POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Ultralow Latency Audio Processing Via FPGA For Networked Music Performance Applications

Supervisors:

Prof. Cristina ROTTONDI

Dott. Riccardo PELOSO

Candidate:

Nicola DOMINI

October 2022

Abstract

Among all computer architectures, Application-Specific Instruction set Processors (ASIP) are one of the solutions that can better host a custom application. Together with the usage of Field-Programmable Gate Array (FPGA) chips, they provide the best tradeoff regarding flexibility and performance, allowing for an efficient design chain.
Transport Triggered Architectures (TTAs) are ASIP-like solutions that can offer quite exceptional features. The possibility of creating custom functional units to accelerate specific tasks and the opportunity of having program instructions that can host several move operations simultaneously lead to a highly optimized and parallel architecture. Furthermore, the possibility of programming TTA cores through a high-level language like C, using custom hardware operations, offers an additional level of freedom to the design, providing an efficient codesign environment for present and future use cases.
This thesis proposes an implementation for an FPGA TTA processor architecture integrated into a Networked Music Performance (NMP) application, an environment where musicians can play together remotely in real-time: custom hardware reduces the typical latencies given by processing, recording, and streaming of audio data.
Results show that the implemented processor can easily record, stream and exchange audio data with the other devices composing the NMP environment, with enough space to add more processing power, eventually. Latency contributions introduced by the architecture are very low since the core can run at a clock speed of 100 MHz with optimized paths, potentially leading to an improvement with respect to software-based solutions.

Acknowledgements

I would like to express my gratitude to all the people who have helped me throughout this thesis work. I would like to thank professor Cristina Rottondi, Riccardo Peloso, Matteo Sacchetto and Leonardo Severi for their precious help and for their patience.
I would like to dedicate this thesis to all the people that have been with me over the years.

Thanks to my parents for raising me and always supporting me in my decisions.

Thanks to my brother Giacomo for always being a point of reference for me.

Thanks to my nephew Livia and her brother who is about to be born for being a flash of pure joy.

Thanks to all of my friends for sharing with me beautiful experiences.

And last but not least, thanks to my dear Gwenda for always being here and supporting me during bad times.

Table of Contents

List of Tables

List of Figures

Acronyms

ASIP Application Specific Instruction-set Processor

FF Flip-Flop

FU Function Unit

HDL Hardware Description Language

I2S Inter-IC Sound

MIDI Musical Instrument Digital Interface

NMP Networked Music Performance

PLB Programmable Logic Block

RPi Raspberry Pi

RX Receiver/Reception

SPI Serial Peripheral Interface

TTA Transport Triggered Architecture

TX Transmitter/Transmission

UART Universal Asynchronous Receiver-Transmitter

TCE TTA-based Co-design Environment

Chapter 1

Introduction

1.1 Motivation

1.1.1 Networked Music Performance

Networked Music Performance (NMP) is a practice by which musicians connected remotely from different and distant places can play together in a real-time environment, leveraging ultralow-delay audio streaming over a telecommunication network.

With NMP, musical performers can overcome the physical distance among them, and the possibilities of application are huge: let's just think about a band whose members need to rehearse but they live in different parts of the country or even of the world. They can avoid being always present while still performing their activity. Even more ambitious tasks can be accomplished with the help of this technology: live performances with members spread at different locations can be supported, opening endless possibilities.

However, the implementation of this technology comes with many problems: transmission and processing latencies of streamed audio have to be very low for the musicians to be synchronized with each other and play virtually at the same time. Luckily, up to some extent, delay is acceptable. Studies reveal that human hearing can tolerate latencies in the range of 20-30 ms [1] while still being able to play simultaneously. This range depends on many factors, such as musicians' skills and practice in coping with delay, as well as the musical context being considered. Let's think about an orchestra: there is a non-negligible distance between the members that leads to delays of up to 30-50 ms. We clearly understand that this is not a problem and many times can even add some "flavor" to the performance. Or even think about an organ player, who is naturally accustomed to significant latencies, even on its own instrument, due to the physical distance between console and pipes that is often present.

Nevertheless, while developing a solution for NMP, one should consider the worst-case scenario and try to minimize as much as possible every source of delay.

Some NMP frameworks try to reduce latency by sending only MIDI information through the network. This is due to the MIDI protocol being extremely lightweight and transmitting only commands without carrying audio signals. At the receiving side, it is sufficient to read these commands and send them to a MIDI-capable device, such as a synthesizer, which will

convert them into audio signals. However, it is clear that this is not a friendly solution for acoustic instruments. More ambitious systems will try to stream audio samples through the network. This time, the situation is much more complicated, since we need to send packets with many audio samples and much more information (depending on the sample rate as well). In addition, there is also the time required for eventual pre/post-processing tasks.

Some of the most challenging aspects of NMP therefore are:

- **Bandwidth**: high bandwidth is needed to obtain high-quality streaming to make the performers feel in a more comfortable environment. On the other hand, high quality also means higher bitrate offered to the network, which may be more challenging to accomodate in highly-congested scenarios.

- **Synchronization**: systems at both ends of the communication should ensure a consistent audio stream. Unfortunately, the internet network is not well known for its reliability. Different delays in packets' arrival and the possibility to lose some of them are problems one has to face. NMP systems at both ends must be able to solve problems related to jitter and try to use the most reliable protocols available.

- **Latency**: it is the time delay that occurs between the moment a musician plays a note and the moment in which this note arrives at the listener's ear. It is probably the most critical parameter for an NMP technology. Many sources contribute to the overall delay, either from the users' sides (sound acquisition, sampling, quantization, processing, buffering of the audio board, audio reconstruction, etc.) or the network side (packetization, transmission and propagation, depacketization, jitter buffering, etc.).

1.2 Thesis objectives

This thesis focuses on the development of a TTA-based processor implemented on an FPGA board that features a series of dedicated hardware functional units for hardware acceleration of specific tasks. The goal is to have an ultra-low latency processor that can stream and process audio samples at a "real-time" rate. Since an FPGA is used, the whole hardware circuitry is completely reconfigurable. In addition to that, the processor should be programmable in C/C++ language and should implement some custom operations to control the function units. In this way, once the architecture is defined, it is sufficient to modify some lines of code to change the overall behavior. And if the situation requires some architectural changes, they can be made as well.

From a functional point of view, the processor should act as an accelerator between one custom audio board [2] and a Raspberry Pi 4. The former features four input and two output audio channels, as well as four input and two output MIDI channels. On the other hand, RPi is used to transmit/receive audio packets to/from the internet network, and pack and unpack them properly.

Communication with the audio board uses the *I2S standard* for the audio signals and a UART-like communication at $31250Hz$ baud rate for *MIDI* messages exchange.

Communication with Raspberry Pi is free but has to be pretty fast. So, using SPI peripheral could be a solution, since it is a pretty fast serial protocol and since FPGA should be good in dealing with such frequencies.

1.3 Thesis outline

This thesis is organized into 6 chapters. Being this one the introduction, chapter 2 shows a digression on NMP history and the state of the art of NMP applications, while chapter 3 overviews the background concepts that are used in the hardware design. Chapter 4 describes the processor architecture that has been implemented, while Chapter 5 reports simulations and measurements that were performed to validate the system. Chapter 6 contains conclusions on the thesis work and possible future work.

Chapter 2

Related Work

This chapter explores some of the most relevant solutions that have been investigated in time to achieve the results expected in the Networked Music Performance field. It also provides some useful articles to compare the state of the art with the results which are expected in this thesis work.

2.1 NMP history

One of the first experiments dealing with a long-distance type of performance can be related to an experiment led by John Cage in 1951, many years before the diffusion of the internet network. The title of the performance was *"Imaginary Landscape No. 4 for Twelve Radios"* and it consisted in the usage of radio frequency transistors to be influenced by each other and at the same time be used as musical instruments [3]. Even without the real usage of a network, this can be considered one of the first steps toward remote performances.
Anyway, the first real experiments with NMP came with the advent of computers and in the late seventies, the ***League of Automatic Music Composers*** exploited the connection of computers in a network to send messages to influence the playing of each other [3].
Later, the same members were involved in a new project called ***The Hub***. They started to perform music adding gradually more distance. In 1987 they were able to play in different streets in New York City [4]. In 1990, they started a MIDI-based version of The Hub, adding a layer of flexibility in the mutual control of machines. In 1997, they were asked to create a new performance called ***"Points of presence"***, connecting musicians at Mills College, California Institute for the Arts with ones at Arizona State University. Anyway, they found out that this was more difficult than previously expected, and that the communication channel was still not ready for this type of musical performance [4][5]. However, it was only at the beginning of the 2000s that the internet started to become fast enough to allow for high-quality audio links [4][6]. One of the first examples is provided by the SoundWire group at Stanford University's CCRMA (Center for Computer Research in Music and Acoustics), which featured bidirectional uncompressed audio streaming[4].

At the same time, at McGill, the Ultra Video-Conferencing Research group started to explore the same field with additional interest in video transmission. Since then, several frameworks have been proposed, such as SoundJack [7] and DIAMOUSES [8]

2.2 Relevant articles

This section provides an overview of some relevant articles that can be related to this thesis work.

Article [1] gives an extensive view of all the problems and technology solutions related to Networked Music Performance, and in particular, in chapter V there is a focus on the *State-Of-Art NMP frameworks*. In particular, a proposal for an FPGA-based solution *"is employed for the routing procedure of media packets (i.e. reception, copy and transmission)"*, without any kernel intervention.

Article [9] analyzes the benefits and the drawbacks of using a Transport Triggered Architecture processor as a soft core in FPGA design, by making a comparison with the well-known VLIW architectures. The main reported drawback is the program size, with an increase varying from 21% to 49% with respect to VLIW. However, employment of TTA architecture led to a resource usage of 67% and an execution time improvement of 88% with respect to VLIW.
These results show that TTA is a good candidate for flexible application-specific tasks aimed at reducing execution time, as in the case analyzed in this thesis.

Article [10] presents the open-source TCE Codesign Environment for TTA processors development. Close attention is paid to its workflow based on a retargetable high-level language compiler, an instruction-set simulator and an RTL generator, which guarantee a fast, flexible and relatively simple toolset for the development of application-specific processors based on TTA architectures.

Article [11] focuses on the benefits of Transport Triggered Architectures when dealing with hardware acceleration of specific tasks to speed up execution time. Traditional architectures introduce overheads when working with hardware acceleration, due to the transfer of data between the processor and the accelerator unit and the signalling required to notify that the accelerator is ready to perform operations. The article explains how TTA architectures reduce these problems by placing the hardware accelerators directly on the processor datapath. In this way, they are seen as normal processor function units, being visible to the programmer and being scheduled by the compiler as any other operations, potentially hiding their latency with other available program operations. This is particularly useful in the case study of this thesis since dedicated units are used for external interface peripherals and others are likely to be added in the future to accelerate specific tasks such as mixing and/or filtering of audio sources.

Chapter 3

Background

This chapter describes general concepts about the employed processor architecture, the FPGA technology and the serial communication protocols used for external communication.

3.1 TTA processors

Transport Triggered Architecture (TTA) is a processor design philosophy where the processors' internal datapaths are exposed in the instruction set. All operation parameter reads and result writes are explicitly stated in the instruction set [12]. Datapath consists of a certain number of functional units (FU), register files, and a series of buses to connect them. Each possible user program is mapped to a discrete number of movements from a trigger port to a destination port through the available buses. Instruction length is usually very long and consists of a move slot for each datapath bus, leading to natural parallelism in the processor.

TTA processors are particularly suitable for **Application Specific Processors (ASIP)**, where custom functional units are used in a modular way to accelerate particular types of tasks (audio-related ones in this case). A picture of a very simple TTA architecture is shown in figure 3.1

3.2 FPGA

Since low latency is our major concern, it is known that dedicated hardware offers many benefits in these terms: having paths and functional units that perform highly specialized operations can lead to minimize the time delay. However, in many situations, this can be too expensive and rigid and it's been years since electronic instruments companies have stopped to use dedicated ICs for their machines: one single bug in the design can bring to the failure of the whole system, without the possibility of debugging.

FPGAs (Field Programmable Gate Array) try to address these problems, by being relatively low cost and flexible hardware solutions: an FPGA can be reconfigured many times with all the circuitry that we need, while still showing great advantages in terms of speed.

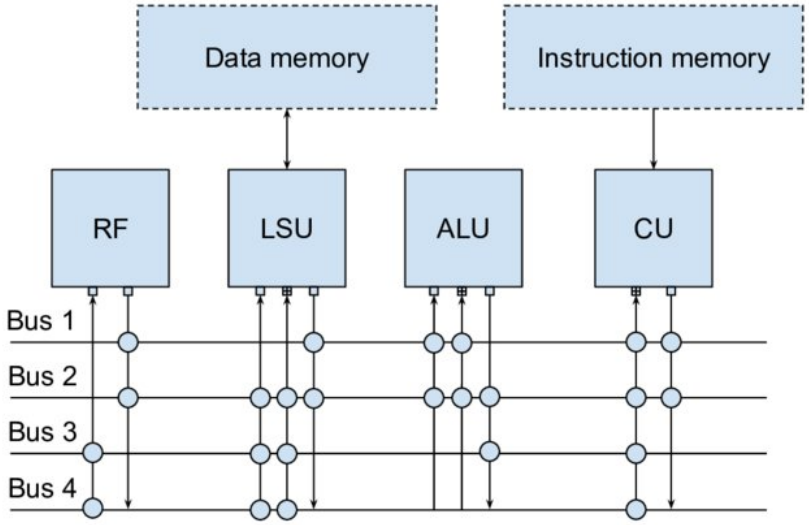


Figure 3.1: Example TTA architecture. Image taken from [13]

Going into detail, FPGAs are semiconductor devices that rely on a matrix of Programmable Logic Blocks (PLB) and reconfigurable connections, usually set up with the help of Hardware Description Languages (HDL) such as VHDL and Verilog, being these two of the most well-known ones worldwide. Memory is also another important building block for FPGAs and it can be in the form of simple Flip-Flops (FF) or more sophisticated memory blocks. In figure 3.2 it is possible to see a simplified representation of the internal organization of an FPGA chip.

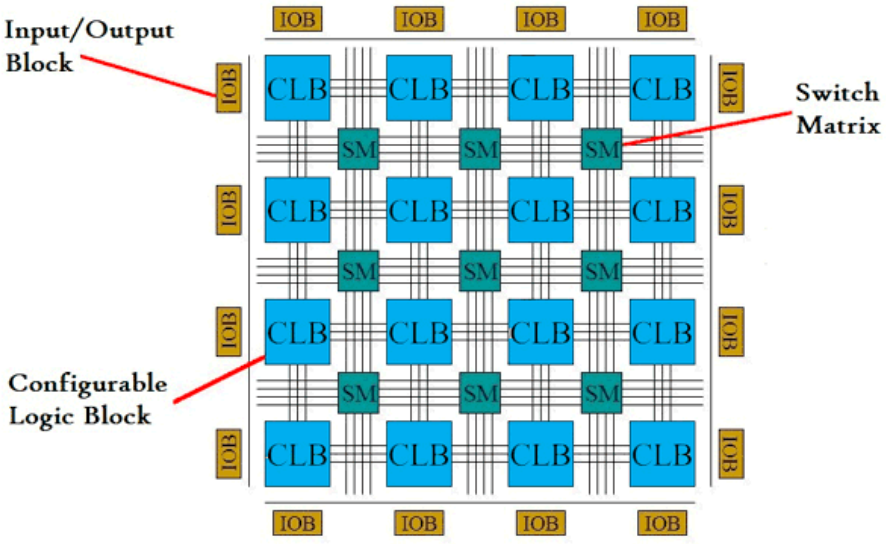


Figure 3.2: Simple representation of FPGA chip. Image taken from [14].

FPGA design flow consists of four main steps:

- **Design elaboration** This is the phase in which the user designs the architecture that has to be implemented in hardware. HDL languages can be used, as well as schematics tools, or even a combination of the two.
- **Synthesis** During this phase, the design entry is analyzed and converted into an electrical circuit composed of logic gates, flip-flops, interconnections, etc. A netlist is generated as a result.
- **Implementation** In this phase, the generated netlist is mapped to the actual resources present in the chosen FPGA chip, by a process called Place&Route, which first maps components to the physical cells and then performs the routing of the interconnections. This last process can be iterated many times to meet the timing requirements of the system that has to be generated. In case timing constraints are not respected, the design is not guaranteed to work properly.
- **Programming** During this last phase, a **bitstream** file is generated with all the information necessary to configure the FPGA chip and make it work.

3.3 Serial communication protocols

3.3.1 MIDI

MIDI, which stands for Musical Instrument Digital Interface, is a standard used to define communication protocol, interface, and physical connectors for the connection between digital musical instruments [15]. In detail, the protocol specifies messages to send control information among devices (no audio samples are exchanged). A message consists of 8-bit words, which can be either state or data words. The communication is asynchronous and is similar to a UART one, with a baud rate of 31250 Hz, a start bit (logic 0), and a stop bit (logic 1) encapsulating the 8-bit message. Messages can be of various types, such as NOTE-ON/NOTE-OFF, CC (usually used to read the value of a certain control, like potentiometers, etc.), and so on. System messages and information about channels and timing can be sent as well.

3.3.2 I2S

I2S (Inter-IC Sound) bus [16] is a synchronous serial communication protocol created to set a standard for digital connection among audio devices, such as A/D and D/A converters, DSP devices and so on.
It features a master/slave configuration and consists of at least three wires:

- **Continuous Serial Clock (SCK) or Bit Clock (BCK)**, generated by the master. It sets the rate at which bits are transmitted. Its frequency is the result of the sample rate f_s times number of bits per channel N_{bch} times number of channels N_{chann}.

$$f_{bck} = f_s \cdot N_{bch} \cdot N_{chann}$$

- **Word Select (WS) or Left-Right Clock (LRCLK)**: this signal specifies to which channel the samples in the SD line belong. WS=0 for channel 1 (left), while WS=1 for channel 2 (right). Its period is equal to the audio sample rate.
- **Serial Data (SD)**: line in which the serialized data is transmitted. Data format is signed with MSB (Most Significant Bit) first.

In **Left-Justified (LJ)** configuration, timing is set as in figure 3.3. When LRCK switches, SD starts simultaneously to transmit the MSB of the word of the selected channel. Then, each bit is transmitted serially at the rate set by BCK (Bit Clock). Although MSB has a fixed position, LSB position depends on the word length.

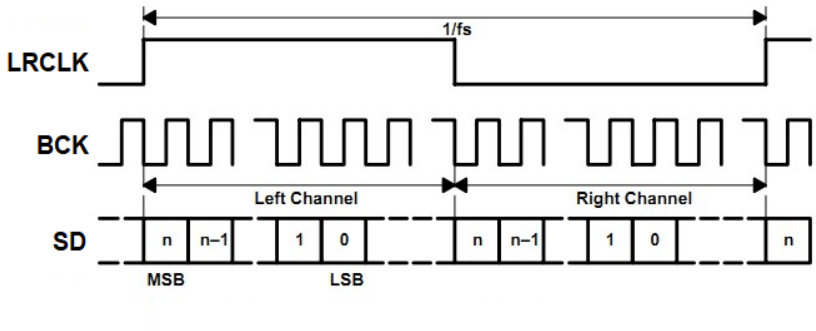


Figure 3.3: Left-Justified configuration timing.

3.3.3 SPI

SPI, standing for Serial Peripheral Interface, is a synchronous serial interface used primarily in embedded electronics. SPI is designed for short-distance communication and supports **full-duplex** communication in a standard master/slave architecture. It is composed of four wires:

- **SCK (Serial Clock)**: it is generated by the master and is the clock signal that determines the rate of communication.
- **MOSI (Master Output Slave Input)**: the line where the master outputs data.
- **MISO (Master Input Slave Output)**: the line where the slave outputs data.
- **CS (Chip Select)**: can be used either to select a specific slave device or to activate the connection with the slave, if only one is present.

SPI supports four modes of operation, depending on the clock polarity (CPOL) and phase (CPHA) concerning data. In figure 3.4 we can see an example of operation in mode 0.

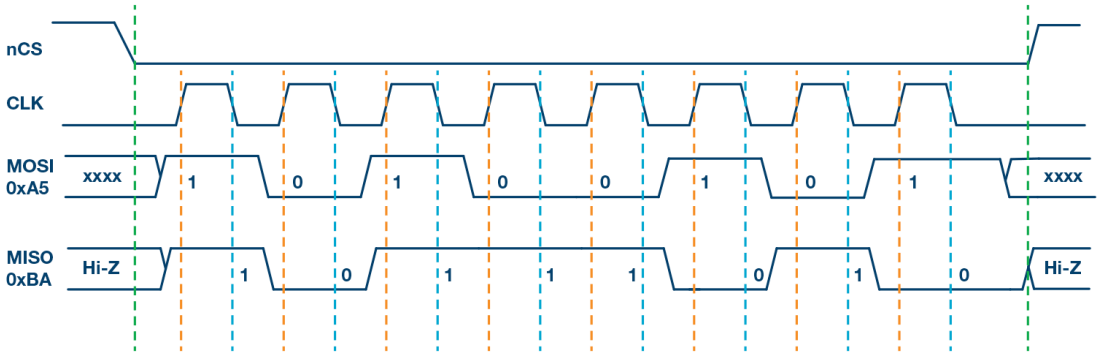


Figure 3.4: Timing diagram for SPI mode 0 (CPOL=CPHA=0). Image taken from [17].

Chapter 4

System Description

This chapter describes the designed system from different points of view, either looking at its interface with the external devices and its internal organization and analyzing in detail the peripherals and the internal functional units used. The exploited hardware is described, as well as the architectural choices that led to the final implementation. The development choices of the firmware that runs inside the processor are also reported.

4.1 Hardware support

To implement the designed system, an **Arty A7-100T** development board by Digilent (figure 4.1) is being used, which features an **XC7A100TCSG324-1** FPGA chip, 15850 logic slices, 65200 flip-flops, 4860 Kbits of block RAM and a 100 MHz internal oscillator clock. It also features some on-board utilities like 4 pushbuttons, 4 switches, 4 LEDs, 4 RGB LEDs, 4 GPIO Pmod connectors, and an Arduino Shield connector.

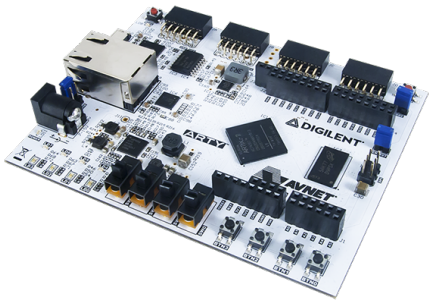


Figure 4.1: Arty A7-100T development board.

In addition, one custom audio board [2] is used to record and play audio data and to exchange it with the FPGA through the I2S protocol. This board has a user interface that features four XLR inputs with pre-amplifier stages, four MIDI inputs, two MIDI outputs, and one headphone output (3.5mm jack). On the other hand, it is connected to

the FPGA board thanks to the two Pmod connectors J12 and J13 (12 pins each). These pins are used to share a common ground and to exchange I2S audio, as well as MIDI data. Internally, it features two PCM1803ADBR Analog to Digital Convertes that convert the four analog input audio channels into I2S data. It also features one PCM1771PW Digital to Analog Converter to convert digital I2S data into an audio signal for the headphones output. Figure 4.2 shows its block diagram.

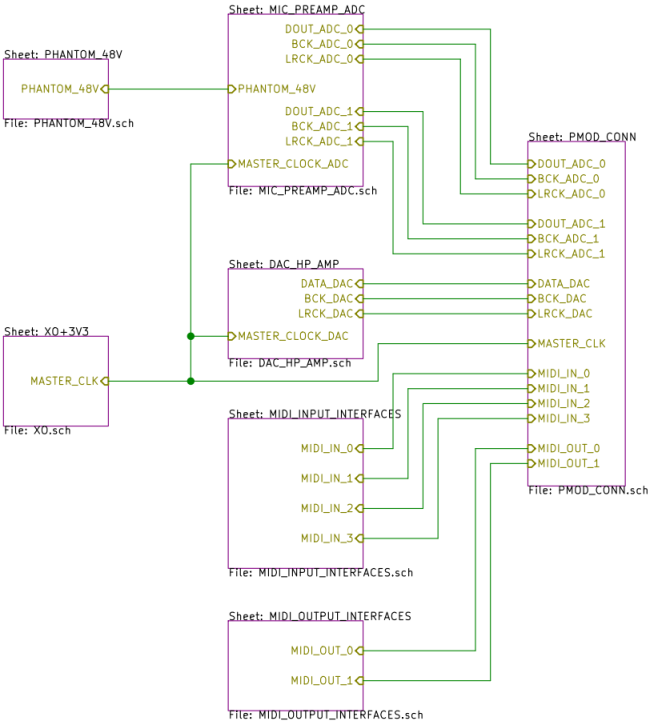


Figure 4.2: Block diagram of the custom audio board.

4.2 Block diagram of the FPGA system

The whole FPGA system consists of many layers, as shown in figure 4.3. The outer one, called "**TOP_PLL**" is composed of two elements, one being the complete processor system ("TOP" layer) and the other one the PLL circuit, which accepts as an input the system clock coming from the Arty A7-100T FPGA board and outputs the master clock *mclk*, which is used by the internal I2S peripherals, as well as the ones on the external custom audio board, to generate the proper timings. Moving one step forward, we find the "**TOP**" layer, which is composed of the **I2S clock generator**, in charge of generating the bit clocks *bck* and the left-right clocks *lrck*, the **data memory**, the **instruction memory**, and the **TTA core**, consisting of all the necessary functional units (see section 4.4.2 for detailed description). Data memory is implemented as a synchronous RAM block and

contains results of operations and global variables, while instruction memory contains the sequence of instructions required to be executed by software.

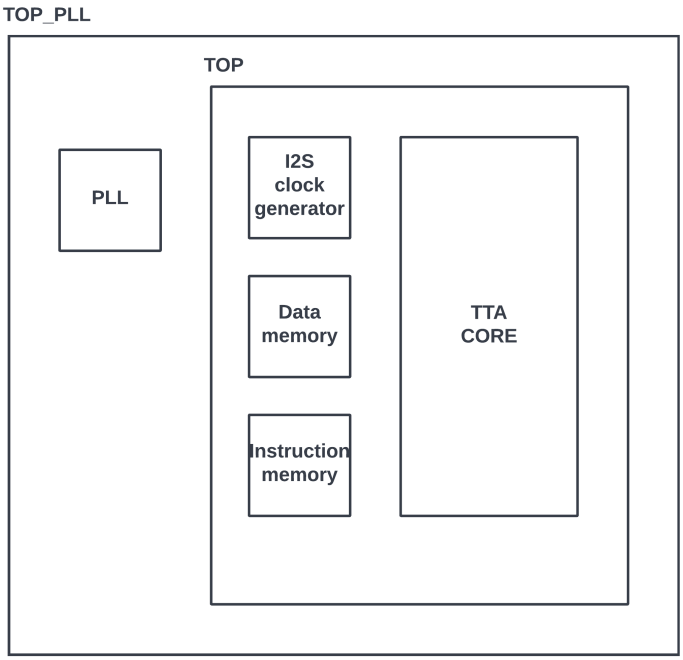


Figure 4.3: Block diagram of FPGA system.

4.3 Interface description

As introduced before, the FPGA design acts as an accelerator unit between a Raspberry Pi 4 board and the custom audio board (see figure 4.4).
Connection with Raspberry Pi is performed by an SPI communication through another Pmod connector. In particular, two peripherals are exploited, one for MIDI data and one for audio data.
Figure 4.5 is the representation of the "**TOP_PLL**" layer described in section 4.2 with all its input and output pins that connect it with the external environment. Note that *u_clk_wiz_0* is the PLL module.
Each of the interface pins is assigned to a physical pin on the FPGA chip, which in turn is assigned to one of the Pmod connectors GPIO pins, as can be seen in table 4.1. In the same table, the custom audio board pin is also shown, that is connected to the Arty A7 pin. Since the FPGA board features four equally spaced PMod female connectors (see figure 4.7a) and the audio board features two male Pmod connectors(figure 4.7b) with the same spacing, it is mandatory that two adjacent FPGA board connectors are assigned to this communication. In addition, JB and JC connectors are labeled as *High-speed* by the

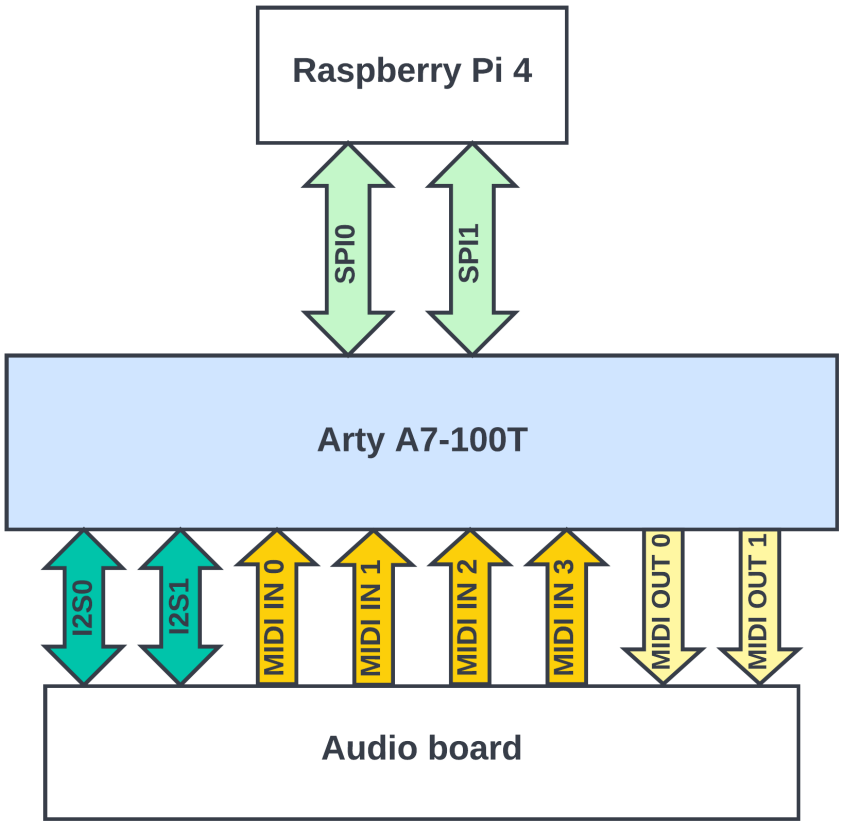


Figure 4.4: External interface block diagram.

manufacturer, and we would like to assign at least one of the two to the SPI communication (since it is the one requiring more bandwidth). Thus, connectors J12 and J13 of the audio board are connected to connectors JA and JB (see figure 4.6), while JC is assigned to high-activity pins of the SPI communication, like miso, mosi and sck.

4.3.1 I2S speed

Since the FPGA system should work as an I2S master device, it should provide the master clock (mclk) and all the other timings to the external audio board. The speed of mclk can be calculated as follows

$$f_{mclk} = f_{sr} * A * B = 11.289{,}6\,\text{MHz}$$

where $A = 4$ is the mclk to blck ratio, $B = 32$ is the bck to lrck ratio, and $f_{sr} = 44{,}100\,\text{Hz}$ is the sampling rate.

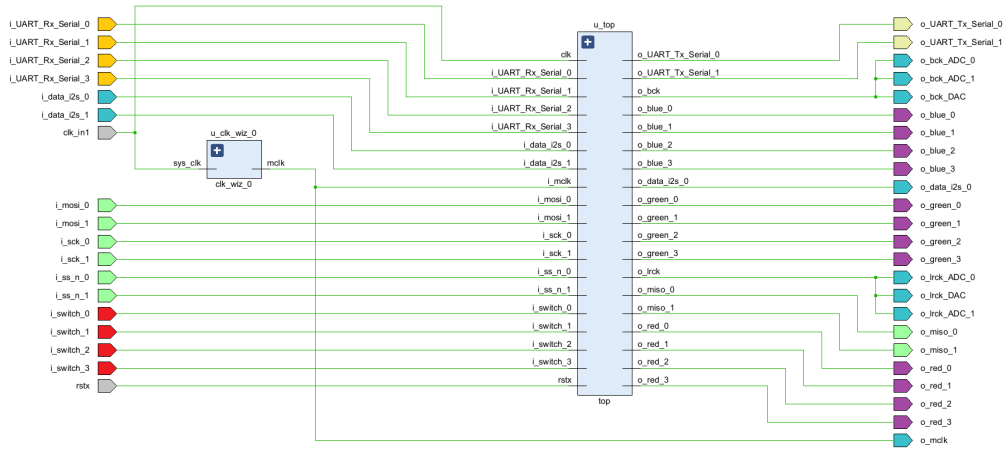


Figure 4.5: TOP_PLL layer schematic.

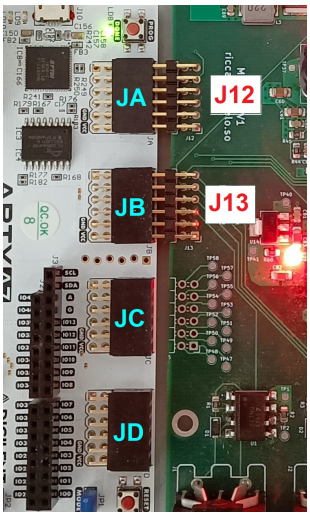
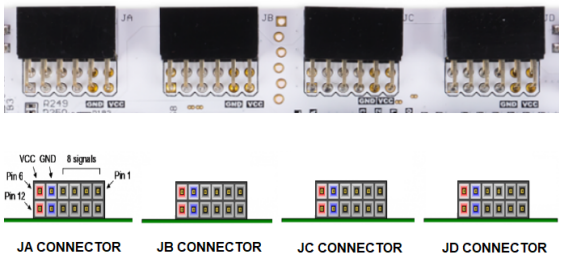


Figure 4.6: Connection between the FPGA (left) and the custom audio board (right).

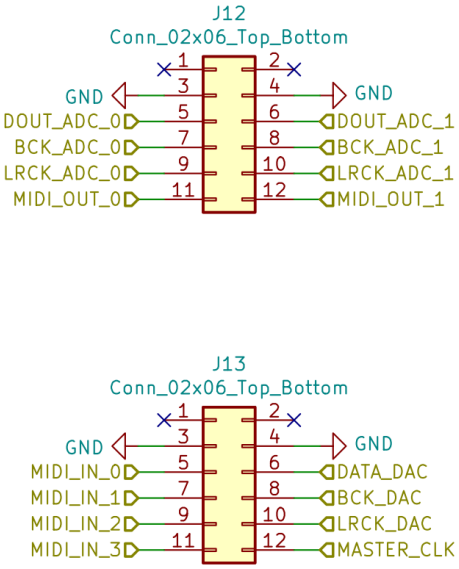
4.3.2 SPI channel bandwidth requirements

According to the number of SPI channels that the SPI link needs to handle, the required bandwidth (the speed of the serial clock) changes. Being the audio sample rate fixed to $f_s = 44{,}100\,\mathrm{k}Hz$ and the number of bits per sample $b_s = 16$, each channel will require at least

$$BW_{chann} = f_s \cdot b_s = 705{,}600\,\mathrm{kbps}$$



(a) Arty A7 Pmod connectors



(b) Audio board Pmod connectors

Figure 4.7: Pmod connectors.

If we want to stay safe and also consider some overhead, let's fix the bandwidth per channel to

$$BW'_{chann} = 750{,}000\,\text{kbps}$$

Hence, the required bandwidth can be easily calculated as

$$BW = n_{chann} \cdot BW'$$

where n_{chann} is the number of channels.

4.4 TTA core

4.4.1 Introduction to TCE

To design the processor, *Tampere University's* **TTA-based Co-design Environment (TCE)** [18][19] was exploited. This open-source tool is aimed at creating and programming customized TTA processors, by providing a set of tools to co-design the RTL architecture and the high-level software running in it.

Table 4.1: Architectural pins assignments.

Arch. Pin	Arty A7 pin	Audio Board pin
clk_in1	E3	
rstx	C2	
i_UART_Rx_Serial_0	C15	J13_5
i_UART_Rx_Serial_1	D15	J13_7
i_UART_Rx_Serial_2	E16	J13_9
i_UART_Rx_Serial_3	E15	J13_11
o_UART_Tx_Serial_0	G13	J12_11
o_UART_Tx_Serial_1	D13	J12_12
i_data_i2s_0	D12	J12_5
i_data_i2s_1	K16	J12_6
o_data_i2s_0	J15	J13_6
o_bck_ADC_0	A11	J12_7
o_bck_ADC_1	A18	J12_8
o_bck_DAC	K15	J13_8
o_lrck_ADC_0	B11	J12_9
o_lrck_ADC_1	B18	J12_10
o_lrck_DAC	J18	J13_10
o_mclk	J17	J13_12
i_mosi_0	V10	
i_mosi_1	T13	
i_sck_0	U12	
i_sck_1	U14	
i_ss_n_0	V11	
i_ss_n_1	U13	
o_miso_0	V12	
o_miso_1	V14	

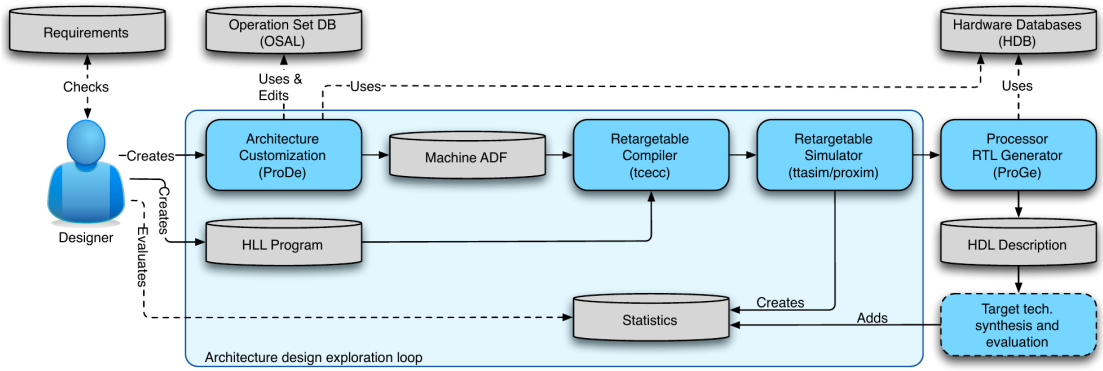


Figure 4.8: TCE design flow [18].

Figure 4.8 is a picture of the design flow of TCE: first of all the user creates an architecture for the processor with all the buses, register files, and function units required. Each function unit features a set of operations, that in first place are associated with a high-level language file that describes their external behavior (OSAL), without taking into account the internal architecture of the unit. In this way, the user can compile the custom program targeting it to the designed machine and then simulate it, without even having to implement it in hardware. After the simulation, the user can extract information about the units and buses usage and can see which ones were exploited the most, and eventually make some changes. Furthermore, it is possible to store the results of some variables, to see if the program is working properly. If everything is coherent with what is expected, the user can proceed to implement the real processor at the RTL level, by expoliting hardware databases **(HDB)** which contain the HDL descriptions of all the function units and their operations.
There are a wide variety of hardware units already present in TCE, each one performing one or more operations. There are even more versions of the same unit, with different output latencies, according to the specific use case. To perform all these tasks, TCE environment provides a series of software tools, each one dedicated to a different part of the design flow:

- **ProDe**: graphical tool to define FUs, registers, and interconnects of the TTA core.
- **tcecc**: to compile the user's code.
- **Proxim**: tool to simulate the firmware running on the basis of the OSAL descriptions.
- **OSEd**: database editor for OSALs.
- **HDBEditor**: database editor for hardware implementation descriptions.
- **ProGe**: to generate the HDL description of the TTA core and its components.

4.4.2 Processor architecture

The internal architecture of the TCE-developed processor is shown in figure 4.9. The four transport buses are 32-bits wide, as well as the data coming from/going to the units. Each function unit is fully connected to any one of the transport buses: thus, the decisions on how to route move operations among FUs are left to the TCE compiler, which will perform it according to the uploaded firmware routine. Note that in the picture only one instance of LED_DRIVER and one of SWITCH_DRIVER are shown for readability, while in reality there are four instances for each one.

4.4.3 Function Units: ports and operations

Having a closer look at the function units in figure 4.9 one can recognize some of the common fundamental building blocks of a processor, such as the **Global Control Unit (GCU)**, **Arithmetic Logic Unit (ALU)**, **Load-Store Unit (LSU)**, and two register

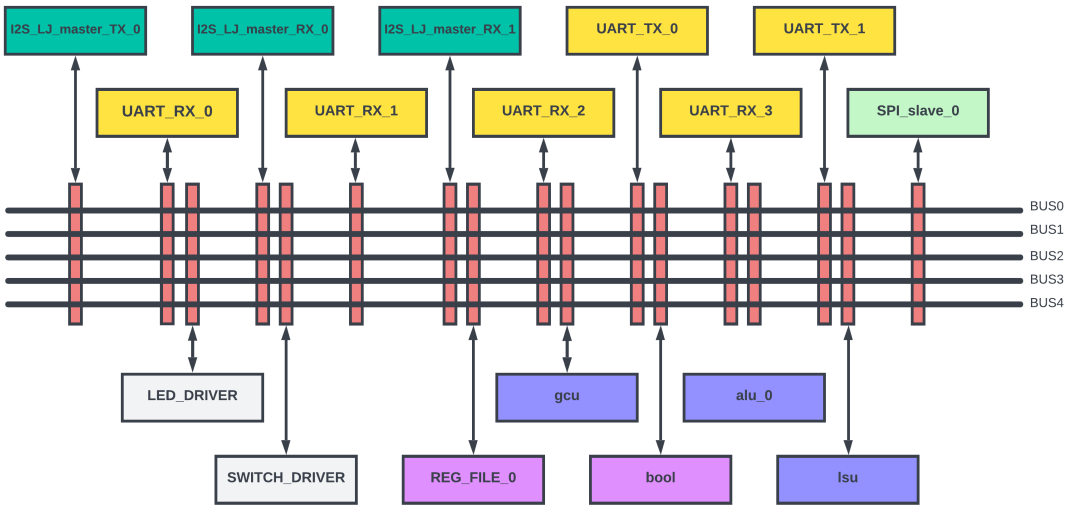


Figure 4.9: Processor internal architecture.

files, one being 32x32 bits and the other a 16x1 vector used to store bool values. The remaining units are custom ones [20] developed to deal with the dedicated serial peripherals.

Each unit features two types of ports, architectural and external.
Architectural ports are the ones that are connected to the transport buses of the TTA processor. They are exposed to the firmware programmer by being arguments of the software operations. At least one of these ports needs to be a **triggering port**, meaning that it is used to initiate the operation. If an architecture port is an input port, it needs also a related load port in the HDL file.
External ports are instead the ones that compose the external interface of the processor. Furthermore, units also feature special ports for the clock, glock, and reset signals and one opcode port to select among the available operations.

It is worth specifying that the serial peripheral units (I2S, UART and SPI) share a similar internal structure: one control unit that contains two elements, one core and one FIFO to buffer the data.
In the case of the receiver circuit, the core takes in input the serial data, ouputs the parallel data and feeds it inside the FIFO. In the case of transmitter circuit, the FIFO takes in input the parallel data, and outputs it to the input of the core, which in turn will output the serial data. Note that each of the FIFOs have separate write and read indexes and that the length of each one is 64. The architecture just described is summarized in figure 4.10.

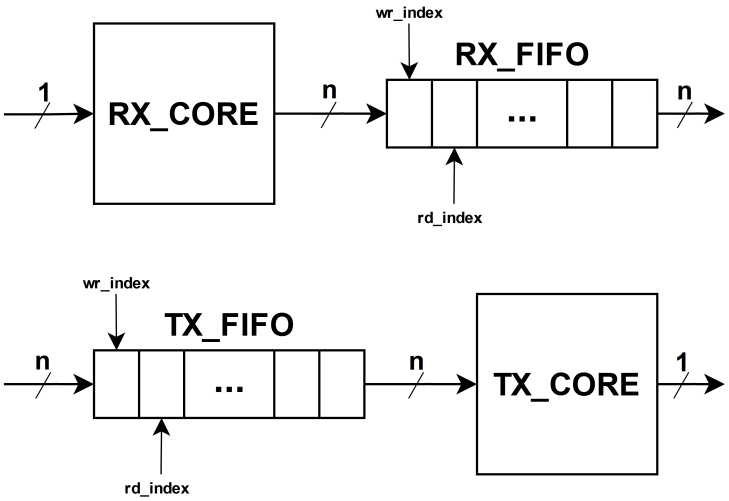


Figure 4.10: Transmitters and Receivers control unit structure. Parallel data is on n bits.

Function Units descriptions Following is a brief description of all the custom units of the processor. However, a better description of ports and operations for each of them is provided in Appendix A.
Each FU is followed by an image of the relative RTL view: note that when the design is simple enough to be printed in a readable format, the RTL view is showed also for one further internal hierarchical level.

I2s_LJ_master_TX_x (fig. 4.11): I2S transmitter with Left-Justified format and 24-bits audio samples on 32-bit frames. Internally, it is composed of two elements: the proper transmitter circuit and two FIFOs for data buffering (one for the left and one for the right channel). It features four operations:

- **PUSH_L, PUSH_R**: to load one word in the respective FIFO, that will then transmit it in order of arrival.
- **STATUS**: Used to output the internal state of FIFOs (Empty, Almost Empty, Almost Full, Full).
- **ZZZ**: Unused.

Ports **i_bck**, **i_bck_eoc**, **i_lrck**, **i_lrck_eoc**, and **i_mclk** are used to receive I2S clocks, eventually generated by a clock generator. **i_data_tx_l** and **i_data_tx_r** are the architectural ports used to transfer parallel input words, while **o_data** is the serial external output and **o_status** outputs internal FIFOs states.

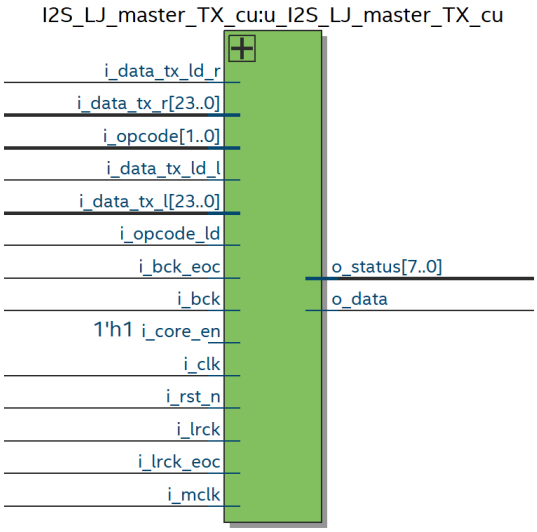


Figure 4.11: I2s transmitter.

I2s_LJ_master_RX_x (fig. 4.12): I2S receiver. Its behavior is symmetrical to that of the I2S transmitter. It features four operations: **POP_L**, **POP_R**, **STATUS**, and **ZZZ**.

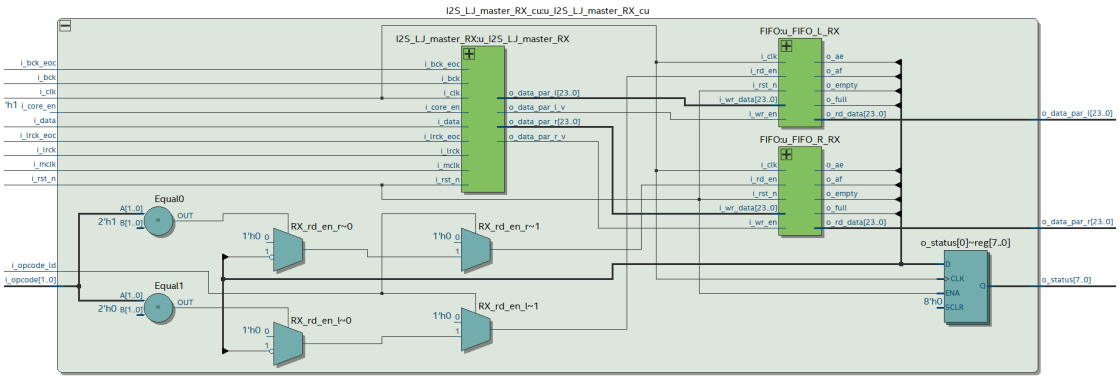


Figure 4.12: I2s receiver.

UART_TX_x (fig. 4.13): UART transmitter at 31250 Hz baud rate, used to send 8-bits MIDI messages, encapsulated between start bit and stop bit. It features two operations, **PUSH** and **STATUS**. **i_data_tx** is the architectural port by which parallel data to be transmitted enters, while **o_UART_Tx_Serial** is the serial external output port.

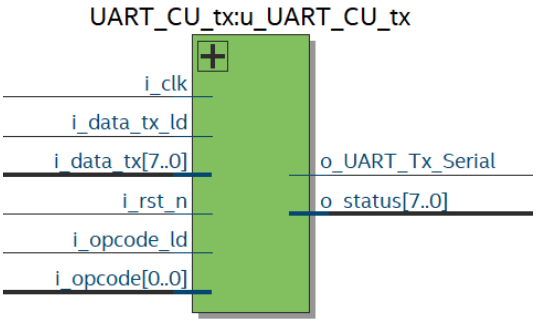


Figure 4.13: UART transmitter.

UART_RX_x (fig. 4.14): UART receiver at 31250 Hz baud rate, used to receive 8-bits MIDI messages. Its behavior is symmetrical to that of the UART transmitter. It features two operations, **PULL** and **STATUS**.

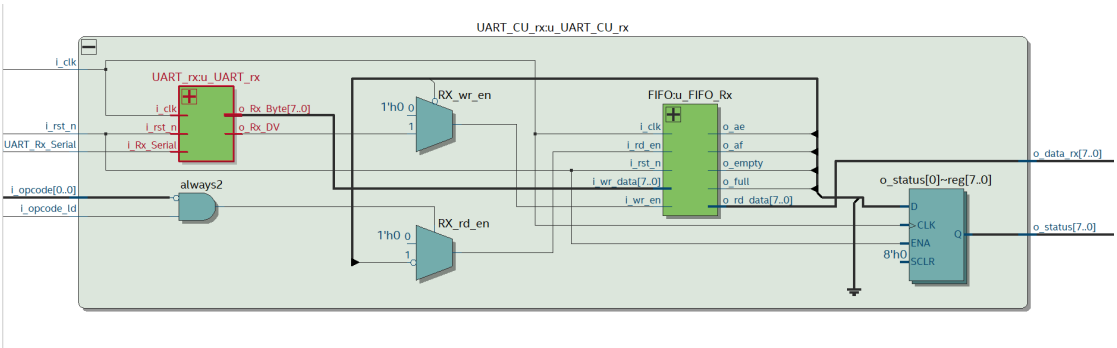


Figure 4.14: UART receiver.

SPI_slave_x (fig. 4.15): SPI transmitter/receiver. It features four operations, that work similarly to the previous ones: **POP**, **PUSH**, **POP_PUSH**, and **STATUS**. **POP_PUSH** merges the first two operations and allows to receive and send one SPI frame in the same operation. Each frame is composed by 8 bits. In addition to the other serial peripherals, it also features a 2 Flip-Flop synchronizer on MOSI, SS_n and SCK inputs to avoid metastability problems.

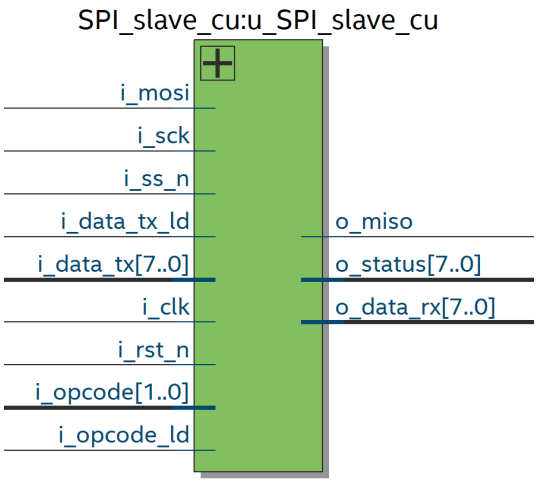


Figure 4.15: SPI slave.

LED_DRIVER_x (fig. 4.16): simple driver to control three led pins. It features only one operation, **SET_LED**, which is used to set the state of the three external outputs **o_blue**, **o_green** and **o_red**, according to the command sent to the **i_color_set** input internal port.

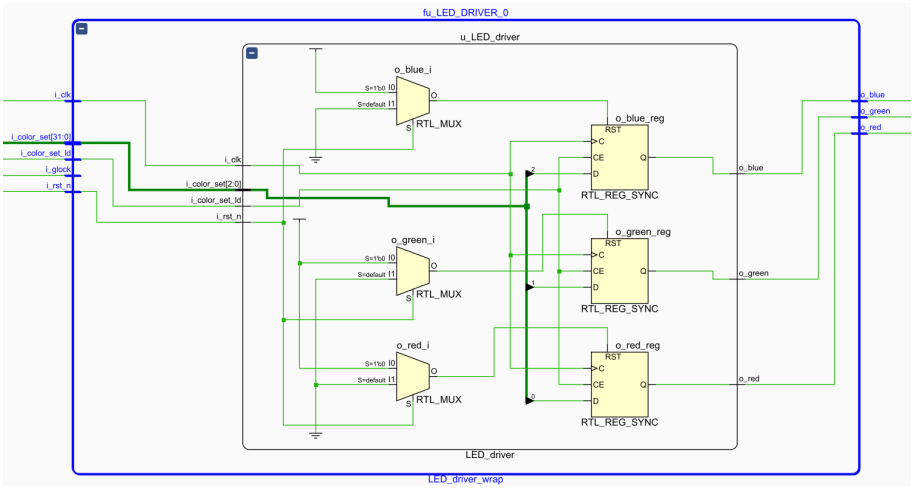


Figure 4.16: LED driver.

SWITCH_DRIVER_x (fig. 4.17): simple driver circuit for reading the state of a switch through external input port **i_switch**. According to it, the state is sent internally to the processor through the **o_status** port and the only operation **switch_status**.

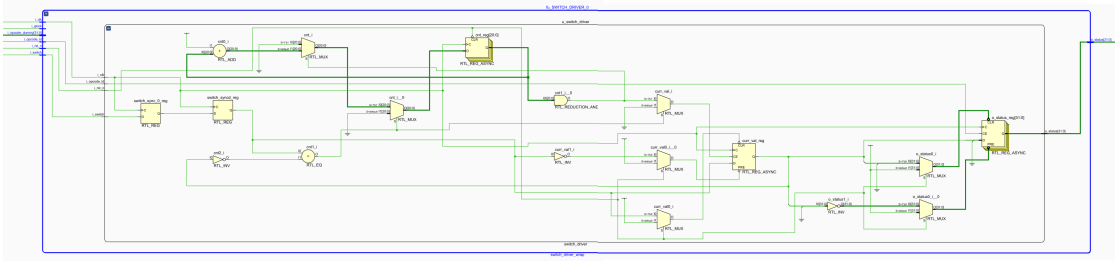


Figure 4.17: Switch driver.

4.5 Firmware

Once the hardware architecture is defined, firmware is the most flexible part of the design, in which we use the defined operations as C-style software functions. According to the purpose of the device, the FPGA system should be able to collect and send data from/to the custom audio board peripherals (I2S and UART, respectively for audio and MIDI samples), and Raspberry Pi4 ones (SPI0 and SPI1).

Modes of operation In figure 4.18 it is possible to see a flow diagram of the firmware routine [21].

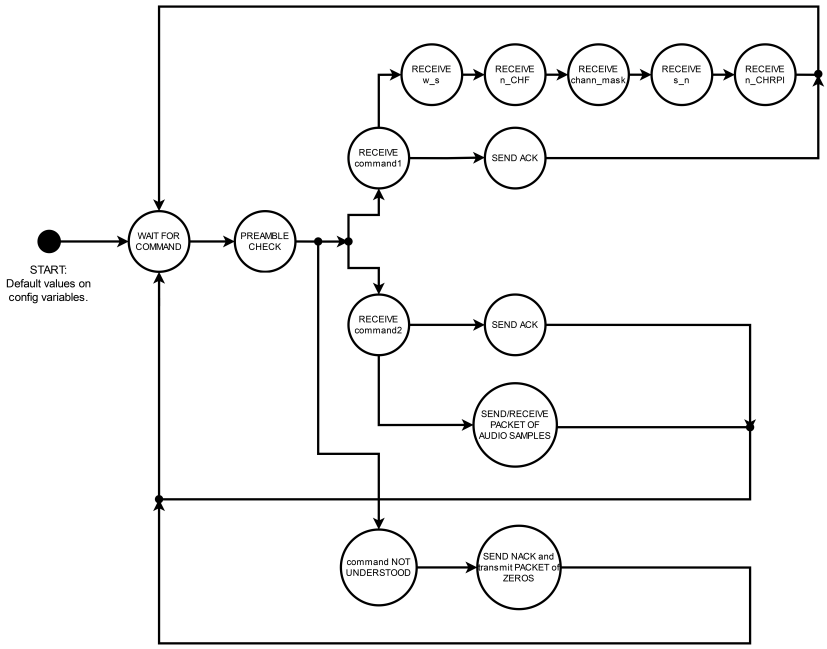


Figure 4.18: Flow diagram of firmware routine.

There is support for two main modes of operation, depending on wheter the mixing of audio samples is done at the RPi level or at the FPGA level (still to be developed). In

the first case, FPGA acquires samples from I2S channels and sends them directly to SPI0. RPi is then in charge of performing the mixing operations. Simultaneously, it also receives already mixed samples from RPi and streams them to I2S output channels. On the other hand, when working in mixing mode, the FPGA is supposed to both send already mixed channels through SPI0 and receive a certain amount of channels on which to perform mixing. This latter mode will lead to a better performance in terms of latency, but will likely have limitations when dealing with a very large amount of audio sources, due to the SPI channel bandwidth (as discussed in section 4.3.2).
There is also a set of configuration variables that can be modified at run-time using a specific command. In detail, they are:

- w_s: it represents the samples bitwidth (can be either 8, 16, 24 or 32 bits). Since SPI frame is on 8 bits, if a sample is composed by more than 8 bits, it will be divided into two or more consecutive frames. By default, w_s is 16 bits.
- n_{CHF}: it indicates the number of channels sent by FPGA in case of RPi-premixing. It can be a value among 1, 2 or 4. By default, it is equal to 4.
- *chann_mask*: a bitmask that sets which inputs to activate in the FPGA-system. Each bit is related to a channel (MSB for channel 3, down to LSB for channel 0). If a channel has to be activated, its bit is set to logic 1. By default chann_mask is equal to "1111", meaning that all the channels are active. The upper four bits of chann_mask are unused and are set to 0. Example: cann_mask equal to "0000 1010" means that only channels 3 and 1 are activated.
- s_n: it is the number of samples inside each packet of audio data. By default, it is set equal to 112.
- n_{CHRPI}: it indicates the number of channels that are coming from RPi-SPI in the case of mixing done by FPGA. By default, n_{CHRPI} is equal to 4.

Since the FPGA-SPI peripheral works as a slave device, it will receive commands from the RPi-SPI, which acts as the master. After starting, the FPGA system will enter an infinite loop, in which first of all it will wait for a command.
A protocol has been developed to try to minimize as much as possible the amount of errors which can occur along the SPI channel (eventually allowing the communication to run at more "unstable" frequencies). It is based on an ACK/NACK mechanism and implements redundancy of critical steps such as command issuing. This is of extreme importance, since it is mandatory that the command sequence is clear to both ends even at high speed, in particular in case of a configuration command: in fact, if a configuration command fails, the whole communication could be compromised.

Each command is preceded by 5 SPI frames (1 byte each) which work as a preamble (see table 4.2 for its value). By FPGA side, if it receives frames that have an Hamming distance less or equal than two with respect to the preamble sequence, and if at least 3 out of the 5 bytes are equal to it, it understands that RPi wants to send a command. Right after sending the preamble, RPi sends the 8 bit command on three consecutive SPI frames and FPGA applies to it the same criteria as for the preamble, except that this

time two out of three of the command frames have to be equal to one of the available commands (figure 4.19 shows the time diagram for this command sequence). Then, RPi receiver (transmitter can work indipendently) waits for an acknowledge (ACK) or an not-acknowledge (NACK) coming from FPGA. Coming back to the FPGA side, after receiving the preamble+command sequence, it has to send the ACK/NACK sequence. If everything worked correctly, an ACK is sent, which consists in sending the same sequence of preamble+command back to RPi (figure 4.20). If instead the command coming from RPi was not clear, the NACK sequence is transmitted (figure 4.21), which consists in the same 5 frames of preamble, but this time followed by the **error sequence** (see table 4.2 for its value).

Table 4.2: Preamble, error and commands sequences values in binary and decimal formats.

SEQUENCE	VALUE
Preamble	10101010 (170)
Error	11110000 (240)
Command 1	00110011 (51)
Command 2	00011000 (24)

Following is the list of the commands RPi can use to control the SPI communication:

- **Command 1**:It is the command for changing the configuration variables. After the preamble+command sequence, RPi sends 5 frames with the value of the first variable, then 5 with the value of the second, and so on (figure 4.22). The order must be the one shown in the list above and each single variable has to be set. If FPGA sends a NACK for command 1, it is of extreme importance that RPi sends back again the command and the configuration, otherwise the whole communication will be compromised.

- **Command 2**: It is the command for starting the streaming of a packet of samples in RPi-mix mode. After sending the preamble+command sequence, RPi starts to send its interleaved audio samples, while waiting for an ACK/NACK. If a NACK is received, no recovery of the lost packet is performed, and the next packet is transmitted, if available. On the FPGA side, after the command reception, the receiver immediately listens on the MOSI line for the samples. The transmitter will instead send the ACK sequence and immediately start the transmission of the FPGA packet of audio samples. Since the audio samples are grouped in packets (determined by $s_n \cdot max(2, n_{CHF})$, since RPi sends two channels even when $n_{CHF} = 1$), for a total of $s_n \cdot max(2, n_{CHF}) \cdot (w_s/8)$ SPI frames, both FPGA and RPi will have internal counters to count how many samples they received/transmitted. When the packet transmission is concluded on both sides, RPi can send again command 2 if another packet is needed to be streamed. A timing diagram for Command 2 is exhibited in figure 4.23.

- **Command 3** will be used in future works for the streaming of audio samples with the mixing operation in FPGA.

It is worth noting that the redundancy introduced in commands does not introduce a great overhead term, in particular in case of **Command 2**: only 8 SPI frames are used at each side, while the SPI packet to stream can be much longer (its length depends on w_s, n_{CHF} and s_n). If the default values are used, the overhead OH introduced by the redundancy is only

$$OH = 8/(((w_s/8) \cdot n_{CHF} \cdot s_n) + 8) = 8/(((16/8) \cdot 4 \cdot 112) + 8) \simeq 0.89\%$$

Synchronicity In case FPGA is not able to understand a certain command, it is important that it sends a packet of zeros to maintain the synchronicity: in fact, if the next command by RPi is still command2, the user will eventually note a glitch given by the previous silence, but the time coherence won't be lost. Furthermore, if the next command is command1, there will be a certain amount of time for the new settings to take place and so we can assume that if a streaming starts again, a new "zero point" in time is being set and so the synchronicity is still guaranteed.
The entire code is provided in Appendix B.

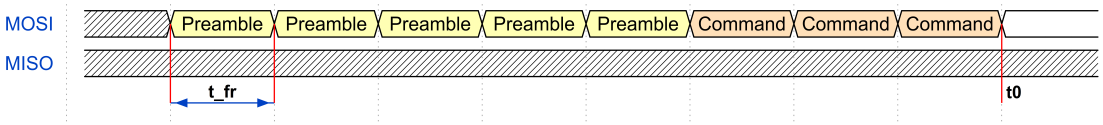


Figure 4.19: MOSI starting a command

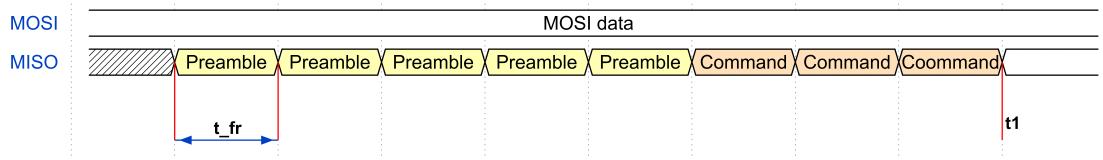


Figure 4.20: MISO ACK

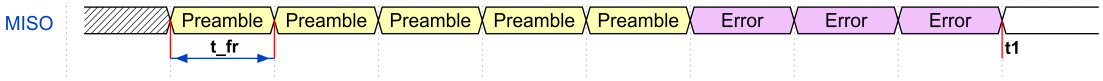


Figure 4.21: MISO NACK

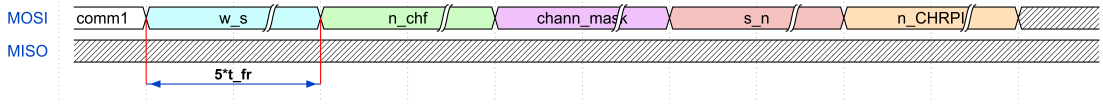


Figure 4.22: Command 1

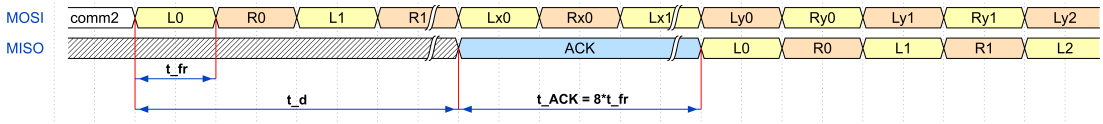


Figure 4.23: Command 2 ($w_s = 8$)

Figure 4.24: Timing diagrams for firmware SPI communication. Horizontal axis is time. *t_fr* is the time of one SPI frame (8 bits), *t_d* is the time required by FPGA to process RPi request before sending ACK and start audio streaming.

Chapter 5

Results Validation

In this chapter, the system behaviour is analysed and evaluated. Many experiments are performed, either in a simulation environment and in real-world scenarios, starting form the single function units and ending to examine the whole FPGA system and its interface with the external devices. The main metrics to be analysed are the results accuracy and the speed performance in terms of latency.

5.1 Test of peripherals and function units

5.1.1 Description of firmware and testbench

A first and simplified version of the firmware (see appendix B for the code) has been developed to perform a loopback test of the serial peripherals: when the board switch is pressed, a LED light is turned on and each RX function unit is first required to pop some data from its FIFOs. Then, the data is pushed back by the relative TX units. If the switch is not pressed, LED turns off and TX units send zeros. Each instance of every peripheral has been tested as well.

A functional simulation of this routine has been performed. The testbench simulates external peripherals attached to the FPGA system. To do so, it sends to the inputs of the RX units a series of "samples" and then performs a control on the TX outputs to see if any data was corrupted during the process. The list of samples sent to the RX units used to generate the following images is shown in table 5.1.

Table 5.1: Testbench samples for the three peripherals.

I2S	UART	SPI
24'h800001	10'b**0**10101010**1**	8'b10001000
24'h7FFFFE	10'b**0**11001100**1**	8'b10101010
24'hAAAAAA	10'b**0**11110000**1**	8'b11001100
24'h555555	10'b**0**01010101**1**	8'b11110000
24'hCCCCCC	10'b**0**11100100**1**	8'b00000000
24'h333333		
24'hF0F0F0		
24'h0F0F0F		
24'hFF00FF		

5.1.2 Behavioral simulation

By looking at figures 5.1 for I2S, 5.2 for UART and 5.3 for SPI, it is possible to verify the simulation results. Note that some of the matching frames are highlighted with rectangles of different colors to improve readability of results.

For I2S, we see that each sample generated by the testbench (**data_RX_i2s signal**) is recognised by the receiver (in **o_data_rx_l** or **o_data_rx_r** depending on which was the lrck value when the sample arrived) after the two clock lrck delays required. Furthermore, the loopback tranmission in the serial **data_Tx_i2s** is correct as well and each of the output samples is aligned with the respective lrck edge, as expected while using the LJ format. We see that the the latency between one frame on the **data_Rx_i2s** line and the relative loopback frame in the d**ata_Tx_i2s** is around 1.5 ms: this is mainly due to the transmitter FIFO. In fact, if one analyzes the internal signals of the two TX and RX I2S peripherals, the following situation will be seen: on the RX FIFO, the write and read indexes would constantly be at the same value, but the write enable signal always arrives before the read enable signal. Thus, the received frame is practically read immediately. On the other hand, the processor gives as input to the TX circuit the value of the received frame, but in this case the read enable signal arrives before the write enable signal. Hence, in order to read the frame, the read index has to go through the whole FIFO until it arrives back to the correct index.

We can also evaluate this latency mathematically: being the mclk period equal to

$$t_{mclk} = 1/(11.289{,}6\,\text{MHz}) = 88.577\,\text{ns}$$

the ratio between master clock and bit clock $MtoB = 4$ and the length of a I2S frame inside the processor equal to $l_{fr} = 32$, the time of one I2S frame is

$$t_{fr} = l_{fr} \cdot t_{mclk} \cdot MtoB = 11.338\,\mu\text{s}$$

Then, since the write and read enable signals are asserted only once each lrck period ($t_{lrck} = t_{fr} * 2$) and since the length of the RX FIFO is $l_{FIFO} = 64$, the resulting latency

of the loopback frame is

$$t_{lat} = l_{FIFO} \cdot t_{lrck} \simeq 1.45\,\text{ms}$$

It is clear that in this case the delay introduced by the processor is negligible. To reduce t_lat one solution could be to use shorter FIFOs (reducing l_FIFO) or trying to modify the TX in order to avoid reading before writing from the FIFO.

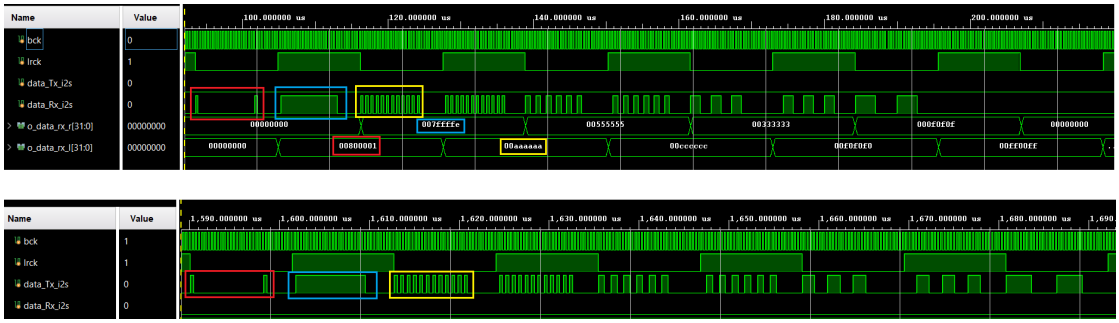


Figure 5.1: Simulation results for I2S peripheral. RX sequence above, TX sequence below.

For UART/MIDI, we have a similar situation, with the serial commands coming from the testbench in **UART_Rx_Serial** signal, the parallelized commands from the receiver in **o_data_rx[31:0]** (with all the bits reversed) and the serial output from the transmitter in **UART_Tx_Serial**. We notice that the samples experience a delay of about 21/22 ms, which is due to the same consideration done for the I2S case. This might sound like a huge latency term for the application, but some considerations have to be made. In fact, this is the case with a FIFO of length 64. Decreasing this size leads to a lower delay term, down to the ideal best case with a FIFO of length 1. In this case, the delay would be equal to

$$t_{lat} = t_{baud} \cdot n_{bits} \cdot l_{FIFO}$$

Being t_{baud} the inverse of the baud rate (31250 Hz for MIDI), $n_{bits} = 10$ the number of bits in a frame (8 bits of data + start bit + stop bit) and $l_{FIFO} = 1$ the length of the FIFO, the following value is obtained

$$t_{lat} = 0.32\,\text{ms}$$

Indeed, this is a quite more acceptable value. So, it is possible to tune l_{FIFO} length to obtain the best trade-off between buffering and latency.

Finally, we consider SPI loopback (figure 5.3). Simulations were performed with a virtual synchronous clock that spans from 1 MHz to 50 MHz (in figure the case with the latter value), obtaining correct results in each case. As for the other peripherals the serial input (**mosi** signal) is correctly received and parallelized by the RX circuit (**o_data_rx[7:0]** and **o_data_rx[31:0]** signals). On the other hand, the processor takes the received values and sends them to the TX circuit (**i_data_tx[7:0] signal**), which correctly outputs the serial data (**miso** signal), aligned with the synchronous clock **sck_sync_d**. Looking at the time, data on the mosi line takes around 11 μs (at $f_{sck} = 50\,\text{MHz}$ and FIFO length equal to 64) to be received and then transmitted back on the miso line.

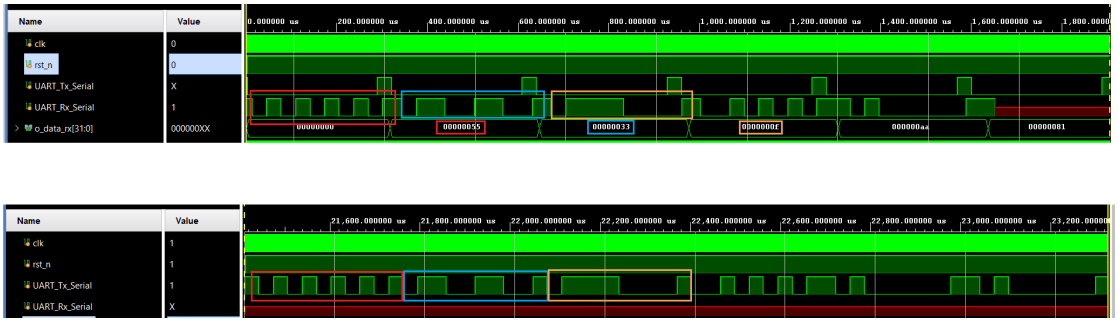


Figure 5.2: Simulation results for UART peripheral. RX sequence above, TX sequence below.

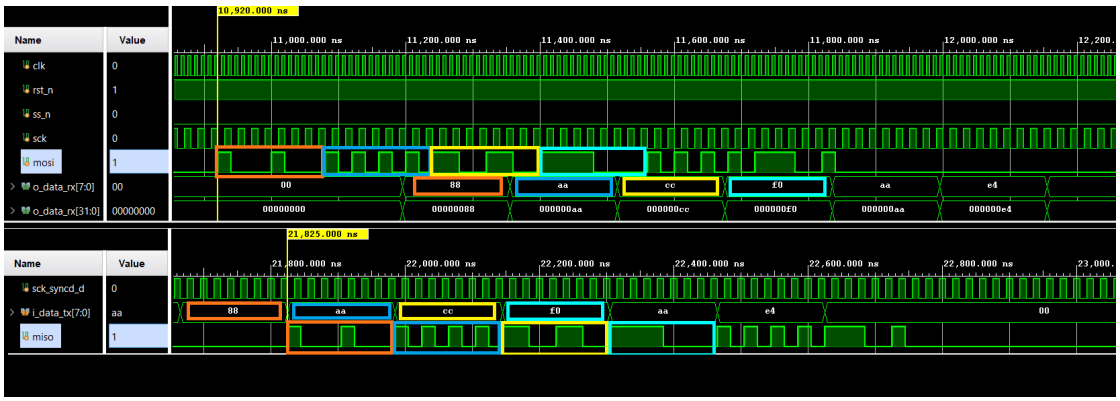


Figure 5.3: Simulation results for SPI peripheral. RX sequence above, TX sequence below.

5.1.3 Real-world Tests

Finally, the processor system was loaded inside the FPGA chip and some tests were performed. First of all, the FPGA board was connected to the audio board through the JA and JB connectors (as explained in chapter 4), to see if the UART/MIDI and I2s loopback routines were performing correctly. Setup for the audio part can be seen in figure 5.4. As expected, when pressing switch button 0, the streaming starts and the audio from the XLR cable is sent to the FPGA board and then back to the headphones output. From the user's point of view, there is virtually negligible latency.

Then, to test the SPI peripheral, the FPGA board has been connected through GPIO pins to an STM32 NUCLEO-F401RE board (see figure 5.5), which acts as a sample generator and then checks the samples sent back by the processor, in a sort of echo test. Succesful results were obtained up to an SCK frequency of 21 MHz. Anyway, we can expect higher frequencies, since this test was performed in non-optimal conditions: in fact, long jumper wires were exploited to connect the NUCLEO GPIO pins to the FPGA board and the firmware simply used a polling strategy to collect and send SPI frames. A better connection and a better software management will for sure lead to better performance in

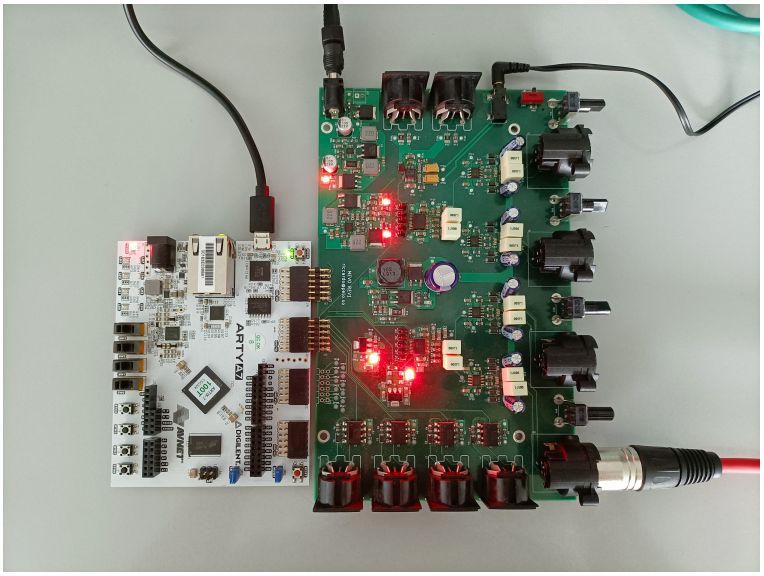


Figure 5.4: Setup for audio board and FPGA interfacing through I2S.

terms of achievable SPI clock speed.

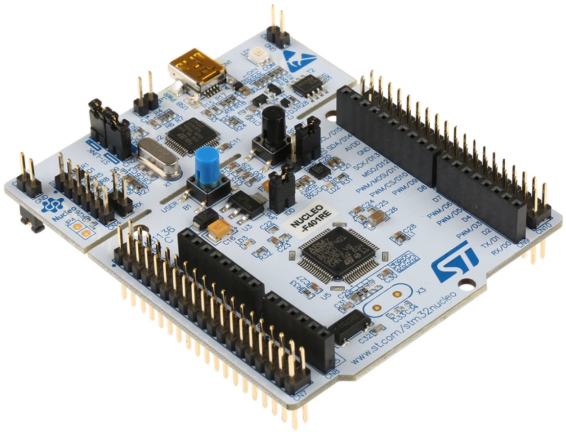


Figure 5.5: NUCLEO STM32 F401RE microcontroller board.

5.1.4 Post-Implementation Simulation

After the implementation with Vivado tool has been performed, a functional simulation of the system with the same testbench has been performed to guarantee that the processor is still working correctly. The results show coherent results with respect to the behavioral simulation.

5.2 Final architecture implementation

In this section are analyzed the steps that led from TCE design to the FPGA implementation of the processor described in chapter 4.

After describing the blocks and buses of the architecture with the help of the Processor Designer tool (ProDe), the generated processor features an instruction word width equal to **176** bits. Then, the final developed firmware (see appendix B) was compiled for the generated architecture and loaded inside the Proxim simulator: this tool generates the machine code that runs in the processor and can execute it step by step: the resulting code has a total of **364** instructions. One interesting thing to be noticed is that even if the code instructions are written in the traditional sequential way, the TCE compiler is able to understand if two software operations can be run in parallel. This is the case depicted in figure 5.6: notice in the highlighted lines how operations involving SPI and I2S TX/RX peripherals are run in parallel in the same instruction thanks to the four buses.

			0: B1	1: B1_1	2: B1_1_1	3: B1_1_2
224			...	1 -> I2S_LJ_master_RX_0.in1t.i2s_rx_pop_r	alu.out1 -> REG_FILE_0.25	...
225			alu.out1 -> SPI_SLAVE_0.in2	255 -> alu.in1t.and	I2S_LJ_master_RX_0.out3 -> alu.in2	1 -> SPI_SLAVE_0.in1t.spi_pop_push
226			0 -> SPI_SLAVE_0.in2	1 -> SPI_SLAVE_0.in1t.spi_pop_push	REG_FILE_0.19 -> alu.in2	4294967295 -> alu.in1t.add
227			...	255 -> alu.in1t.and	SPI_SLAVE_0.out1 -> alu.in2	alu.out1 -> SPI_SLAVE_0.in2
228			...	alu.out1 -> REG_FILE_0.19	REG_FILE_0.4 -> alu.in2	alu.out1 -> alu.in1t.eq
229			...	alu.out1 -> REG_FILE_0.20	0 -> I2S_LJ_master_RX_1.in1t.i2s_rx_pop_l	...
230			1 -> SPI_SLAVE_0.in1t.spi_pop_push	I2S_LJ_master_RX_1.out1 -> alu.in2	255 -> alu.in1t.and	alu.out1 -> bool.1
231			0 -> SPI_SLAVE_0.in2	2 -> SPI_SLAVE_0.in1t.spi_push	1 -> I2S_LJ_master_RX_1.in1t.i2s_rx_pop_r	!bool.1 221 -> gcu.pc.jump
232			alu.out1 -> SPI_SLAVE_0.in2	2 -> SPI_SLAVE_0.in1t.spi_push	I2S_LJ_master_RX_1.out3 -> alu.in2	255 -> alu.in1t.and
233			REG_FILE_0.25 -> I2S_LJ_master_TX_0.in2	0 -> I2S_LJ_master_TX_0.in1t.i2s_tx_push_l	0 -> SPI_SLAVE_0.in2	2 -> SPI_SLAVE_0.in1t.spi_push
234			REG_FILE_0.20 -> I2S_LJ_master_TX_0.in3	1 -> I2S_LJ_master_TX_0.in1t.i2s_tx_push_r	alu.out1 -> SPI_SLAVE_0.in2	2 -> SPI_SLAVE_0.in1t.spi_push
235			...	...	...	36 -> gcu.pc.jump

Figure 5.6: Parallelization of FUs instructions.

Furthermore, the Proxim tools can give an estimation of the usage of each part of the generated processor after a certain number of code cycles. Since many of the instructions involving SPI and I2S peripherals are executed only when the command sequence is received from Raspberry Pi through SPI, the firmware has been modified to always enter the command2 condition (see section 4.5). The statistics show that all the four buses are equally exploited, with an average utilization of 84% each. As expected, we see a perfect balance between the I2S_TX_PUSH_L and I2S_TX_PUSH_R operations (50% usage of I2S_LJ_master_TX_0 unit each). Hence, the same result holds also for I2S_RX_POP_L and I2S_RX_POP_R for the two I2S RX peripherals and for SPI_POP_PUSH and SPI_PUSH in the SPI_0 peripheral.

5.2.1 Synthesis and implementation

Once the Proxim analysis was concluded, synthesis and implementation were performed in the Vivado software, with a target system clock speed of $f_{clk} = 100\,\text{MHz}$. For **synthesis**, the directive *PerformanceOptimized* was exploited, since speed of the final system is the main

objective. **Retiming** was also used in order to add a degree of freedom for the synthesizer to reorganize the registers. Regarding the **implementation**, the *Performance_Auto_1* strategy was used, with *opt_design* and *phys_opt_design* directives checked. After all the steps, the processor was succesfully implemented, and all the timing constraints were verified. In the following sections, various reports for the implemented design are presented.

Resources usage

In tab 5.2 we can see a summary of the resources usage after synthesis and after implementation steps, while in tab 5.3 we see a more detailed summary of only the latter one. We see that since the amount of utilization of LUT with respect to the available amount is quite low (about 20%), there is a large amount of LUTRAM exploited, since it is generally faster than BRAM. In particular, if we look in detail inside the reports, we see that the majority of LUTRAM is exploited in form of Distributed RAM, used to implement the synchronous data memory (8192 out of 8378 LUTRAM blocks). It is also possible to see that the only two BRAM blocks are used in the FIFOs of I2S receivers and that the only three DSP blocks used are assigned to the ALU module.

Table 5.2: Resources usage after synthesis and after implementation steps.

	LUT	FF	BRAM	URAM	DSP
Post-synthesis	13916	3881	2	0	3
Post-implementation	13545	3980	2	0	3

Table 5.3: Detailed resources usage after implementation step.

Resource	Utilization	Available	Utilization %
LUT	13545	63400	21.36
LUTRAM	8378	19000	44.09
FF	3988	126800	3.15
BRAM	2	135	1.48
DSP	3	240	1.25
IO	42	210	20.00
MMCM	1	6	16.67

Power report

Even if power is not the major concern for this project, it is worth reporting the total On-Chip Power consumption estimated by Vivado. By looking at figure 5.7, it is clear that the major contribution to the total power consumption (equal to 0.326 W) is determined by the 70% of the dynamic power. In particular, MMCM, which is the PLL used to create the I2S master clock, is dissipating more than 50% of this power alone. The other three major contributions are given by Clocks (system clock and I2S master clock), by signals and by logic.

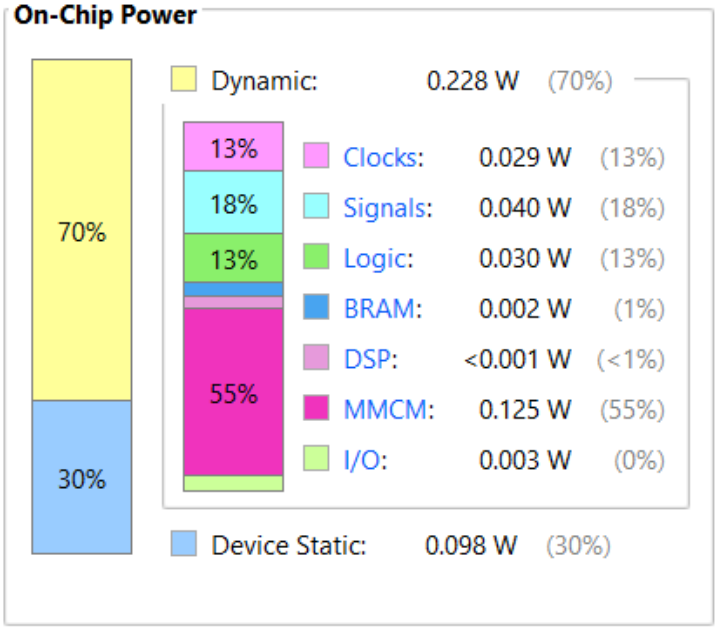


Figure 5.7: Power consumption estimation.

Timing

After completing the implementation step, all the timing constraints are satisfied and in tab. 5.4 it is possible to see a brief summary of the results regarding Setup, Hold and Pulse Width constraints. One thing to be observed is that the worst slack and the resource usage seem to be sensitive to variations in the firmware code: more complex is the latter, lower is the slack and higher is the utilization of resources. Anyway, in case of future improvements that will require a larger firmware (let's think to implementation of command 3 mentioned in section 4.5) it is predictable that more hardware support will be added to the processor and so the slack trend will be mitigated. Also, more aggressive Vivado synthesis and implementation strategies can be explored.

Table 5.4: Timing report values.

	Setup	Hold	Pulse Width
Worst Slack	0.159ns	0.033ns	3.000ns
Total Slack	**0**	**0**	**0**
Failing endpoints	0	0	0
Total endpoints	90684	90684	12376

Below is a report showing the critical path and in figure 5.8 is an RTL view of it. It is

shown that the critical path starts from the C input of the

$$rf\_REG\_FILE\_0\_wr\_opc\_reg\_reg[1]\_rep$$

register of the instruction fetch module, and ends in the D input of the

$$addr\_reg\_reg[1]\_replica\_1$$

register of the load-store unit. Analyzing the report, it is clear that the highest contribution in the total delay (equal to 9.456ns) is given by the Net Delay (equal to 7.988), contributing for a 84.48% over the total, while the logic delay contributes only for a 15.52%.

```
Timing Report

Slack (MET) :             0.159ns  (required time - arrival time)
  Source:                 u_top/core/inst_decoder/rf_REG_FILE_0_wr_opc_reg_reg[1]_rep/C
                            (rising edge-triggered cell FDCE clocked by clk_in1  {rise@0.000ns fall@5
    ↪ .000ns period=10.000ns})
  Destination:            u_top/core/fu_lsu/addr_reg_reg[1]_replica_1/D
                            (rising edge-triggered cell FDCE clocked by clk_in1  {rise@0.000ns fall@5
    ↪ .000ns period=10.000ns})
  Path Group:             clk_in1
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (clk_in1 rise@10.000ns - clk_in1 rise@0.000ns)
  Data Path Delay:        9.456ns  (logic 1.468ns (15.525%)  route 7.988ns (84.475%))
  Logic Levels:           6  (LUT4=1 LUT6=4 MUXF7=1)
  Clock Path Skew:        -0.292ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.923ns = ( 14.923 - 10.000 )
    Source Clock Delay      (SCD):    5.402ns
    Clock Pessimism Removal (CPR):    0.187ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
-------------------------------------------------------------------    -------------------
                         (clock clk_in1 rise edge)    0.000     0.000 r
    E3                                                0.000     0.000 r  clk_in1 (IN)
                         net (fo=0)                   0.000     0.000    clk_in1
    E3                   IBUF (Prop_ibuf_I_O)         1.482     1.482 r  clk_in1_IBUF_inst/O
                         net (fo=1, routed)           2.025     3.506    clk_in1_IBUF
    BUFGCTRL_X0Y16       BUFG (Prop_bufg_I_O)         0.096     3.602 r  clk_in1_IBUF_BUFG_inst/O
                         net (fo=12180, routed)       1.799     5.402    u_top/core/inst_decoder/
    ↪ clk_in1
    SLICE_X41Y20         FDCE                                         r  u_top/core/inst_decoder/
    ↪ rf_REG_FILE_0_wr_opc_reg_reg[1]_rep/C
-------------------------------------------------------------------    -------------------
    SLICE_X41Y20         FDCE (Prop_fdce_C_Q)         0.456     5.858 r  u_top/core/inst_decoder/
    ↪ rf_REG_FILE_0_wr_opc_reg_reg[1]_rep/Q
                         net (fo=128, routed)         1.684     7.542    u_top/core/rf_REG_FILE_0/
    ↪ TX_wr_data_temp_reg[0]_i_60_0
    SLICE_X28Y33         LUT6 (Prop_lut6_I2_O)        0.124     7.666 r  u_top/core/rf_REG_FILE_0/
    ↪ addr_reg[1]_i_70/O
                         net (fo=1, routed)           0.000     7.666    u_top/core/rf_REG_FILE_0/
    ↪ addr_reg[1]_i_70_n_0
    SLICE_X28Y33         MUXF7 (Prop_muxf7_I1_O)      0.217     7.883 r  u_top/core/rf_REG_FILE_0/
    ↪ addr_reg_reg[1]_i_58/O
                         net (fo=1, routed)           0.882     8.765    u_top/core/rf_REG_FILE_0/
    ↪ addr_reg_reg[1]_i_58_n_0
    SLICE_X22Y33         LUT6 (Prop_lut6_I5_O)        0.299     9.064 r  u_top/core/rf_REG_FILE_0/
    ↪ addr_reg[1]_i_31/O
                         net (fo=4, routed)           0.924     9.988    u_top/core/inst_decoder/
    ↪ rf_REG_FILE_0_r1data_wire[3]
    SLICE_X13Y22         LUT4 (Prop_lut4_I3_O)        0.124    10.112 r  u_top/core/inst_decoder/
    ↪ addr_reg[1]_i_13/O
                         net (fo=1, routed)           0.575    10.687    u_top/core/inst_decoder/
    ↪ addr_reg[1]_i_13_n_0
    SLICE_X13Y22         LUT6 (Prop_lut6_I3_O)        0.124    10.811 r  u_top/core/inst_decoder/
    ↪ addr_reg[1]_i_3/O
                         net (fo=14, routed)          1.076    11.887    u_top/core/inst_decoder/
    ↪ databus2[3]
    SLICE_X25Y24         LUT6 (Prop_lut6_I1_O)        0.124    12.011 r  u_top/core/inst_decoder/
    ↪ addr_reg[1]_i_1/O
                         net (fo=13, routed)          2.846    14.858    u_top/core/fu_lsu/addr_reg_reg
    ↪ [13]_12[1]
```

```
48    SLICE_X63Y79          FDCE                                          r   u_top/core/fu_lsu/addr_reg_reg
        ↪ [1]_replica_1/D
49    ------------------------------------------------------------------------------------------------------
50
51                          (clock clk_in1 rise edge)    10.000    10.000 r
52    E3                                                  0.000    10.000 r   clk_in1 (IN)
53                          net (fo=0)                    0.000    10.000     clk_in1
54    E3                    IBUF (Prop_ibuf_I_O)          1.411    11.411 r   clk_in1_IBUF_inst/O
55                          net (fo=1, routed)            1.920    13.331     clk_in1_IBUF
56    BUFGCTRL_X0Y16        BUFG (Prop_bufg_I_O)          0.091    13.422 r   clk_in1_IBUF_BUFG_inst/O
57                          net (fo=12180, routed)        1.500    14.923     u_top/core/fu_lsu/clk_in1
58    SLICE_X63Y79          FDCE                                          r   u_top/core/fu_lsu/addr_reg_reg
        ↪ [1]_replica_1/C
59                          clock pessimism               0.187    15.110
60    SLICE_X63Y79          FDCE (Setup_fdce_C_D)        -0.058    15.016     u_top/core/fu_lsu/addr_reg_reg
        ↪ [1]_replica_1
61    ------------------------------------------------------------------------------------------------------
62                          required time                          15.016
63                          arrival time                          -14.858
64    ------------------------------------------------------------------------------------------------------
65                          slack                                   0.159
```

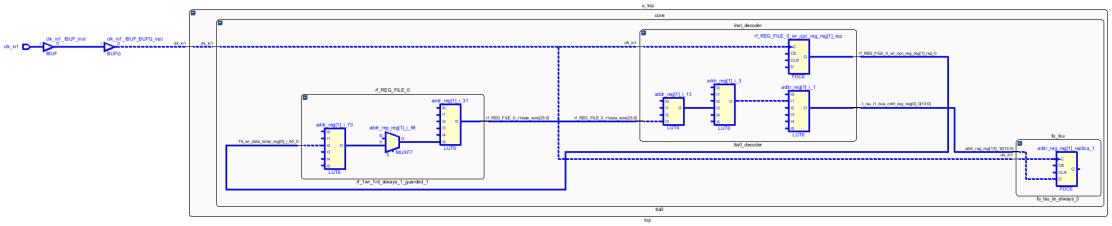


Figure 5.8: RTL view of critical path.

Chapter 6

Conclusions

In this thesis, the base structure for an FPGA-based processor integrated inside a Networked Music Performance application environment has been designed and analyzed. The need to examine hardware solutions comes from the strict latency requirements that the study case asks for, while the FPGA solution was chosen due to its great characteristic of being easily reconfigurable. Among all architectures, the Transport Triggered (TTA) one has been chosen due to its suitability for application-specific tasks and the possibility to easily parallelize different operations. Hence, the processor has been developed using a set of tools aimed at helping during the codesign of software and hardware. The final design resulted in a four buses architecture featuring custom units that deal with the serial links connecting the FPGA system to the external custom audio board and Raspberry Pi 4.

After that, some loopback tests were performed to verify the correct behavior of the serial peripherals when integrated inside the processor, and the latency introduced by these operations was calculated. Results show that latency is directly proportional to the transmitter FIFO length and that this value can be adapted by taking into account a trade-off between buffering and delay. Calculated latencies vary from microseconds in the SPI case, to few milliseconds for I2S and UART.

Later, a communication protocol has been proposed for the interaction between the FPGA system and Raspberry Pi 4 through the SPI bus. It is based on an ACK/NACK mechanism and uses a preamble to identify commands. A certain amount of redundancy is introduced in the commands to reduce as much as possible the possibility of errors in the communication and guarantee the reliability of the link.

Finally, this protocol has been adapted and translated into code to implement audio streaming with mixing performed at Raspberry Pi level. The resulting firmware has been loaded inside the designed processor and finally implemented for the target FPGA chip at a system clock rate of 100 MHz. The resulting architecture shows good utilization of the available resources, in particular an even usage of the four transport buses and a good parallelization of the tasks. All the timing requirements are satisfied and the resources and power reports show that there is still a lot of free space for further improvement of the designed processor.

Future work Considering possible future developments, the first step should be focused on the "real-world" testing of the developed interface protocol between Raspberry Pi and the designed FPGA. Then, an incremental movement of tasks from Raspberry Pi firmware to FPGA hardware should be started. In these terms, the first assignment could be to port the mixing operation inside the processor, as introduced in section 4.5, by eventually developing a dedicated functional unit. In this way, the latency of this task could benefit from the speed enhancement introduced by a hardware-oriented solution. Finally, another step to be developed could consist in adding some filtering units to the processor to perform some equalization directly on the audio sources coming from the custom audio board.

Appendix A

Function units ports and operations

This appendix shows the input/output ports and the operations of the custom function units used in the TTA processor. A brief description of each one is provided. Note that architectural ports are written in bold.

Table A.1: fu_UART_TX_x

fu_UART_TX_x			
INPUT PORTS	**Description**	**OUTPUT PORTS**	**Description**
i_clk	System clock	o_UART_Tx_Serial	Serial output
i_glck	Global lock	**o_status[32]**	TX status
i_rstn	Neg. reset		
i_opcode[1:0]	Opcode		
i_opcode_ld	Opcode load		
i_opcode_dummy[32]	Triggering Opcode		
i_data_tx[32]	Parallel input		
i_data_tx_ld	Par. IN load port		
OPERATION		**Description**	
0. uart_tx_push		Push data frame to be transmitted	
1. uart_tx_status		Return TX status	

Table A.2: fu_UART_RX_x

fu_UART_RX_x			
INPUT PORTS	**Description**	**OUTPUT PORTS**	**Description**
i_clk	System clock	**o_data_rx[32]**	Parallel output port
i_glck	Global lock	**o_status[32]**	RX Status
i_rstn	Neg. reset		
i_opcode[1:0]	Opcode		
i_opcode_ld	Opcode load		
i_opcode_dummy[32]	Triggering Opcode		
i_UART_Rx_Serial	Serial input		
OPERATION		**Description**	
0. uart_rx_pop		Get one frame of received data	
1. uart_rx_status		Return RX status	

Table A.3: fu_I2S_LJ_TX_x

fu_I2S_LJ_TX_x			
INPUT PORTS	**Description**	**OUTPUT PORTS**	**Description**
i_clk	System clock	o_data	Serial output
i_glck	Global lock	**o_status[32]**	TX status
i_rstn	Neg. reset		
i_opcode[1:0]	Opcode		
i_opcode_ld	Opcode load		
i_opcode_dummy[32]	Triggering Opcode		
i_mclk	I2S master clock		
i_bck	I2S bit clock		
i_bck_eoc	bit clock end of count		
i_lrck	I2S Left-Right clock		
i_lrck_eoc	LR clock end of count		
i_data_tx_l[32]	Parallel input left		
i_data_tx_ld_l	Par. IN left load		
i_data_tx_r[32]	Parallel input right		
i_data_tx_ld_r	Par. IN right load		
OPERATION		**Description**	
0. i2s_tx_push_l		Push left data frame to transmit	
1. i2s_tx_push_r		Push right data frame to transmit	
2. i2s_tx_status		Return TX status	
3. i2s_tx_zzz		Not assigned	

Table A.4: fu_I2S_LJ_RX_x

fu_I2S_LJ_RX_x			
INPUT PORTS	**Description**	**OUTPUT PORTS**	**Description**
i_clk	System clock	**o_data_rx_l[32]**	Parallel left output
i_glck	Global lock	**o_data_rx_r[32]**	Parallel right output
i_rstn	Neg. reset	**o_status[32]**	RX status
i_opcode[1:0]	Opcode		
i_opcode_ld	Opcode load		
i_opcode_dummy[32]	Triggering Opcode		
i_mclk	I2S master clock		
i_bck	I2S bit clock		
i_bck_eoc	bit clock end of count		
i_lrck	I2S Left-Right clock		
i_lrck_eoc	LR clock end of count		
i_data	Serial input		

OPERATION	**Description**
0. i2s_rx_pop_l	Get one frame of left channel received data
1. i2s_rx_pop_r	Get one frame of right channel received data
2. i2s_rx_status	Return RX status
3. i2s_rx_zzz	Not assigned

Table A.5: fu_SPI_SLAVE_x

fu_SPI_SLAVE_x			
INPUT PORTS	**Description**	**OUTPUT PORTS**	**Description**
i_clk	System clock	**o_data_rx[32]**	Parallel output
i_glck	Global lock	o_miso	SPI MISO
i_rstn	Neg. reset	**o_status[32]**	SPI status
i_opcode[1:0]	Opcode		
i_opcode_ld	Opcode load		
i_opcode_dummy[32]	Triggering Opcode		
i_ss_n	Neg. slave select		
i_sck	SPI Synch. clock		
i_mosi	SPI MOSI		
i_data_tx[32]	Parallel input		
i_data_tx_ld	Par. IN load port		

OPERATION	**Description**
0. spi_pop	Get one frame of received data
1. spi_pop_push	Get frame of received data and push frame to be transmitted
2. spi_push	Push one frame of data to be transmitted
3. spi_status	Return SPI slave status

Table A.6: fu_LED_DRIVER_x

fu_LED_DRIVER_x			
INPUT PORTS	**Description**	**OUTPUT PORTS**	**Description**
i_clk	System clock	o_blue	Blue LED output
i_glock	Global lock	o_green	Green LED output
i_rstn	Neg. reset	o_red	Red LED output
i_color_set[32]	Set color		
i_color_set_ld	Set color load		

OPERATION	**Description**
0. set_led	Turn ON/OFF LEDs

Table A.7: fu_SWITCH_DRIVER_x

fu_SWITCH_DRIVER_x			
INPUT PORTS	**Description**	**OUTPUT PORTS**	**Description**
i_clk	System clock	**o_status[32]**	Switch status
i_glock	Global lock		
i_rstn	Neg. reset		
i_opcode[1:0]	Opcode		
i_opcode_ld	Opcode load		
i_opcode_dummy[32]	Triggering Opcode		
i_switch	Read switch status		
OPERATION		**Description**	
0. switch_status		Return switch status	

Appendix B

Code

B.1 Loop-back firmware

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <status_flags.h>

int main (){
    //UART variables
    int status_UART = AE_RX_UART | E_RX_UART;
    int UART_data_TX = 0;
    int UART_data_RX = 0;
    int UART_data_0  = 0;
    //SPI variables
    int status_SPI = AE_RX_SPI | E_RX_SPI;
    int SPI_data_TX = 170;
    int SPI_data_RX = 0;
    int SPI_data_0 = 0;
    //I2S variables
    int status_I2S = AE_RX_I2S | E_RX_I2S;
    int I2S_data_l_RX = 0;
    int I2S_data_r_RX = 0;
    int I2S_data_0_RX = 0;
    int I2S_data_l_TX = 10066329;
    int I2S_data_r_TX = 10066329;
    int I2S_data_0_TX = 0;
    int status_switch = 0;


    _TCEFU_SET_LED("LED_DRIVER_0", 0);     //Switch OFF all LEDS
    while(1){

        _TCEFU_SWITCH_STATUS("SWITCH_DRIVER_0", 1, status_switch);

        if (status_switch == 1){ //When we press the switch we start
                                 //reading audio from outside while streaming

            //I2S
            _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_1",0, I2S_data_l_RX, status_I2S);
            I2S_data_l_TX = I2S_data_l_RX; //Left channel
            _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0",0,I2S_data_l_TX, status_I2S);

```

```
41              _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_1",1, I2S_data_r_RX, status_I2S);
42              I2S_data_r_TX = I2S_data_r_RX; //Right channel
43              _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0",1,I2S_data_r_TX, status_I2S);
44
45              //UART
46              _TCEFU_UART_RX_POP("UART_RX_0", 0, UART_data_RX, status_UART);
47              UART_data_TX = UART_data_RX;
48              _TCEFU_UART_TX_PUSH("UART_TX_0", 0, UART_data_TX, status_UART);
49
50              //SPI
51              _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0",1,SPI_data_TX,SPI_data_RX,status_SPI)
    ↪ ;
52              SPI_data_TX = SPI_data_RX;
53
54          _TCEFU_SET_LED("LED_DRIVER_0", 1); //Turn ON LED
55
56          }
57          else {  //We always send something when switch not pressed (zeros/silence)
58
59              //I2S
60              _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0",0,I2S_data_0_TX, status_I2S);
61              _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0",1,I2S_data_0_TX, status_I2S);
62
63              //UART
64              _TCEFU_UART_TX_PUSH("UART_TX_0", 0, UART_data_0, status_UART);
65
66              //SPI
67              _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0",1,SPI_data_0,SPI_data_RX,status_SPI);
68
69          _TCEFU_SET_LED("LED_DRIVER_0", 0);       //Turn OFF LED
70          }
71
72      }
73
74      return 0;
75  }
```

B.2 Final firmware (as described in section 4.5)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stdint.h>
4   #include <status_flags.h>
5
6   int hamming_dist(int a, int b);
7   void send_ack(uint8_t command);
8   void send_error(int s_n);
9
10
11  int main (){
12      int status_SPI_0  = AE_RX_SPI | E_RX_SPI;
13      int SPI_data_TX_0 = 0;     //SPI data sent on SPI0
14      int SPI_data_RX_0 = 0;       //SPI data received on SPI0
15
16      int status_I2S_TX_0 = AE_RX_I2S | E_RX_I2S;
17      int status_I2S_RX_0 = AE_RX_I2S | E_RX_I2S;
18      int status_I2S_RX_1 = AE_RX_I2S | E_RX_I2S;
19
20      uint8_t I2S_data_RXa = 0;
21      uint8_t I2S_data_RXb = 0;
22      uint8_t I2S_data_TXa = 0;
23      uint8_t I2S_data_TXb = 0;
24
25      int mode = 0;        //Store the value of the current mode (determined by the command received)
26      uint8_t comm = 0;    //Command received
27
28      //CONFIGURATION VARIABLES
```

```
29      uint8_t  w_s = 16;           //Samples bitwidth
30      uint8_t  n_chf = 4;          //Number of channels FPGA sends in case of RPi-mixing
31      uint8_t  chann_mask = 0x0F; //BitMask to set the input channels in FPGA. Put a 1 on the selected
        ↪ channels
32                                   //es.:  channel_mask = 0000 1010 (only channels 1 and 3);
33                                   //      channel_mask = 0000 0100 (only channel 2);
34      uint8_t s_n = 112;           //Samples in a packet
35      uint8_t n_chpre = 4;         //Number of channels RPi sends in case of FPGA-mixing
36      int config_num = 5;          //Number of config variables
37      uint8_t SPI_frames_per_sample = w_s/8;
38
39      //SEQUENCES OF BITS
40      uint8_t preamble    = 0b10101010;
41      uint8_t error       = 0b11110000;
42      uint8_t command1    = 0b00110011;
43      uint8_t command2    = 0b00011000;
44
45      //Counters for majority check
46      uint8_t general_cnt = 0;     //
47      uint8_t preamble_cnt = 0;
48      uint8_t command_cnt = 0;
49      uint8_t preamble_check = 0;
50      uint8_t command_check = 0;
51
52      _TCEFU_SET_LED("LED_DRIVER_0", 0);     //Switch OFF all LEDS
53
54      while(1){    //Entering the main loop
55
56      //WAIT FOR A COMMAND
57          _TCEFU_SPI_POP("SPI_SLAVE_0", 0, comm, status_SPI_0);
58          if (hamming_dist(comm, preamble) <= 2) //Check for preamble
59          {
60              general_cnt++;
61              if (comm == preamble)
62              {
63                  preamble_cnt++;
64              }
65          }
66          else
67          {
68              general_cnt = 0;
69              preamble_cnt = 0;
70          }
71
72          if (general_cnt == 5){
73              if (preamble_cnt >= 3) //If preamble detected
74              {   //Wait for the command
75                  preamble_check = 1;
76                  general_cnt = 0;
77                  preamble_cnt = 0;
78                  while (preamble_check == 1){
79                      _TCEFU_SPI_POP("SPI_SLAVE_0", 0, comm, status_SPI_0);
80                      if ((hamming_dist(comm, command1) <= 2) || (hamming_dist(comm, command2) <= 2))
81                      {
82                          general_cnt++;
83                          if (comm == command1)
84                          {
85                              command_cnt++;
86                              mode = 1;
87                          }
88                          if (comm == command2)
89                          {
90                              command_cnt++;
91                              mode = 2;
92                          }
93                      }
94                      else {
95                          general_cnt = 0;
96                          preamble_cnt = 0;
97                          preamble_check = 0;
98                          send_error(s_n);
99                      }
100
101                     if (general_cnt == 3)
102                     {
103                         if (command_cnt >=2)
104                         {
105                             command_check = mode;
106                         }
107                         else
108                         {
109                             general_cnt = 0;
110                             preamble_cnt = 0;
111                             preamble_check = 0;
112                             send_error(s_n);
113                         }
114                     }
115                 }
116             }
```

```
117             else{
118                 preamble_cnt = 0;
119                 general_cnt = 0;
120                 //send_error(s_n);
121             }
122         }
123 
124     //Activate only in case of proxim testing
125     //command_check = 2;
126 
127     switch(command_check)
128         {
129             case 1:
130                 send_ack(command_check);
131                 _TCEFU_SPI_POP("SPI_SLAVE_0", 0, w_s, status_SPI_0);
132                 _TCEFU_SPI_POP("SPI_SLAVE_0", 0, n_chf, status_SPI_0);
133                 _TCEFU_SPI_POP("SPI_SLAVE_0", 0, chann_mask, status_SPI_0);
134                 _TCEFU_SPI_POP("SPI_SLAVE_0", 0, s_n, status_SPI_0);
135                 _TCEFU_SPI_POP("SPI_SLAVE_0", 0, n_chpre, status_SPI_0);
136                 command_check = 0;
137                 break;
138 
139             case 2: //Streaming with RPi-mix
140                 _TCEFU_SET_LED("LED_DRIVER_0", 1);  //LED switches ON to tell streaming is working
141                 send_ack(command_check);
142 
143                 for (int i = 0; i<s_n; i++)
144                 {
145                     uint8_t SPI_byte[4] = {0, 0, 0, 0};
146                     if (n_chf == 1)
147                     {
148                         if (chann_mask == 0b00000001)    //CH.0
149                         {
150                             _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_0", 0, I2S_data_RXa, status_I2S_RX_0)
    ↪ ; //Record ch.0
151                             for (int i = 0; i < SPI_frames_per_sample; i++){
152                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.0 and send ch.0
153                             }
154                             I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8) + (
    ↪ SPI_byte[0]);
155 
156                             _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa, status_I2S_TX_0
    ↪ ); //Play ch.0
157                             for (int i = 0; i < SPI_frames_per_sample; i++){
158                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.0
159                             }
160                             I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8) + (
    ↪ SPI_byte[0]);
161 
162                             _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb, status_I2S_TX_0
    ↪ ); //Play ch.1
163                         }
164                         else if (chann_mask == 0b00000010)  //CH.1
165                         {
166                             _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_0", 0, I2S_data_RXa, status_I2S_RX_0)
    ↪ ; //Record ch.1
167                             for (int i = 0; i < SPI_frames_per_sample; i++){
168                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i] , status_SPI_0); //Receive ch.0 and send ch.1
169                             }
170                             I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8) + (
    ↪ SPI_byte[0]);
171                             _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa, status_I2S_TX_0
    ↪ ); //Play ch.0
172                             for (int i = 0; i < SPI_frames_per_sample; i++){
173                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.1
174                             }
175                             I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8) + (
    ↪ SPI_byte[0]);
176                             _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb, status_I2S_TX_0
    ↪ ); //Play ch.1
177                         }
178                         else if (chann_mask == 0b00000100)  //CH.2
179                         {
180                             _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_1", 0, I2S_data_RXa, status_I2S_RX_0)
    ↪ ; //Record ch.2
181                             for (int i = 0; i < SPI_frames_per_sample; i++){
182                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.0 and send ch.2
183                             }
184                             I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8) + (
    ↪ SPI_byte[0]);
185                             _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa, status_I2S_TX_0
    ↪ ); //Play ch.0
186                             for (int i = 0; i < SPI_frames_per_sample; i++){
```

```
187                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.2
188                             }
189                             I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8) + (
    ↪ SPI_byte[0]);
190                             _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb, status_I2S_TX_0
    ↪ ); //Play ch.1
191                         }
192                         else if (chann_mask == 0b00001000)  //CH.3
193                         {
194                             _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_1", 0, I2S_data_RXa, status_I2S_RX_0)
    ↪ ; //Record ch.3
195                             for (int i = 0; i < SPI_frames_per_sample; i++){
196                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.0 and send ch.3
197                             }
198                             I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8) + (
    ↪ SPI_byte[0]);
199                             _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa, status_I2S_TX_0
    ↪ ); //Play ch.0
200                             for (int i = 0; i < SPI_frames_per_sample; i++){
201                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.3
202                             }
203                             I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8) + (
    ↪ SPI_byte[0]);
204                             _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb, status_I2S_TX_0
    ↪ ); //Play ch.1
205                         }
206 
207                     }
208                     else if (n_chf == 2)
209                     {
210                         if (chann_mask == 0b00000011)   //Channels 0 and 1
211                         {
212                             _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_0", 0, I2S_data_RXa,
    ↪ status_I2S_RX_0); //Record ch.0
213                             for (int i = 0; i < SPI_frames_per_sample; i++){
214                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i] , status_SPI_0); //Receive ch.0 and send ch.0
215                             }
216                             I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
217                             _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa,
    ↪ status_I2S_TX_0); //Play ch.0
218                             _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_0", 1, I2S_data_RXb,
    ↪ status_I2S_RX_0);    //Record ch.1
219                             for (int i = 0; i < SPI_frames_per_sample; i++){
220                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXb >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.1
221                             }
222                             I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
223                             _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb,
    ↪ status_I2S_TX_0); //Play ch.1
224                         }
225                         else if (chann_mask == 0b00001100)  //Channels 2 and 3
226                         {
227                             _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_1", 0, I2S_data_RXa,
    ↪ status_I2S_RX_0); //Record ch.2
228                             for (int i = 0; i < SPI_frames_per_sample; i++){
229                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.0 and send ch.2
230                             }
231                             I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
232                             _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa,
    ↪ status_I2S_TX_0); //Play ch.0
233                             _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_1", 1, I2S_data_RXb,
    ↪ status_I2S_RX_0);    //Record ch.3
234                             for (int i = 0; i < SPI_frames_per_sample; i++){
235                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXb >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.3
236                             }
237                             I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
238                             _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb,
    ↪ status_I2S_TX_0); //Play ch.1
239                         }
240                         else if (chann_mask == 0b00001010)  //Channels 1 and 3
241                         {
242                             _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_0", 0, I2S_data_RXa,
    ↪ status_I2S_RX_0); //Record ch.1
243                             for (int i = 0; i < SPI_frames_per_sample; i++){
244                                 _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.0 and send ch.1
245                             }
246                             I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
```

```
247                                 _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa,
    ↪ status_I2S_TX_0); //Play ch.0
248                                 _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_1", 1, I2S_data_RXb,
    ↪ status_I2S_RX_0);     //Record ch.3
249                                 for (int i = 0; i < SPI_frames_per_sample; i++){
250                                     _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXb >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.3
251                                 }
252                                 I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
253                                 _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb,
    ↪ status_I2S_TX_0); //Play ch.1
254                             }
255                             else if (chann_mask == 0b00000101)  //Channels 0 and 2
256                             {
257                                 _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_0", 0, I2S_data_RXa,
    ↪ status_I2S_RX_0); //Record ch.0
258                                 for (int i = 0; i < SPI_frames_per_sample; i++){
259                                     _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.0 and send ch.0
260                                 }
261                                 I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
262                                 _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa,
    ↪ status_I2S_TX_0); //Play ch.0
263                                 _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_1", 1, I2S_data_RXb,
    ↪ status_I2S_RX_0);     //Record ch.2
264                                 for (int i = 0; i < SPI_frames_per_sample; i++){
265                                     _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXb >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.2
266                                 }
267                                 I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
268                                 _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb,
    ↪ status_I2S_TX_0); //Play ch.1
269                             }
270                             else if (chann_mask == 0b00001001)  //Channels 0 and 3
271                             {
272                                 _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_0", 0, I2S_data_RXa,
    ↪ status_I2S_RX_0); //Record ch.0
273                                 for (int i = 0; i < SPI_frames_per_sample; i++){
274                                     _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.0 and send ch.0
275                                 }
276                                 I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
277                                 _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa,
    ↪ status_I2S_TX_0); //Play ch.0
278                                 _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_1", 1, I2S_data_RXb,
    ↪ status_I2S_RX_0);     //Record ch.1
279                                 for (int i = 0; i < SPI_frames_per_sample; i++){
280                                     _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXb >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.1
281                                 }
282                                 I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
283                                 _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb,
    ↪ status_I2S_TX_0); //Play ch.1
284                             }
285                             else if (chann_mask == 0b00000110)  //Channels 1 and 2
286                             {
287                                 _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_0", 0, I2S_data_RXa,
    ↪ status_I2S_RX_0); //Record ch.0
288                                 for (int i = 0; i < SPI_frames_per_sample; i++){
289                                     _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.0 and send ch.0
290                                 }
291                                 I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
292                                 _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa,
    ↪ status_I2S_TX_0); //Play ch.0
293                                 _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_1", 1, I2S_data_RXb,
    ↪ status_I2S_RX_0);     //Record ch.1
294                                 for (int i = 0; i < SPI_frames_per_sample; i++){
295                                     _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXb >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.1
296                                 }
297                                 I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
298                                 _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb,
    ↪ status_I2S_TX_0); //Play ch.1
299                             }
300                     }
301                     else if (n_chf == 4)
302                     {
303                                 _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_0", 0, I2S_data_RXa,
    ↪ status_I2S_RX_0); //Record ch.0
304                                 for (int i = 0; i < SPI_frames_per_sample; i++){
```

```
305                                     _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.0 and send ch.0
306                                 }
307                                 I2S_data_TXa = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
308                                 _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, I2S_data_TXa,
    ↪ status_I2S_TX_0); //Play ch.0
309                                 _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_0", 1, I2S_data_RXb,
    ↪ status_I2S_RX_0);     //Record ch.1
310                                 for (int i = 0; i < SPI_frames_per_sample; i++){
311                                     _TCEFU_SPI_POP_PUSH("SPI_SLAVE_0", 1, I2S_data_RXb >> (8*(
    ↪ SPI_frames_per_sample-1-i)), SPI_byte[i], status_SPI_0); //Receive ch.1 and send ch.1
312                                 }
313                                 I2S_data_TXb = (SPI_byte[3]<<24) + (SPI_byte[2]<<16) + (SPI_byte[1]<<8)
    ↪  + (SPI_byte[0]);
314                                 _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, I2S_data_TXb,
    ↪ status_I2S_TX_0); //Play ch.1
315                                 _TCEFU_I2S_RX_POP_L("I2S_LJ_master_RX_1", 0, I2S_data_RXa,
    ↪ status_I2S_RX_0); //Record ch.2
316                                 for (int i = 0; i < SPI_frames_per_sample; i++){
317                                     _TCEFU_SPI_PUSH("SPI_SLAVE_0", 2, I2S_data_RXa >> (8*(
    ↪ SPI_frames_per_sample-1-i)), status_SPI_0);               //Send ch.2
318                                 }
319                                 _TCEFU_I2S_RX_POP_R("I2S_LJ_master_RX_1", 1, I2S_data_RXb,
    ↪ status_I2S_RX_0);     //Record ch.3
320                                 for (int i = 0; i < SPI_frames_per_sample; i++){
321                                     _TCEFU_SPI_PUSH("SPI_SLAVE_0", 2, I2S_data_RXb >> (8*(
    ↪ SPI_frames_per_sample-1-i)), status_SPI_0);               //Send ch.3
322                                 }
323                         }
324                     }
325 
326                 _TCEFU_SET_LED("LED_DRIVER_0", 0);  //LED switches OFF to tell streaming is closed
327                 command_check = 0;
328                 break;
329 
330                 case 3: //Streaming with FPGA-mix
331                     break;
332                 default:
333                     break;
334             }
335 
336         }
337         return 0;
338 }
339 
340 //FUNCTIONS
341 
342 int hamming_dist(int a, int b)  //Simple functions that calculates Hamming distance between two numbers
343                                 //(Credits to https://www.geeksforgeeks.org/hamming-distance-between-
    ↪ two-integers/)
344 {
345     int c = a ^ b;
346     int cnt = 0;
347 
348     while (c > 0) {
349         cnt += c & 1;
350         c >>= 1;
351     }
352     return cnt;
353 }
354 
355 void send_ack(uint8_t command)
356 {
357     uint8_t preamble    = 0b10101010;
358     int status_SPI_0;
359     for (int i = 0; i < 5; i++)     //Send 5 Bytes of preamble
360     {
361         _TCEFU_SPI_PUSH("SPI_SLAVE_0", 2, preamble, status_SPI_0);
362     }
363     for (int i = 0; i < 3; i++)     //Send 3 Bytes of error sequence
364     {
365         _TCEFU_SPI_PUSH("SPI_SLAVE_0", 2, command, status_SPI_0);
366     }
367 }
368 
369 void send_error(int s_n)        //This function sends an error to RPi and contemporarly
370 {                           //reproduces a packet of zeros to maintain synchronicity
371     uint8_t preamble    = 0b10101010;
372     uint8_t error       = 0b11110000;
373     int status_SPI_0;
374     int status_I2S_TX_0;
375     for (int i = 0; i < 5; i++)     //Send 5 Bytes of preamble
376     {
377         _TCEFU_SPI_PUSH("SPI_SLAVE_0", 2, preamble, status_SPI_0);
378     }
379     for (int i = 0; i < 3; i++)     //Send 3 Bytes of error sequence
380     {
381         _TCEFU_SPI_PUSH("SPI_SLAVE_0", 2, error, status_SPI_0);
```

```
382     }
383     _TCEFU_SPI_PUSH("SPI_SLAVE_0", 2, preamble, status_SPI_0);
384
385     int SPI_frames = 2*s_n;
386     for (int i = 0; i < (SPI_frames); i++){ //PLAY PACKET OF ZEROS
387         _TCEFU_I2S_TX_PUSH_L("I2S_LJ_master_TX_0", 0, 0, status_I2S_TX_0);
388         _TCEFU_I2S_TX_PUSH_R("I2S_LJ_master_TX_0", 1, 0, status_I2S_TX_0);
389     }
390
391 }
```

bibliography

[1] C. Rottondi, C. Chafe, C. Allocchio, and A. Sarti. «An Overview on Networked Music Performance Technologies». In: *IEEE Access* 4 (Dec. 2016), pp. 13–16 (cit. on pp. 1, 5).

[2] Riccardo Peloso. *Custom Hardware audio board*. 2021 (cit. on pp. 2, 11).

[3] A. B. Renaud, A. Carôt, and P. Rebelo. «Networked Music Performance : State Of The Art». In: *AES 30th International Conference (Saariselkä, Finland, 2007 March 15–17)* (Mar. 2007) (cit. on p. 4).

[4] L. Gabrielli S. Squartini. *Wireless Networked Music Performance*. Springer briefs in electrical and computer engineering. Springer, Dec. 2015. Chap. 2. ISBN: 978-981-10-0335-6 (cit. on p. 4).

[5] J. Bischoff and Brown C. «Crossfade». In: *Retrieved 2009-11-26* (). URL: `http://crossfade.walkerart.org/brownbischoff/` (cit. on p. 4).

[6] *Networked music performance*. URL: `https://en.wikipedia.org/wiki/Networked_music_performance`. (accessed: 2022/08/29) (cit. on p. 4).

[7] A. Carôt, A. Renaud, and B. Verbrugghe. «Network Music Performance (NMP) with Soundjack». In: *Networked Performance Workshop Paper. Presented at the 6th NIME Conference* (June 2006) (cit. on p. 5).

[8] C. Alexandraki, P. Koutlemanis, P. Gasteratos, N. Valsamakis, D. Akoumianakis, G. Milolidakis, G. Vellis, and D. Kotsalis. «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. 5).

[9] Pekka Jääskeläinen, Aleksi Tervo, Guillermo Paya-Vaya, Timo Viitanen, Nicolai Behmann, Jarmo Takala, and Holger Blume. «Transport-Triggered Soft Cores». In: *IEEE International Parallel and Distributed Processing Symposium, Workshops (IPDPSW)* (2018) (cit. on p. 5).

[10] Pekka Jääskeläinen, Timo Viitanen, Jarmo Takal, and Berg Heikki. «HW/SW Co-design Toolset for Customization of Exposed Datapath Processors». In: *Computing Platforms for Software-Defined Radio (book chapter pp. 147-164)* (2017) (cit. on p. 5).

[11] Pekka Jääskeläinen, Teemu Kultala Heikki Pitkänen, and jarmo Takala. «Reducing the Overheads of Hardware Acceleration Through Datapath Integration». In: *Electronic Imaging 2008 (San Jose, USA, January 2008)* () (cit. on p. 5).

[12] *About Transport-Triggered Architectures*. URL: http://openasip.org/tta.html. (accessed: 2022/08/29) (cit. on p. 6).

[13] Tomi Äijö, Pekka Jääskeläinen, Tapio Elomaa, Heikki Kultala, and Jarmo Takala. «Integer Linear Programming-Based Scheduling for Transport Triggered Architectures». In: *ACM Transactions on Architecture and Code Optimization* 12 (Dec. 2015), pp. 1–22. DOI: 10.1145/2845082 (cit. on p. 7).

[14] *Introduction to FPGA and It's Programming Tools*. URL: https://circuitdigest.com/tutorial/what-is-fpga-introduction-and-programming-tools. (accessed: 2022/08/29) (cit. on p. 7).

[15] *MIDI*. URL: https://en.wikipedia.org/wiki/MIDI. (accessed: 2022/08/29) (cit. on p. 8).

[16] Philips Semiconductors. *I^2S bus specification (PDF)*. June 1996. URL: https://web.archive.org/web/20070102004400/http://www.nxp.com/acrobat_download/various/I2SBUS.pdf. Archived from the original (PDF) on January 2, 2007 (accessed: 2022/08/29) (cit. on p. 8).

[17] *Introduction to SPI Interface*. URL: https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html. (accessed: 2022/08/29) (cit. on p. 10).

[18] *TCE*. URL: http://openasip.org/. (accessed: 2022/08/29) (cit. on pp. 16, 17).

[19] 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*. Ed. by Waqar Hussain, Jari Nurmi, Jouni Isoaho, and Fabio Garzia. Springer International Publishing, 2017, pp. 147–164. ISBN: 978-3-319-49679-5. DOI: 10.1007/978-3-319-49679-5_8. URL: https://doi.org/10.1007/978-3-319-49679-5_8 (cit. on p. 16).

[20] *Custom HDL function units written by and with the help of Riccardo Peloso* (cit. on p. 19).

[21] *Firmware routine developed with the help of Matteo Sacchetto and Leonardo Severi* (cit. on p. 24).