**. La interpretación del archivo MIDI llevada a cabo en el piano digital, no carece de limitaciones. Dependiendo del archivo MIDI, este puede no ser interpretado con fidelidad debido a un desfase rítmico durante la reproducción del mismo. Esto es sin lugar a dudas debido a una falta de profundización sobre la gestión de los mensajes MIDI de establecimiento de tempo, siendo necesario un mayor estudio sobre los mismos. A su vez se recomienda la búsqueda de software

libre en el que se implementen algoritmos y reglas para la lectura de archivos MIDI.

Una vez teniendo clara una posible mejora, sería necesario retocar principalmente el componente MidiParser (§4.7) y el subcomponente ReadTrackChunk (§4.7.6).

- **Añadir el reconocimiento y aplicación del efecto *dumper* de piano**. Esta mejora consistiría en ser capaz de reconocer los mensajes MIDI de activación y desactivación del pedal derecho de un piano (*dumper On/Off*). La función de este pedal consiste en dejar sonando las notas cuando estas no están siendo tocadas, permitiendo así que los sonidos se mezclen. Las notas que no estén siendo pulsadas dejaran de sonar cuando se reciba un mensajes de desactivación de este comportamiento.

  Este añadido se caracteriza principalmente por requerir mejorar e incluso tener que cambiar la arquitectura interna del componente KeyboardCntrl ya que es necesario tener un control muy preciso de que notas están siendo pulsadas y cuales no, postergando la orden de desactivación de notas en caso de que el *dumper* este activado. Una posible primera aproximación, haría uso de un buffer FIFO que almacenase los comandos de teclado de desactivación de notas mientras el efecto *dumper* este activo. Posteriormente con su desactivación, se aplicarían los comandos almacenados en dicho buffer de uno en uno. Por otro lado habría también que modificar ligeramente el subcomponente ReadHeaderChunk para identificar estos mensajes MIDI de activación/desactivación de *dumper*.

# Bibliografía


[1] Marío Luiz Rodríguez Cobos. *Humanizar La Tierra*. León Alado Ediciones, 2013.

[2] History of roland digital pianos. https://www.roland.com/es-es/promos/piano_design_awards/history/.

[3] Historia del negocio de los instrumentos musicales de casio. https://www.casio-intl.com/latin/es/emi/history/.

[4] V3 sound sonority xxl, thoman website. https://www.thomannmusic.com/v3_sound_sonority_xxl.htm.

[5] Miditech pianobox mini, thoman website. https://www.thomann.de/es/miditech_pianobox_mini.htm.

[6] Polyphonic_fpga_synthesizer. https://github.com/ComicAddict/Polyphonic_FPGA_Synthesizer.

[7] Digital vhdl piano. https://devpost.com/software/digital-piano#updates.

[8] A-simple-electronic-piano-on-fpga. https://github.com/eltsai/A-Simple-Electronic-Piano-on-FPGA.

[9] Digital keyboard. https://github.com/killalea/digital-keyboard.

[10] Javier Cañada Toledo. Diseño e implementación de un sintetizador modular de audio, June 2017. http://oa.upm.es/47603/1/TFG_JAVIER_GONZALO_CANADA_TOLEDO.pdf.

[11] Guillem Ibañez Masanés. Piano digital midi a s/pdif, 2018 - 2019. https://upcommons.upc.edu/handle/2117/173692.

[12] Sistema de notación musical anglosajón. https://composicionmusical.es/sistema-de-notacion-musical-anglosajon/.

[13] Kiran Bondalapati and Viktor K. Prasanna. Reconfigurable computing: Architectures, models and algorithms. *Proceedings of the IEEE*, April 2000. https://www.researchgate.net/publication/298250159_Reconfigurable_computing_Architectures_models_and_algorithms.

[14] Maurice O'Connor. Presentation on theme: "week #12 programmable logic technologies". https://slideplayer.com/slide/12360110/.

[15] Digilent. *Nexys4 DDR$^{TM}$ FPGA Board Reference Manual*, April 2016. https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-4-ddr/nexys4ddr_rm.pdf.

[16] 7 series fpgas data sheet: Overview, August 2017. https://www.datasheets360.com/pdf/8955149337532931898.

[17] Samsung Electronics. *DDR2 SDRAM Device Operating & Timing Diagram*. https://www.samsung.com/semiconductor/global.semi/file/resource/2017/11/ddr2_device_operation_timing_diagram_may_07-0.pdf.

[18] Cypress. *S25FL127S 128-Mb (16 MB) 3.0 V SPI Flash Memory*, April 2019. https://www.cypress.com/file/177961/download.

[19] Digilent. *Pmod I2S2 Reference Manual*. https://reference.digilentinc.com/reference/pmod/pmodi2s2/reference-manual.

[20] Digilent. *Pmod BT2 Reference Manual*. https://reference.digilentinc.com/reference/pmod/pmodbt2/reference-manual.

[21] The MIDI Manufacturers Association. *The Complete MIDI 1.0 Detailed Specification*, 2014.

[22] Logotipo midi. https://www.doctorproaudio.com/content.php?44-midi-sonorizacion-sonido.

[23] Wave pcm soundfile format. https://web.archive.org/web/20040317073101/http://ccrma-www.stanford.edu/courses/422/projects/WaveFormat/.

[24] Sophocles J. Orfanidis. *Introduction to Signal Processing*. Prentice Hall, August 1995. https://www.ece.rutgers.edu/~orfanidi/intro2sp/.

[25] Freepats acoustic grand piano. http://freepats.zenvoid.org/Piano/acoustic-grand-piano.html.

[26] Interpolación lineal. https://es.wikipedia.org/wiki/Interpolaci%C3%B3n_lineal.

[27] Espinosa-Durán John M. Velasco-Medina Jaime Liévano-Torres, Pedro P. Implementación de algoritmos para efectos de audio digital con alta fidelidad usando hardware programable. Technical report, Pontificia Universidad Javeriana Colombia, enero-junio 2013. https://www.redalyc.org/articulo.oa?id=47728625006.

[28] Randy Yates. Fixed-point arithmetic: An introduction, August 2007. https://courses.cs.washington.edu/courses/cse467/08au/labs/l5/fp.pdf.

[29] José Manuel Mendías Cuadros. Modelos de programación de software empotrado. http://www.fdi.ucm.es/profesor/mendias/PSyD/docs/PSyDtema6.pdf.

[30] Pmod i2s2 reference manual. https://reference.digilentinc.com/reference/pmod/pmodi2s2/reference-manual.

[31] Cirrus Logic. *CS4344/5/8 10-Pin, 24-Bit, 192 kHz Stereo D/A Converter*, July 2013. https://statics.cirrus.com/pubs/proDatasheet/CS4344-45-48_F2.pdf.

[32] Roving Networks. *Bluetooth Data Module Command Reference & Advanced Information User's Guide*, March 2013. http://ww1.microchip.com/downloads/en/DeviceDoc/bluetooth_cr_UG-v1.0r.pdf.

[33] José Manuel Mendías Cuadros. Fsm con ruta de datos comunicación con un terminal a través de un bus rs-232. http://www.fdi.ucm.es/profesor/mendias/DAS/docs/DASlab5.pdf.

[34] Dora María Ballesteros. *Modulación PWM en FPGA basado en máquinzas de estado finito*. Scientia Et Technica, June 2008. https://www.redalyc.org/articulo.oa?id=84903874.

[35] Enrique Vicente Bonet Esteban. Lenguaje c. https://informatica.uv.es/estguia/ATD/apuntes/laboratorio/Lenguaje-C.pdf.

[36] Sublime text 3 official website. https://www.sublimetext.com/.

[37] Logotipo vivado. https://canada.newark.com/xilinx/ef-vivado-design-nl/vivado-hl-design-suite-designnl/dp/33AC4348.

[38] Juan González. Introducción al lenguaje de descripción hardware vhdl. http://www.iearobotics.com/tmp/seminarios-vhdl.pdf.

[39] Marcos Sánchez-Élez. Introducción a la promgramación en vhdl. https://eprints.ucm.es/26200/1/intro_VHDL.pdf.

[40] Logotipo android. https://asesoriasredesycomputadores.com/servicio-tecnico-de-computadores-bogota/android-logo/.

[41] Historia del lenguaje java. http://www.manualweb.net/java/historia-java/.

[42] Cirrus Logic. *CS5343/4 98 dB, 96 kHz, Multi-Bit Audio A/D Converter*, March 2015. https://statics.cirrus.com/pubs/proDatasheet/CS5343-44_F5.pdf.

[43] Xilinx intellectual property home page. https://www.xilinx.com/products/intellectual-property.html.

[44] Xilinx. *Mixed-Mode Clock Manager(MMCM) Module (v1.00a)*, June 2009. https://www.xilinx.com/support/documentation/ip_documentation/mmcm_module.pdf.

[45] Xilinx. *7 Series FPGAs Memory Interface Solutions, User Guide*, January 2012. https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v1_4/ug586_7Series_MIS.pdf.

[46] Xilinx. *XADC Wizard v3.0, LogiCORE IP Product Guide*, April 2015. https://www.xilinx.com/support/documentation/ip_documentation/xadc_wiz/v3_0/pg091-xadc-wiz.pdf.

[47] José Manuel Mendías Cuadros. Lógica secuencial, lectura de señales asíncronas y módulos genéricos. http://www.fdi.ucm.es/profesor/mendias/DAS/docs/DASlab2.pdf.

[48] José Manuel Mendías Cuadros. Máquinas de estados finitos, acondicionamiento de las señales de reloj y reset. http://www.fdi.ucm.es/profesor/mendias/DAS/docs/DASlab3.pdf.

[49] Pong P.Chu. *RTL Hardware Design Using VHDL Coding for Efficiency, Portability and Scalability.* John Wiler & Sons, Inc, Hoboken, New Jersey, 2006.

[50] Xilinx. *Integrated Logic Analyzer v6.2*, October 2016. https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf.

# Apéndice A

# Introduction

## A.1. Motivation

Music has always come along with humans since the beginning of mankind's history, allowing their intellectual and spiritual development. Due to itself, mankind it had been and will be, able to express and share undying emotions through generations building a cultural skyscraper which continue growing up until the present days. Music, like everything in life, is in constant evolution, the last changes it had suffered mainly involves the technological development of these latest years, the digitalization. The digitalization has caused the democratization of music in all Its levels. Because of it, the way the music instruments are made has changed a lot, allowing the reduction in occupied space, and in manufacturing costs, the piano could be a good example.

Since I have started learning piano, I felt curious about how it is possible for the hardware systems, included in digital pianos, to be able to play a song with a completely natural sound of an acoustic piano. Doing some research I discovered the MIDI standard, a standard that is used for almost all the digital pianos of today's world. Without thinking twice, at my third university year I decided my final university project will be the creation of a digital piano design that can understand MIDI files.

## A.2. Objetives

The purpose of this project is the creation of a base open-design to develop a polyphonic digital keyboard that can play MIDI files. Anyone can use and improve this design in the future. The main goal is to learn about MIDI standard, wave sounds, and music.

Due to the enormous amount of different actions which are represented in MIDI files, the process of understanding them is not trivial. This project aims to be a solid base for future improvements in MIDI file compatibility. At the same time, this project intends to be the first approach to Wavetable sound synthesis which is commonly used in current digital pianos.

The hardware design is developed to be implemented in an FPGA platform, allowing the component reconfiguration of the system. This feature will simplify the development of future versions of the design. This design follows a modular philosophy and could be configured to adjust the polyphony degree and the maximum number of MIDI tracks that the system can play.

## A.3. Background

In relation to the commercial world, in the early's years of the 70's decade, the electronic piano technology had finally be established, this type of instrument is natural predecessors of modern digital pianos. The EP-10 model of the Japanese company Roland is a good example of this fact.

**Figura A.1**: *Roland EP-10, first electronic piano commercialized by Roland* [2]

In the next decade with the recent definition of MIDI standard, it had started to appear the first digital pianos. Thanks to MIDI standard instruments from different companies are able to understand each other, allowing to the musicians to expand their creativity. In this generation we can include the models Roland HP-300 and HP-400, and the Casio CPS-201

**Figura A.2**: *First digital piano from Casio, model CPS-201*
[3]

**Figura A.3**: *First design of Roland digital pianos, models HP-300 y HP-400*
[2]

Over time, the number of different brands that have taken part in the digital piano market, some of them focusing on the creation of models with a good price-quality relation

and others tries to offer the best qualities in their products. Between these brands, we can find Yamaha, which is famous thanks to the Calvinova line products. Also, Kawai is another example of a pioneering company in the development of digital pianos.

On the other hand, nowadays it is possible to find systems that try to mix the functionality of a rhythm machine with the sound generation, these systems are named "GM sound modules" (GM from "General MIDI"). The behavior of these modules is based on the reception of MIDI messages and the corresponding sound note generation. Furthermore, the module can play a huge variety of sound instruments at the user's choice. The MIDI messages can reach the module using a USB or MIDI connection, regardless of the instrument interface that is being used. The main advantage of these products is their portability, allowing to anybody enjoy high-quality sounds in any place.

Below this line, you will find a couple of pictures with some examples of these products:

**Figura A.4**: *Modelo V3 Sound Sonority XXL de calidad profesional* [4]

**Figura A.5**: *Modelo Miditech Pianobox Mini, gama de entrada* [5]

In relation to the academic world, almost all projects we can find focus only on music notes synthesis, which is implemented using a software or hardware platform. You can check some examples of them in the following lines:

- Polyphonic_FPGA_Synthesizer [6], this project implements a polyphony keyboard (4 voices) in an FPGA. Using PWM modulation this project generates different notes. The sound synthesis is quite simple compared to the sound generation of this project.

- Digital VHDL Piano [7], this project generates the notes using a clock divider and a self-build amplifier circuit. Both the clock divider and the user interface are implemented in an FPGA.

- A-Simple-Electronic-Piano-on-FPGA [8], this project implements a piano keyboard using a PS2 keyboard as a user interface.

- Digital Keyboard [9], this project involves a python application, showing to the user a GUI interface to play the notes. This project follows universal software design principles.

However, I was able to find two projects with similar characteristics.

One of them belongs to the *Universidad Politécnica de Madrid* named as "Diseño e implementación de un sintetizador digital modular de audio"[10]. In this project, an FPGA

board is connected to a PC and then, the FPGA board can understand the MIDI messages which have been sent by the PC to generate different notes. The sound synthesis used in this project performs sample reading which is stored in the FPGA (no use of RAM or Flash memory). These samples represent different waveforms such as square, triangular, ...

The title of the second project is "Piano Digital" [11] and belongs to the *Universitat Politécnica de Catalunya*. This project involves the implementation process of a hardware system using an FPGA platform which can understand MIDI messages generated by a MIDI keyboard. With the reception of these messages, the FPGA starts to send sound samples using the S/PDIF protocol. The sound samples of this project are stored in the FPGA without using of RAM or Flash memory, and represent a pre-calculated waveform.

## A.4. Work schedule

In order to achieve successful completion of this project, all the work have been organized as an order group of stages, which each one of them has their own goals and deadlines.

1. **Study and comprehension of MIDI standard** (july of 2019 to agust of 2019). Develop a console application that can extract MIDI messages.

2. **Familiarize with Xilinx Vivado 2018.2 tool** (septiember of 2019 to october of 2019). To accomplish it, a design of previous years had been migrated from a Xilinx ISE environment implemented for an FPGA Spartan 3. Thanks to this exercise, I was able to develop the I2S interface component used to send sound samples, and the Ram2Ddr used to get efficiently a bytes stream from the external RAM memory.

3. **Elaboration of a interpolation formula to use in Wavetable synthesis** (october of 2019 to november of 2019). it has been required the development of a console

application in order to check the viability of the interpolation formula and make some corrections in the sound synthesis process.

4. **FPGA implementation of the digital piano, following a modular design** (december of 2019 to january of 2020). The implementation process starts by creating the module which represents a piano keyboard in the system. The most important moments during this process happened in the Christmas holidays when I was trying to create a real-time sound synthesis. Afterward, I decided how the parallel architecture of the system will be, using memory commands to manage the read/write requests.

5. **FPGA implementation of the MIDI file interpreter, following a modular design,** (january of 2020 to february of 2020). The MIDI file interpreter module, which it had been developed with help of simulations, was finally added to the digital piano. At the same time, the correct behavior of the parallel architecture was tested.

6. **Android app development** (march of 2020 to april of 2020). This application implements an external interface simplifying the user interaction with the digital piano. To be able to communicate the app and the digital piano between them, a Bluetooth connection is established.

7. **Improvements and final revison of the Android app and the hardware design in order to achive a good interaction between them** (april of 2020 to may of 2020). This stage focuses in clean up all the source code, fixing bugs, and the parametrization of the design.

8. **Elaboration of the project documentation** (may of 2020 to june of 2020).

## A.5. Document organization

The next lines provide a brief description of the chapters contained in this document:

1. **Introduction**: In this chapter, it is commented on the main motivations and objectives. It also includes a brief description of previous academic projects and the work schedule followed in the development of this project.

2. **Technological context of the project**: This chapter involves the explanation of different tools and technologies used in this project. The explanation about how it is made the sound synthesis (§2.3) is included in this chapter as well.

3. **Console applications**: This chapter explains some implementation details about the console applications Midi Parser and Note Conversors. Also, it contains the corresponding conclusion of each program.

4. **Hardware arquitecture of the digital piano, MIDI_Soc project** : This chapter includes an explanation of the different components and sub-components which form the whole design of the digital piano.

5. **App Android MySoc, user interface for the digital piano**: This chapter explains both a few implementation details and how to use the app.

6. **Final conclusiones** : This last chapter gives some conclusions such as the grade of resources utilization in the FPGA implementing the final design with a polyphonic degree of 16 notes and a 6 MIDI tracks compatibility. The rest sections involve the different knowledge gained in this project and some ideas for future works to improve the design.

# Apéndice B

# Final Conclusions

This chapter contains the conclusions after finishing this project. This includes both a summary of the main characteristics of the system and some considerations about this project followed by the acquired knowledge. Lastly, some future work suggestions figure at the end of this chapter.

## B.1. Results

To summarise, the final result has been the successful realization of a digital piano that can play MIDI files designed as a complete hardware system. The sound synthesis used in this project is based on linear interpolation using two sound samples which are contained in a Wavetable, which represents a real piano note (§2.3). The designed architecture stands out for his highly parallel degree, his high scalability, and for the use of advanced hardware design methods such as fixpoint arithmetic to reduce cost, segemented data paths to improve the performance, memory cache to optimize the access to the extern DDR2 SDRAM, etc.

Furthermore, this design is also high parametrizable allowing to easily configure both, the maximum number of MIDI supported tracks and the polyphony degree. This system it has been implemented in an FPGA platform with the following specifications:

| Technical specifications | |
|---|---|
| **System** | Frequency: **75MHz**<br>Supply: **5V**<br>(*)Dissipated power FPGA: **1,282 W**<br>(*)Dissipated power DDR2 SDRAM: **300 mW**<br>(**)Dissipated power SPI Flash: **negligible**<br>Máx. compute time limit per sample: **3072 cycles** |
| **Audio** | Audio type: **mono**<br>Sampling rate: **48,8KHz**<br>Memory samples resolution: **16 bits**<br>Audible samples resolution : **24 bits**<br>(*)Dissipated power of Codec: **53 mW** |
| **MIDI file interpreter** | MIDI file formats supported: **0 y 1**<br>Delta-time type supported: **metrical time**<br>Maximum number of supported MIDI tracks: **6**<br>Minimum number of supported MIDI tracks: **1**<br>Max MIDI file size supported : **117MiB** |
| **Digital piano** | Polyphony degree: **16**<br>Number of notes: **88**<br>Sonority/timbre: **Piano**<br>Sound effect: ***Reverb*, 100ms** |
| **Bluetooth** | Version: **2.1**<br>(*)Dissipated power: **100 mW** |

The (*) symbol refers to a value obtained by estimations that the Vivado tool had made. Due to the recent COVID-19 lockdown, it hasn't been possible to do any real power consumption analysis because the lack of the necessay equipment.

(**) The access to Flash memory only occur during the system setup.

This implementation of the digital piano approximately uses a 22 % from all the FPGA resources [16]. The next table shows the amount of the different resources used by this implementation in the Nexys 4 DDR [15] board:

| Resource | Utilization | Available | Utilization degree |
|---|---|---|---|
| LUT | 17685 | 63400 | 27.89 % |
| LUTRAM | 432 | 19000 | 2.27 % |
| FF | 18640 | 126800 | 14.70 % |
| BRAM | 8 | 135 | 5.93 % |
| DSP | 94 | 240 | 39.17 % |
| IO | 102 | 210 | 48.57 % |
| BUFG | 3 | 32 | 9.38 % |
| MMCM | 2 | 6 | 33.33 % |
| PLL | 1 | 6 | 16.67 % |

**Tabla B.1**: *Table which represents the different utilization degrees of the resources in the FPGA C7A100T-1CSG324C*

[16]

In case you want to increase the maximum number of supported MIDI tracks and/or the polyphony degree, you must be aware of the hardware platform resources. The most used resources in this design are the DSP and the IO. This design also requires a hardware platform with at least 12MiB of RAM and Flash memory, which approximately 1MiB of it is used to store the MIDI files.

At the same time, this project includes the development of an Android app named MySoc. This app is used as an external interface, simplifying the user interaction with the digital piano. Using a Bluetooth connection this app allows the user to upload a MIDI file to the RAM memory of the board, start/stop playing a MIDI file, and start/stop the reverb effect. Furthermore, this app also allows the user to play in real-time the notes of a certain piano octave, which will sound even if the system is playing a MIDI file.

## B.2. Final considerations and acquired knowledge

This project proof the viability of implementing a MIDI file player as a complete hardware system. This complete hardware system in comparison with a System On Chip implementation which uses a microcontroller, obtains a better energy efficiency giving up in

system scalability and hardware amortization. These latest facts are due to a microcontroller implementation that would cede the MIDI file interpretation and the management of the other hardware components to a software entity (executed program in the microcontroller). Each type of implementation has its pros and cons, ultimately the use of one or another mainly depends on the objectives that you want to achieve. This project has been developed as a complete hardware system to avoid the waste of time in learning the use of complex tools, to be able to use a microcontroller. Thanks to this decision, all the efforts have been focused on the development of good architecture, increasing the value of this project.

The Wavetable synthesis mechanism used in this project has resulted in very convenient and thanks to it, the generated sound has great quality and without any doubt, the sonority corresponds to a piano. The cost/quality relation of this mechanism is really good because it saves almost the double memory space used to store the Wavetables, making possible the generation of 58 notes from 30 stored notes and then, resulting in the 88 piano notes. In this project, it has been possible to approximately save 21MiB of memory space. However, the main limitation of this sound synthesis mechanism is related to the memory space which must be enough to save all the Wavetable of the base notes. This limitation is caused due to the committed error produced in the generation of a note located at a whole step from the base note (which is stored in the Wavetable), resulting in a quite bad sonority. Because of it, is necessary to store a minimum of 4 notes per piano octave, to generate the rests of notes contained one octave.

The MIDI interpretation process carried out by the digital piano, it is limited because it hasn't been possible to generalize the tempo management. Consequently, some MIDI files present a rhythm desynchronization while there are played.

In relation to the acquired knowledge, this project has greatly helped me in my un-

derstanding of hardware design, allowing me to understand much better how the different components which make any electronic/digital system, can correctly communicate between them. Also, thanks to this project I have learned the behavior of the SPI and I2S protocols which are commonly used in the nowadays systems.

At the same time, I have understood much better the behavior of wave sounds and the main changes that it suffers from the application of simple arithmetic operations on the samples which globally represent the wave sound. All the wave theory included in this document was firstly unknown by me at the beginning of this project.

This project also gives me the opportunity to learn about the MIDI standard and the read file process.

Since always the hardware development has been more complex than software development, which led me to learn more about the EDA tools and HDL [49]. Knowledge such as the proper use of debug components (use of the ILA component [50]) to be able to check the real waveform of different signals of the system, it has been something essential. Furthermore, the realization of component simulations has also been essential (§4). Without it, it would be almost impossible to design a huge group of modules, like MidiParser and all the contained subcomponents (§4.7). I also acquired more knowledge about the VHDL language, thanks to the comprehension of new syntax, including the *attribute* word. Thanks to this kind of label, I was able to customize the synthesis process of my designs, avoiding optimizations during this workflow stage. This has supposed an enormous time-saving in the generation of all the *bitstream* during the development process. Simultaneously, this fact has simplified the recognition of the signals to debug. Besides, it hasn't been trivial the acquisition of knowledge related to fixpoint arithmetic, which is mainly used in hardware systems and software optimization.

This project also served me as a good introduction to mobile device programming, thanks

to the development of the app MySoc. Thanks to it, I have settled my knowledge about Java language, object-oriented programing, and the "Model View Controller" paradigm.

Without any doubt, all this knowledge will be really useful in my labor and academic future.

## B.3. Future work

The following lines describe some suggestions for future improvements in this project:

- **Improve sound quality**. The quality of the sound could be easily improved, it is only needed more Flash memory and make some minors changes in iisInterface_75Mhz component and UniversalNoteGen subcomponent. In order to optimize the memory space, I would prioritize the increment in the sampling rate instead of the sample resolution. This decision is supported by the fact that incrementing the sampling rate is going to also increase the granularity in the sound synthesis process, which will induce a minor error in the application of the interpolation formula (§2.1).

  In case to generate stereo sound, it would be necessary to make some change in UniversalNoteGen subcomponent and Reverb subcomponent.

- **Add more sound instruments and/or improve note intensity**. The inclusion of more instrument sounds is relatively simple and will depend again on the use of more Flash memory. This memory space will be used to store new notes Wavetables of the corresponding instruments we want to add. To improve the note intensity, it is also needed to store a new Wavetable per note supported intensity of the system. Each one of these Wavetables will be used as a real approximation when a softer or strong sound is generated by the instrument

  In both cases, it will be necessary to a correct mapping of the news Wavetables to be able to be indexed by the UniversalNoteGen subcomponents.

- **Improve the sustain effect of the notes**. This might be the most difficult improvement to achieve. As chapter 2 explains (§2.4), this effect allows us to indefinitely repeat the Release phase of a note by the generation of a loop that always reads the same interval of sound samples. To generate the sound without the emergence of glitches, the first and the last sample of the interval must correspond with the firsts samples of a period in the wave sound.

  This project doesn't lack this kind of sound glitches due to it hasn't been done yet a deep study of the wave sounds stored on the board. This is something that has to be done to identify the proper start period samples and consequently the definition of all the sustain intervals.

- **Improve the tempo managment**. The MIDI interpretation which takes part in the digital piano has some limitations. Depending on the MIDI file, this could be not interpreted with absolute fidelity because a rhythm lag occurs while the MIDI file is played. This happens due to a lack of study in the Set tempo MIDI message, something that must be done to fix this problem. Besides, it is recommended to search for some free software that implements the algorithm and rules to follow to completely understand the Set tempo message.

  Knowing how to proceed, the changes to be done in the design will be in the MidiParser component and ReadTrackChunk subcomponent.

- **Add the recognition and application of the dumper effect fo the piano**. This improvement consists of being able to recognize the MIDI messages of activation and deactivation of the right pedal of a piano (dumper On/Off). When this pedal is "On", allow the notes to keep generating sound even if the note is not pressed. Thanks to it all the sounds generated by the different notes could be mix together creating new complex sounds. The notes which are not being pressed will mute when a "turn Off" pedal order is received.

This improvement probably will suppose an architecture change of KeyboardCntrl, because it is needed to carefully manage the “press” and “depress” actions which have to be simulated by the behavior of this component. One first implementation approach could be the use of a FIFO buffer to saves the turn Off note keyboard commands while the dumper effect is active. With the deactivation of this effect, the commands saved in the before-mentioned buffer will be processed one by one. At the same, some minors changes have to be done in ReadTrackChunk to recognize the new dumper MIDI messages.