POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico di Torino

Master's Degree Thesis

FPGA-based implementation of audio effects for ultralow-latency Networked Music Performance applications

Supervisors

Prof. Cristina ROTTONDI

Ph.D. Riccardo PELOSO

Candidate

Diego BERT

April 2023

Abstract

A Networked Music Performance (NMP) is a real-time musical interaction that aims at allowing musicians to play together from remote locations. One of the greatest challenges of networked performances is keeping latency, i.e. the mouth-to-ear delay between users, as low as possible. In recent years, several solutions, mainly software ones, have been developed to ensure a satisfactory communication. A very promising approach to further reduce latency is the implementation of a dedicated audio processor. Indeed, this solution permits improvements in terms of speed, while keeping some degree of flexibility through an implementation based on a Field-Programmable Gate Array (FPGA). Among the many architectures available, one of the most suitable for an audio processor with these characteristics is the Transport Triggered Architecture (TTA). This processor design is a one-instruction set architecture that is based on different function units connected by buses. The only available operation is the move one, which allows the execution of more complex instructions by moving data from one unit to another.
This thesis proposes a hardware implementation of new function units to perform two of the most widely used functionalities in the music environment: audio mixing and reverb effect. Specifically, the former is the process of combining different audio sources to generate a single output sound. This feature is crucial in NMPs because it allows multiple input channels and different users to be heard together in real-time. The audio mixing implementation is described starting from its algorithm, going through a first firmware implementation and achieving a dedicated hardware design by exploiting a toolset called TTA-based Co-design Environment (TCE). The second functionality added to the system is the reverb effect, an audio unit to simulate the acoustic phenomenon of reverberation. Since the goal of NMP applications is also to provide natural listening conditions to musicians, adding reverberation artificially can give the sense of being part of a shared acoustic space. The algorithm chosen for the reverb implementation is the one theorized by Manfred Robert Schroeder. Regarding its development, a different approach is considered: MATLAB HDL Coder is employed to generate a hardware implementation starting from a Simulink model. Finally, the entire audio processor is implemented on FPGA exploiting Xilinx Vivado, a synthesis tool for hardware design.

Acknowledgements

Dopo averla completata, mi sono reso conto di come questa tesi non sia altro che l'ennesimo cerchio che si chiude. Diversi anni fa, infatti, quando decisi di iscrivermi ad ingegneria elettronica, ero guidato dal desiderio di capire come fosse possibile realizzare dispositivi in grado di acquisire e riprodurre il suono. Oggi, grazie a questo progetto, sento di aver avuto l'occasione di poter concludere il mio percorso rispondendo ad alcune delle domande che mi ero sempre posto in passato. Ovviamente, tutto questo è stato possibile anche grazie all'aiuto di numerose persone, che voglio ringraziare.

Per prima cosa, desidero ringraziare la Prof. Cristina Rottondi e il Ph.D. Riccardo Peloso, per avermi coinvolto in questo progetto ricco di sfide. Inoltre, un ringraziamento lo devo anche a Matteo Sacchetto e Leonardo Severi, per avermi supportato durante tutta la mia permanenza in dipartimento.

Desidero poi ringraziare tutte le persone che ho conosciuto al Politecnico in questi anni. In particolar modo, voglio ringraziare Paul, Bex, Fili e Vlad. Siete stati i migliori alleati che potessi trovare.

Ringrazio i miei amici, gli Svincolati: Wong, Publio, Synth, Rugio, Gabbo e Briga. Ogni bevuta fatta insieme in questi anni mi ha permesso di scrivere circa un paragrafo della tesi. Fate voi il conto. Ringrazio inoltre Federico, per essere stato sempre il metro di paragone, la persona da battere.

Un ringraziamento enorme va poi ai miei genitori, che hanno sempre appoggiato le mie scelte, e a mio fratello e mia sorella, in grado di strapparmi un sorriso anche nei momenti più difficili.

Infine, desidero ringraziare con tutto me stesso Federica. Sei il motivo principale per cui credo così tanto in me stesso e per cui non ho mai mollato. Per questo motivo, questa tesi è dedicata a te.

Table of Contents

List of Tables

List of Figures

Acronyms

ASIP Application-Specific Instruction set Processor

DSP Digital Signal Processing
DUT Design Under Test

FPGA Field-Programmable Gate Array
FSM Finite-State Machine
FU Function Unit

HDL Hardware Description Language

I2S Inter-IC Sound
IP Intellectual Property

LUT Lookup Table

MIDI Musical Instrument Digital Interface

NMP Networked Music Performance

OSAL Operation Set Abstraction Layer

PLC Packet Loss Concealment
PLL Phase-Locked Loop

SPI Serial Peripheral Interface

TCE TTA-based Co-design Environment
TCP Transmission Control Protocol
TTA Transport Triggered Architecture

UART Universal Asynchronous Receiver-Transmitter
UDP User Datagram Protocol

Chapter 1

Introduction

For thousands of years, music has been one of the greatest ways to link people to each other, and even today it plays an extremely important role within society in creating communities. Its unique ability to bring people together, overcoming language barriers and cultural differences, is one of the main reasons why it is so loved. Thanks to the technological progresses which have been made over the last decades, several other methods of connecting people have emerged. One of them in particular, called Networked Music Performance (NMP), is becoming of interest nowadays and is succeeding in uniting people once more through music.

A Networked Music Performance is a musical interaction in which musicians can play together from remote locations. The technology that supports this kind of interaction aims at connecting performers by exploiting a telecommunication network to stream audio signals, so to link them even if they are not physically in the same room. The main purpose of NMP is to provide a real-time scenario, in which musicians can feel like they are playing close to each other. As said, over the last decades, the interest in remote performances has notably increased thanks to the progress in the telecommunication field and in particular, in the last few years, as a consequence of the COVID-19 pandemic.

The importance of this topic has increased so much, that many researchers have started to study both technological and social aspects related to NMP. According to them, the introduction of remote performances in society can bring a lot of benefits to different kinds of users, such as expert musicians, artists, music teachers and students [1]. Among the benefits for professional musicians, there is for sure the possibility to save the huge amount of time they spent commuting from one place to another. It is well known, indeed, that a consistent part of a performer's life is made up of travels. Networked performances may then introduce also new ways of thinking about music and change considerably the creative process of artists.

The opportunity to get in touch with other musicians all over the world can be a boost for creativity and new inspiration. Furthermore, NMP can become also an interesting approach to music study, providing teachers and students with an environment in which they can practice from everywhere.

Like other ground-breaking technologies, also NMP is a very challenging field and has many different issues to overcome. First of all, the necessity of maintaining a stable real-time communication between two or more users, which is one of the most problematic but crucial features to achieve. Indeed, as confirmed by a large number of studies, very strict latency requirements need to be met to ensure a successful musical interaction. The time delay recognized to be optimal for NMP is approximately around 20-30 ms [2]. Achieving this kind of latency is not an easy task, as there are usually several bottlenecks, starting with the telecommunication network. Indeed, one may wonder why not use modern videoconferencing software. These tools are acceptable for speech, but are not fast enough for musical interactions, since the mouth-to-ear delays they introduce are in the order of hundreds of ms. For this reason, dedicated solutions need to be developed to solve the problem.

The solution considered in this thesis consists of a dedicated hardware system designed to deal with extremely low latency. As it will be explained in the next chapters, this solution is mainly based on a Field-Programmable Gate Array (FPGA), used to implement a custom audio processor. Compared to currently available software, this approach can bring several benefits, that will be analyzed in detail later.

1.1 Thesis objectives

This master's thesis investigates a possible hardware implementation for two of the most widely used functionalities in the music environment: audio mixing and reverb effect. Specifically, these would be included as Function Units (FUs) in a custom audio processor to support NMP applications and, for this reason, their design needs to take in consideration several aspects. First, they should introduce the lowest possible latency. By doing so, the further time delay added can be considered a negligible contribution. Moreover, since their physical implementation is then performed on an FPGA, also resource utilization becomes a major concern.

In addition to this main goal, the thesis has also some other purposes. First, to give a greater contextualization, it aims to describe in detail the overall system in which both the new units and the audio processor are placed. Then, its intent is also to present some possible optimizations to the considered FPGA-based solution, so that better performances can be achieved.

1.2 Thesis overview

Following this introduction, the remainder of the thesis is organized into other 6 chapters:

- Chapter 2, which describes more in detail the NMP field, by giving an overview of latency studies, the history of networked performances, currently available solutions and open challenges.
- Chapter 3, which offers a full description of the proposed FPGA-based approach, examining the different components of the system and the interfaces between them.
- Chapter 4, which breakdowns the workflow adopted for the development of the mixing function unit on the audio processor, starting from a firmware solution and achieving a dedicated hardware structure.
- Chapter 5, which explains the reverb feature design by exploiting MATLAB HDL Coder and a Simulink modelization of Schroeder's reverberator.
- Chapter 6, which analyzes the optimizations performed on the audio processor in terms of latency and resource utilization, taking then a look at the final outcomes.
- Chapter 7, which summarizes the thesis work and shows future perspectives.

Chapter 2

Networked Music Performance

This chapter gives an overview of NMP's state of the art. First, it focuses on one of the main actors which influence performances: latency. Then, it describes some of the most important currently available solutions for networked performances, introducing also some meaningful milestones in NMP's history. Finally, future and still open challenges on the topic are presented.

2.1 Latency perception

One of the major concerns when talking about NMP is latency, which can be defined as the time delay that occurs between two musicians when playing together. Indeed, every kind of musical performance is influenced by this delay and by the way humans perceive it. In the last decades, as the interest in NMP started to increase, also research and studies on latency started to rise. In particular, the most common topics were latency perception in different environmental conditions and strategies to cope with it. The main target of these studies was usually the understanding of which are the parameters that influence latency perception and which is a suitable latency range to support real-time performances.

Different experiments on latency perception can be found in literature. Researchers not only focus their attention on changing environmental and acoustical conditions, but they also concentrate their efforts on proving the effect of different instruments, musical pieces and tempos. However, most of the experiments follow the same setup for testing purposes. Usually, musicians are placed in two different anechoic rooms, so that they can't see each other. They can listen to what they are playing through some kind of communication system that has a negligible time delay. Latency

is then introduced in their live interaction artificially to understand its impact. Another kind of setup used in some experiments also includes video transmission of each musician to figure out possible differences in perception.

Experiments showed that in general an audio latency of 20-30 ms [2] can be considered as a suitable range to have an acceptable performance. However, it seems that different studies have their own opinion on acceptable latency ranges. Indeed, as observed in [3], it is difficult to determine a common latency threshold between humans. This is not only because several parameters need to be taken into account during a performance, but also because subjectivity plays a crucial role. This study, in which musicians were asked to perform different rhythmical patterns, demonstrates how they perceive latency in quite different ways.

Another research that deserves to be taken into consideration is the one realized at the Norwegian University of Science and Technology [4]. In this set of experiments, participants were both musicians and non-musicians. This was done to understand the differences between people with some musical training and those who have not. They were asked again to perform some rhythmical patterns and the results seem to confirm that being used to music practice has an impact on performances.

As it can be noticed, latency perception in humans is a quite complex field of study, because it is influenced by many different conditions and also by subjectivity. This is the reason why the goal of an NMP environment is to reduce as much as possible the delay introduced, to achieve performative conditions similar to those of musicians playing together in the same physical space. For a more detailed overview of the most important studies on latency, the reader can refer to [2].

2.2 History

Networked Music Performances have their roots long before the spread of currently available telecommunication systems and networks. Indeed, remote music collaborations and the possibility to play from remote locations are concepts that have been explored many years ago.

In such context, John Cage's "*Imaginary Landscape No.4 for twelve radios*" is accredited to be one of the first NMP experiments [5]. Executed for the first time in 1951, this composition was written to be performed by 12 interconnected radio transistors. Cage's will to investigate new ways of creating music was evident but, of course, strongly limited by the technological achievements of that time.

Another interesting experiment, which sounds visionary for that period, was held in 1957 by Paul Robeson [6]. The artist performed a concert for people sitting in the

St Pancras Town Hall in London, while being in a recording studio in New York. This was possible thanks to the first transatlantic telephone cable system, called TAT-1. Even if it was a one-way-only musical interaction, Robeson's performance can be considered a milestone in music technology since it was one of the first times that a concert was offered to a remote audience.

One of the first real improvements in creating remote interconnections between musicians was brought by the spread of early personal computers in the 1970s. In particular, it was provided by a band called The League of Automatic Music Composers, which had its origin in the San Francisco Bay Area [7]. This group of innovators used early networked computers to exchange data and influence each other on what they were playing. After their first attempts, the same members of The League of Automatic Music Composers were involved in a new project, which was named The Hub [7]. From 1987, when they became able to play between different buildings in New York, they started to increase the distance among players. One of their most famous experiments was related to a live performance between the Mills College, California Institute for the Arts and the Arizona State University. Unfortunately, they recognized that remote performances were heavily limited by communication networks, so their experiment was a failure.

The greatest advancement in the creation of a system for networked performances was possible at the beginning of 2000. Thanks to the latest improvements in the computer networks field, the first bidirectional communications at acceptably low latency were possible. This was mainly due to the development of Internet2, a network which provides bandwidth-intensive services for the research and education community in the United States. The first solution which exploited this kind of infrastructure was developed at Standford University's CCRMA (Center for Computer Research in Music and Acoustics) and was led by Chris Chafe. This project was named SoundWIRE [8] and had as a main goal the design of a set of tools for high-quality uncompressed bidirectional audio communications in real-time.

Another important proposal that was made possible by Internet2 is the Distributed Immersive Performance (DIP) project [9]. Its objective was to develop a technology for real-time and multi-site live musical interactions. The implementation of the system was supported by a series of different experiments.

2.3 Current approaches

During the last two decades, the interest in NMP has increased a lot, as witnessed by a large number of solutions that have been proposed. This interest has increased so much that also some companies started to create their own product for remote

musical interactions. Many different approaches to NMP are available, each one with its characteristics such as the use of a software or a hardware solution, compressed or uncompressed audio streaming, digital audio processing algorithms and also video support.

One of the first solutions, developed in the early 2000s by Chris Chafe and Juan-Pablo Cáceres, is Jacktrip [10]. This is an open-source software designed after the previous SoundWIRE project at Standford University's CCRMA. Jacktrip works on most general-purpose OSs and supports a number of audio channels limited only by the network bandwidth. The main approach employed for the real-time interaction is to send uncompressed audio through Internet2. To support the project, a collaboration between Standford University's CCRMA and Silicon Valley software entrepreneurs was established in 2020, leading to the birth of the Jacktrip Foundation [11]. Thanks to this, a complete NMP environment called Jacktrip Visual Studio was built. Moreover, they developed also the JackTrip Analog Bridge, a hardware solution that acts as a sound card to further reduce latency with respect to currently available commercial solutions. According to them, the time delay introduced by this device is about 1 ms.

One more solution inspired by the SoundWIRE project is SoundJack [12], a software application developed by Alexander Carôt at the International School of New Media in Lübeck in 2006. The followed approach is similar Jacktrip's one, but as a difference, both a compressed audio format (Opus audio coded) or uncompressed sound can be used. Furthermore, to simplify user accessibility, also a web version of the application called SJ-lite was developed [13].

Another software approach that can be used on most general-purpose OS is Jamulus [14]. Also his software was released in 2006 and since then it has increased its popularity among musicians, creating quite a large user base over the years. It is an open-source software licensed under the GNU General Public License (GPL) and has a friendly and intuitive user interface. Some key features of Jamulus include the use of an Opus compressed audio format and the possibility to add reverb effect to audio channels. Moreover, Jamulus offers also the opportunity to set up audio and network parameters for more expert users to achieve better performances.

A research project that is worth mentioning is DIAMOUSES [15]. Also this software is an open-source one, but it works only on Linux OS. Its main characteristics are that it uses uncompressed audio signals and that can be used to stream also video. DIAMOUSES's main purpose is to create an environment in which different application scenarios are present to support user experience, such as rehearsals, teaching lessons or collaborative sessions for jamming.

One last open-source project is SonoBus [16]. This software was released in 2020, after the boom of users in NMP usage. It includes almost all the features of other solutions already described, but it has two main adds-on: it can use both uncompressed or Opus compressed audio and includes many different effects for audio processing such as compression, noise gate, equalizers and reverb.

Among the commercial solutions, one of the first products that can be considered is JamKazam [17], a proprietary software that again works on most general-purpose OSs. Among its features, it can be recalled the possibility to stream video, record sessions and play pre-recorded tracks in the background.

A different approach is the one adopted by the Elk Audio company [18]. The main difference, with respect to the ones described as far, is that they developed a complete setup for real-time remote performances. In particular, they designed a hardware audio interface called Elk Bridge that can be used together with a proprietary software called Elk LIVE. To have greater control over hardware, they implemented also Elk Audio OS, which is a Linux-based OS optimized for low-latency audio applications. Thanks to their dedicated audio interface and OS, they are able to achieve very promising results in terms of latency. According to them, in suitable conditions, their setup has "about 15-20 ms of latency over 1000 km (621 miles) on a fiber connection".

This section tried to describe some of the possible implementations of NMP environments. Of course, many other solutions are currently available, as said mainly software ones. For a more schematic and detailed comparison of the best existing solutions, the reader can refer to [19]. In the next chapter, a different kind of approach, based on an FPGA implementation of an Application-Specific Instruction set Processor (ASIP) for NMP applications, will be investigated.

2.4 Open challenges

As already said, NMP is a very challenging topic, since it involves many different expertises. For this reason, even if a lot of progress has been done in the last two decades, several open challenges persist.

Since latency is one of the main issues to consider and network delays are usually the bottleneck of every NMP application, some possible future improvements should address them. Indeed, nowadays NMP traffic is treated the same as other kinds of services. This is of course a problem for real-time interactions because they have to share the same telecommunication infrastructure of other content. To increase performances in this way, commercial agreements for a reserved bandwidth could be taken into account [1].

Another issue to deal with is the portability and ease of use of NMP environments. In fact, all the currently available solutions require an adequate network connection, which is not always possible to meet. This is the case for example of wireless connections [1]. In addition to the portability of NMP, also its ease of use is an important point. Indeed, most of the currently available solutions are not enough intuitive for common users and require some minimum knowledge of audio devices and their parameters.

Finally, one of the most challenging topics to deal with is Packet Loss Concealment (PLC), which is a method used to cope with packet losses on networks. Indeed, in telecommunication networks it is quite common for a data packet to be corrupted in some way or arrive too late at the receiver side. In most of the cases, thanks to the Transmission Control Protocol (TCP), this is avoided by retransmitting lost/corrupted packets, thus ensuring a reliable delivery. Unfortunately, since NMP requires stringent delay guarantees, TCP can't be used and User Datagram Protocol (UDP) is used instead. UDP is a lighter transmission protocol but does not ensure reliable transfer, so it requires some kind of algorithm to deal with packet losses. For this reason, PLC techniques can have an important impact on UDP transmissions and as a consequence on NMP quality. Moreover, most of the current solutions also transmit uncompressed audio to save time, which does not leverage intrinsic PLC methods implemented by traditional audio codecs. For all these reasons, PLC techniques need to be implemented in an NMP environment and several different solutions have already been developed. Current studies are also trying to understand a possible approach based on deep learning [20]. One of the greatest challenges, in this regard, is the possibility to use machine learning models to predict the next packets before their actual arrival.

Chapter 3

System description

This chapter analyzes the preliminary version of the system developed for NMP applications, which will be then modified in the next chapters. First, it gives an overview of its components and the interfaces between them. Then, each element of the system is explained, focusing in particular on the FPGA-based audio processor, which is the main candidate for the improvements treated in this thesis. Finally, the software development tools used are also introduced.

3.1 System overview

As mentioned in section 2.3, there are several ways to develop an NMP environment. However, since latency requirements are very severe, a purely software-based approach is not always the best option. An alternative solution, described also in [2], is the use of a system based on an FPGA, where audio packets are completely handled by a dedicated hardware. In this way, the many sources of additional latency that can exist in a general-purpose architecture, such as the kernel intervention, are not present.

The system considered in this thesis is a hybrid solution between a software approach and the mentioned FPGA-based one. In particular, as shown in figure 3.1, it can be divided into three main components, each of them implemented on a different hardware support:

- Audio board
- Audio processor
- Raspberry Pi

The former is a custom hardware support that acts as a sound card for the considered system. Its main purpose is to acquire and play sound samples by exploiting several audio interfaces. The audio board is also the component that is responsible for interfacing directly with users, by making them control, for example, the volume of each input audio channel. The second element of the system, the audio processor, is an ASIP implemented on an FPGA development board. In this preliminary version of the system, it is simply used to handle the communication between the audio board and the Raspberry Pi device. However, as introduced in the next chapters, the audio processor can be also used to perform digital signal processing operations on the acquired audio samples. Then, the last component of the system is the Raspberry Pi. This device is in charge of packetizing and depacketizing audio samples and of managing the network communication through an Ethernet port. In this preliminary version of the system, it is also used to mix incoming audio data streams.

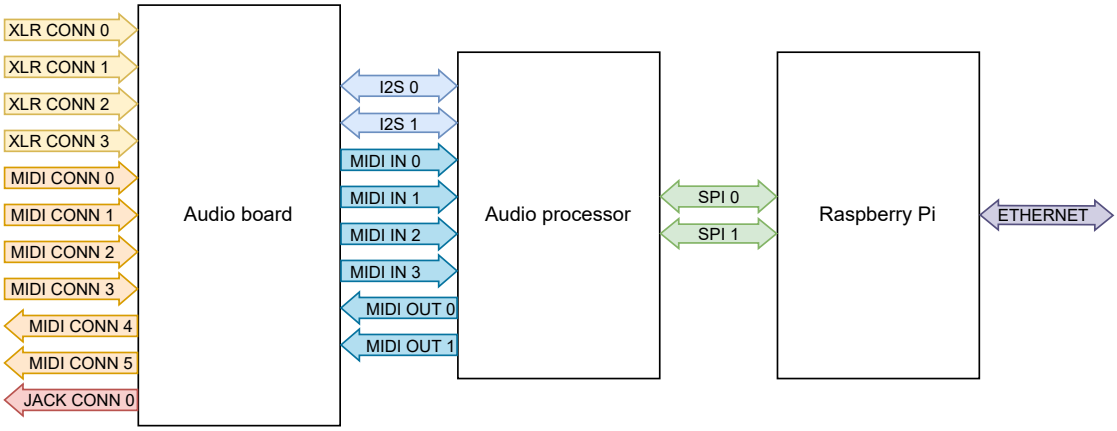

Figure 3.1: System block diagram

With respect to the solutions analyzed in section 2.3, the proposed system can bring some advantages to an NMP application. One of the most important is definitely the low latency introduced compared to purely software approaches. Indeed, the system is mainly composed of dedicated hardware, being the Raspberry Pi the only general-purpose device. With this kind of solution, the latency introduced by commonly available sound cards, due to a consistent buffer size, is not present. Moreover, as mentioned before, no OS runs on the ASIP, reducing further the time delay introduced. Of course, this solution comes also with some drawbacks, such as a reduced flexibility compared to pure software solutions.

3.2 Communication protocols

As it can be seen from figure 3.1, the three main components of the system exploit different communication protocols to interface with each other:

- I2S protocol, between audio board and audio processor
- MIDI protocol, between audio board and audio processor
- SPI protocol, between audio processor and Raspberry Pi

3.2.1 I2S protocol

The Inter-IC Sound (I2S) [21] is a synchronous serial communication protocol designed by Philips Semiconductors (nowadays known as NXP Semiconductors), introduced in 1986 to deal with the lack of a standard interface for digital audio systems.

I2S communications are based on a 3-line serial bus composed by:

- Continuous Serial Clock (SCK) or Bit Clock (BCLK): determines the bits transmission rate
- Word Select (WS) or Left-Right Clock (LRCLK): indicates the channel being transmitted (typically WS=0 means left channel and WS=1 means right one)
- Serial Data (SD): is the serial audio data line on which data are usually transmitted in an MSB-first two's complement representation

The protocol is based on a master/slave communication and permits the presence of multiple transmitters and receivers. The master device needs to drive the SCK and WS lines, while the SD one can be driven also by slaves. In particular, if the master is a transmitter, it will drive also the SD line, while if it is a receiver, SD will be driven by a slave. In the considered system, the audio processor acts as master, while the audio board is a slave. Specifically, to work properly with the I2S interface on the audio board, the processor needs to drive the SCK and WS lines (also referred to as BCLK and LRCLK), by respecting the following timing:

$$\begin{aligned} f_{bclk} &= 64 \cdot f_s \\ f_{lrclk} &= f_s \end{aligned} \tag{3.1}$$

where f_s is the sampling frequency of the system, equal to 44.1 kHz.

I2S bus specification also supports different transmission formats. In the case of the considered system, the one shown in figure 3.2 is adopted. This configuration represents audio data on 24 bits with a left-justified, MSB-first format (LRCLK=1 means left channel and LRCLK=0 means right one). This representation was chosen since both Analog-to-Digital Converter (ADC) and Digital-to-Analog Converter (DAC) on the audio board have an embedded I2S interface with these characteristics (as explained later in section 3.3).

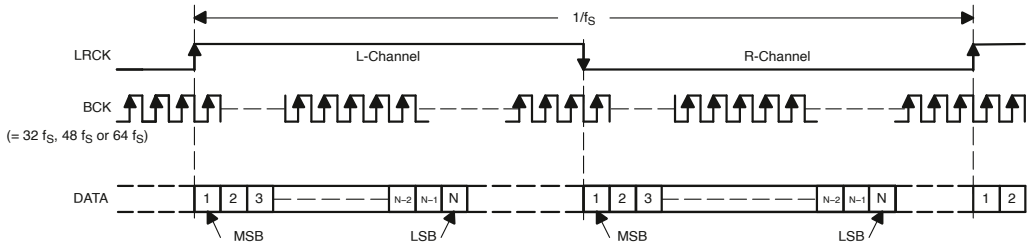

Figure 3.2: I2S left-justified transmission format (24-bit, MSB-first) [22]

3.2.2 MIDI protocol

Musical Instrument Digital Interface (MIDI) [23] is a technical standard among modern digital audio devices introduced by Dave Smith and Chet Wood in 1981. In particular, this term describes not only a communication protocol but also physical connectors designed to achieve a standardized interface between different electronic musical instruments. Unlike I2S, MIDI is not used to transfer audio samples but messages among devices to exchange control information.

MIDI messages are composed by words of 8 bits, that can represent both a status or data information. The first bit of each word is used to identify the kind of message (0 for data and 1 for status).

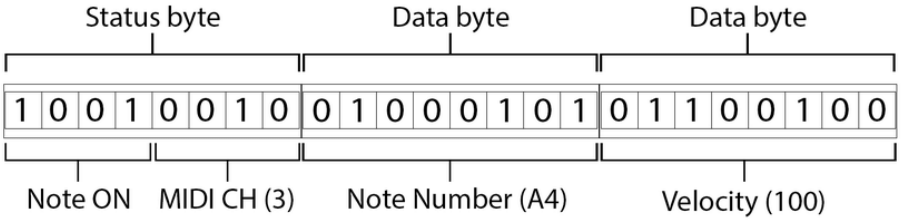

Figure 3.3: Example of MIDI messages [24]

The communication protocol is UART-like: a start (logic 0) and stop (logic 1) bit are used to define a frame and data are transmitted with a baud rate of 31.25 kbit/s.

As said, the MIDI standard defines also electrical interfaces. For this purpose, originally, 5-pin DIN connectors were used, carrying information on two pins and omitting the use of the other three to achieve better electrical isolation. However, nowadays MIDI messages can also be exchanged on other types of connectors, such as USB and Ethernet ones.

3.2.3 SPI protocol

The Serial Peripheral Interface (SPI) [25] is a full duplex synchronous serial communication protocol introduced in the 1980s by Motorola. It was mainly intended for short-distance communication between integrated circuits on the same printed board and over the years has become a de facto standard.

The SPI interface is composed of 4 lines:

- Serial Clock (SCLK): the serial synchronous clock
- Master Out Slave In (MOSI): the master outputs data line
- Master In Slave Out (MISO): the slave outputs data line
- Chip Select (CS): the line that indicates if a slave is selected (often active low). If there is more than one slave, multiple CS lines are present.

Also this protocol is based on a master/slave configuration, where usually a single master manages one or more slave devices. SCLK, CS and MOSI lines are driven by the master, while the MISO one is driven by the slave. Usually, the SCLK frequency is determined by the master according to which frequency range slaves can support. In the case of the considered system, the Raspberry Pi is the master device, while the audio processor is the slave.

The Serial Peripheral Interface protocol supports different modes of operation. In particular, it can be configured in four distinct ways according to the value of two parameters: CPOL, the polarity of the clock during the idle state, and CPHA, which determines if the rising or falling clock edge is used to sample the data. Usually, how the bytes exchanged need to be interpreted is defined and specified for each slave device. In the case of the considered system, the two parameters are both set to 0, meaning that the clock idle state is low and data are sampled on the rising edge (while they are shifted on the falling one). This mode of operation, also known as mode 0, is shown in figure 3.4.

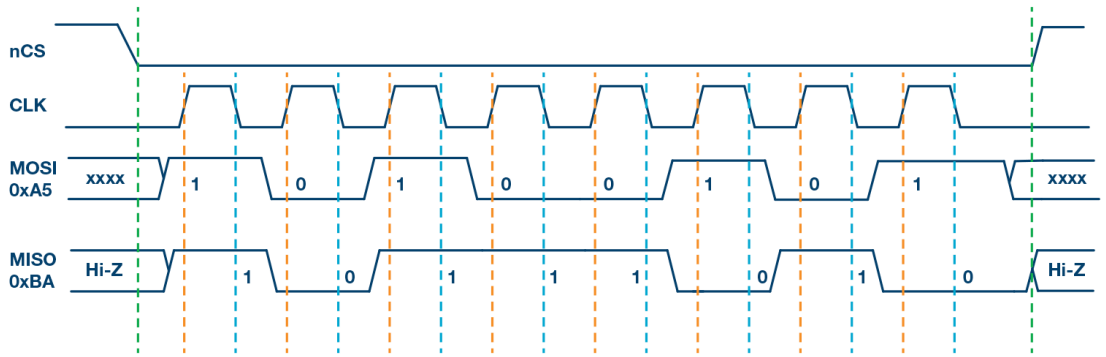

Figure 3.4: SPI transmission mode 0 (CPOL=0, CPHA=0) [26]

3.3 Audio board

The first component of the considered system is the audio board [27], a hardware circuit conceived as a custom sound card that can be used to meet extremely low latency requirements. A picture of it can be seen in figure 3.5.

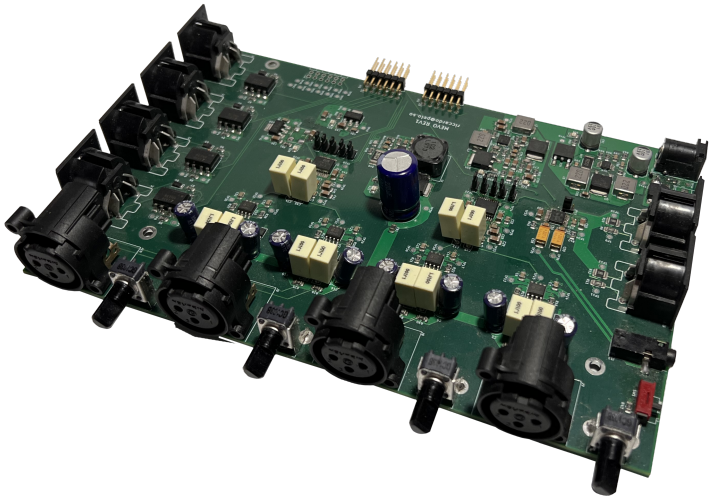

Figure 3.5: Custom audio board

As already mentioned, the main role of this component is to acquire and play sound signals by means of different connectors and electrical interfaces. For this reason, the audio board is equipped with:

- four XLR connector inputs, to acquire analog audio signals
- one headphone output (3.5 mm jack), to play analog audio signals (stereo output)

- four 5-pin DIN connector inputs, to receive MIDI messages
- two 5-pin DIN connector outputs, to transmit MIDI messages

To effectively carry out its tasks, i.e, converting analog signals to digital ones and vice versa, the audio board includes two ADCs and one DAC. Both of these devices are designed for audio purposes and are equipped with I2S modules, configurable as masters or slaves depending on the application. Since in the considered system the master device is the audio processor, these interfaces are set as slaves. Regarding the 5-pin DIN connector, input interfaces composed of optocouplers are present to satisfy MIDI transmission requirements. Moreover, the board features two Pmod connectors to interact with the audio processor. These are used as physical supports for both the I2S and the MIDI communication protocols.

One last consideration which needs to be made is that, even if the audio board features MIDI interfaces, the new function units added to the system will refer only to signals coming from XLR input channels. This is because MIDI protocol works with messages and requires different techniques to manage them.

3.4 Audio processor

The second component of the considered system is the audio processor, implemented as a custom ASIP. Its goal in this preliminary architecture is to manage the communications between the custom audio board and the Raspberry Pi device. In this first version, it is still not employed for sound processing intent. However, in the next chapters and future development, it will be designed to do so.

3.4.1 Development board

Since manufacturing a custom audio processor on an Application-Specific Integrated Circuit (ASIC) is very expensive, at the early stages of development another solution is preferred, usually an FPGA-based implementation. This choice comes also with the necessity of flexibility that a processor under development requires. For these reasons, a development board that embeds an FPGA was adopted to implement the audio ASIP and test it easily. The development platform used for this purpose is the Arty A7-100T board produced by Digilent and built around the Artix-7 model XC7A100TCSG324-1 FPGA from Xilinx. This board is a ready-to-use one and requires only to install Vivado Design Suite as a software tool. As a flexible platform, Arty A7-100T board is equipped with many devices such as 4 switches, 4 buttons, 8 LEDs (including 4 RGBs), 4 Pmod connectors and a shield connector Arduino / chipKIT compatible, while the built-in Artix-7 FPGA features 15850 Configurable Logic Blocks (CLB) slices, 240 Digital Signal Processing (DSP) slices,

4860 Kbits of Block RAM (BRAM) and an internal clock generation logic that can exceed 450 MHz [28]. A picture of the FPGA development board can be seen in figure 3.6.

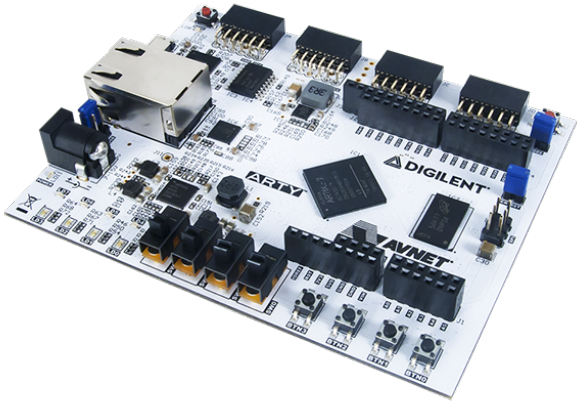

Figure 3.6: FPGA development board

3.4.2 Top level design

The preliminary version of the audio processor consists of many layers. As it can be seen from figure 3.7, it is implemented as a wrapper unit described in the Hardware Description Language (HDL) file *top_pll.sv*. This module is organized into two main blocks: a 11 MHz Phase-Locked Loop (PLL) and a top level, respectively *pll.v* and *top.vhdl*. The former block is the element in charge of generating an accurate 11.2896 MHz clock to support the I2S communication protocol. This module takes the main clock *clk* as input and produces the 11 MHz *mclk* as output. To easily implement the PLL unit, a Vivado customizable Intellectual Property (IP) is used. The second element contained in the wrapper *top_pll.sv* is the top level unit, which in turn is organized into different modules:

- TTA core: is the actual processor, containing all the different FUs
- I2S clock generator: generates the bit clock *bclk* and the left-right clock *lrclk* for the I2S protocol (since the audio processor is the master device)
- Instruction memory: contains the sequence of instructions to be executed by the processor
- Data memory: is used to store data and global variables

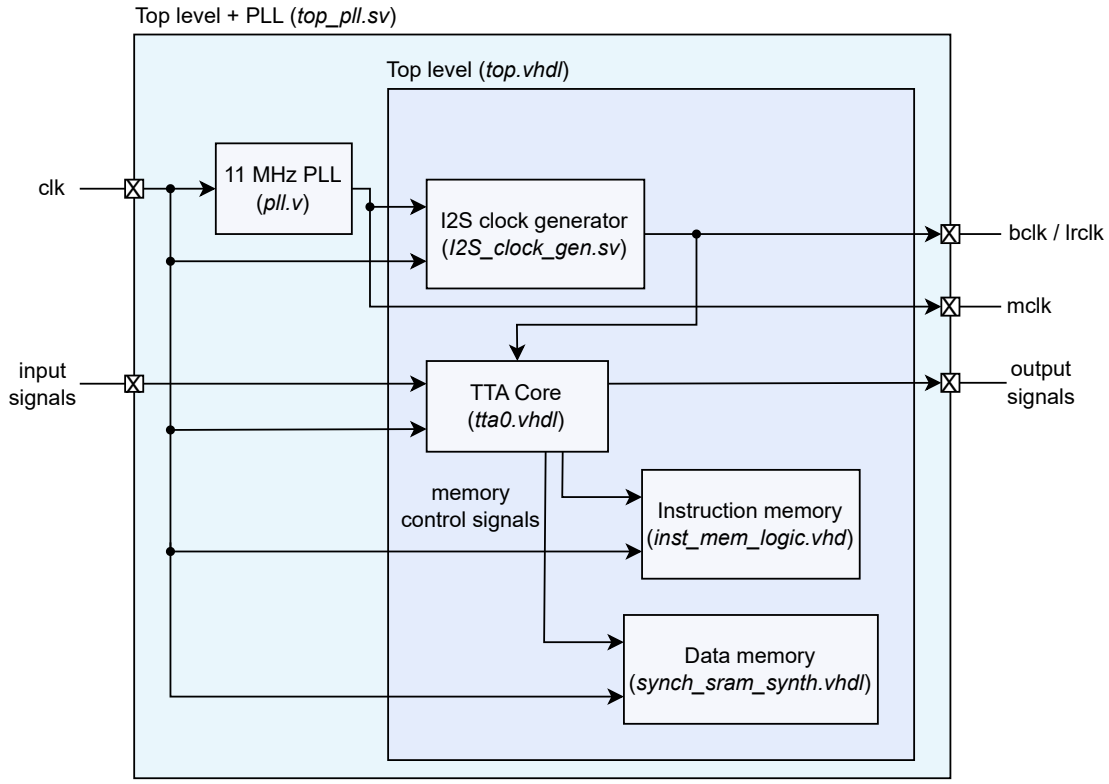

Figure 3.7: Audio processor top level block diagram

3.4.3 TTA core

The actual core of the audio processor is implemented using a Transport Triggered Architecture (TTA). This processor design is a one-instruction set architecture, based on several FUs and some register files connected by buses [29]. In this kind of architecture, the only available operation is the move one, which allows the execution of more complex instructions by moving data from one unit to another. For this reason, programs directly control the transport buses since the internal datapaths are exposed to the instruction set. The execution of operations happens as a side-effect of the moves between FUs: moving data into the trigger port of a FU makes it start working.

A TTA processor design has several advantages that make it particularly suitable for the system considered. First, it is an optimal solution for the development of an ASIP, since its characteristics match well with specific application domains, such as sound processing in this case. Moreover, TTA processors provide also opportunities to increase the functional parallelism of operations. The instructions to be executed are usually very long and permit parallelizing the job of several

function units, according to the number of buses designed. One more advantage, of a TTA implementation, comes from the possibility to exploit register bypassing. This optimization technique is usually adopted in general-purpose processors when the result of an operation is also the input to one of the following operations. The write-back register is bypassed using a hardware structure, saving a clock cycle. The same concept can be applied to a TTA processor. with the difference that data are directly moved from one unit to another, so registers to store results are always avoided if the compiler optimizes the operation execution. In addition, a TTA is also a suitable solution to implement an energy-efficient design. The only major drawback in using this kind of architecture can be found when talking about general-purpose processors. In this context, indeed, a TTA implementation is usually not recommended.

The designed TTA core is reported in figure 3.8 and, as it can be seen, it contains several units and two register files fully connected by means of four transport buses. In particular, the architecture is composed by the following FUs:

- *I2S_LJ_master_TX_0*: the I2S master-transmitter module used to transmit audio samples to the audio board
- *I2S_LJ_master_RX_x*: the I2S master-receiver modules used to receive audio samples from the audio board
- *UART_TX_x*: the UART transmitter modules used to transmit MIDI messages to the audio board
- *UART_RX_x*: the UART receiver modules used to receive MIDI messages from the audio board
- *SPI_slave_x*: the SPI slave modules used to transmit and receive audio samples and MIDI messages to/from the Raspberry Pi device
- *LED_DRIVER_x*: a module to drive the onboard RGB LEDs
- *SWITCH_DRIVER_x*: a module to read the status of the onboard switches
- *TIMER_0*: a timer unit
- *alu*: an Arithmetic-Logic Unit (ALU)
- *lsu*: a Load-Store Unit (LSU)

Furthermore, the processor architecture contains some other fundamental blocks: a Global Control Unit called *gcu*, that manages all the operations to be performed, and two register files called *REG_FILE_0* and *bool*, used respectively to store 32x32 bits and 16x1 bits values.

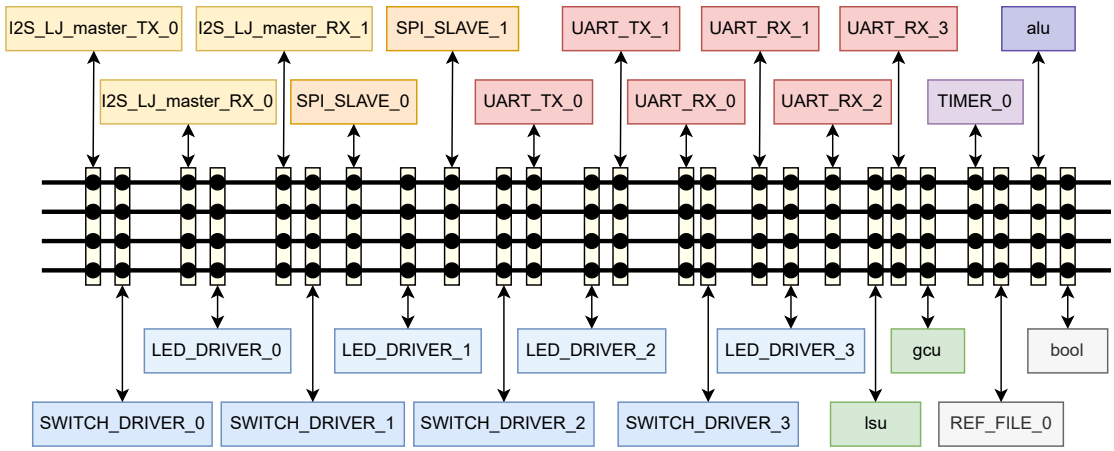

Figure 3.8: TTA core

3.4.4 Firmware

The developed audio processor can execute different firmware. This can be done just by changing the content of the instruction and data memories. Two firmware have been developed for the processor as far: *test_loopback.c* and *main_final.c*.

To test the correct behavior of the designed FUs, the firmware called *test_loopback.c* can be used. This code has been developed primarily to test the loopback capabilities of the system, which means checking if the transmitting and receiving interfaces are working. While executing the firmware, the processor is supposed first to acquire audio samples and MIDI messages from inputs and second to send them back. Of course, each elaborated audio data has to remain unaltered and this can be checked from the sender device after coming back. Using this firmware, the correct operation of the units can be tested and evaluated.

The *main_final.c* firmware is instead the main code to be executed by the audio processor to make the whole system work. Unlike *test_loopback.c*, it is not used to transmit audio samples and MIDI messages back to the sender, but to stream them from the audio board to the Raspberry Pi device and vice versa. In particular, the firmware supports two modes of operation: one in which the audio samples mixing is performed by the Raspberry Pi and another one in which, instead, the mixing is performed by the custom processor (a feature still to be developed in this initial version). Being the Raspberry Pi device the master of SPI communication, it has to properly set the mode of operation of the system. Moreover, it is also in charge of sending some configuration parameters to the audio processor, so that a correct communication can take place.

In particular, the Raspberry Pi can configure:

- w_s: samples bitwidth (default value is 16 bits, but should be 24 for the considered system)
- n_{CHF}: number of channels sent by the processor in case of mixing performed by Raspberry Pi (default value is 4 channels, all of them)
- *chann_mask*: bitmask to understand which channels are activated (default value is 0x0F, all channels activated)
- s_n: number of samples inside each audio packet (default value is 112 samples)
- n_{CHRPI}: number of channels coming from SPI in case of mixing performed by the audio processor (default value is 4 channels)

Moreover, a custom protocol has been developed to communicate over SPI between the audio processor and the Raspberry Pi. Specifically, a preamble sequence (0b10101010) has been introduced to understand when the communication is going to start and three custom command sequences have been developed:

- Command 1 (sequence 0b00000011): used to the change configuration variables
- Command 2 (sequence 0b00001100): used to start streaming audio samples, with mixing performed by Raspberry Pi
- Command 3 (sequence 0b00110000): used to start streaming audio samples, with mixing performed by the audio processor (feature still to be developed in this initial version)

To minimize the possibility that an error occurs during an SPI communication, the protocol involves also some redundancy. Indeed, the preamble sequence has to be repeated 5 times, while commands 3 times. More detailed information about the initial version of the audio processor and the main firmware can be found at [30].

3.5 Raspberry Pi

The last element that composes the system described so far is a Raspberry Pi. In particular, the adopted device is a Raspberry Pi 4 model B [31], released in 2019 by the Raspberry Pi Foundation. As already mentioned in section 3.1, this single-board computer aims to manage the transmittion and reception of audio packets to and from the network to support NMP applications. Since the latency requirements are very stringent and the delay associated with data networking is typically the bottleneck of real-time communications, the choice of an appropriate device was

crucial. To take into account these major concerns, the Raspberry Pi 4 series was chosen since it features a full gigabit Ethernet. This was not true for other series, which have a throughput limited by the internal USB connection. Moreover, the B model was adopted because it features an Ethernet port (Raspberry model A does not have Ethernet circuitry). A picture of the Raspberry Pi 4 model B device can be seen in figure 3.9.

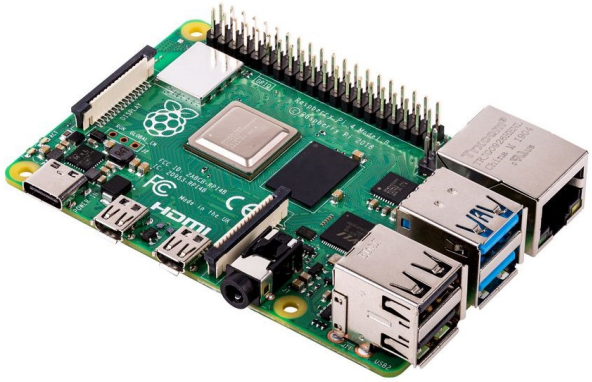

Figure 3.9: Raspberry Pi 4 model B

Other important features of the Raspberry to be considered are a 1.5 GHz 64-bit quad-core ARM Cortex-A72 processor, 8 GB RAM, two USB 2.0 ports, two USB 3.0 ports, two micro-HDMI ports and on-board wireless and Bluetooth modules. Moreover, it features 28 General Purpose Input/Output (GPIO) pins supporting various interface options, including the possibility to handle 5 different SPI communications. For the considered system, only one of them is actually needed to interact with the FPGA board. Regarding the adopted OS, for speed purposes, a real-time Linux kernel is running on the device. This was done since having hard time limits on system call execution can bring improvements to the system.

From a networking perspective, two main aspects must be taken into account to achieve low latencies. The first of them is the chosen communication protocol. Indeed, to guarantee suitable performances UDP was used. This kind of protocol is particularly convenient for time-critical data transmissions since it is faster than TCP. UDP does not require handshake techniques to set up a connection before transmitting data and, for this reason, can speed up the overall communication. However, as a main drawback, UDP communications are also less reliable and usually need PLC methods to deal with missing data packets. The other important aspect to be considered is the use of a peer-to-peer architecture between users of the NMP environment, which can decrease even more the perceived latency with

respect to a client-server solution. The developed software routines running on Raspberry Pi adopt the above-mentioned design choices [32].

3.6 Tools

3.6.1 TTA-based Co-design Environment (TCE)

TTA-based Co-design Environment (TCE) [33] is a toolset that can be used to design and program a custom TTA-based processor. Its development is led by the Customized Parallel Computing (CPC) group at Tampere University, Finland. This toolset permits to adapt several points of the architecture, such as function units, register files and interconnection networks, to the designer's needs. Such customizations are useful to obtain different kinds of performances from the processor and moreover to analyze the advantages and disadvantages of several design choices.

One of the greatest advantages of using TCE is that it contains a complete set of tools to design the system from a high-level representation to low-level hardware. In particular, as a co-design environment, it offers the possibility to understand if a software task can be accelerated by means of a dedicated hardware function unit or not. This is done by employing high-level custom operations which emulate the non yet existing hardware function units. With this procedure, a designer can analyze what are the advantages of having a dedicated function unit for a certain task, without even know how to implement it in hardware.

The typical design flow that can be adopted when using TCE is the one shown in figure 3.10. As it can be seen, to achieve an optimal architecture several steps are required. In particular, in the case of the considered system, since the audio processor is already present, only new FUs need to be managed. The first step is usually the creation of a firmware version of the feature to be added, by employing high-level languages. Then, to evaluate if the introduction of a dedicated structure can bring some advantages, a FU can be added to the processor (defining also its custom operations). In this way, it is possible to simulate its behavior and determine whether to implement it in hardware or not.

To follow this design approach, several tools can be exploited:

- ProDe: Processor Design, a graphical tool for editing the custom TTA processor and select hardware implementations for each component
- tcecc: TCE compiler, which compiles high-level language sources according to the processor's characteristics

- proxim: TTA Processor Simulator, which simulates programs on the target processor
- ProGe: Processor Generator, which produces a synthesizable hardware description of the processor
- OSEd: Operation Set Editor, a graphical application for managing the library which contains operation definitions called Operation Set Abstraction Layer (OSAL)
- HDBEditor: HDB Editor, a graphical tool for creating and modifying the databases that contain hardware implementations

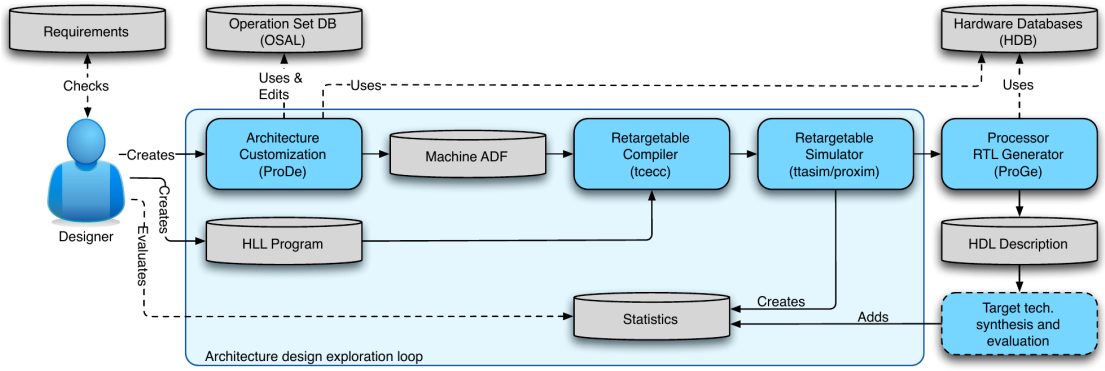

Figure 3.10: TCE design flow [34]

Furthermore, another feature of TCE is a tool called Processor Cost/Performance Estimator, which can be used to estimate die area, energy consumption and the maximum clock rate of the designed processor. Unfortunately, no estimations could be run on the considered audio processor, because some cost estimation plugins were missing. Anyway, even if this data could not be retrieved, more accurate estimations and results will be discussed after describing the synthesis and implementation processes on FPGA.

To speed up the execution of the different steps required for the implementation of new function units in TCE, a bash script was implemented and can be found in appendix A.1.

3.6.2 Vivado

Vivado [35] is a software suite developed by Xilinx to synthesize and implement HDL designs on its proprietary FPGA devices. As a software suite, Vivado includes

different tools to improve the digital design flow of a hardware architecture. As well as being able to synthesize a design, Vivado can also simulate it by defining a simulation source file (generally a HDL testbench) and integrate customizable IPs from the Xilinx library. Moreover, after the synthesis step and the implementation one (also known as Place & Route), the software provides several metrics to analyze the final result in terms of resource utilization (so employed area), timing and power consumption.

Chapter 4

Mixing Function Unit

This chapter describes the first feature added to the system: the audio mixing. First, it introduces what mixing is and what are the main reasons to implement it on an FPGA. Then, the adopted workflow is explained, starting from a high-level firmware implementation and getting to a hardware function unit.

4.1 Introduction to mixing

Audio mixing is the process of combining different sound sources (also called tracks) together to obtain a musical piece as a final result [36]. Its main goal is to balance the volume of each track to make some of these sound louder than others. Nowadays, this practice is widely employed both in live performances and in music recordings.

Since the introduction of multi-track recorders, audio mixing has played a crucial role in music production. Before that, in fact, all sounds were mixed together during live performances, with less control over recording results. Without separate tracks, there was no possibility to perform sound processing, add effects or adjust the volume of each instrument. Moreover, if some kind of mistake occurred during the performance, the only way to proceed was to record again the entire piece. In the 50s, the birth of multi-track recorders brought some more flexibility for musicians and producers. Indeed, it allowed them to record each instrument onto a separate track and as a consequence to have greater control over how the final record sounded. But with the possibility to record separate tracks, the need of combining them into a single product rose. The solution to this problem was achieved thanks to an electronic device known as mixer. The first kind of mixers were hardware devices that allowed the control of each track through a series of physical knobs and sliders. However, in the 90s, with the increasing use of the early Digital Audio Workstations (DAWs), also software mixers became popular.

Nowadays, modern mixers are not only used to combine sounds, but can also offer several features such as equalization, filtering and panning management [37]. In particular, the latter is an extremely common element in music production and will be also implemented in the mixing function unit described in the next sections. Panning is an audio technique used to give a particular direction to audio signals and distribute them so that a listener can enjoy a more realistic sound environment. This can be done by introducing the concept of stereophonic sound (or stereo), which is a method to reproduce sound to give a multi-directional experience to listeners. This is usually performed using two different sound sources: left and right channels. Indeed, if a single output source reproduces all the sounds, the listener is not going to perceive spatial dimensions. This different approach is also known as monophonic sound reproduction (or mono).

4.2 Motivation

As for musical production and live performances, audio mixing has an important role also in NMPs. Indeed, like in live interactions, also in this case a way of combining different sound sources and setting their volumes is needed. In the preliminary version of the system considered, sound mixing was implemented in software and ran on the Raspberry Pi device. The purpose of this chapter is to move the mixing operation from software to a hardware unit on the audio processor.

Implementing an audio mixer in hardware, exploiting a dedicated function unit, could have some advantages for the overall system. In particular, it can:

- Reduce resource utilization on the Raspberry Pi, so that it has only to care about acquiring samples from the SPI interface, packetizing and sending them to the network.

- Reduce the amount of time taken by the mixing process, since a hardware-dedicated solution can usually bring improvements in terms of speed and latency.

However, this kind of approach has also some drawbacks to take into account, which are related to:

- An increased use of area on FPGA (not a problem for the considered system, since there are still many resources available)

- Possible limitations when dealing with a very large amount of audio sources, due to the SPI communication bandwidth

4.3 Implementation

4.3.1 Mixing algorithm

Before going into the firmware and hardware implementation details, a brief theoretical explanation of the mixing algorithm is needed. As described in the previous sections, a mixer can have different features, but the most basic ones are:

- Combining different sound sources
- Adjusting the volume of each track
- Performing panning

The first of these features can be accomplished by performing sum operations between the different sound sources (input channels). Indeed, this will result in a single output which is the combination of all of them. The second feature, instead, can be performed by exploiting a multiplication between each input source and a coefficient (gain) that determines the volume. In particular, if this coefficient is lower than 1, then the volume is decreased accordingly, while if is greater, it is increased. Moreover, a coefficient equal to 1 means unaltered sound, while a coefficient equal to 0 means silence on output. This concept can be easily extended to obtain also the panning feature. Indeed, this can be implemented by considering two coefficients for each input channel instead of one. In this way, each input source is multiplied by two different values which quantify the amount of sound going into the left and right output channels respectively.

The mixing algorithm is based on the concept of gain matrix. This algebraic structure contains all the coefficients explained above, thus the information regarding the output volume and the panning of each input channel. This matrix has a number of rows which is equivalent to the number of input channels and a number of columns which is equivalent to the number of output sources. For this reason, the gain matrix developed for the considered system is composed of 4 rows (4 input channels) and 2 columns (2 output channels) and can be written as:

$$G = \begin{bmatrix} G_{00} & G_{01} \\ G_{10} & G_{11} \\ G_{20} & G_{21} \\ G_{30} & G_{31} \end{bmatrix} \tag{4.1}$$

That said, the mixing algorithm performs nothing but a matrix multiplication between an input sound sources vector and the gain matrix. The result of this operation is a vector containing output channels. The complete equation of the

mixing algorithm is reported in formula 4.2, where again the case of the considered system is taken into account.

$$\begin{bmatrix} CH_0 & CH_1 & CH_2 & CH_3 \end{bmatrix} \cdot G = \begin{bmatrix} CH_L & CH_R \end{bmatrix} \tag{4.2}$$

Rewriting equation 4.2 in an expanded form, it can be seen that output channels can be computed as:

$$\begin{aligned} CH_L &= CH_0 \cdot G_{00} + CH_1 \cdot G_{10} + CH_2 \cdot G_{20} + CH_3 \cdot G_{30} \\ CH_R &= CH_0 \cdot G_{01} + CH_1 \cdot G_{11} + CH_2 \cdot G_{21} + CH_3 \cdot G_{31} \end{aligned} \tag{4.3}$$

To implement the mixing function unit, the workflow suggested by TCE was adopted. As already explained in subsection 3.6.1, the workflow can be split into several steps, the first of which is to implement a firmware version of the algorithm.

4.3.2 Initial firmware

To correctly evaluate the benefits that a hardware mixing unit can bring to the system, the first step performed was to write an initial test firmware. This program executes a software version of the mixing algorithm and runs it by exploiting the preliminary version of the architecture, so the basic ALU unit for additions and multiplications. In the next subsection, another version of this firmware, containing custom operations to perform mixing, will be described to compare performances.

The developed initial firmware is called *test_mixing.c* and is similar to the loopback test firmware (*test_loopback.c*). What this program does is to acquire audio channels from the I2S interfaces, mix them by exploiting matrix multiplication and then transmit the resulting output channels on the I2S interface again.

To make the test firmware as complete as possible, different mixing conditions can be configured by acting on the physical switches on the FPGA development board. In particular, what a user can do is deciding which couple of channels he/she wants to listen to as outputs. Switches 0 (SW0) and 1 (SW1) can be used to turn ON/OFF the couple of channels CH0/CH1 and CH2/CH3 respectively, while switches 2 (SW2) and 3 (SW3) can be used to test mono/stereo configurations for each couple. As an example, if SW1 is turned ON while SW3 is OFF, the user can hear the couple CH2/CH3 in mono on output. If then SW3 is turned ON too, the system switches to stereo conditions, so CH2 can be heard on the left output channel, while CH3 on the right one. A complete list of the different mixing configurations is shown in table 4.1.

To easily debug the code, while performing onboard testing, also two RGB LEDs were exploited. Each of them is related to a couple of channels and is turned on with a red color for mono mode, while with a blue one for stereo.

SW0	SW1	SW2	SW3	CH0/CH1	CH2/CH3
OFF	OFF	OFF	OFF	OFF	OFF
ON	OFF	OFF	OFF	MONO	OFF
OFF	ON	OFF	OFF	OFF	MONO
ON	ON	OFF	OFF	MONO	MONO
OFF	OFF	ON	OFF	OFF	OFF
ON	OFF	ON	OFF	STEREO	OFF
OFF	ON	ON	OFF	OFF	MONO
ON	ON	ON	OFF	STEREO	MONO
OFF	OFF	OFF	ON	OFF	OFF
ON	OFF	OFF	ON	MONO	OFF
OFF	ON	OFF	ON	OFF	STEREO
ON	ON	OFF	ON	MONO	STEREO
OFF	OFF	ON	ON	OFF	OFF
ON	OFF	ON	ON	STEREO	OFF
OFF	ON	ON	ON	OFF	STEREO
ON	ON	ON	ON	STEREO	STEREO

Table 4.1: Mixing configurations in *test_mixing.c*

Additionally, two considerations need to be made on the *test_mixing.c* firmware. The first regards the fact that all the coefficients in the gain matrix have been assumed to be integer numbers so far. This is because the preliminary version of the considered architecture is based on 32 bits and it is not yet taking into account a possible fixed-point representation. The second consideration, instead, concerns the possibility to deal with different gain matrixes, one for each user of the system on a networked performance. In this initial firmware, this feature is not taken into account, but it will be implemented when considering mixing custom operations.

The compilation of the firmware was carried out using the `tcecc` command and gives a total number of instructions to be executed by the audio processor, equal to 187. After this operation, the `proxim` command was executed to run the instruction set simulator. This TCE tool simulates TTA cores at an architectural level, without caring about their hardware implementation, and it produces statistics about cycle counts and utilization of units. This simulator is a very powerful tool since it permits us to understand which are the characteristics of a firmware and to compare it with another one. Indeed, to understand differences, the same procedure will be performed also in the next subsection when dealing with the new custom operations.

The simulation provides different kinds of results. Some of the most significant ones are reported here. Since the firmware under analysis does not have an ending

point, a timeout was set for the simulation. This was possible thanks to the `simulation_timeout` parameter, which permits specifying the number of seconds before the timeout expires. In particular, for this firmware, a simulation timeout of 60 seconds was applied. This is because it was found that 60 seconds were enough to converge to stable results.

During the simulation, a total number of 47595527 clock cycles were evaluated. First, the tool provides some information regarding the utilization of each bus, which can be seen in table 4.2. As it can be observed, buses B3 and B4 are the most used ones (more than 60%) and have a very high utilization rate with respect to buses B1 and B2 (less than 25%). Bus utilization is a consequence of how operations are distributed by the compiler. Since the fourth bus is used very few times (2.55% of the simulated instructions), for this firmware it seems that 3 buses are enough to execute the code without impacting too much on performance.

B1	2.55% (1212593 writes)
B2	21.97% (10456742 writes)
B3	63.38% (30164514 writes)
B4	75.31% (35845080 writes)

Table 4.2: Bus utilization (without a mixing FU)

Another useful information which the simulator gives is the usage of each function unit. In table 4.3, the most used units are reported along with their utilization percentage and number of triggers. As it can be seen, the ALU unit is the most used FU with a number of triggers which is almost 55% of the total instructions. This means that more than a half of the executed instructions consist of some kind of arithmetical or logical operations. In the same table, it can be also noticed that I2S utilization is around 1% for TX and 5% for RX, while LSU is triggered 3.83% of the times. These data will be useful for comparison in the next subsection.

alu	54.45% (25917431 triggers)
I2S_LJ_master_TX_0	0.96% (455540 triggers)
I2S_LJ_master_RX_0	5.09% (2424920 triggers)
I2S_LJ_master_RX_1	5.09% (2424920 triggers)
lsu	3.83% (1822161 triggers)

Table 4.3: FU utilization (without a mixing FU)

The simulator can also retrieve a more detailed analysis on the percentage for each operation in ALU. The resulting data are presented in table 4.4. As expected, it

can be seen that two of the most used operations are the ADD (16.10% of total ALU instructions) and the MUL (7.03% of total ALU instructions).

ADD	16.10% of FU total (4173471 executions)
AND	34.19% of FU total (8862212 executions)
EQ	11.40% of FU total (2954203 executions)
IOR	15.20% of FU total (3938761 executions)
SHL	8.48% of FU total (2197151 executions)
XOR	7.60% of FU total (1969380 executions)
MUL	7.03% of FU total (1822160 executions)

Table 4.4: Operations executed in ALU (without a mixing FU)

4.3.3 OSAL

After testing the first firmware solution, another version of *test_mixing.c* was developed by using custom operations to implement the mixing algorithm. This was done by using the `osed` command, which opens a graphical application to manage the OSAL database. To implement mixing, a new module called *MIXER* was added to the OSAL. Then, two custom operations were added to the new module. This procedure can be used to accelerate application-specific functionalities.

In particular, the first operation to speed up is the one which permits to mix audio samples. This operation is called *MIXER_MIX_CHANNELS* and can be performed using the custom macro:

MIXER_MIX_CHANNELS(USER, CH0, CH1, CH2, CH3, CHL, CHR)

where USER is an unsigned input number associated with the gain matrix of each user, CH0 to CH4 are signed 2's complement input numbers containing the samples to mix and CHL/CHR are signed output numbers containing the mixed samples.

The second operation added to the *MIXER* module is *MIXER_SET_GAIN*. This operation can be used to change the coefficients in one of the gain matrixes. This can be performed using the custom macro:

MIXER_SET_GAIN(USER, G0, G1, G2, G3, G4, G5, G6, G7)

where USER is an unsigned input number that is used to determine the correct gain matrix to be changed and G0 to G7 are the gain coefficients written as unsigned input numbers.

To use these operations in the new version of *test_mixing.c*, a dedicated function unit needed to be added to the processor's architecture. For this reason, the

MIXER_0 unit was conceived. When creating a FU, an important parameter that needs to be determined for both custom operations is latency. This parameter is fundamental to simulate the new architecture since it tells how many clock cycles every operation will take. The latency setting is almost a trial and error process at the early stage of the project because usually a developer still doesn't know the hardware that he/she is going to develop. For this reason, he/she does not even know how many clock cycles are required for its execution. However, the results which can be found in the remaining part of the section refer to the final latency chosen (obtained after knowing the implemented hardware). Latency has been fixed to 1 clock cycle for the *MIXER_SET_GAIN* operation and to 5 cycles for *MIXER_MIX_CHANNELS*.

Having all the necessary ingredients, the new version of *test_mixing.c* was developed. This was done by replacing the lines of code related to the mixing with the *MIXER_MIX_CHANNELS* custom operation and the ones related to the configuration changing with a *MIXER_SET_GAIN* one.

To evaluate the impact of these two new operations, the instruction set simulator was executed again, with a simulation timeout of 60 seconds. The first difference with respect to the previous solution is that the program is composed of 166 instructions instead of 187. This means that mixing custom operations decreases the total number of instructions to be executed by 11.2%. This improvement has as a major consequence an increase in speed, since fewer instructions need to be performed by the processor. Moreover, since each instruction has a length of 176 bits, having less instructions permits also to decrease the size of the instruction memory (462 bytes in this case).

For the new firmware, the tool simulates 48823712 clock cycles, so the following data will refer to them. Again, results on bus utilization were retrieved and can be seen in table 4.5. As it can be observed, the use of each bus follows almost the same trend as before, but with some differences. Indeed, it seems that both buses B1, B2 and B4 share an increased usage, while the B3 one is decreasing.

B1	3.01% (1471781 writes)
B2	24.25% (11837905 writes)
B3	59.72% (29157755 writes)
B4	76.37% (37284294 writes)

Table 4.5: Bus utilization (with a mixing FU)

Regarding the FUs, their utilization is reported in table 4.6. As it can be seen, with the new firmware the ALU unit was triggered a lower amount of times. This

is of course happening because the new FU is taking care of the mixing operations. The ALU utilization decreased to a 53.21% of total instructions, with respect to a 54.45% of the previous version. The *MIXER_0* unit was triggered only 0.48% of the time, meaning that once the samples have been sent to it, the unit is quite independent from the rest of the architecture. From the table, it can be seen also that I2S interfaces were triggered more times. This is probably because, in almost the same number of simulated cycles, more of them can be dedicated to I2S handling. Instead, as expected, LSU is triggered very few times, since no gain coefficients need to be retrieved from the data memory.

alu	53.21% (25979857 triggers)
I2S_LJ_master_TX_0	0.96% (469042 triggers)
I2S_LJ_master_RX_0	6.03% (2943296 triggers)
I2S_LJ_master_RX_1	6.03% (2943296 triggers)
lsu	2.05e-06% (1 triggers)
MIXER_0	0.48% (234560 triggers)

Table 4.6: FU utilization (with a mixing FU)

The evidence that the ALU unit is no more used for mixing purposes can be found in table 4.7. Looking at it, it can be noticed that both the ADD and MUL operations have a lower number of executions. The ADD execution decreases from the previous 16.10% to 4.76%, while MUL from 7.03% to no executions at all.

ADD	4.76% of FU total (1237128 executions)
AND	42.86% of FU total (11134145 executions)
EQ	14.29% of FU total (3711488 executions)
IOR	19.05% of FU total (4948508 executions)
SHL	9.52% of FU total (2474255 executions)
XOR	9.52% of FU total (2474254 executions)
MUL	0% of FU total (0 executions)

Table 4.7: Operations executed in ALU (with a mixing FU)

Finally, also results related to the mixing FU can be seen in table 4.8. As expected, most of the simulated operations are *MIXER_MIX_CHANNELS*, since changing the gain matrix is quite rare with respect to mixing new samples.

MIXER_SET_GAIN	0.017% of FU total (39 executions)
MIXER_MIX_CHANNELS	99.983% of FU total (234521 executions)

Table 4.8: Operations executed in MIXER_0

4.3.4 Hardware implementation

Since the simulation of the custom operations gave the expected improvements in terms of number of instructions to be executed, the next step was to design the hardware FU. In particular, the goal was to develop a mixing unit that can combine audio samples using a different set of gain coefficients for each user. To achieve this result, the hardware module shown in figure 4.1 was conceived.

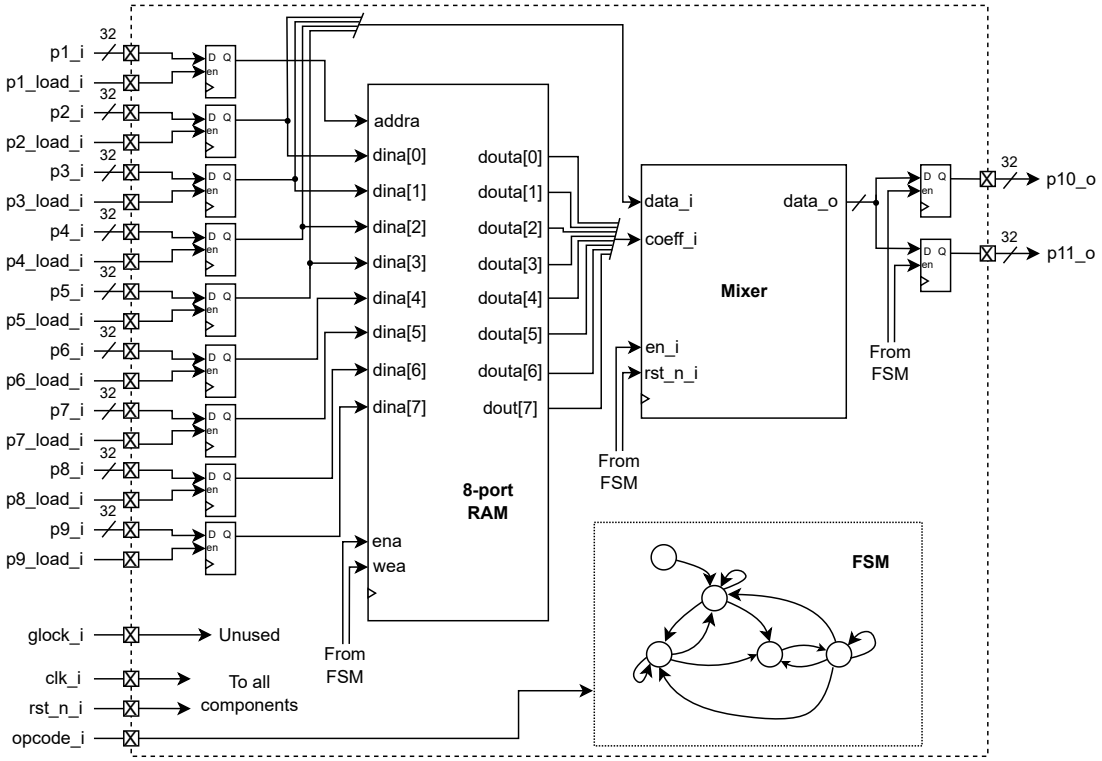

Figure 4.1: Mixer wrap

The developed module, called *mixer_wrap.sv*, is mainly composed by three different blocks: an 8-port RAM, a *Mixer* component and a Finite-State Machine (FSM). Each of them has an important role inside the architecture and will be explained in detail later. First, however, some considerations on the wrap module need to be made. From figure 4.1, it can be observed that the unit has several ports. In addition to the clock (*clk_i*) and the reset (*rst_n_i*) ones, there are two other input ports used for control signals: *glock_i*, which can be exploited for locking purposes but is unused so far, and *opcode_i*, which is used to determine the correct custom operation to be performed. Then, as it can be seen, the developed FU features other input ports with 32-bit width. Each of them is associated with a load signal and it is connected from outside to all the transport buses of the TTA processor. Port

p1_i, which is also the trigger one, is used to receive the unsigned number related to the user, while the others have different meanings depending on the custom operation to be executed. Specifically, when performing *MIXER_SET_GAIN* all the other input ports are used to receive the gain coefficients (8 values on 32 bits, one per port), while when executing *MIXER_MIX_CHANNELS* only ports from *p2_i* to *p5_i* are actually exploited to receive the samples to be mixed (4 values on 32 bits). Moreover, the input ports assume different meanings in terms of numerical representation depending on the operation. When dealing with gain coefficients, the input values to be provided need to have a 24p8 fixed-point representation, so that fractional numbers can be used (where 24 is the integer part and 8 is the fractional one). Instead, audio samples need to have a 32p0 representation. Obviously, having the same numerical representation for all the inputs is the most convenient way to proceed when designing a hardware unit, but in this case the choice was imposed by the numerical representation of the overall processor, that is not taking into account fractional numbers. For this reason, numeric conversions are applied inside the FU and in particular in the *Mixer* component so far (a possible improvement for the processor should take in consideration a fixed-point representation for the entire architecture). Also the output ports of the unit are connected to all the transport buses of the processor. In particular, ports *p10_i* and *p11_i* are used to output respectively left and right output channels. Since they are used for audio samples only, they are always characterized by a 32p0 numeric representation.

As it can be seen from figure 4.1, input and output registers were introduced into the mixing FU. In this way, the new unit is not likely to increase critical paths that can exist outside of it. Moreover, input registers are also used to properly sample the data, by enabling them only when the corresponding load signal occurs. The enabling of output registers, instead, is managed by an internal control signal.

The first element to be described is the FSM, which was realized to properly handle all the control signals in the hardware unit, depending on the custom operation that needs to be performed. Its state diagram is shown in figure 4.2 and as it can be seen, the FSM can evolve in 5 different states:

- RESET: the entry point of the FSM and the state that can be reached only in case of errors
- IDLE: the state in which the unit is waiting for a new operation to be triggered
- SET_GAIN: the state in which the unit is storing a set of gain coefficients for a specific user
- READ_MEM: the state in which the unit is reading a set of gain coefficients to start the mixing operation for a specific user

- MIX_CHANNELS: the state in which audio mixing is actually performed

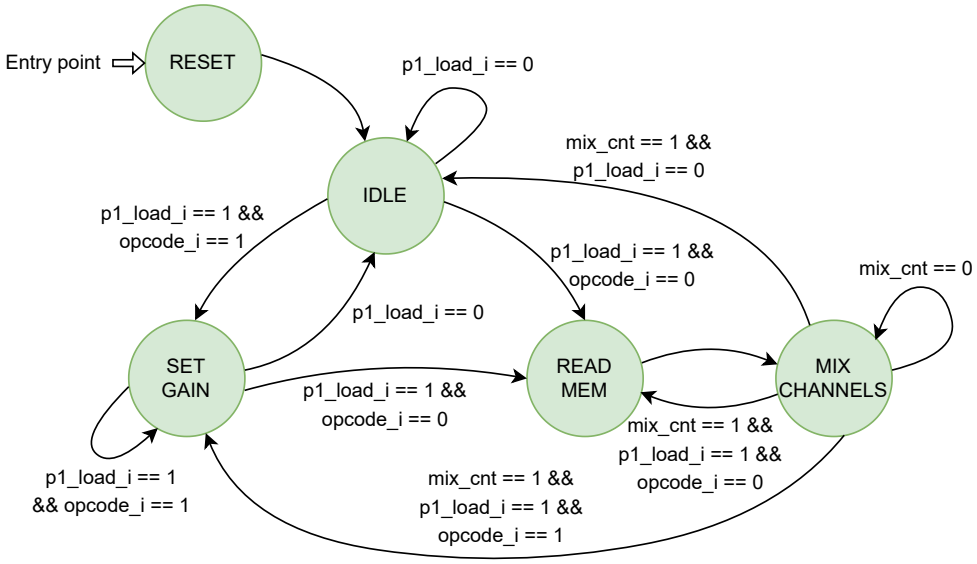

Figure 4.2: Mixer FSM

As it can be observed, the transition between states usually happens when a new custom operation is triggered, so when *p1_load_i*, i.e., the trigger of the FU, is equal to 1. Moreover, the *opcode_i* is used to determine which kind of operation needs to be executed. The only two states that are behaving in a slightly different way are READ_MEM and MIX_CHANNELS. In fact, the former has always a transition to the latter, regardless of the value of other signals, while MIX_CHANNELS always lasts two clock cycles. These behaviors are the consequence of the fact that the *MIXER_MIX_CHANNELS* custom operation has a latency greater than one clock cycle to be executed.

The second main component of the mixing FU is the 8-port RAM. This block is used to store all the gain coefficients needed by each user. The first solution attempted to implement this module was based on an HDL file called *nport_ram.sv*, that was describing a generic and configurable N-port RAM. However, during the synthesis step, Vivado had some problems exploiting the most appropriate resources for its implementation, so another solution was considered. In particular, memories from the Vivado IP catalog were chosen. Since by using this component it was not possible to directly implement an 8-port RAM, the solution was to design 8 different single port memories. To store all the needed coefficients, each of them has a word width of 32 bits (coefficients have a 24p8 numerical representation) and a depth of 256 words (equal to the number of maximum supported users so far).

The last component of the architecture is called *Mixer* and it is the one that is actually in charge of mixing the audio samples. This module is described by means of the HDL file called *mixer.sv* and its schematic is reported in figure 4.3.

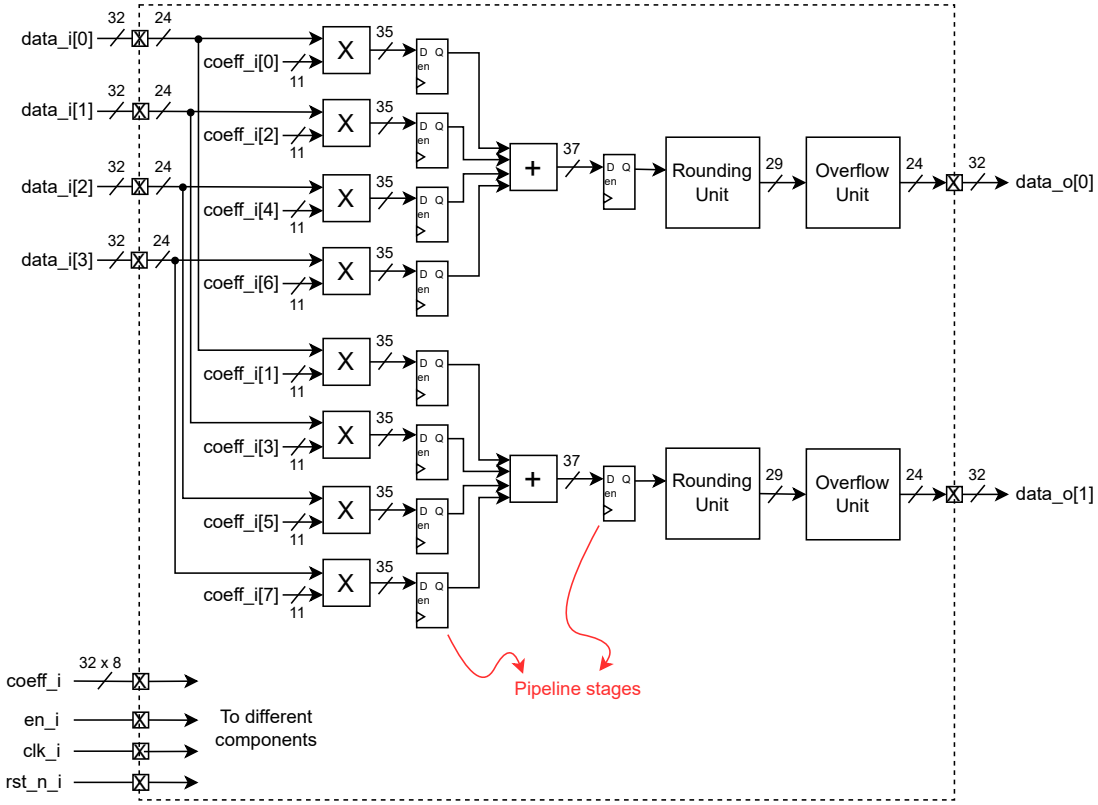

Figure 4.3: Mixer component

As it can be observed, the *Mixer* component performs the algorithm described in subsection 4.3.1, by parallelizing as much as possible the different arithmetical operations, so without sharing common hardware resources. This was done in order not to introduce too much latency on audio samples, although it is at the expense of the area used. Audio mixing is computed by means of an internal dedicated data width and a fixed-point representation to support fractional numbers. Indeed, as already mentioned, coefficients have a 24p8 numeric representation and arithmetic operators need to take them into account properly. Furthermore, two other elements were introduced in the *Mixer* module: a rounding unit and an overflow one. The former is used to round the final results of the mixing operation to integer values by exploiting the rounding to the nearest method, while the latter performs an arithmetic saturation on possible overflows.

Two major improvements were performed on the *Mixer* component to optimize

it. First, to save some area, a smaller data width was actually exploited to compute the algorithm. Indeed, even if the audio samples coming from outsides are represented on 32 bits, only 24 of them are actually used (audio board's ADCs provide samples with a 24-bit width). For this reason, a lower number of bits can be used. A similar reasoning can be made for the gain coefficients, indeed to have less complex arithmetic operators, their dynamic was limited too (11 bits with a 3p8 representation were used). The second optimization performed regards, instead, the introduction of two pipeline register stages. This was done to improve the maximum frequency achievable by the unit, because otherwise too long paths would be implemented. As a drawback, these stages add two clock periods of latency, but in this way the required timing can be met.

To have a better understanding of how the designed unit behaves, a timing diagram, depicted in figure 4.4, was realized. In particular, as it can be seen, the two implemented custom operations are shown: first, *MIXER_SET_GAIN* and then, *MIXER_MIX_CHANNELS*.

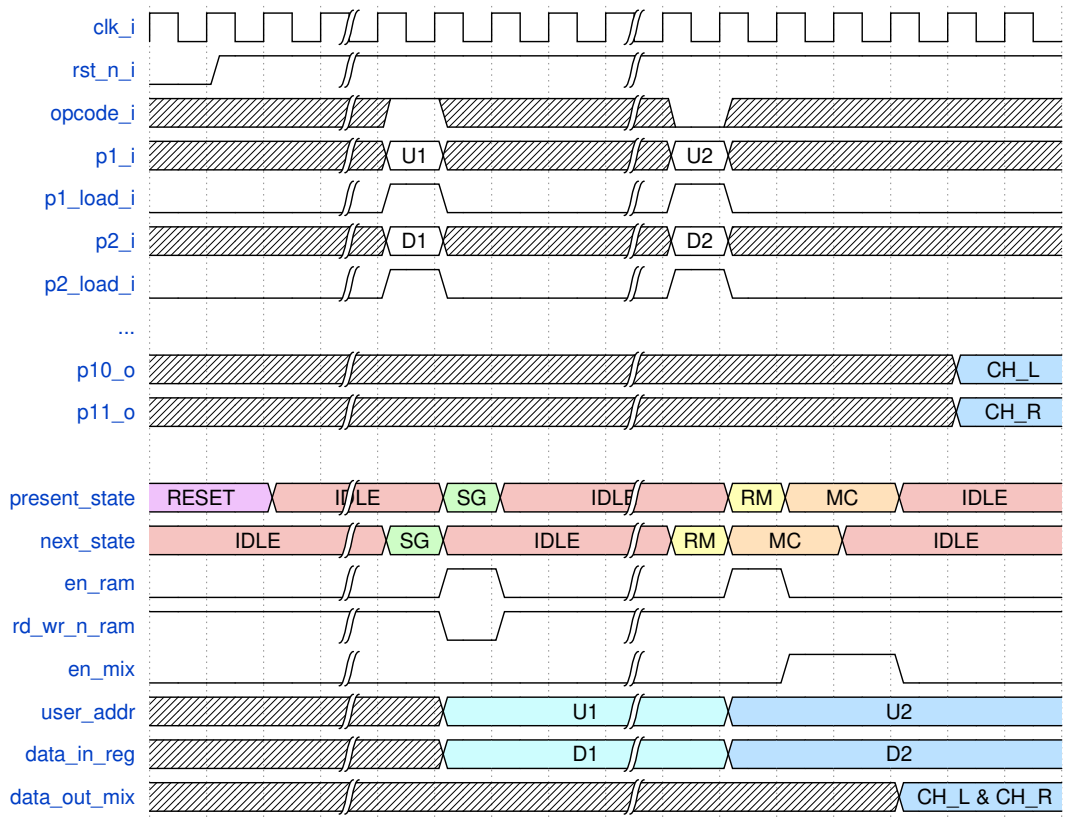

Figure 4.4: Mixer timing diagram

It can be noticed that *present_state* and *next_state* are the signals that determine

the state machine transitions. By taking a look at them and considering input/output registers, it is possible to observe that the overall latency of the FU is equal to 1 for the *MIXER_SET_GAIN* operation and to 5 for *MIXER_MIX_CHANNELS*.

After its design, the mixing FU was added to the Hardware Database using the `hdbeditor` command. Then, `generateprocessor` was executed to generate the complete audio processor from the hardware source files and `generatebits` was launched to create the contents of both instruction and data memories.

4.3.5 Hardware simulation

Before synthesizing the designed unit on FPGA, another step is usually performed in a digital design flow. Indeed, the implemented module, which takes the name of Design Under Test (DUT), needs to be simulated by means of a hardware simulation tool. For this purpose, Xilinx Vivado simulator was exploited. Moreover, to check if the DUT behaves correctly or not, a verification environment needs to be built. As it can be seen from figure 4.5, to test the design in the most generic way, an approach based on a C golden reference model was adopted.

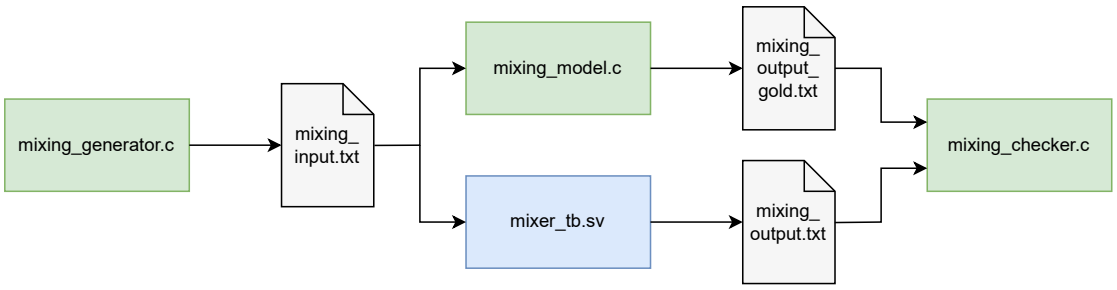

Figure 4.5: Mixer FU simulation approach

The full verification environment is composted by different elements:

- *mixing_generator.c*
- *mixing_model.c*
- *mixer_tb.sv*
- *mixing_checker.c*

The first component of the simulation environment is *mixing_generator.c*. This is a C program whose purpose is to generate random operations to be executed by the DUT. It creates a text file called *mixing_input.txt* containing an instruction for each line. Since the DUT can perform two different operations, each line starts with an

'S' character if *MIXER_SET_GAIN* needs to be executed or with an 'M' character in the case of *MIXER_MIX_CHANNELS*. Some parameters of *mixing_generator.c* can be configured by acting on internally defined macros. In particular, one can change the number of random samples generated, the maximum range of both samples and gain coefficients and the occurrence of the *MIXER_SET_GAIN* operation.

The second element is *mixing_model.c*, another C program that acts as the golden reference model to be compared with the DUT. It takes *mixing_input.txt* as a parameter and generates a text file called *mixing_output_gold.txt* containing the result of each instruction read. To have a reliable reference model, *mixing_model.c* computes its output values using a floating-point representation on 64 bits.

In the meanwhile, the same input file is given to *mixer_tb.sv*, which is the actual testbench designed in SystemVerilog. During the simulation, this file extracts the instructions that the DUT has to perform from *mixing_input.txt*, generating input signals according to its content. As mentioned earlier, a *MIXER_SET_GAIN* operation is executed when an 'S' character is read, while a *MIXER_MIX_CHANNELS* one is performed when the testbench reads an 'M' character. This is done by acting on the *opcode_i* input port of the DUT, which is forced to 0 to mix input samples and to 1 to set one of the gain matrixes. Finally, *mixer_tb.sv* writes each value computed by the DUT on an output file called *mixing_output.txt* so that a comparison with the reference model can be performed.

The component in charge of comparing the mixed output samples of the DUT with respect to the golden ones is called *mixing_checker.c*. In practice, this C program computes the difference between them to check if there is some kind of inconsistency. Here, further considerations should be taken into account. As already explained in subsection 4.3.4, the output values computed by the DUT are already rounded internally to the module, since a rounding unit is implemented in *mixer.sv*. Because of this, it is clear that the difference computed by *mixing_checker.c* could be different from 0: reference values are computed using floating-point and they own an intrinsic better precision. In particular, since the rounding to the nearest method was applied in the *Mixer* component, the expected difference could be at most 0.5. For this reason, differences below such threshold are not considered as errors. Unfortunately, a more accurate estimation of the internal precision of the mixing unit can't be performed without having access to internal signals.

After executing all these programs and performing the hardware simulation on Vivado, no differences greater than 0.5 were found. For this reason, the design flow proceeded with its last phase: the implementation on FPGA.

4.3.6 Synthesis and implementation on FPGA

The final step performed is the synthesis of the design into a digital circuit and its implementation on FPGA. These operations were performed again by means of Vivado, which made some improvements possible. Indeed, since this software knows on which FPGA it is going to implement the digital circuit, it is also able to optimize resource utilization. The two main elements exploited to achieve improvements in the mixing unit are BRAM blocks and DSP slices.

BRAM blocks are RAM embedded into the FPGA that can be used to optimize data storage. Each BRAMs can store up to 36 kbit and, to further optimize storage, it can be divided in two completely independent blocks of 18 kbit. To do that, each BRAM has two independent ports that share nothing but the stored data. Artix-7 family FPGAs are equipped with a different amount of storage capacity depending on the device. For the considered XC7A100T model, the total number of BRAMs is equal to 135, giving a maximum storage capacity of 4860 kbit [28].

For the design of the mixing unit, BRAMs were exploited to implement the 8-port RAM component. In this way, it was possible to store gain coefficients more efficiently. Indeed, by employing these embedded memory blocks, better results in terms of speed could be obtained and it was possible to meet the clock frequency requirement of 100 MHz. Each single port memory has a word width of 32 bits and a depth of 256 words, so it requires a total capacity of 8192 bits (1 BRAM block of 18 kbit). Since 8 memories are used in the FU, 4 BRAM blocks of 36 kbit were needed to properly store all the coefficients.

The second enhancement made during the synthesis step regards the use of DSP slices to improve arithmetic operations. Artix-7 family's FPGAs provide specific hardware accelerator called DSP48E1 for signal processing purposes. In particular, the XC7A100T device contains 240 of these slices, so that many arithmetic and logic operations can be performed in parallel [28]. An internal representation of a DSP48E1 slice is shown in figure 4.6. As it can be noticed, each DSP slice is mainly composed by [38]:

- A pre-adder for power saving purposes
- A 25×18 two's-complement multiplier
- A 48-bit three-input adder
- A pattern detector to support rounding and overflow / underflow detection
- Optional pipeline registers and dedicated buses for slices cascading

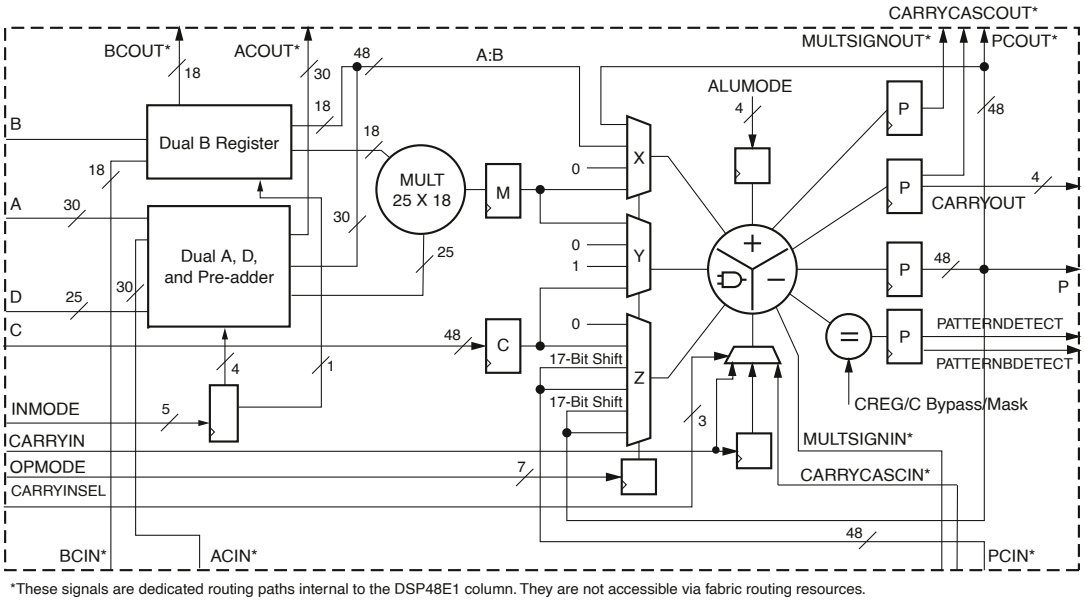

Figure 4.6: DSP48E1 slice (image taken from [38])

After performing the synthesis, the inferred logic was analyzed by taking a look at the *runme.log* file generated by Vivado. The retrieved results about DSP blocks utilization are shown in table 5.5.

DSP Mapping	A Size	B Size	P Size	MREG	PREG
(A'*B)'	30	18	0	0	1
PCIN+(A'*B)'	30	18	0	1	0
PCIN+(A'*B)'	30	18	0	1	0
(PCIN+(A'*B)')'	30	18	37	1	1
(A'*B)'	30	18	0	0	1
PCIN+(A'*B)'	30	18	0	1	0
PCIN+(A'*B)'	30	18	0	1	0
(PCIN+(A'*B)')'	30	18	37	1	1

Table 4.9: DSP final report for the reverb FU (the ' indicates corresponding REG is set)

As it can be observed, Vivado exploits 8 different DSP blocks to implement the arithmetic operations. Specifically, the first and the fifth slices are used to perform only a multiplication, while the others compute also an addition. The tool exploits the dedicated internal bus called PCIN to perform slices cascading. To better understand how Vivado decided to implement audio mixing by means of its DSP

slices, a graphical representation of the synthesized hardware is reported in figure 4.7. Looking at this picture, the purpose of the tool becomes clear: Vivado implemented the architecture exploiting slices cascading so that a higher clock frequency could be achieved.

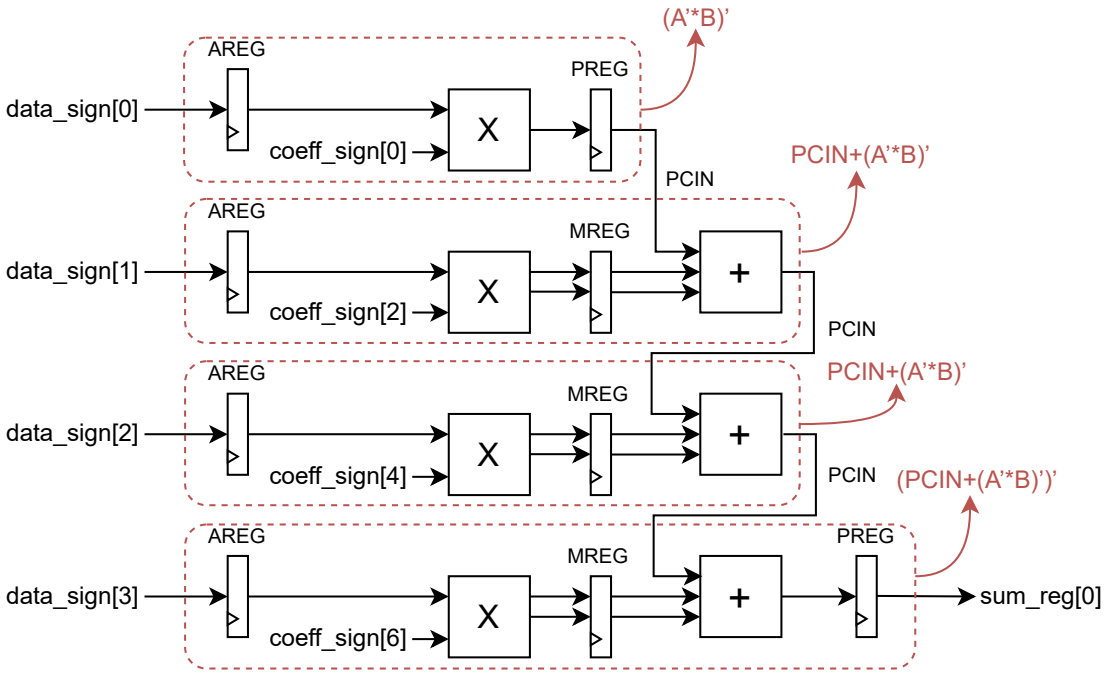

Figure 4.7: Mixer implementation on DSP slices

Chapter 5

Reverb Function Unit

This chapter describes the hardware implementation of the reverb FU. First, it introduces the concept of reverberation and its relevance as an element of an NMP environment. Then, it analyses the chosen reverb algorithm, its implementation and the applied design improvements.

5.1 Introduction to reverb

5.1.1 Acoustic reverberation

The reverb is an audio effect that can be applied to a sound signal to simulate reverberation [39], a physical phenomenon present in every acoustic environment. Reverberation can be defined as the persistence of sound after it is produced by a source and is the main consequence of the reflection of signals by objects in space [40]. Indeed, when a sound is generated, it can get both reflected and absorbed by the different surfaces present in the environment. Reflections add up to one another and make the sound persist in the space, while absorptions make their amplitude decay at a certain rate. The result is an acoustic experience that is not perceived as separate sound events [41].

Reverberation is influenced by different conditions, which can have also an impact on the loudness and timbre of the perceived sound. In particular, one of the most important parameters to consider when talking about this phenomenon is the reverberation time. This is defined as the amount of time required by reflections to reduce to inaudible levels. Different mathematical equations were proposed to describe this parameter, but the most common definition is the time it takes for the sound pressure level to reduce by 60 dB. Depending on the kind of acoustic space considered and its reverberation time, different type of reverb can occur.

Acoustic environments can also present other characteristics which can influence sound perception. In fact, in the same family of reverberation, more phenomena can be found, such as echo and delay. While the reverb is characterized as a random reflection of sounds, the echo is a more structured acoustic perception in which every reflection can be heard as an independent sound. Usually, reverberation occurs when reflections come to the listener in less than 50 ms, while when the time delay is longer distinct echoes are detectable.

5.1.2 History of reverb

Since its first use, the reverb effect has played a key role in audio productions. The Harmonicats' "*Peg o' My Heart*", released in 1947, is credited to be one of the first songs that made intentional use of acoustic reverberation [42]. For the musical piece, the producer Bill Putnam recreated this sound effect by recording instruments into a reverberant environment. The recording setup was fairly simple: a loudspeaker was placed in the studio's bathroom (a typical reverberant space) and the produced sound was then recorded by means of a microphone. This very cheap but efficient solution has been taken as an example by a large number of musicians over the years for its simplicity.

Thanks to those first experiments, in the following years there was an increasing interest in adding artificial reverberation to musical pieces by many recording studios and musicians. This led to the birth of the first echo chambers, special rooms designed to sound as reverberant environments. Even if nowadays other methods are used to create this effect, some of them are still used, such as the one built at Capitol Records' studios [42]. However, echo chambers have several drawbacks: the reverberation produced is not so flexible and editable, because of their fixed structure, and they are also very expensive.

The first solution that aims to solve these kinds of problems was the plate reverb. Introduced in the 50s with the EMT 140 by Elektromesstechnik, this electro-mechanic system could add artificial reverberation by exploiting the mechanical vibrations of large metal plates [39]. Indeed, to produce the desired sound effect, a transducer was used to convert sound signals into vibrations that were then picked up by one or more contact microphones. However, although this solution added some possibility to tune the output sound, it was still very expensive.

The real spread of the reverb effect occurred in the 60s with the introduction of the spring reverb [42]. Its wide use was possible thanks to its cheapness and portability with respect to other available solutions. Moreover, the spring reverb became famous and iconic very quickly because of surf music, popular at that time. To simulate reverberation, this system exploited a set of springs mounted inside a box.

The working principle was similar to the one used in plate reverb, but this time the transducer and the pickup were placed at the ends of each spring.

Finally, thanks to improvements in electronics, starting from the 80s, most of these electro-mechanic devices were replaced by digital reverbs. That is because, in 1976, the EMT 250 was released by Elektromesstechnik. It was the first commercial digital reverb and increased the number of studies on algorithms to add artificial reverberation. In particular, one of the most noticeable solutions was developed in 1999 by Sony, when they released the DRE S777, the first real-time convolution reverb. This device aims at recreating the reverberation of real physical spaces exploiting their impulse response. As known, indeed, if an acoustic space is treated as a Linear Time-Invariant (LTI) system, the output sound can be computed as the convolution operation between the input source and the room impulse response. Thanks to this characteristic, the convolution reverb became widely used to simulate different environmental conditions, especially in film production [39].

5.2 Motivation

As described previously, reverberation is an acoustic phenomenon of fundamental importance. Several studies have confirmed that acoustic reverberation is a critical characteristic for musicians, that often rely on it to interact properly with other players [2][9]. Indeed, the absence of a reverberant environment is perceived as unnatural and can also have a relevant impact on sensitivity to delay. In particular, a research has pointed out that "anechoic conditions lead to a higher imprecision, and a larger asymmetry than reverberant conditions" [4]. All these factors thus highlight how the introduction of a reverb unit can be essential to recreate a convincing immersive experience [41].

In addition, as discussed in subsection 5.1.2, the reverb effect is also highly appreciated by producers as a tool to add creativity to musical pieces. Indeed, nowadays, reverb is one of the most popular and used sound effects. For this reason, a reverb unit could be developed also for merely creative purposes and to meet different musical tastes. For example, one of the thing usually done is to add reverb to simulate imaginary and unrealistic spaces in order to make an instrument or a voice sounds in a particular way.

A hardware approach to the reverb implementation requires for sure considerable effort, but it has some advantages, especially for NMP purposes. First of all, a great improvement in speed with respect to a software solution is possible. This is because a hardware unit can parallelize most of the operations, which would instead be executed sequentially. As a consequence of that, also the introduced latency can

be impacted. Furthermore, this solution can bring some improvements regarding power consumption. Obviously, a hardware approach has also some weaknesses and, in particular, flexibility is one of those. Indeed, a dedicated unit can usually perform fewer functionalities compared to a software one and, to have the same features, usually requires a greater effort to be developed.

5.3 Implementation

During the last decades, several algorithms were developed to simulate reverberation, so a large number of different implementations were possible. To understand which of them would be the most suitable for an NMP application, some considerations were taken into account. First of all, the implemented reverb effect needed to give musicians an experience as immersive as possible, making them feel like they are playing in a real environment. Secondly, users need to have the possibility to customize their experience, being able to control how their reverb effect sounds.

A possible solution to meet these requirements could be the implementation of a convolution reverb, which is one of the best options to simulate the acoustic of existing spaces in real-time. However, this approach has some drawbacks since it is quite inflexible and expensive from a computational point of view. This solution does not allow for a real-time control over reverb sound characteristics, due to the fact that usually a large number of coefficients needs to be recomputed when changing a parameter. For this reason, another group of reverb effects was taken into account, i.e., those based on comb and all-pass filters. The forefather of this family is Schroeder's reverberator, which is also one of the first digital reverbs conceived ever. Over the years, several improvements and similar approaches were considered, but as a starting point, this implementation seemed to have reasonable characteristics for the considered system and to achieve a more natural NMP environment. A summary of other important available solutions can be found in [41], where the pros and cons of each implementation are discussed in detail.

5.3.1 Schroeder's reverberator

Schroeder's reverberator is one of the first algorithms introduced to create a digital artificial reverberation. In 1962, in his seminal paper [43], Manfred Robert Schroeder described different methods and configurations for generating a reverberation by purely electronic means. The main purpose of his studies was to find a way to generate a natural-sounding reverb effect, that can overcome the shortcomings of the reverberators available at that time.

As underlined by Schroeder, the two main aspects to consider when designing

a reverb unit are its frequency response and its echo density. The former is a key point, since it determines if the reverberated sound acquires some unpleasant coloration, i.e. the amplitude of some frequency ranges is altered. Among the proposed methods, Schroeder introduced a flat response reverb unit. The second major concern regards echo density, i.e. the number of echos per second generated by the reverb unit. This is essential to reproduce a natural reverberation, since real rooms can generate a very large number of reflections. The main phenomenon to avoid is fluttering, a condition that occurs when the generated echo density is too low, which leads to output sound more similar to the one a listener can experience with the delay audio effect. Indeed, if this density is too low, distinct echoes can be heard and interpreted as independent acoustic events. Empirical experiments conducted by Schroeder and his team demonstrated that approximately 1000 echos per second are required to avoid fluttering.

All the different configurations proposed by Schroeder, are mainly based on two components: the comb filter and the all-pass one. The former can be considered as the simplest way to produce multiple echos characterized by an exponential sound decay. A block scheme of a generic comb filter can be seen in figure 5.1. As it can be observed, this component is mainly composed of a delay line inserted into a feedback loop and is characterized by two different parameters, that make its behavior tunable. Specifically, τ is the time delay introduced by the delay line, while g is the gain of the feedback loop (always less than 1 to have a stable loop).

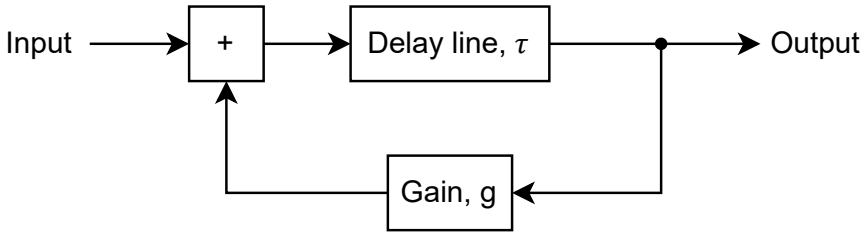

Figure 5.1: Generic comb filter

To better understand the behavior of the comb filter, both its impulse and frequency responses can be analyzed. In particular, the former can be written as:

$$h(t) = \delta(t - \tau) + g\delta(t - 2\tau) + g^2\delta(t - 3\tau) + ... \tag{5.1}$$

where $\delta(t)$ is the impulse function, also known as delta function.

A graphical representation of the comb filter's impulse response can be seen in figure 5.2, where τ=10 ms and g=0.7 were assumed.

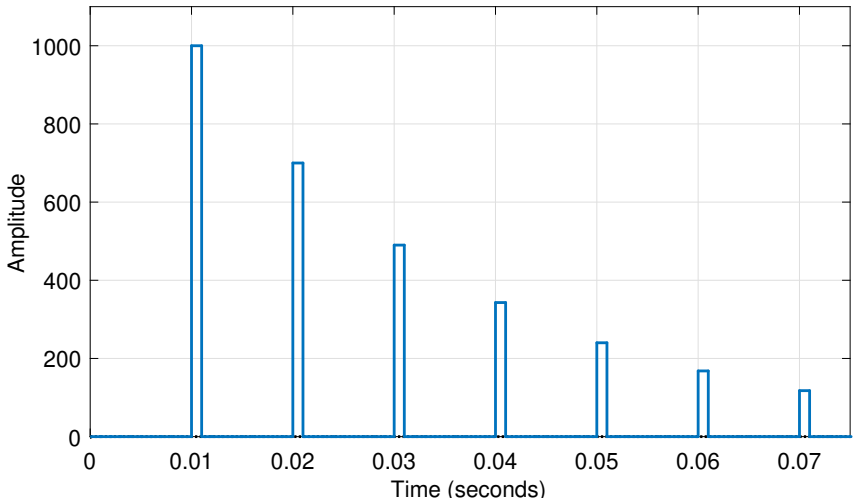

Figure 5.2: Comb filter's impulse response

By exploiting the formula for summing geometric series, instead, the frequency response of the comb filter can be written, and then its magnitude derived as:

$$H(\omega) = \frac{e^{-i\omega\tau}}{1 - ge^{-i\omega\tau}} \;\Rightarrow\; |H(\omega)| = \frac{1}{\sqrt{1 + g^2 - 2g\cos\omega\tau}} \tag{5.2}$$

This magnitude is shown in figure 5.3, where the same parameters reported above were assumed. As it can be observed, the filter's response is not flat, but resembles the teeth of a comb (after which it is named), so it can influence the output sound.

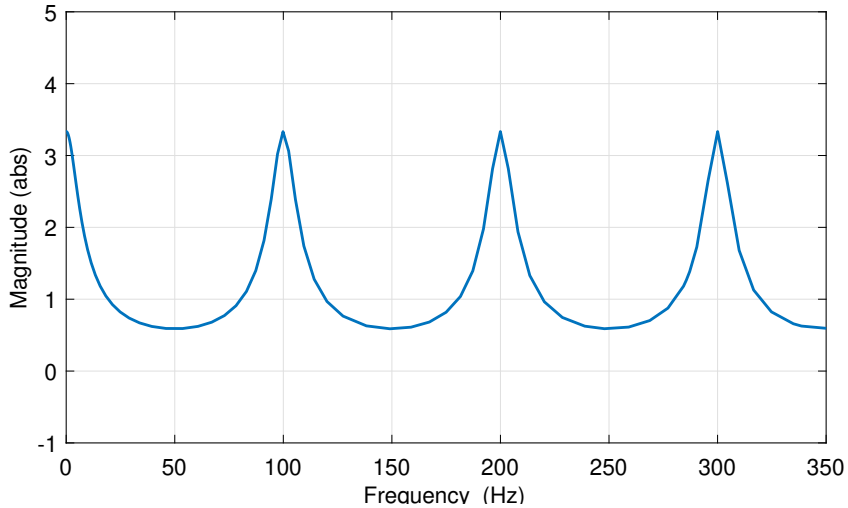

Figure 5.3: Comb filter's frequency response

The second main element introduced by Schroeder is the all-pass filter. This component is again capable of generating several echos, but this time, it features also a flat frequency response, which makes it very suitable to avoid unpleasant coloration of sounds. A block scheme of a generic all-pass filter can be seen in figure 5.4. As it can be noticed, to obtain an equal response for all frequencies, a mixture of the delayed sound and the undelayed one is exploited.

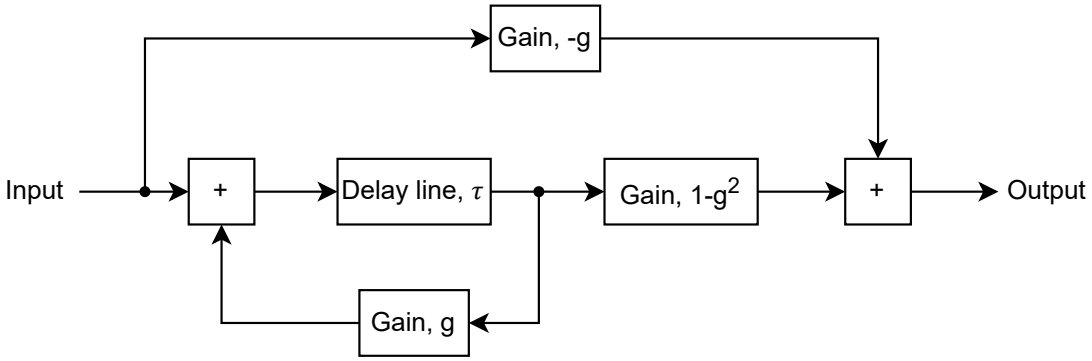

Figure 5.4: Generic all-pass filter

In this case, the generic impulse response of the filter can be written as:

$$h(t) = -g\delta(t) + (1 - g^2)[\delta(t - \tau) + g\delta(t - 2\tau) + ...] \tag{5.3}$$

A graphical representation of this impulse response is reported in figure 5.5, where again τ=10 ms and g=0.7 were assumed.

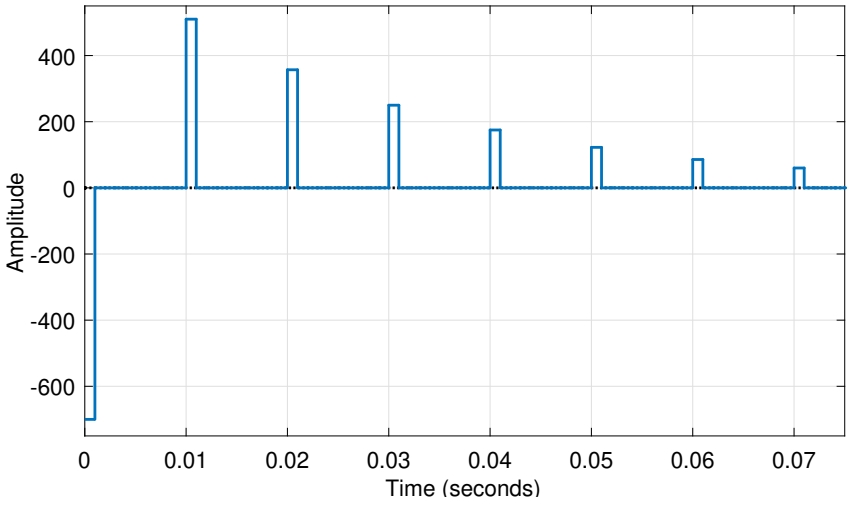

Figure 5.5: All-pass filter's impulse response

Regarding the all-pass filter's frequency response and in particular its magnitude, it is possible to obtain:

$$H(\omega) = e^{-i\omega\tau} \cdot \frac{1 - ge^{i\omega\tau}}{1 - ge^{-i\omega\tau}} \;\Rightarrow\; |H(\omega)| = 1 \tag{5.4}$$

As it can be seen from figure 5.6, the filter exhibits a flat response, thereby introducing a time delay without altering the sound.

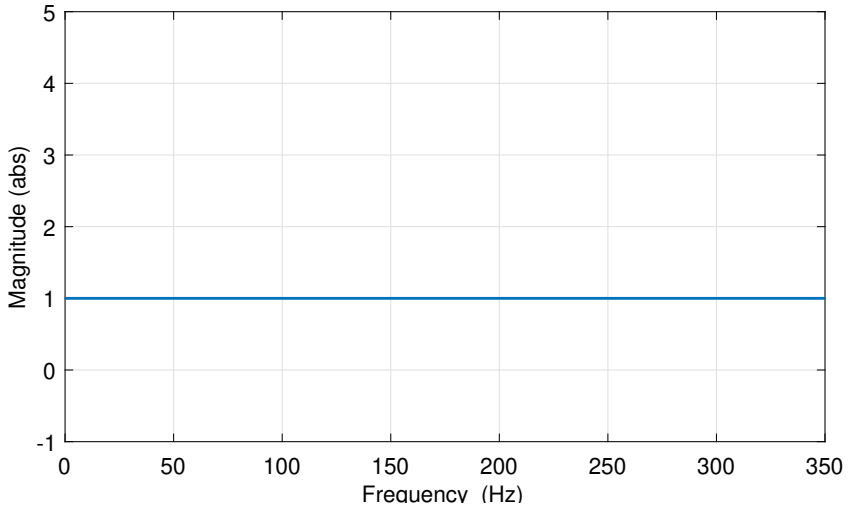

Figure 5.6: All-pass filter's frequency response

Schroeder proposed several configurations to solve the problems related the frequency response and the echo density of the reverberator. Among all of them, the one depicted in figure 5.7 was actually chosen to be implemented as a FU. According to Schroeder, this configuration, also named "the comb filter approach", has an output sound that is indistinguishable from the one of real rooms. Indeed, it can be demonstrated that it features an exponential decay, avoiding the fluttering phenomenon by employing different delay lines. As it can be observed, however, this architecture lacks a flat frequency response, since it includes comb filters. Schroeder explained that this is not a problem, because physical rooms are usually characterized by irregular responses. In particular, the fact that comb filters do not have a flat response can be made imperceptible if the density of their peaks and valleys is high enough. For this reason, four different comb filters in parallel could be used to obtain a result very similar to the one of real environments. Furthermore, since the echo density was still insufficient, two all-pass filters were added in series to the architecture. In this way, the number of echos could be amplified without impacting the overall frequency response. As it can be observed, the final output of the unit is then computed as the sum between the direct input sound and the

reverberated one.

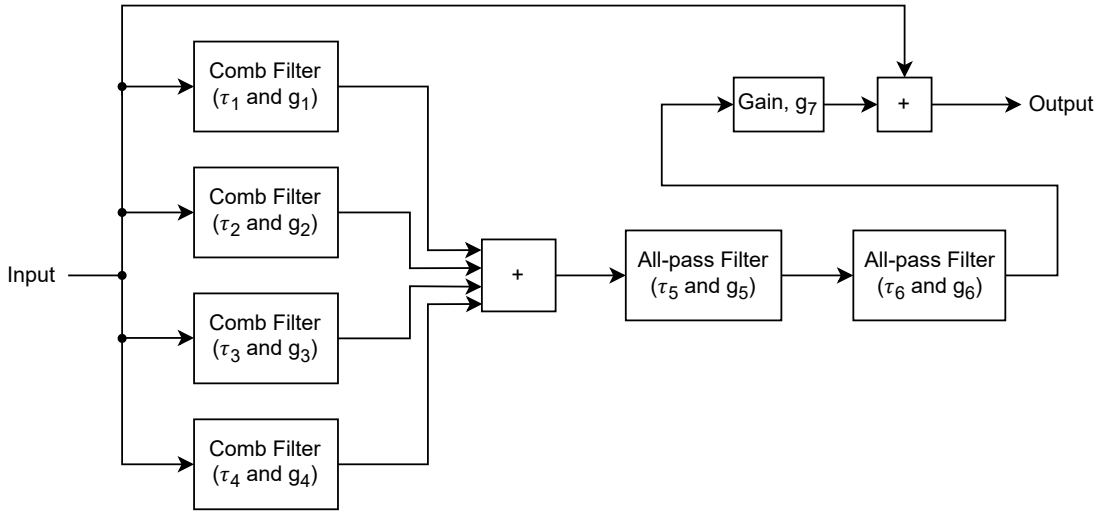

Figure 5.7: Schroeder's reverberator

In his analysis, Schroeder proposed also a way to determine the different parameters for each filter. In particular, he suggested spreading the time delay values from τ_1 to τ_4 between 30 and 45 ms, while setting τ_5=5 ms and τ_6=1.7 ms. Regarding instead the gain parameters, and in particular the comb filters' ones, he explained how they can be used to tune the reverberation time of the architecture, by employing the equation:

$$T = \frac{3\tau_n}{-\log|g_n|} \;\Rightarrow\; g_n = 10^{-\frac{3\tau_n}{T}} \tag{5.5}$$

where T is the desired reverberation time.

Since real rooms are usually characterized by a reverberation time of about 1 second, this value was also chosen for the FU implementation. In this way, the gain parameters shown in table 5.1 were computed.

Filter	τ_n	g_n
Comb 1	30 ms	0.813
Comb 2	35 ms	0.785
Comb 3	40 ms	0.759
Comb 4	45 ms	0.733

Table 5.1: Comb filter's parameters

For what concern the gain g_5 and g_6, Schroeder suggested to conventionally set them equal to 0.7, while g_7 could be chosen arbitrarily since it determines the amount of reverberated sound which is going to be heard in output with respect to the direct one. In the hardware implementation, gain g_7 will be configurable from outside and referred to as wet parameter.

5.3.2 OSAL

The implementation of the reverb FU was carried out by adopting a different workflow than the one used for audio mixing. Indeed, in this case, the first step of the TCE design flow was skipped. The development of an initial firmware version of the reverb was not performed because fractional numbers were required to represent the filters' gain coefficients. Since, as already mentioned, the architecture of the audio processor is based on 32-bit integers, this step could not be performed.

Before proceeding with the hardware implementation, however, a new module called *REVERB* and its custom operation *REVERB_ADD* needed to be included in the OSAL. In particular, this operation can be used to add the reverb effect on input audio samples, while collecting reverberated outputs and can be exploited in firmware using the macro:

REVERB_ADD(INPUT_CH, WET_PARAM, OUTPUT_CH)

where INPUT_CH and OUTPUT_CH are respectively input and output signed numbers containing the corresponding audio samples, while WET_PARAM is an unsigned input number associated with the wet parameter. As mentioned before, this is the value that determines the amount of reverberated sound to be heard in output and, for this reason, it can range from a minimum of 0 (no reverb added) to 1 (no direct sound in output, only reverberated one). However, since the architecture can not deal with fractional numbers, this value can actually range from 0 to 256, being then rescaled in the FU according to a fixed-point representation that will be explained in the next subsection.

After adding the module and its custom operation to the OSAL, a new dedicated FU called *REVERB_0* was added to the processor's architecture. In this way, it was possible to design the *test_reverb.c* firmware. This program aims at testing the new operation by exploiting a loopback on I2S. Specifically, according to the position of switch 0 (SW0), the reverb effect can be turned ON/OFF, while acting on switch 1 (SW1), the user can change the wet parameter supplied to the FU. When the status of SW1 is equal to 0, the value 64 is provided (which means 0.25 in the internal fixed-point representation), while when its status is equal to 1, 192 is given as a parameter (0.75 in the internal fixed-point representation).

5.3.3 Hardware implementation

The hardware implementation of the reverb FU started with the design of the top level wrapper called *reverb_wrap.sv*, shown in figure 5.8. As it can be seen, this unit is mainly composed by another component called *Reverb*, which is the one that is actually taking care of the algorithm developed by Schroeder and that will be explained in detail later. It can be also observed that the FU has different input ports. In particular, the control ones are the same as for the mixing unit, but this time *opcode_i* is not present. This is because, as mentioned in subsection 5.3.2, the unit features only one custom operation. The reverb FU has also two 32-bit width ports: *data_i* and *wet_param_i*. The former is the one related to input audio samples, while the latter refers to the wet parameter. Even if both these ports exploit 32 bits, they have different numerical representations and, for this reason, conversions are applied to the input and output of the FU. In particular, *data_i* requires a 32p0 value, while *wet_param_i* has a 24p8 fixed-point representation. Regarding the output port *data_o*, the same considerations made on *data_i* apply.

As for the mixing feature design, input and output registers were implemented. Also in this case, this was done in order not to increase critical paths that can exist outside of the unit.

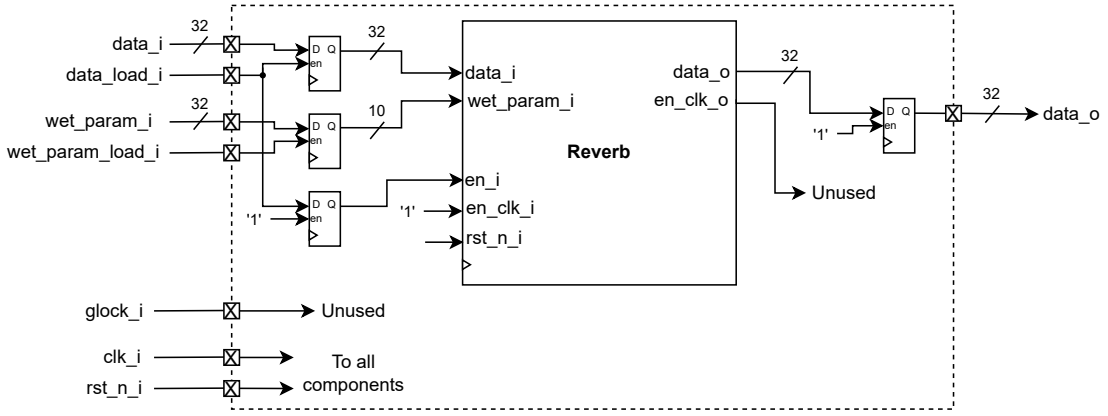

Figure 5.8: Reverb wrap

The design of the *Reverb* component was performed using a tool called HDL coder [44], a MATLAB add-on to generate HDL code from Simulink models. This software permits to translate a graphical representation of an architecture into a hardware logic, written both in VHDL or in Verilog language. It is a very useful tool to increase productivity and make several enhancements to a design. Moreover, since the model was built on Simulink, it was possible to hear the output sound of the reverb before actually implementing it on FPGA.

The reverb model on Simulink was implemented by exploiting a hierarchical design. First, the two main components of Schroeder's reverberator were developed. The comb filter model is shown in figure 5.9, while the all-pass one in figure 5.10.

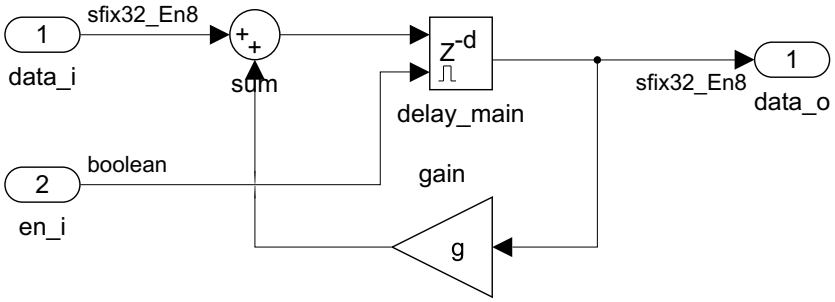

Figure 5.9: Comb filter (Simulink model)

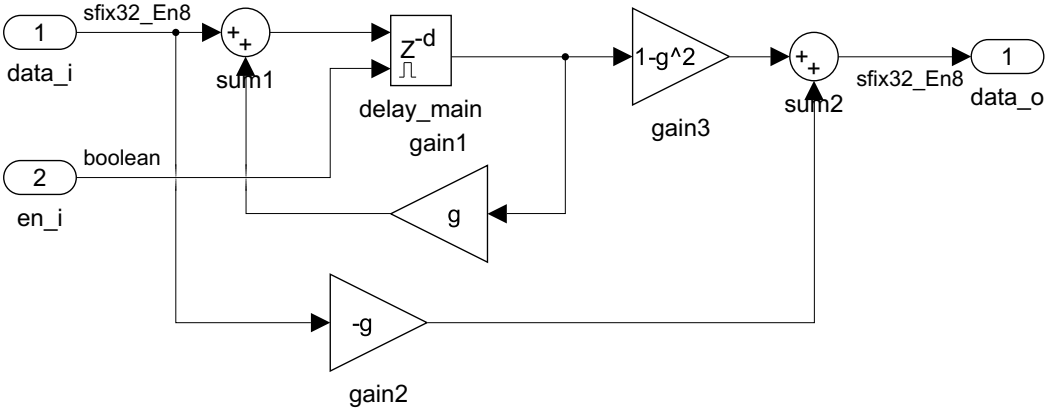

Figure 5.10: All-pass filter (Simulink model)

It can be observed that the Simulink models reflect the ones in figures 5.1 and 5.4. In particular, the gain blocks were realized using the corresponding operators, while to design the delay lines, shift register components were exploited. In particular, to make each filter of the unit behave according to the parameters suggested by Schroeder, the length of each delay line in terms of number of samples was computed. This was done by employing the following expression:

$$d_i = f_s \cdot \tau_i \tag{5.6}$$

where f_s is the sampling frequency and τ_i is the desired time delay.

The obtained values can be seen in table 5.2, where also the resulting time delay of each component is reported. Indeed, since the number of registers can assume

only integer values, the latency introduced by each filter is quantized. Despite of that, time delays are pretty much the same as those expected and the maximum deviation introduced by this design procedure is 0.2% (percentage concerning τ_5, so all-pass filter 1, while for other components is lower).

Filter	d_i	τ_i (quantized)
Comb 1	1323	30 ms
Comb 2	1543	34.99 ms
Comb 3	1764	40 ms
Comb 4	1984	44.99 ms
All-pass 1	220	4.99 ms
All-pass 2	75	1.7 ms

Table 5.2: Time delay parameters for each filter

Another aspect to be considered for each filter is the quantization of gain parameters. In fact, as known, real numbers may not be characterized accurately using a fixed-point representation and some approximations can be introduced. Moreover, to save FPGA resources, the multiplicative coefficients were represented on 9 bits using a 1p8 representation. The outcomes of this design choice can be seen in table 5.3. It can be noticed that, this time, the maximum deviation introduced is 0.18% (percentage concerning gain g_4 and lower for other components).

Filter	g_i (quantized)
Comb 1	0.8125
Comb 2	0.78515625
Comb 3	0.7578125
Comb 4	0.734375
All-pass 1	0.69921875
All-pass 2	0.69921875

Table 5.3: Gain parameters for each filter

By combining all the developed components, it was possible to achieve the reverb structure shown in figure 5.11. As it can be noticed, this architecture is very similar to the one depicted in figure 5.7, but it has some differences. First, arithmetic shifts were exploited after every sum operation to reduce signal dynamics. Indeed, by adding together four different delayed samples, saturation can occur. Second, to properly mix the reverberated and the direct sound, two different multipliers were exploited to achieve again signal dynamics comparable to that of the input.

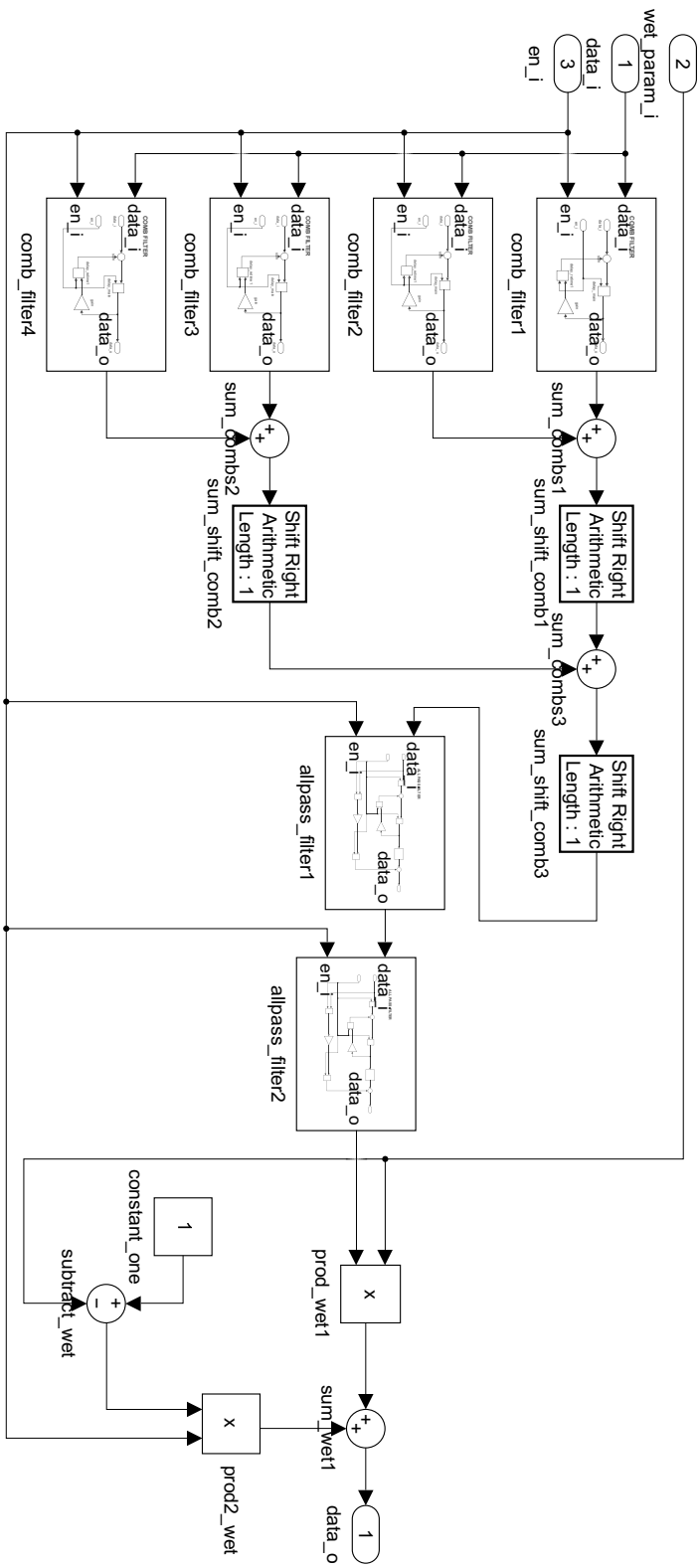

Figure 5.11: Reverb core (Simulink model)

The main issue related to this first version of the architecture is the maximum achievable clock frequency. Indeed, as it can be observed, the *Reverb* component is characterized by very long paths starting from the output of each comb filter and arriving at the *data_o* output. Since the goal was to reach a clock frequency of 100 MHz, some improvements needed to be made. The first optimization performed was register retiming. This technique can be exploited to optimize timing paths in a digital circuit by moving registers, without affecting its behavior. In Simulink models, it was applied on shift registers, leading to the components shown in figures 5.12 and 5.13. In this way, the critical path of the circuit was strongly reduced.

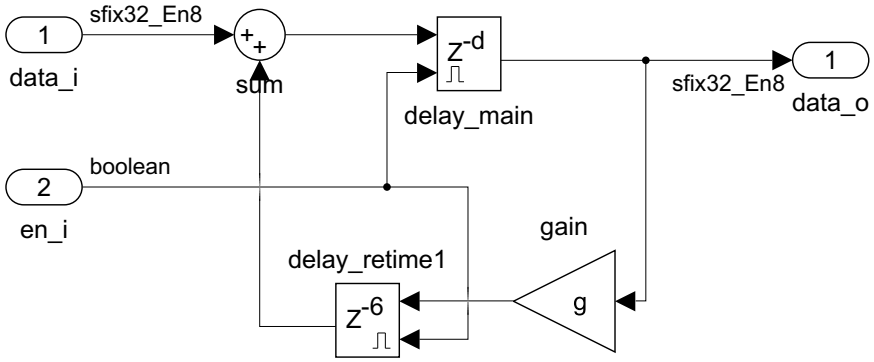

Figure 5.12: Comb filter improved (Simulink model)

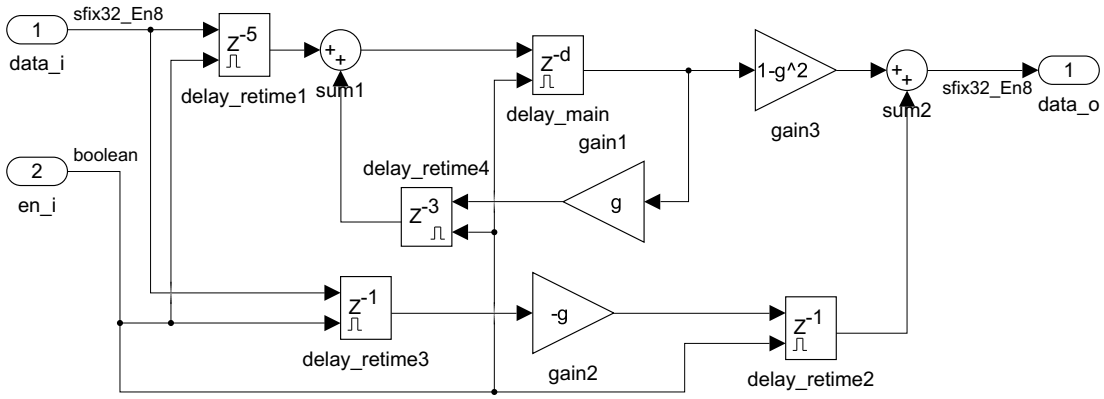

Figure 5.13: All-pass filter improved (Simulink model)

Then, the second optimization applied was pipelining, a technique that can be used to further reduce timing paths, but at the expense of latency. As it can be seen from figure 5.14, the final architecture achieved features a pipeline stage to split the timing path of *subtract_wet* and *prod2_wet* in two different contributes.

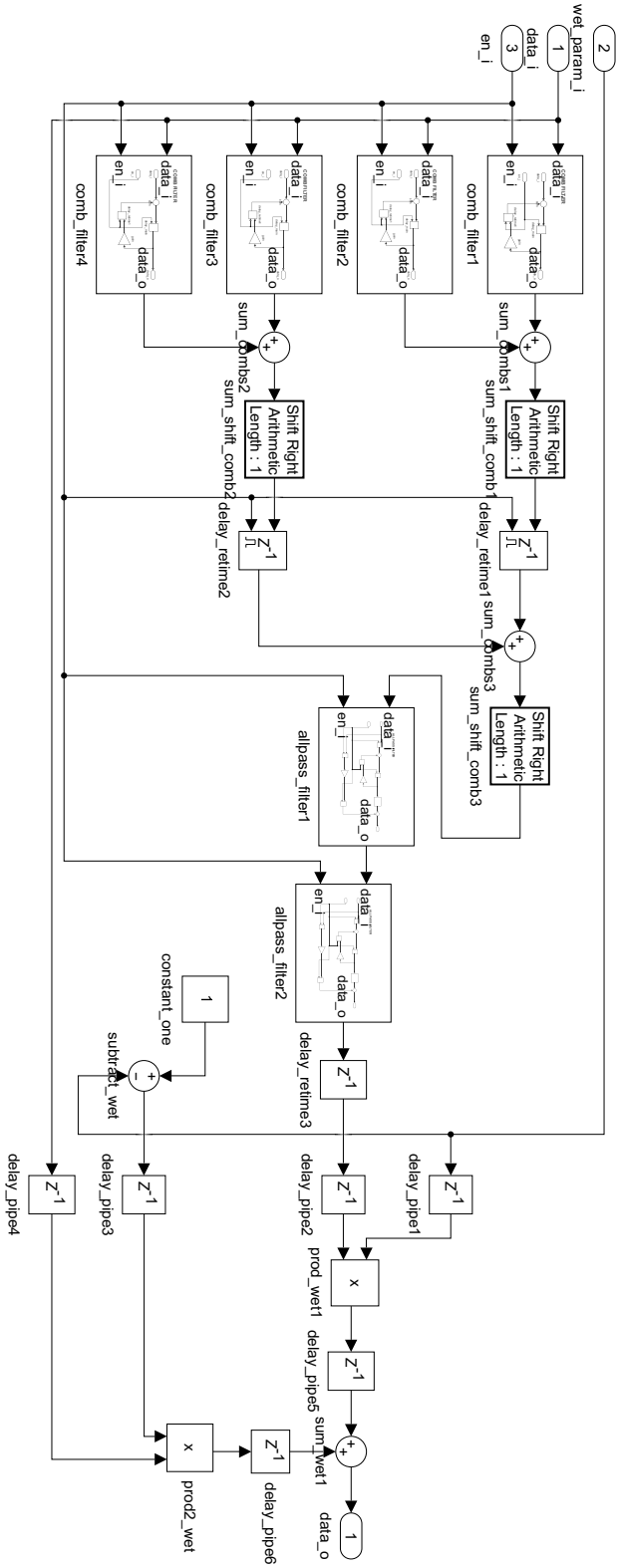

Figure 5.14: Reverb core improved (Simulink model)

5.3.4 Hardware simulation

As already done for the mixing unit, also the hardware implementation of the reverb needed to be simulated. In this case, however, a different procedure was adopted to check the correctness of the computed output samples. This alternative verification approach is reported in figure 5.15. As it can be seen, this time a MATLAB script called *export_data.m* was used to generate both the input file (*reverb_input.txt*) and the output golden reference one (*reverb_output_gold.txt*). Then, to simulate the implemented DUT on Vivado, the same input values were provided to *reverb_tb.sv*, a testbench in SystemVerilog. Thanks to this, output values were written on *reverb_output.txt* and were finally compared to the reference ones using a C program called *reverb_checker.c*.

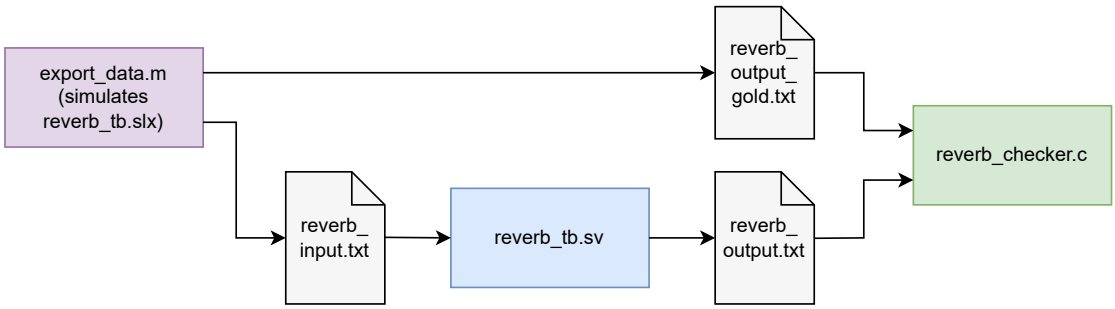

Figure 5.15: Reverb FU simulation approach

As mentioned before, the first element of the whole simulation procedure is *export_data.m*. This MATLAB script performs two main activities: it simulates the behavior of the reference reverb on Simulink employing a testbench model (*reverb_tb.slx*) and it exports the input and output data on different text files. However, the real core of this script is the Simulink testbench model itself, shown in figure 5.16. As it can be observed, the designed reverb module is the DUT of this testbench and several other blocks are present to properly evaluate its behavior. First of all, the *data_i* input is provided by means of a multimedia file reader, which takes an audio file called *RockGuitar.wav* (sample file from MATLAB library) and converts it to raw data. In particular, it can be seen that its audio samples are represented on 16 bits, in stereo mode (two channels) and with a sampling frequency of 44.1 kHz. To make these input data consistent with the one expected from the DUT, two other blocks are needed: a selector, to retrieve only one audio channel from the .wav file, and a converter, to adapt data to the reverb internal fixed-point representation. Then, the *wet_param_i* input is set to a constant value of 0.25. To make the simulation more complex, this value can be made time-dependent in future developments. Next, the *en_i* input port is connected instead to the output of a pulse generator, providing an enable signal every 44.1 kHz. Also in this

case, a converter is needed since the output data type of the generator is a double value, while the DUT is expecting a boolean. Finally, the *data_o* output values are converted again to a 16 bits representation and are written to *output_reverb.wav* using a multimedia file writer. Moreover, in the testbench model, two blocks to export data on the MATLAB workspace and a scope element to see the simulated waveforms in real-time are present. To test the DUT, the simulation time was set to 1 second, so that a total number of 44100 samples were computed.

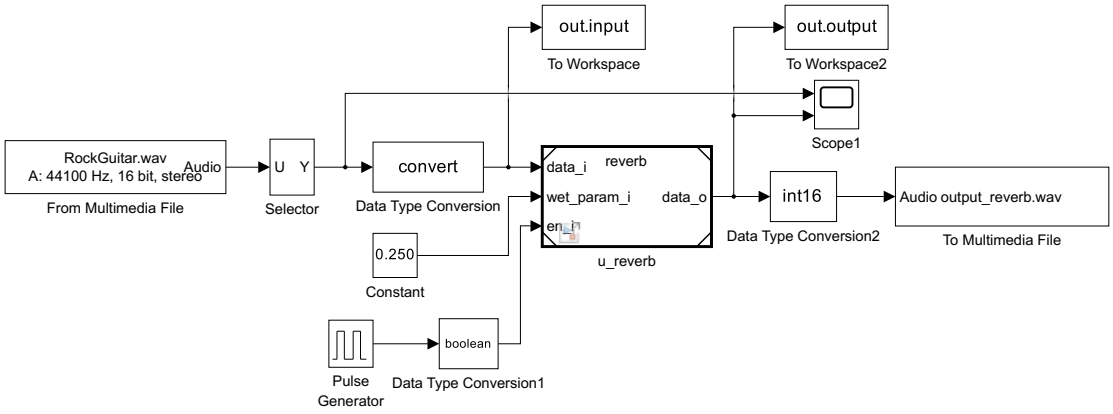

Figure 5.16: Reverb core testbench (Simulink model)

The second component employed for the hardware simulation is *reverb_tb.sv*, the HDL testbench to be simulated on Vivado. This module reads the *reverb_input.txt* file and provides appropriate control signals to the DUT, that in this case is the designed hardware FU (*reverb_wrap.sv*). In the meanwhile, *reverb_tb.sv* also writes the computed output samples to the output file *reverb_output.txt*.

Then, to check the correctness of the hardware implementation with respect to the Simulink model, *reverb_checker.c* was executed. This program compares the output samples from *reverb_output.txt* to the ones from *reverb_output_gold.txt* and computes some statistics. Before going to the results, it is necessary to make a last clarification. To properly check the accuracy, the output values generated by the Simulink model are not yet rounded and are still including their fractional part, while the ones from the hardware unit are rounded to integer and represented on 24 bits (as required by the top level architecture). For this reason and since floor rounding is applied, a maximum deviation (difference compared to reference values) of 1 LSB is acceptable. In particular, for the performed simulation, it was found that the maximum deviation from the golden values is equal to 0.996 and the mean difference is equal to 0.4996. As it can be observed, these results confirmed the expected ones, since, as known, the floor function introduces a maximum difference of 1 and a mean error of about 0.5 on the rounded output values.

5.3.5 Synthesis and implementation on FPGA

After running the hardware simulations, the synthesis and implementation steps were performed again using Vivado. To improve resource utilization on the FPGA, some optimizations were performed on the delay lines and the arithmetic operators of the reverb FU.

The first enhancement regards the implementation on FPGA of the large delay lines contained in the designed reverb unit. To do so, the Artix-7 family's FPGAs feature dedicated cells to design and optimize shift register, called SRL16E and SRLC32E. These resources are integrated into look-up tables and permit to efficiently implement 1-bit shift registers without using flip-flop resources. However, what Vivado usually does without being instructed properly is to infer delay registers as distributed logic, instead of exploiting dedicated resources. This is because, if some precise coding guidelines are not followed, Vivado is unable to understand the designer's purposes, causing occasionally synthesis issues and some difficulties to met the timing requirements. For this reason, Vivado was instructed to use dedicated shift registers by means of *mevo_constraints.xdc*, the synthesis file used to define the constraints that the synthesizer needs to meet. To do so, the following line was added:

set_property SHREG_EXTRACT yes [get_cells core/fu_REVERB_0/u_reverb]

The second improvement performed regards the use of DSP slices for the arithmetic operations. Also in this case, Vivado needed to be instructed on how to infer these blocks because adders, subtractors, and accumulators are by default implemented using generic logic. This will not happen only if a multiplication operation is involved too. For this reason, to force Vivado to use DSP slices also for additions, another line was added to *mevo_constraints.xdc*:

set_property USE_DSP yes [get_cells core/fu_REVERB_0/u_reverb]

Thanks to both these improvements, it was again possible to meet the established timing requirement and achieve a clock frequency of 100 MHz.

After performing the synthesis, the inferred logic was analyzed by taking a look again at the *runme.log* file generated by Vivado. The results about shift registers utilization are provided in table 5.4. As it can be seen, the delay lines contained in each comb and all-pass filter are mapped both into an SRL16E or an SRLC32E block. In particular, the largest ones are inferred as SRLC32E, while the others exploit SRL16E cells. Also, it is possible to recognize that the comb filter's delays are correctly divided into two contributions: the first is the main delay, while the second is due to the registers re-timed during the hardware design on Simulink.

Module name	Length	Width	SRL16E	SRLC32E
u_comb_filter1	1317	32	0	1344
u_comb_filter1	6	32	32	0
u_comb_filter2	1537	32	0	1536
u_comb_filter2	6	32	32	0
u_comb_filter3	1758	32	0	1760
u_comb_filter3	6	32	32	0
u_comb_filter4	1978	32	0	1984
u_comb_filter4	6	32	32	0
u_allpass_filter1	216	32	0	224
u_allpass_filter2	73	32	0	96

Cell	Total number
SRL16E	128
SRLC32E	6944

Table 5.4: Static shift register reports for the reverb FU

Furthermore, the retrieved results about DSP blocks utilization are shown in table 5.5.

DSP Mapping	A Size	B Size	C Size	P Size	MREG	PREG
C+A:B	30	18	48	33	0	0
C+A:B	30	18	48	33	0	0
C+A:B	30	18	48	33	0	0
C+A:B	30	18	9	11	0	0
C'+A':B'	30	18	48	33	0	0
A"*B"	17	18	0	48	0	0
PCIN»17+A"*B"	30	18	0	25	0	0
A"*B'	17	18	0	48	0	0
PCIN»17+A"*B'	30	18	0	25	0	0

Table 5.5: DSP final report for the reverb FU (the ' indicates corresponding REG is set)

It can be observed that Vivado maps the sum operations between the comb filter's outputs into three different blocks (first three lines), then the wet parameter subtraction and the final sum between direct and reverberated data into other two DSP slices (lines four and five). The remaining blocks are used to implement the

two multipliers present in the design. As it can be seen, two DSPs are exploited for each product operation since the multiplier contained in a slice can have a maximum dimension of 25 x 18 bits.

Chapter 6

System optimizations and final results

This chapter describes some of the most important improvements made to the system and, in particular, to the audio processor. To achieve better performances, both architectural and firmware optimizations were introduced and can be found in detail in the next sections. Finally, also the most important results provided by Vivado on the final FPGA implementation are analyzed.

6.1 Memory optimization

The first optimization performed on the audio processor is the one related to the implementation of memories. In the preliminary architecture, these components were implemented by means of two HDL files called *inst_mem_logic.vhd* and *synch_sram_synth.vhdl*, which described the instruction and data memory respectively. In particular, *synch_sram_synth.vhdl* was a hand-written file, while *inst_mem_logic.vhd* was generated by TCE exploiting the `generatebits` command. This tool was also used to automatically create the contents of each memory, called *mevo_proc_imem_pkg.vhdl* for instructions and *mevo_proc_data.img* for data.

The optimization of these components consisted of replacing them with two customizable memory elements taken from the Vivado IP catalog. Specifically, the Block Memory Generator tool was used to implement the new components. The instruction memory was designed as a single port RAM, with a word width of 176 bits (the length of each instruction in the TTA processor) and a depth of 1024 words (an arbitrary value, large enough to contain the different firmware developed so far). In this way, a 10-bit addressable memory module called *inst_mem.vhd* was

obtained. The data memory, instead, was realized as a single port RAM, with a word width of 32 bits (the data width in the TTA processor) and a depth of 1024 words (again an arbitrary value, sufficient for the considered firmware). By doing so, a 10-bit addressable memory called *data_mem.vhd* was obtained. However, the content of these IP modules still needed to be initialized (in particular using a .coe file). This was possible by coming back to the TCE tool and changing some of the options provided to the `generatebits` command. In this way, two memory initialization files called *mevo_proc_inst.coe* and *mevo_proc_data.coe* were generated and were then provided to Vivado for the synthesis process.

This new approach to memory implementation can introduce several benefits for the architecture. First of all, FPGA resources can be better exploited. Indeed, as it can be observed comparing tables 6.1 and 6.2, the large number of LUTRAMs used initially to implement the memories (43.76% of the total availability) can be saved by using BRAMs instead (only 6 more of them are enough). A LUTRAM block, also known as distributed RAM, is a memory device contained inside FPGA's Lookup Tables (LUTs) and is the best way to design a very fast memory component. However, if a large memory needs to be implemented, BRAM usually performs better in terms of speed, since there is no need to implement complex interconnections between the exploited LUTs. This is because BRAMs are dedicated RAM blocks, that do not consume any additional LUT.

Resource	Utilization	Available	Utilization %
LUT	12117	63400	19.11%
LUTRAM	8314	19000	43.76%
BRAM	3	135	2.22%

Table 6.1: Memory resource utilization before optimizations

Resource	Utilization	Available	Utilization %
LUT	3519	63400	5.71%
LUTRAM	122	19000	0.64%
BRAM	9	135	6.67%

Table 6.2: Memory resource utilization after optimizations

Another additional improvement introduced by the use of BRAMs is the speed up of both synthesis and implementation processes (since a lower number of complex interconnections needs to be routed). The only drawback that this implementation choice can bring is that the architecture becomes less portable on other platforms and devices that are not specifically designed by Xilinx.

6.2 Latency optimization

Since the latency introduced by the system is a crucial parameter to be minimized for an NMP application, some optimizations were made in this regard as well. In particular, the improvements introduced refer to the time delay of audio samples and are not taking into account MIDI messages. Indeed, only the I2S communication is considered so far, but a similar approach can be used also to improve both UART and SPI interfaces.

As a case study to evaluate the latency introduced by the system, the *test_loopback.c* firmware was chosen. As already explained in subsection 3.4.4, this program is used to test the loopback capabilities of the system, so to verify if an input sound can be acquired and played back without being modified. In this process several factors contribute to the total latency introduced: the sound needs to be digitized, transferred using I2S, managed by the processor, transferred back and then converted again into an analog signal. For this reason, the overall latency introduced by the loopback test can be split into different contributions:

$$T_{lat} = T_{ADC} + T_{I2S} + T_{proc} + T_{I2S} + T_{DAC} \tag{6.1}$$

where T_{ADC} and T_{DAC} are the time delays associated with audio conversions, T_{I2S} is the one regarding data transfers (considered two times since data are going back and forth) and T_{proc} refers to the latency introduced by the processor.

The first contributions that can be analyzed are T_{ADC} and T_{DAC}. Determining these delays is very important for the total latency since they represent a significant part of it. This is due to the fact that both signal conversions, featured in the audio board, are based on Delta-Sigma modulation, a method used to reduce noise and increase the accuracy of samples. To make this technique work, a digital filter is always present in the conversion structure, being responsible for the introduction of a considerable delay. Indeed, as Linear Time-Invariant (LTI) system, this component can add a delay associated with its phase response, called group delay and defined as:

$$Group\ delay\ (\omega) = -\frac{\mathrm{d}\Phi(\omega)}{\mathrm{d}\omega} \tag{6.2}$$

where $\Phi(\omega)$ is the phase response of the system and ω is the angular frequency.

The digital filters exploited in this conversion process are low-pass ones and usually present a sinc function response in the frequency domain. For this reason, they also show a linear phase and as a consequence a constant group delay, that can be generally computed in a deterministic way as a function of the sampling frequency

(that in the considered case is 44.1 kHz). To take into account the latency introduced by both the ADCs and the DAC, the information on group delay was retrieved from their datasheets. In particular, it was found that:

$$\begin{aligned} T_{ADC} &= \frac{17.4}{f_s} = \frac{17.4}{44.1\text{ kHz}} = 394.56\text{ µs} \\ T_{DAC} &= \frac{20}{f_s} = \frac{20}{44.1\text{ kHz}} = 453.51\text{ µs} \end{aligned} \tag{6.3}$$

The second contribution to the overall latency is the one associated with the I2S communication protocol and in particular with the time required to transfer an audio sample. In this case, the delay introduced can be computed as the product between the BCLK period and the number of bits for each I2S transfer. Otherwise, it can be also expressed as the LRCLK semi-period, getting to:

$$T_{I2S} = T_{bclk} \cdot 32 = \frac{T_{lrclk}}{2} = 11.34\text{ µs} \tag{6.4}$$

The latency introduced by the audio processor is the last contribution to be considered. Indeed, the acquired samples need to be correctly retrieved from the I2S receiver and sent to the transmitter. In the initial version of *test_loopback.c*, this procedure takes a lot of time due to the misuse of the I2S transmission FIFO. The program was supposed to fill this audio buffer with zeros until SW0 is in OFF position, but this was done without respecting the correct timing behavior of the I2S communication. For this reason, the delay introduced for each sample depended on the FIFO length. This latency can be computed as:

$$T_{proc} = T_{lrclk} \cdot l_{FIFO} = 22.68\text{ µs} \cdot 64 = 1.45\text{ ms} \tag{6.5}$$

By looking at the different latency contributions described so far, it is clear that *T_proc* is the only parameter to act on to reduce the total delay. In particular, what could be done is to manage properly the I2S transmission. The firmware was then modified to ensure an appropriate timing (transmission FIFO was filled every 44.1 kHz). In this way, the following theoretical delay can be achieved:

$$T_{proc} = 3 \cdot \frac{T_{lrclk}}{2} = 3 \cdot 11.34\text{ µs} = 34.02\text{ µs} \tag{6.6}$$

Summing together all these time delays, it is possible to compute the total theoretical latency associated to the initial version of *test_loopback.c* and the optimized one:

$$\begin{aligned} T_{lat} &= T_{ADC} + T_{I2S} + T_{proc} + T_{I2S} + T_{DAC} = 2.31\text{ ms} \\ T_{lat,opt} &= T_{ADC} + T_{I2S} + T_{proc} + T_{I2S} + T_{DAC} = 0.89\text{ ms} \end{aligned} \tag{6.7}$$

To evaluate the latency improvements introduced by the optimization explained above, a real-world test was performed. By using a tool to measure audio latency, it was possible to compare the first version with the optimized one. Results show that the initial loopback firmware had a total latency of 2.115 ms, while the improved version can achieve 0.859 ms. These values are subject to an uncertainty of 0.005 ms, since the sampling time of the measurement tool is equal to 192 kHz. As it can be observed, these outcomes almost reflect the expected ones and give the system a decrease in latency of 59.4%.

6.3 Top level restructuring

Some other improvements were performed on the architecture, before optimizing the firmware. For example, to simplify the audio processor representation, its top level structure was restructured. In particular, one of the intermediate layers of hierarchy was removed, leading to the final architecture reported in figure 6.1. As it can be seen, all the main components of the processor (already described in subsection 3.4.2) are contained into a single design module called *top.vhdl*.

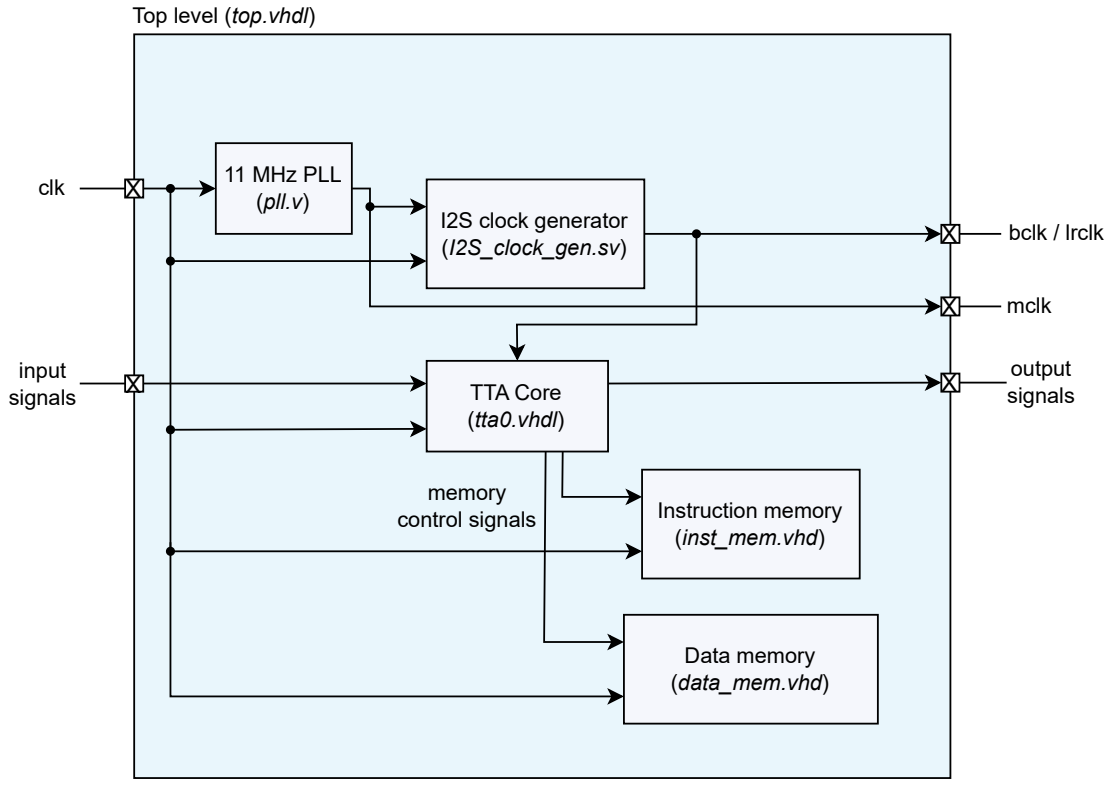

Figure 6.1: Audio processor final top level block diagram

6.4 Main firmware restructuring

In addition to architectural optimizations, also the main firmware running on the audio processor was improved. In particular, an approach based on a software FSM was considered to make the code more clear and to bring some advantages for future developments. A state machine is usually a very flexible way to model the behavior of a system and its code can be highly maintainable. Moreover, functionalities can be quite isolated, so that new features can be introduced without the need to consider already existing ones. This can be a great advantage, especially in a project run by several people over time.

The new version of the main firmware, *mevo_main.c*, is replacing the old one called *main_final.c*. Its state diagram can be seen in figure 6.2.

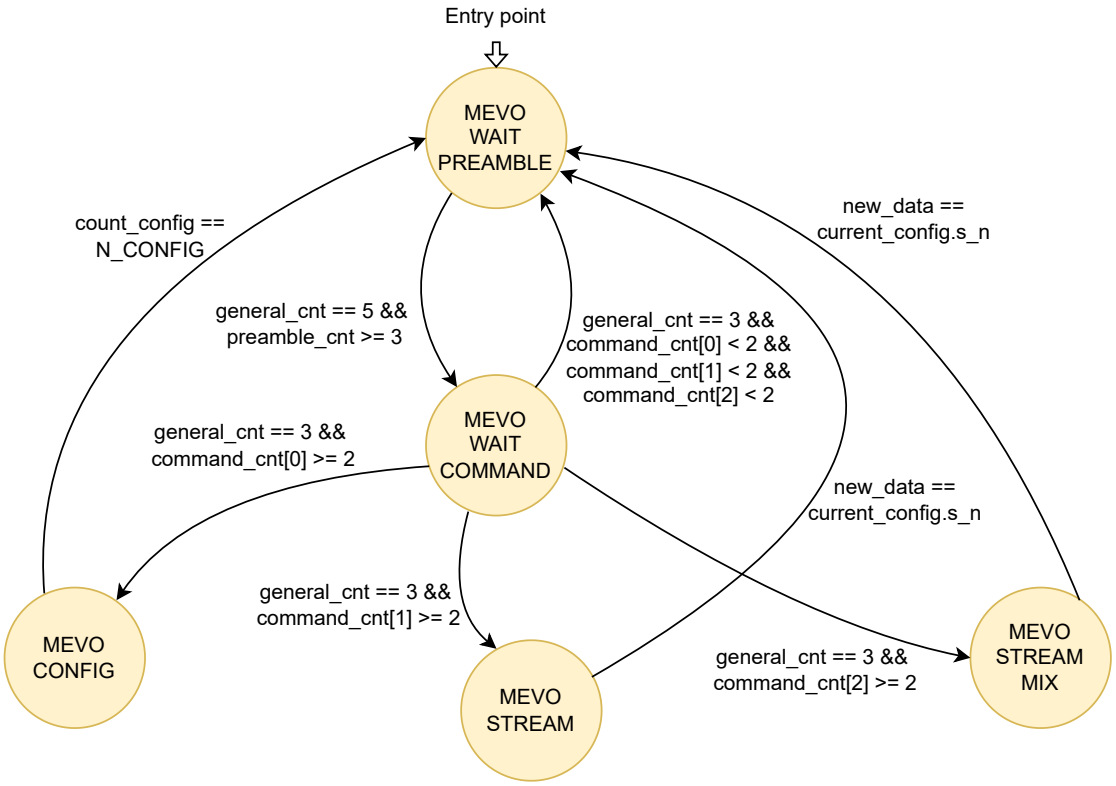

Figure 6.2: Firmware state diagram

Besides the changes introduced in its structure, some other improvements were performed. In particular, the same considerations already described in section 6.2 for *test_loopback.c* were made to properly handle communication protocols. Furthermore, the command 3 missing feature, about handling the mixing operation

on FPGA, was added. Unfortunately, even if all the new FUs are present in the processor architecture, a discussion about how to include the reverb feature on the main firmware has not been made yet.

Back to the FSM, as it can be seen in figure 6.2, the firmware was reorganized in 5 different states:

- MEVO_WAIT_PREAMBLE: in which the processor is waiting for the preamble sequence to occur from SPI
- MEVO_WAIT_COMMAND: in which the processor is waiting for a command to occur from SPI
- MEVO_CONFIG: in which the processor is changing its configuration parameters according to the values received from SPI
- MEVO_STREAM: in which the processor is streaming audio samples and MIDI messages from the audio board to the Raspberry Pi and vice versa, without performing audio mixing
- MEVO_STREAM_MIX: in which the processor is streaming audio samples and MIDI messages from the audio board to the Raspberry Pi and vice versa, performing also audio mixing

To understand the difference between the new main firmware and its previous version, both of them were compiled and the `proxim` command was run. Results show that the old program consisted of 362 instructions, while the new one has 608. This is probably the consequence of introducing a proper way of handling interfaces. Despite all, the new structure of the main firmware can be seen almost as a starting point for future development, since there is definitely a lot of room for improvements.

6.5 Final results

After developing the new required FUs and optimizing the audio processor from an architectural and firmware point of view, the final results provided by Vivado can be analyzed in detail. This tool can report different information about the implemented design and, specifically, outcomes in terms of resource utilization, timing and power consumption. To provide an exhaustive analysis of the obtained results, a comparison with respect to the initial architecture can be made.

6.5.1 Resource utilization

The first results that can be considered are the ones regarding the final utilization of FPGA resources. These outcomes are important to understand possible future developments of the processor and in particular which components should require further optimizations. Vivado can provide some very detailed reports on resource allocation. A comparison between the initial utilization and the final one can be made by taking a look at tables 6.3 and 6.4.

Resource	Utilization	Available	Utilization %
LUT	12117	63400	19.11%
LUTRAM	8314	19000	43.76%
FF	3312	126800	2.61%
BRAM	3	135	2.22%
DSP	3	240	1.25%
IO	42	210	20.00%
BUFG	3	32	9.38%
MMCM	1	6	16.67%

Table 6.3: Resource utilization (initial architecture)

Resource	Utilization	Available	Utilization %
LUT	13479	63400	21.26%
LUTRAM	7193	19000	37.86%
FF	7662	126800	6.04%
BRAM	13	135	9.63%
DSP	20	240	8.33%
IO	42	210	20.00%
BUFG	3	32	9.38%
MMCM	1	6	16.67%

Table 6.4: Resource utilization (final architecture)

The first thing which can be observed is that the number of exploited LUT has increased. Of course, this is mostly the consequence of adding two FUs. It can be seen, however, that the increment in LUT utilization is quite limited (only 2% more are used), because dedicated cells were used as much as possible during the development of these new features. Regarding the LUTRAM utilization, it can be seen that their number has decreased, but not as much as expected. As it was observed previously in table 6.2, indeed, after optimizing the memory optimization, a very low number of LUTRAM were exploited by Vivado (merely 0.64%). However,

this value was not yet taking into account the reverb FU and specifically its large shift registers. Considering them, it is possible to see that a final utilization of 37.86% can be achieved. Some future works may aim to see if this value can be improved by exploring new algorithms and implementations for the reverb FU. A few other considerations can be made also regarding BRAMs and DSP resources. As it can be observed, their utilization reflects the expectations made during the design of the new units. In the initial architecture, only 3 BRAMs were exploited, in particular for I2S buffers, while now 13 of them are used. This increment can be explained by considering that 6 cells are needed for data and instruction memories, while 4 more are used to store audio mixing coefficients. Instead, concerning DSP, the architecture is increasing their utilization from 3 to 20. First, this kind of resource was exploited only for the ALU, while now 8 more are used for audio mixing and 9 to implement the reverb algorithm.

6.5.2 Timing results

The timing results are the second type of report that can be analyzed. As already mentioned, a clock frequency of 100 MHz was set as a goal for the audio processor implementation on FPGA. This requirement was met in both architectures, but some differences in timing behaviors need to be commented. In tables 6.5 and 6.6, it is possible to observe some of the fundamental metrics used to compare the previous architecture with the final one.

Type	Worst slack	Total violation	Failing endpoints	Tot endpoints
Setup	0.023 ns	0 ns	0	89179
Hold	0.035 ns	0 ns	0	89179
Pulse width	3 ns	0 ns	0	11648

Table 6.5: Timing results (initial architecture)

Type	Worst slack	Total violation	Failing endpoints	Tot endpoints
Setup	0.217 ns	0 ns	0	31360
Hold	0.015 ns	0 ns	0	31360
Pulse width	3 ns	0 ns	0	14918

Table 6.6: Timing results (final architecture)

As said, timing violations and failing paths are not present in any of them, but some variations could be noticed about the worst slack, in particular on the setup time. This design characteristic refers to the longest existing path in the circuit and usually determines the maximum achievable clock frequency. As it can be

observed, the initial architecture has a worst slack on setup time of 0.023 ns, while the final processor exhibits a greater value equal to 0.217 ns. These results show that the new architecture is faster than the previous one and that, theoretically, it can achieve a maximum frequency that is almost 2% higher (100.23 MHz of the previous vs. 102.22 MHz of the new one). The reason for this improvement in terms of speed can be found mostly in the optimization performed on memories (described in section 6.1). Indeed, since BRAMs are now exploited for both instruction and data memories, the architecture has shorter critical paths. Moreover, for the same reason, it can also be proved that the implementation results are no more dependent on the firmware used.

6.5.3 Power consumption

The last outcomes that can be considered are the ones related to power consumption. Since Vivado has not enough information about the actual switching activity of internal signals, these results were estimated with a low confidence level. This is because usually to retrieve a satisfactory evaluation of the power consumption, different simulations in some real case scenarios need to be performed. To simplify this process, the switching activity arithmetically estimated by Vivado was considered.

The power consumption for both the initial and the final architectures can be found in tables 6.7 and 6.8. As it can be seen, Vivado can provide detailed information on the type and the different contributions that make up the total value. Specifically, the first distinction that can be made is between dynamic and static power. The former, which includes both the switching and the short-circuit factors, is the one related to the activity of signals and the frequency of the system. The latter, instead, regards the leakage of transistors and the power consumed when there is signals are not switching.

Type	Estimation	Contribute	Estimation
Dynamic	0.185 W (65%)	Clock	0.024 W
		Signals	0.014 W
		Logic	0.016 W
		BRAM	0.002 W
		DSP	< 0.001 W
		MMCM	0.125 W
		IO	0.004 W
Static	0.098 W (35%)	PL static	0.098 W

Table 6.7: Power consumption (initial architecture)

Type	Estimation	Contribute	Estimation
Dynamic	0.252 W (72%)	Clock	0.031 W
		Signals	0.047 W
		Logic	0.024 W
		BRAM	0.013 W
		DSP	0.007 W
		MMCM	0.125 W
		IO	0.004 W
Static	0.099 W (28%)	PL static	0.099 W

Table 6.8: Power consumption (final architecture)

Comparing these tables, it is possible to observe that the dynamic power estimated for the final architecture is higher. This is because, as expected, the new developed features have an impact on consumption. It can be seen that all the different contributions of the dynamic power increased, in particular the one regarding signals. One consideration that can be made concerns the contribution of BRAM and DSP resources. It can be noticed that, even though several new blocks were exploited during the design, the energy associated with these cells is quite low, probably because they are dedicated facilities. For this reason, a greater use of them needs definitely to be considered in future optimizations. The only dynamic contribution that remained the same is MMCM, the one associated mainly with the PLL component. As it can be observed, this estimated power is the greatest one for both the architectures. Regarding static power, it is possible to notice that it has remained almost unchanged. Summing each contribution, the architecture went from an initial consumption of 0.283 W to a final value of 0.352 W.

Chapter 7

Conclusions

This thesis proposed a hardware implementation of two different audio effects to be included in a system for NMP applications. Specifically, they were designed to be used as FUs in a custom TTA processor for audio purposes. This solution, based on an FPGA implementation, was described in detail and compared with the currently available ones, which are mainly software. As seen, this approach can bring several enhancements in terms of latency to a networked performance, since a dedicated hardware architecture is exploited.

Audio mixing was the first unit added to the custom processor. NMP applications need a way to mix the samples coming from different channels and users in real-time and, for this reason, a dedicated hardware was conceived. By adopting the TCE design flow, an initial firmware implementation, to be executed by the already existing audio processor, was developed first. Then, to evaluate whether the introduction of a hardware mixing unit could give a performance boost to the system, this implementation was compared to another one in which the mixing process was performed by some custom operations. In particular, *MIXER_SET_GAIN* and *MIXER_MIX_CHANNELS* were added to the architecture and a new version of the firmware was developed, by replacing previous arithmetical operations with them. By doing that, it was possible to observe that the total number of instructions to be executed decreased by 11.2%. For this reason, a hardware mixing unit was then designed. Its implementation had to consider different aspects and, in particular, the fact that it needed to manage the two custom operations mentioned before. An FSM was developed to properly handle control signals according to the operation to be performed, while an 8-port RAM and a Mixer component were implemented to execute them. The correctness of the design was then checked using a verification procedure based on a SystemVerilog testbench and a golden reference model written in C language. Finally, during the FPGA implementation, some improvements on the developed FU were performed. In particular, optimizations in terms of BRAM

and DSP blocks permit to achieve a clock frequency of 100 MHz and led to better utilization of resources.

The second unit introduced into the system was the one implementing the reverb effect. Over the years, several studies have confirmed that the absence of reverberation can sound unnatural to musicians and may have an impact on the way they interact. Since networked performances are susceptible to this phenomenon, a technique to add artificial reverberation was implemented. However, in this case, a different procedure was adopted to develop the dedicated hardware unit. No initial firmware could be realized, since the audio processor was unable to support fractional numbers and as a consequence no comparison with new custom operations was possible. For this reason, the reverb implementation started directly with the hardware design, which was carried on by exploiting MATLAB HDL Coder and a Simulink model of the unit. In particular, the architecture chosen for the development of this audio effect was the one proposed by Manfred Robert Schroeder. This solution, mainly consisting of comb filters and all-pass ones, was optimized to meet the timing requirements and occupy as little area as possible. To check the hardware implementation, a verification process was again executed. This time a Simulink testbench was used as a golden reference, instead of a C model. Finally, also in this case, optimizations were performed on the FPGA implementation, focusing in particular on DSP and SRL blocks. Thanks to them, the architecture was able to meet the target clock frequency of 100 MHz.

As an additional purpose, some other improvements were applied to increase the performance of the audio processor. Optimizations on memories were introduced by taking advantage of the Vivado library and in particular of its customizable IPs. Thanks to them, LUTRAM resource utilization decreased from 43.76% to 0.64%, leaving more room for other future features. Moreover, since NMP applications have very strict latency requirements, optimizations were made to the system in this regard. Using the *test_loopback.c* firmware as a case study, it was possible to demonstrate that lower time delays could be achieved by properly handling I2S interfaces. Real-world tests show that a decrease of 59.4% was obtainable, leading to a lower loopback latency of 0.859 ms. After performing all these optimizations, the final results of the FPGA implementation were retrieved. Resource utilization and power consumption have increased as a consequence of the fact that two new FUs were added to the processor. Despite that, positive outcomes for these metrics were achieved through the use of dedicated FPGA blocks, such as DSP slices. Instead, regarding timing results, an improved setup time slack was obtained. The final architecture could theoretically be synthesized at a maximum clock frequency of 102.22 MHz, which is almost 2% higher than before.

7.1 Future work

The system considered in this thesis and in particular the FPGA-based audio processor offers several opportunities for future work. Indeed, as seen in the previous chapters, the exploited resources are far from being exhausted and many features can still be implemented.

First, some improvements and work can be done on the already existing functionalities. An optimization that could be explored is the possibility to bring the entire architecture to a unique fixed-point representation on 32-bit, maybe also taking advantage of TCE's future developments. Another enhancement could be related to the reverb FU designed in chapter 5. Many solutions are present in literature about algorithms to create an artificial reverberation and some of them are surely good candidates to improve the quality and the naturalness of the developed one. Moreover, some plans about how to include the reverb feature in the main firmware running on the processor should be made. The designed code needs to be optimized and properly tested in real-world, focusing in particular on the interface with the Raspberry Pi device.

Finally, regarding new features that can be investigated for a possible FPGA-based implementation, two possibilities may be mentioned. First, PLC techniques can be taken into account from a hardware point of view, to realize a system that is able to deal with the lack of audio samples in real-time. Second, some research on the possibility to include new hardware audio effects could be examined. Among all of them, some suggested options are a compressor unit to dynamically handle the volume of each track or a filtering unit to perform sound equalization.

Appendix A

Code

A.1 *mevo_run.sh* script

mevo_run.sh is a bash script that can be used to automatize some of the most used steps in the TCE workflow. In practice, what it does is to execute in sequence `tcecc`, `generateprocessor` and `generatebits` commands, while doing also some housekeeping operations on project folders. *mevo_run.sh* can be used from the command-line by specifying options to achieve different kinds of results.

The available options are the following:

- `-a <ARCHITECTURE_NAME>` : option to specify the name of the architecture (.adf file) you want to use. Default is *mevo_proc.adf*.
- `-c <FIRMWARE_NAME>` : option to specify the name of the firmware (.c file) you want to use. Default is *fw_src/fw_test/test_loopback.c*.
- `-h` : option to display the help manual
- `-i <IMPLEMENTATION_NAME>` : option to specify the name of the implementation (.idf file) you want to use. Default is *mevo_proc.idf*.
- `-t <TPEF_PROGRAM_NAME>` : option to specify the name of the TPEF program (.tpef file) you want to use. Default is *mevo_proc.tpef*.
- `-x` : option to perform only the C source code compilation (by means of `tcecc`). `generateprocessor` and `generatebits` are not performed.

```
#!/bin/bash

################################################################################
## File          :    mevo_run.sh
## Author        :    Diego Bert
################################################################################
## Description   :    This script is used to automatize some of the
##                    most used steps in the TCE workflow.
## Note          :    Before using this script you MUST create .adf,
##                    .idf and .c files.
################################################################################
## Revisions     :
## Date          Version      Author       Description
## 2022-11-18    1.0          Diego Bert   Created
## 2022-12-11    1.1          Diego Bert   New command options added
################################################################################


# Variables
ADF_SOURCE="mevo_proc.adf"
IDF_SOURCE="mevo_proc.idf"
C_SOURCE="fw_src/fw_test/test_loopback.c"
TPEF_SOURCE="mevo_proc.tpef"
HDL_FOLDER="proge-output"
TOP_FOLDER="top-output"
ONLY_TCECC=0

# Choose options
while getopts "a:i:c:t:xh" OPTION
do
    case $OPTION in
        a)
            # Change architecture name
            ADF_SOURCE=$OPTARG
            ;;
        i)
            # Change implementation name
            IDF_SOURCE=$OPTARG
            ;;
        c)
            # Change C program name
            C_SOURCE=$OPTARG
            ;;
        t)
            # Change TPEF program name
            TPEF_SOURCE=$OPTARG
            ;;
        x)
```

```
49              # Perform only TCECC source code compilation
50              ONLY_TCECC=1
51              ;;
52          h)
53              man ./mevo_run_man.1
54              exit
55              ;;
56          *)
57              exit
58      esac
59  done
60
61  # Start message
62  echo "------ Automated TCE workflow ------"
63
64  # Tcecc command
65  echo "[*] TCECC..."
66  tcecc -O3 -a $ADF_SOURCE -o $TPEF_SOURCE $C_SOURCE
67
68  if [ $ONLY_TCECC == 0 ]
69  then
70
71      # Generateprocessor command
72      echo "[*] GENERATEPROCESSOR..."
73      generateprocessor -i $IDF_SOURCE -o proge-tmp/ $ADF_SOURCE
74
75      # Generateprocessor housekeeping operations
76      cp $HDL_FOLDER/README.md proge-tmp/
77      rm -r $HDL_FOLDER
78      mv proge-tmp/ $HDL_FOLDER
79
80      # Generatebits command
81      echo "[*] GENERATEBITS..."
82      generatebits -w 4 -f coe -d -o coe -p $TPEF_SOURCE -x $HDL_FOLDER
        $ADF_SOURCE
83
84      # Generatebits housekeeping operations
85      mv mevo_proc.coe $TOP_FOLDER/mevo_proc_inst.coe
86      mv mevo_proc_data.coe $TOP_FOLDER
87
88  fi
89
90  # End message
91  echo "------ Done ------"
```

Bibliography

[1] Cristina Rottondi. «Networked music performances in the COVID19 pandemic era: achievements and open challenges». In: *IEEE Student Branch: W[I]E Present!* 2020 (cit. on pp. 1, 9, 10).

[2] Cristina Rottondi, Chris Chafe, Claudio Allocchio, and Augusto Sarti. «An Overview on Networked Music Performance Technologies». In: *IEEE Access* (2016) (cit. on pp. 2, 6, 11, 49).

[3] Alexander Carôt, Christian Werner, and Timo Fischinger. «Towards a Comprehensive Cognitive Analysis of Delay-Influenced Rhythmical Interaction». In: *Proceedings of the International Computer Music Conference*. 2009 (cit. on p. 6).

[4] Snorre Farner, Audun Solvang, Asbjørn Sæbø, and U. Peter Svensson. «Ensemble Hand-Clapping Experiments under the Influence of Delay and Various Acoustic Environments». In: *Journal of the Audio Engineering Society* (2009) (cit. on pp. 6, 49).

[5] Alain B. Renaud, Alexander Carôt, and Pedro Rebelo. «Networked Music Performance : State Of The Art». In: *AES 30th International Conference*. 2007 (cit. on p. 6).

[6] *Hidden Histories of the Information Age - Tat-1*. URL: `https://www.bbc.co.uk/programmes/b04m3bcc` (cit. on p. 6).

[7] *Introduction : Experimental Music in the Bay Area*. URL: `http://crossfade.walkerart.org/brownbischoff/` (cit. on p. 7).

[8] Chris Chafe, Scott Wilson, Randal Leistikow, Dave Chisholm, and Gary Scavone. «A simplified approach to high quality music and sound over IP». In: *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*. 2000 (cit. on p. 7).

[9] Alexander A. Sawchuk, Elaine Chew, Roger Zimmermann, Christos Papadopoulos, and Chris Kyriakakis. «From remote media immersion to Distributed Immersive Performance». In: *Proceedings of the 2003 ACM SIGMM workshop on Experiential telepresence*. 2003 (cit. on pp. 7, 49).

[10] Juan-Pablo Cáceres and Chris Chafe. «JackTrip: Under the Hood of an Engine for Network Audio». In: *Journal of New Music Research* (2010) (cit. on p. 8).

[11] *JackTrip Foundation*. URL: https://www.jacktrip.org/ (cit. on p. 8).

[12] Alexander Carôt, Alain B. Renaud, and Bruno Verbrugghe. «Network Music Performance (NMP) with Soundjack». In: *Proceedings of the 6th NIME Conference*. 2006 (cit. on p. 8).

[13] *SoundJack*. URL: https://www.soundjack.eu/ (cit. on p. 8).

[14] *Jamulus*. URL: https://jamulus.io/ (cit. on p. 8).

[15] Chrisoula Alexandraki, Panayotis Koutlemanis, Petros Gasteratos, Nikolas Valsamakis, Demosthenes Akoumianakis, and Giannis Milolidakis. «Towards the implementation of a generic platform for networked music performance: The DIAMOUSES approach». In: *EProceedings of the ICMC 2008 International Computer Music Conference (ICMC)*. 2008 (cit. on p. 8).

[16] *SonoBus*. URL: https://sonobus.net/ (cit. on p. 9).

[17] *JamKazam*. URL: https://jamkazam.com/ (cit. on p. 9).

[18] *Elk Audio*. URL: https://www.elk.audio/ (cit. on p. 9).

[19] *Comparison of Remote Music Performance Software*. URL: https://en.wikipedia.org/wiki/Comparison_of_Remote_Music_Performance_Software (cit. on p. 9).

[20] Prateek Verma, Alessandro I. Mezza, Chris Chafe, and Cristina Rottondi. «A Deep Learning Approach for Low-Latency Packet Loss Concealment of Audio Signals in Networked Music Performance Applications». In: *2020 27th Conference of Open Innovations Association (FRUCT)*. 2020 (cit. on p. 10).

[21] *I2S bus specification*. URL: https://www.nxp.com/docs/en/user-manual/UM11732.pdf (cit. on p. 13).

[22] *DAC PCM1771 datasheet*. URL: https://www.ti.com/lit/ds/symlink/pcm1771.pdf (cit. on p. 14).

[23] *MIDI*. URL: https://en.wikipedia.org/wiki/MIDI (cit. on p. 14).

[24] Bernardo Breve, Stefano Cirillo, Mariano Cuofano, and Domenico Desiato. «Perceiving space through sound: mapping human movements into MIDI». In: *26th International Conference on Distributed Multimedia Systems*. 2020 (cit. on p. 14).

[25] *SPI Block Guide V04.01*. URL: https://www.nxp.com/files-static/microcontrollers/doc/ref_manual/S12SPIV4.pdf (cit. on p. 15).

[26] *Introduction to SPI Interface*. URL: https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html (cit. on p. 16).

[27] Riccardo Peloso. *Custom hardware audio board*. 2021 (cit. on p. 16).

[28] *7 Series FPGAs Data Sheet: Overview*. URL: https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview (cit. on pp. 18, 43).

[29] *About Transport-Triggered Architectures*. URL: http://openasip.org/tta.html (cit. on p. 19).

[30] Nicola Domini. *Ultralow Latency Audio Processing Via FPGA For Networked Music Performance Applications*. 2022 (cit. on p. 22).

[31] *Raspberry Pi 4 Model B*. URL: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf (cit. on p. 22).

[32] Matteo Sacchetto and Leonardo Severi. *Software development on Raspberry Pi* (cit. on p. 24).

[33] Pekka Jääskeläinen, Timo Viitanen, Jarmo Takala, and Heikki Berg. «HW/SW Co-design Toolset for Customization of Exposed Datapath Processors». In: *Computing Platforms for Software-Defined Radio*. 2017 (cit. on p. 24).

[34] *TTA-based Co-design Environment*. URL: http://openasip.org/ (cit. on p. 25).

[35] *Vivado ML Overview*. URL: https://www.xilinx.com/products/design-tools/vivado.html (cit. on p. 25).

[36] *Audio mixing (recorded music)*. URL: https://en.wikipedia.org/wiki/Audio_mixing_(recorded_music) (cit. on p. 27).

[37] *Mixing Music: What is Sound Mixing?* URL: https://online.berklee.edu/takenote/mixing-music-what-is-sound-audio-mixing/ (cit. on p. 28).

[38] *7 Series DSP48E1 Slice - User Guide (UG479)*. URL: https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1 (cit. on pp. 43, 44).

[39] *Reverb effect*. URL: https://en.wikipedia.org/wiki/Reverb_effect (cit. on pp. 47–49).

[40] *Reverberation*. URL: https://en.wikipedia.org/wiki/Reverberation (cit. on p. 47).

[41] Mark Kahrs and Karlheinz Brandenburg. *Applications of Digital Signal Processing to Audio and Acoustics*. The Springer International Series in Engineering and Computer Science. Springer US, 1998 (cit. on pp. 47, 49, 50).

[42] William Weir. «How Humans Conquered Echo». In: *The Atlantic* (2012). URL: https://www.theatlantic.com/entertainment/archive/2012/06/how-humans-conquered-echo/258557/ (cit. on p. 48).

[43] Manfred R. Schroeder. «Natural Sounding Artificial Reverberation». In: *Journal of the Audio Engineering Society* (1962) (cit. on p. 50).

[44] *MATLAB HDL Coder*. URL: https://it.mathworks.com/products/hdl-coder.html (cit. on p. 57).