An Open Audio Processing Platform with Zync FPGA

1st Kevin Vaca
Engineering Department
University of Houston Clear Lake
Houston, USA

2nd Mitchell M. Jefferies
Engineering Department
University of Houston Clear Lake
Houston, USA

3th Xiaokun Yang*
Engineering Department
University of Houston Clear Lake
Houston, USA
yangxia@uhcl.edu

***Abstract*—This paper presents an open audio processing platform on Zync7000 Field-Programmable Gate Array (FPGA), capable of collecting analog frequency signals through a microphone, and pushing out a data set of frequencies and amplitudes to a UART interface. The validity of this platform has been proved by a design on a real-time automatic music transcription (AMT) system, mainly containing three algorithms run on the Zync processor including Discrete Fourier Transform (DFT), vector building with Pitch Class Profile (PCP), and pattern matching. Experimental results show that the accuracy of the chord detection can reach 90% while playing open chords, and the latency of analog sampling is around 250 ms.**

We make this platform publicly available to encourage future research in audio processing such as speech recognition and natural language processing, and to serve educational studies on digital system and embedded system implementations with hardware-software co-design FPGAs.

***Index Terms*—audio processing, digital system design, field-programmable gate array (FPGA), hardware-software co-design**

I. Introduction

Today, audio processing systems, such as the automatic music transcription (AMT), automatic speech recognition (ASR), and the wider field of natural language processing (NLP), show a clear potential for both economic and societal impact [1]. As a fascinating example of their applications, the devices of Amazon Alexa and Google Home have been widely used as control centers of smart home/building platforms, including audio signal processing and artificial intelligence (AI) in enabling technologies with Amazon Cloud and Google Cloud. Furthermore, due to the benefits of processing data on the AI-enabled integrated circuit (IC), for example the low-latency response and parallel computing, the implementation on the audio processing system-on-chips (SoC) is becoming one of the most inexpensive and energy-efficiency solutions for such real-time systems [14], [15].

Under this context, this paper presents an open platform on audio processing with Zync FPGA, providing a way to test the functionality with FPGA before taping out the application-specific IC (ASIC). More important, our proposed system can be used during the earlier stage of the project due to the integration of the software programmability of an ARM processor with the hardware programmability of the FPGA.

As shown in Fig. 1, more specifically, the traditional FPGA design starts from the register-transfer-level (RTL) programming and ends up with the final synthesis netlist. The functionality is not able to be demonstrated until the completion of the FPGA design flow. However, our proposed work allows the hardware-software co-design on Zync FPGA: the algorithms can be directly run on the ARM processor, or the programming system (PS); the designs with hardware programming language (HDL) and intellectual properties (IPs) provided by the FPGA can be programmed on the programming logic (PL) [11], [12]. In such a way the design with FPGA can be tested at an earlier stage to shorten the time-to-market and reduce the cost of development and tape-out risk.

Overall, the main contributions of this work are:

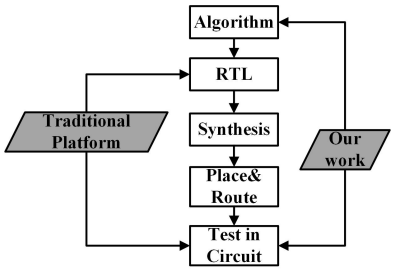

Fig. 1. FPGA Design Flow

- We presented an open audio processing platform containing the block-based design and the synthesis results in terms of slice count and power consumption. We expect that the public release of this platform is able to lead to better implementations in the future, as well as to generate many application-specific designs in teaching and doing research in audio processing, speech recognition, natural language processing, etc.
- We tested the validity of using this platform with an application of automating music transcription, which is able to extract the frequencies from a live recording and then analyze the given frequencies to find specific chords played by the user. Experimental results show that the design on hardware and software can be seamlessly integrated by using the application programming interface (API) and drivers provided by the Zync FPGA.

The organization of this paper is as follows. Section II briefly introduces the related works and III presents our work

with design architecture. Section IV discusses the implementation of the proposed system. In Section V, the experimental results in terms of accuracy and latency of the system, as well as the slice cost and power consumption on FPGA are shown. Finally, Section VI presents the concluding remarks in our target architecture.

II. Related works

Prior research in audio processing platforms mainly based on embedded systems or DSPs, such as the implementation of personal sound amplification product (PSAP) using binaural microphones/earphones and a Raspberry Pi [4], an advanced sound system operated by DSP boards [3], an object-based audio representation [2], a platform for embedded speech analysis and synthesis [5], location-aware speakers [6], and predictive real-time beat tracking [7].

To make data processing in real time, particularly for time-sensitive applications like the music transposer and speech recognition, accelerators like FPGA have been widely used in audio processing prototypes. For example, in [9] an FPGA acceleration for tracking audio effects in movies has been presented, and [10] proposed a custom multi-core hardware accelerators for both algorithms and map them onto Virtex6 FPGAs. Results showed that FPGA implementations can provide more power-effective solutions, due to the parallel computing on hardware design and capability to drive more complex microphone and loudspeaker setups than PC-based approaches [10]. Additionally an implementation of audio signal processing and display system has been proposed in [8]. Since the design specification is not publicly available, the hardware cost and system performance are not able to be compared.

To tackle the aforementioned issues, a scalable audio processing system is presented with the hardware-software co-design platform on Zync7000 FPGA. The availability of this open platform is able to minimize setup-time when testing and benchmarking new audio processing projects. The application of this platform has been tested on an automatic music transposer, enabling the automatic generation of detected chords by means of real-time recordings.

III. Proposed open platform

Fig. 2 shows the hardware architecture of our proposed work. It can be divided into two parts. The first part is the programming logic (PL), including the HDL design and IPs offered by the specific FPGA. In our case, an analog audio codec is instantiated to provide integrated digital audio processing to the Zynq7000 SoC. It allows for stereo record and playback at sample rates from 8 kHz to 96 kHz. The audio data is transferred via the I^2S protocol as shown in Fig. 3.

To use the audio codec in a design with non-default settings, it needs to be configured over an I^2C bus. The device address is binary 0011010 or hexdecimal $1A$. The audio path needs to be established by configuring the (de)multiplexers and amplifiers in the codec. Some digital processing can also be done in the codec. Configuration is read out and written by

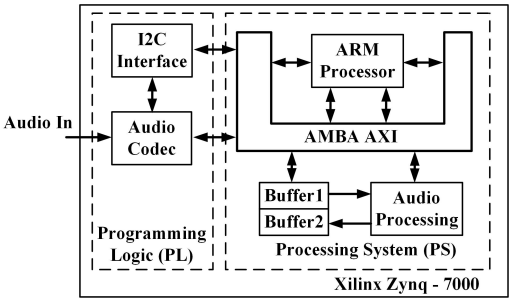

Fig. 2. Hardware Architecture

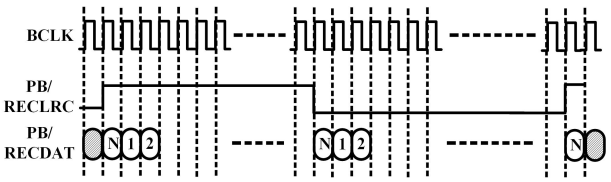

Fig. 3. I^2S Timing Diagram

accessing the register map via I^2C transfers shown in Fig. 4. The register map is described in the SSM2603 datasheet [13].

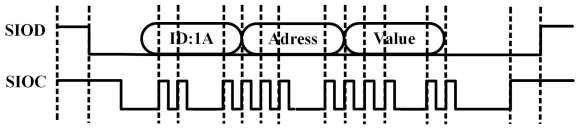

Fig. 4. I^2C Timing Diagram

The second part is the processing system (PS), including the algorithms of audio processing run on the ARM processor. The hardware interface between the processing system and processing logic is based on the AXI3 specification [18].

During the ASIC/FPGA design flow, the design and verification in the behavioral model or register-transfer level is very time consuming. Using the programming system offered by Xilinx Zync, it is able to test the functionality in the algorithm level, which is an earlier stage compared to the register-transfer level, as shown in Fig. 1. In other words, all algorithms are able to be written as C-code software to be run by the Cortex A9 processor present on the Zybo Z7. Together with the IP blocks and resource provided by the Zync SoC, the design on the system can be tested before performing the RTL coding.

IV. Implementation

In this section, the open platform introduced in Section III is implemented. After that, a demonstration of guitar music transcription is used to test the validity of our proposed work.

A. Block Based Design with Proposed Work

Fig. 5 shows the block based design on Zync FPGA. The development of the first stage using the Zybo Z7 board has been through the use of two main software: Xilinx Vivado and Xilinx Software Development Kit (SDK).

On the hardware itself, the parts used includes the MIC IN interface for use with a 3.5mm coaxial cable connected to a microphone, as well as the microUSB port to communicate with the external devices. The Zybo Z7 has an Analog Devices SSM2603 Audio Codec which provides the audio processing required by our project. The SSM2603 is programmed on-board through the I^2C protocol.

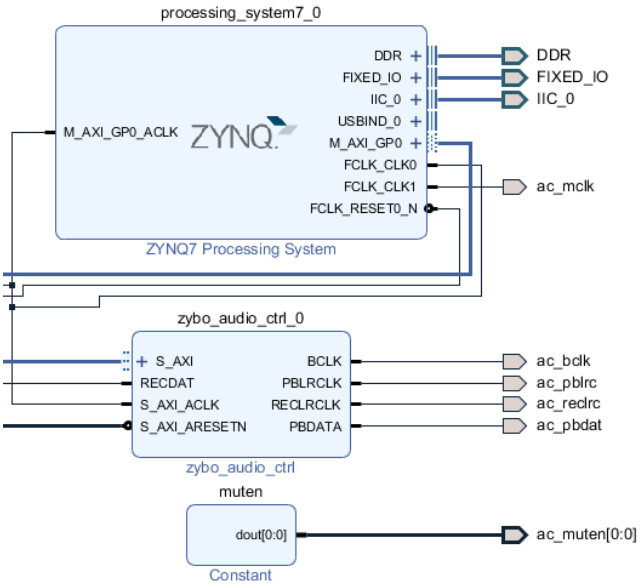

Fig. 5. The Block Based Design.

a) Block Based Design: Using Xilinx Vivado, we create a hardware level design which includes an IP for the audio device that was present in one of the tutorial designs available from the Zynq book website [25]. The Zybo Audio Control IP is used to communicate with and program the on-board SSM2603 audio codec. The hardware block diagram is implemented and the bitstream is generated which is exported to the SDK. The implementation of the design includes the aforementioned IP, an IP to program the Zynq processor, and C code that handles all data manipulation and processing algorithms.

Xilinx SDK is where the software control for the design is created. The SDK allows us to create projects using both C and C++. Each IP from Xilinx Vivado has specific source and header files that must be imported into the project in order to call the functions associated with the IP. For example, the IP used to program the SSM2603 has function calls which follow the I^2C protocol to properly write data to the registers on the audio codec.

The imported bitstream from Xilinx Vivado includes the memory address map for the design and defines all of the device's memory addresses in a parameters header file. It is the SDK that allows you to program the FPGA with the generated bitstream, and then launch the C or C++ level code on the FPGA.

b) Audio Codec Programming and Sampling: Upon the exportation of the bitstream from Xilinx Vivado, the first part in developing the C code includes programming the audio codec and sampling the microphone. By referencing the SSM2603 data sheet [13], one can program the SSM2603 registers, using I^2C code, in order to adjust microphone/line-in enables, input and output volume control, sample data bit size, sampling speed, and much more.

Our project changes a few things from the default register configuration: we turn power on for the microphone input ($R6$), we adjust sidetone attenuation to -15dB, select DAC, disable bypass, set microphone as the in select, and disable the mute on the microphone data path ($R4$), we disable the DAC mute ($R5$), set the data to 32 bit unsigned integers ($R7$), and set the sampling rate to $8,000Hz$ ($R8$). All these changes are necessary to ensure that the audio codec properly samples data for use in our program.

After the audio codec is properly configured in the code, the second process is to sample the data. In order to achieve a bin width of $< 4Hz$, with a sampling rate of $8,000Hz$, our sample size, denoted as N, would need to be 2048. Instantiating an array of 2048, unsigned long (32 bit) integers allow us to declare memory space for the sample sets. Our design reads from the audio codec register that stores data sampled from the microphone and saves that data into the array. To ensure that each data entry is unique, a short if-statement is introduced to compare the current index with the previous index.

B. Demonstration

With the utilization of the open platform, a music transcription system is demonstrated to test the validity of our open platform. A typical data representation used in a music transcription system is introduced in Fig. 6. It basically takes an audio waveform as input, computes a time-frequency representation, and outputs a representation of pitches over time.

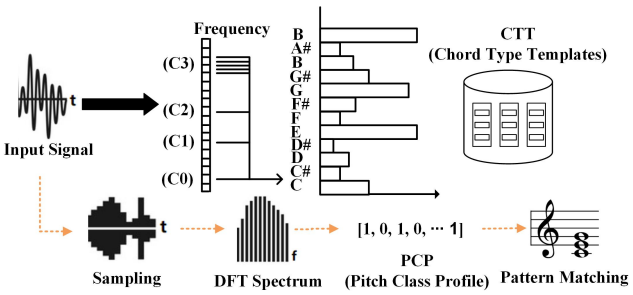

Fig. 6. Datapath for an AMT system with use in chords

More specifically, the system enables the extraction of the frequencies from a live recording and then analyzes the given frequencies to find specific chords played by the user. The method by which we analyze samples is by Discrete Fourier Transform (DFT).

The resultant frequency spectrum must then be processed by the Pitch Class Profile (PCP) algorithm to extract the pitch

class of the samples [19]. The pitch class is then referenced to existing templates in order to store the found chord to the database. The database will keep records of the chords input, as well as the time the chord received. With both chord data and time data, it is possible to write out sheet music corresponding to the user's playing.

a) Discrete Fourier Transform: All audio exists as sinusoidal waves in the frequency spectrum and the human ear can detect from as low as 20 Hz and as high as 20,000 Hz [21]. Considering a guitar music transcription, the lowest frequency on a standard tuned guitar is the second octave E at 82.407 Hz, and the highest note ranges around the fifth octave B at 987.767 Hz [24], as shown in Fig. 7. By recording audio and analyzing the sinusoidal waves for its frequency spectrum, one can determine what frequencies are present in a given audio signal.

		1F	2F	3F	4F	5F	6F	7F	8F	9F	10F	11F	12F	13F	14F	15F	16F	17F	18F	19F	20F
1st	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C
	329	349	370	392	415	440	466	494	523	554	587	622	659	698	740	784	831	880	932	988	1047
2nd	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G
	247	262	277	294	311	329	349	370	392	415	440	466	494	523	554	587	622	659	698	740	784
3rd	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D	D#
	196	208	220	233	247	262	277	294	311	329	349	370	392	415	440	466	494	523	554	587	622
4th	D	D	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#
	147	156	165	175	185	196	208	220	233	247	262	277	294	311	329	349	370	392	415	440	466
5th	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F
	110	117	123	131	139	147	156	165	175	185	196	208	220	233	247	262	277	294	311	329	349
6th	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C
	82	87	92	98	104	110	117	123	131	139	147	156	165	175	185	196	208	220	233	247	262

Fig. 7. Guitar Frets & Their Notes vs Frequencies

The basic equation used to convert the sampled analog input into the frequency domain is

$$X_k = \sum_{n=0}^{N-1} x_n \times (\cos(\frac{2\pi kn}{N}) – i \times sin(\frac{2\pi kn}{N})) \qquad (1)$$

where N denotes the number of samples (2048), k represents the index value for the frequency domain (20 to 256), and n indicates the index value for the time domain (0 to 2047). X_k denotes the magnitude of the frequency bin at k, and x_n represents the time-domain sample data at n.

Because of the $3.9Hz$ bin width, and the frequency range of a standard tuned guitar ($80Hz$–$1000Hz$), in this case the bins that actually needed calculation numbered 236. Our for-statement for calculating the DFT begins at index 20 ($78Hz$) and ends at index 256 ($998.4Hz$).

b) Pitch Class Profile Algorithm: Pitch Class Profiles quantify the intensity of each pitch class with twelve numbers and can be used for chord recognition. In order to calculate a PCP, each bin of the DFT must be mapped to a pitch class. This mapping is done by the equation for $M(l)$, which is shown as

$$M(l) = \begin{cases} -1, & \text{for } l = 0 \\ round\left[12\log_2(\frac{F_s \times l}{N}/F_{ref})\right] mod 12 & \text{for } l = 1, 2, \\ & ....., N/2-1 \end{cases} \qquad (2)$$

where F_s is the sampling frequency, l is the bin index, N is the DFT length, and F_{ref} is the reference frequency.

$M(l)$ uses the cent value of a DFT bin and modulus math to return an integer valued zero through eleven. This integer corresponds to the pitch class of a DFT bin; zero represents pitch class C, one represents pitch class $C\#$ and so on. The term $\frac{F_s \times l}{N}$ is the frequency in Hz that a bin represents. $M(l)$ calculates the cent value of a particular frequency bin, with respect to a constant reference frequency, 65.4 Hz in our case. Each pitch class is 100 cent apart so dividing by modulus twelve returns an integer valued 0-11.

The PCP effectively sums the square of the intensity of all bins that fall under a particular pitch class as is shown in equation 3.

$$PCP(p) = \sum_{l.s.t.M(l)=p} \|X(l)\|^2 \qquad (3)$$

Thus, the PCP results in twelve values representing the intensity of each pitch class. Once a PCP is calculated for an input signal, matching algorithms must be implemented to find the most likely chord played.

In order to do this, Chord Type Templates (CTTs) which are binary vectors with twelve values are used. These twelve values corresponding to the twelve pitch classes much like the PCP. Different methods like Weighted Sum and Nearest Neighbor can be performed to compute scores for each CTT. These scores are maximized or minimized respectively to return the most likely chord played.

c) Exponential Smoothing: Exponential smoothing, shown in equation 4 is implemented to reduce noise and provide a more consistent output. The exponential smoothing value is α, the smoothed output is the discrete function $y(n)$, and the un-smoothed input is the discrete function $x(n)$.

$$y(n) = \alpha x(n) + (1-\alpha)y(n-1) \qquad (4)$$

The instant after a chord is struck, the frequency spectrum of the signal looks rather unfamiliar before becoming the recognizable chord. Exponential smoothing minimizes this effect by weighing the current PCP with previous PCPs. To implement this, we use a variable to store the previous PCP and loop through each PCP value and set it equal to a new smoothed value. After a little bit of tuning, our final system uses an exponential smoothing factor of 0.5.

d) Data Structures: The CTTs are stored in a two-dimensional array. Our system detects all major and minor chords; thus our array size is 24×12 corresponding to the number of chords in our data structure and twelve PCP values for each chord respectively.

The CCT data structure starts with the $Cmaj$ chord and increments by semi-tone until reaching the last element of $Bmin$. Order is important as a parallel data structure is needed to map each CTT to a string that represents the chord name. An array of structs is used to hold the string values. Both data structures follow the same order in that the index 0 corresponds to a $Cmaj$, index 1 to $C\#maj$, and so on until $Bmin$.

e) Weighted Sum: Our algorithm implements Weighted Sum method to match each calculated PCP to the most similar CTT, shown in equation 5.

$$Score_c = \sum_{p=0}^{11} W_c(p) \times PCP(p) \tag{5}$$

Essentially, each PCP value is multiplied by the corresponding CTT value and then all values summed. This score is calculated for every CTT and then maximized to return the most likely chord played.

Nested for-loops are used to access every chord and then all twelve values of the given CTT. As each score is calculated, our program updates the maximum scores as well as the index of the CTT. When the nested for loops are finished iterating, a simple print statement is used to access the chord name and send it to the UART interface.

f) Silence Detection: Silence detection is needed to prevent the detection of chords when nothing is being played. The system must recognize when nothing is being played as early as possible to prevent unnecessary running of chord recognition code.

Our program uses the average of all DFT values of each frame for a quantified volume of the input. This average is compared to a threshold; if the threshold is not met, a chord is still detected. However, if the threshold is not met three times in a row, our program skips all chord recognition code and returns "nothing detected".

V. Experimental Results

In this section, the system performance in terms of accuracy and computation speed of the chord recognition is evaluated. In what follows, the hardware cost on the Zync FPGA is further estimated.

A. Accuracy and Computation Speed

Having implemented all the algorithms in the previous sections, we created a bootable BIN file for the Zybo SoC to boot into through the SD card. We did testing using a standard tuned guitar, a PopVoice Lapelle Microphone, and TeraTerm on a laptop computer to read the UART output. The platform is shown in Fig. 8, and the experimental results are demonstrated in Fig. 9.

Fig. 8. FPGA Demonstration

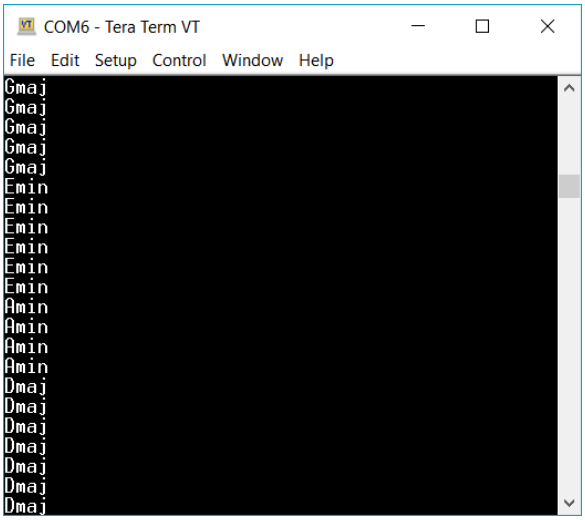

Fig. 9. Chord Recognition Results

TABLE I
FPGA Resource Cost

Resource	Utilization	Available	Utilization %
LUT	547	17600	3.11
RAM	60	6000	1.00
FF	791	35200	2.25
IO	9	100	9.00
BUFG	2	32	6.25

On average, when testing while playing open chords, we manage to get close to 90% accuracy for the chord detection. While playing barre chords, the accuracy drops close to 65%.

The delay between chord outputs averages near 0.6 seconds. Considering the sampling frequency and number of samples, it can be observed that close to 0.25 seconds are used in the sampling process, and the remaining 0.35 seconds are used for the rest of our algorithms. In all, the speed and accuracy of our system exceed the system requirements we had set for ourselves to consider the system $real - time$, which is 350 ms time constrain for sampling one chord [16].

B. FPGA Cost

The hardware cost on FPGA is summarized in Table I, including 547 slices of look-up-table (LUT) and 791 slices of flip-flops (FFs). The IOs consumption is 9% of the total IOs on Zybo FPGA.

Using Xilinx Power Analyzer, the power consumption is estimated in Fig. 10. The static power dissipation is 120 mW, which is 8% of the total power. Using the ARM PS7, the dynamic power cost can reach 1.405 W, which is 94% of the total power due to the frequent register configurations for performing the algorithms with software. Hence, there is a great potential to reduce power consumption by implementing all the algorithms by FPGA as the future work.

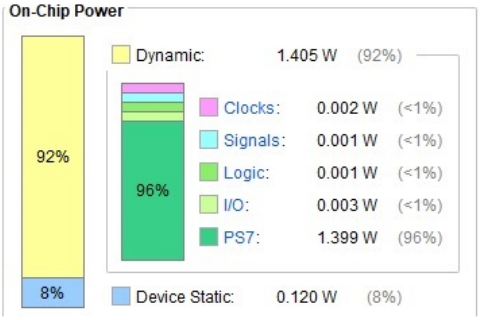

Fig. 10. Power Consumption

VI. Conclusion

This paper presents a scalable audio processing platform on FPGA, containing an open design on Zync7000 FPGA and performance evaluation in terms of accuracy, latency, slice cost, and power consumption. More important, we proved the validity of our proposed platform with an application of automating music transcription, capable of finding specific notes played by users through extracting and analyzing the frequencies from a live recording. This platform is reusable and expandable to a diverse range of applications in audio processing and speech recognition with FPGA. We hope that the public release of the platform will lead to multiple designs in the future, and serve as a framework to projects of research and education.

VII. Future work

The future work of this open platform focuses on a real-time music transposer with designs on FPGA. There were many powerful computer programs that allow for musicians and hobbyists to write out sheet music, such as TuxGuitar, GuitarPro, and others. They allow for the user to compose music using both the traditional staff, as well as through tablature. Though powerful, these programs still require the user to manually input the music. This process can take a very long time, especially when the musician is composing an original piece. For example, Lunaverus has developed a software called AnthemScore which takes in audio files like mp3, WAV, etc., and uses computer vision and convolutional neural networks to extract the notes from visualized spectrogram [17].

All these approaches to automating music transcription have an overhead of three to five minutes, compared to our intended range of millionths of seconds. The ideal end goal of our work is a pure hardware implementation on an ASIC, which is able to be integrated as an IP for system-on-chips (SoCs) for devices like Amazon Echo and Google Home.

VIII. Acknowledgment

This research comes from one of the Senior Projects in Fall 2018 - Spring 2019 semesters at University of Houston Clear Lake. In this paper we focus on the design of an audio processing system with FPGA. The entire system, including the database and phone application, were completed through a team effort. We thank Mr. Rigo De Leon for assistance with the database on the Raspberry Pi, Mr. Harold Mao for assistance with the interfacing between the FPGA and Raspberry Pi, and both with their assistance in the APP development.

References

[1] E. Benetos, S. Dixon, Z. Duan, and S. Ewert, "Automatic Music Transcription," IEEE Signal Processing Magazine, pp.21-30, 2019.
[2] P. Coleman, A. Franck, J. Francombe, et al., "An Audio-Visual System for Object-Based Audio: From Recording to Listening," IEEE Trans. on Multimedia, vol. 20, no. 8, pp. 1919-1931, Aug. 2018.
[3] J. Jiang, "Audio processing with channel filtering using DSP techniques," 2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC), pp. 545-550, 2018.
[4] Irwansyah, M. B. Andra, K. Kiyota, K. Mitarai and T. Usagawa, "Open-Source Raspberry Pi Hearing Assistance Device with Consumer Hardware," 2018 IEEE 7th Global Conference on Consumer Electronics (GCCE), Nara, pp. 164-165, 2018.
[5] F. Raffaeli and S. Awad, "Portable low-cost platform for embedded speech analysis and synthesis," 2016 12th International Computer Engineering Conference (ICENCO), Cairo, pp. 117-122, 2016.
[6] C. H. Lee, "Location-Aware Speakers for the Virtual Reality Environments," in IEEE Access, vol. 5, pp. 2636-2640, 2017.
[7] I. Al-Hussaini et al., "Predictive Real-Time Beat Tracking from Music for Embedded Application," 2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR), pp. 297-300, 2018.
[8] J Zhang, G. Ning, and S. Zhang, "Design of Audio Signal Processing and Display System Based on SoC," 2015 4th International Conference on Computer Science and Network Technology (ICCSNT 2015), pp. 824 - 828, 2015.
[9] M Psarakis, A. Pikrakis, G. Dendrinos, "FPGA-based Acceleration for Tracking Audio Effects in Movies," 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, pp. 85-92, 2012.
[10] D. Theodoropoulos, G. Kuzmanov, G. Gaydadjiev, "Multi-Core Platforms for Beamforming and Wave Field Synthesis," IEEE Trans. on Multimedia, vol. 13, no. 2, pp. 235 - 245, 2011.
[11] "Zynq-7000 SoC Data Sheet: Overview," V1.11.1, Xilinx, July 2, 2018.
[12] "Zynq-7000 SoC Technical Reference Manual," V1.12.2, Xilinx, July 2018.
[13] "Low Power Audio Codec – SSM2603 Data Sheet," Rev. D, Analog Devices, Inc., Norwood, MA, US, 2008–2018.
[14] W. Shi, et al. "Edge Computing: Vision and Challenges," IEEE Internet of Things, vol 3, no. 5, pp. 637–646, Oct. 2016.
[15] X. Yang, et al., "A Vision of Fog Systems with Integrating FPGAs and BLE Mesh Network," Journal of Communications (JC) , Vol. 14, No. 3, PP. 210-215, March 2019.
[16] K. Vaca, A. Gajjar, and X. Yang, "Real-Time Automatic Music Transcription (AMT) with Zync FPGA," IEEE Computer Society Annual Symposium on VLSI (ISVLSI), In Press, 2019.
[17] Lunaverus, "Automatic Music Transcription Software," 2019. [Online]. Available: https://www.lunaverus.com/
[18] "AMBA AXI Protocol Specification," Axis. Sunnyvale, CA, USA, 2003.
[19] T. Fujishima, "Real time chord recognition of musical sound: a system using common lisp music," in Proceedings of the International Computer Music Conference (ICMC1999), pp. 464–467, 1999.
[20] T. Palace, "Stand-Alone Device for Chord Detection," Capstone Design Project, Muse Union, 2015.
[21] G. Husain, W. Thompson, and E. Schellenberg, "Effects of Musical Tempo and Mode on Arousal, Mood, and Spatial Abilities," An Interdisciplinary Journal, vol. 20, no. 2, pp. 151-171, 2002.
[22] X. Yang and J. Andrian, "A High Performance On-Chip Bus (MSBUS) Design and Verification," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., Vol. 23, Issue: 7, PP. 1350-1354, Sept. 2015.
[23] X. Yang and J. Andrian, "An Advanced Bus Architecture for AES-Encrypted High-Performance Embedded Systems," Patent, US20170302438A1, Oct. 19, 2017.
[24] R.M. Mottola, "Table of Musical Notes and Their Frequencies and Wavelengths," [Online]. Available: https://www.liutaiomottola.com/formulae/freqtab.htm, Sept. 2018.
[25] Crockett et al., "The Zynq Book," [Online]. Available: http://www.zynqbook.com/, 2015.