Journal of Smart Technology Applications
Vol.3, No.1 (2022), pp.1-20
http://dx.doi.org/10.21742/jsta.2022.3.1.01

FPGA based Implementation of an Audio Signal Processing System on Zedboard

Shensheng Tang[1], Siong Moua[2], Yi Xie[3] and Yi Zheng[4*]

[1,2,4]Department of Electrical and Computer Engineering, St. Cloud State University, St. Cloud, MN 56301, USA
[3]School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou 510006, China
[1]stang@stcloudstate.edu, [2]smoua@go.stcloudstate.edu, [3]xieyi5@mail.sysu.edu.cn, [4]zheng@stcloudstate.edu

Abstract

An FPGA based audio signal processing system is designed and implemented through an input and output circuit created on the breadboard, a Vivado project and an SDK application program involving the Zedboard, and a C# GUI application for the serial port communication with the Zedboard. In the SDK application development, we use the Matlab filter designer tool for an FIR LPF design and generated the filter coefficients file for the SDK programming. The filtering is designed by the convolution operation and implemented through C/C++ programming. A modified mean normalization algorithm is proposed to apply to the filtered data samples. Finally, the integrated system is tested through a mixed music and interference tone signal. Experiment results verified the successful implementation of the audio signal processing system on FPGA (Zedboard). This paper provides hands-on experience in FPGA based embedded system design and implementation through Xilinx Vivado and SDK tools as well as C# GUI programming.

Keywords: *FPGA, Vivado, SDK, Zedboard, C/C++, C# GUI, FIR Filter, Convolution, Normalization*

1. Introduction

An audio signal is a representation of sound using either continuous analog signals or binary numbers of digital signals. Audio signal processing is a method where intensive algorithms and techniques are applied to audio signals. It is used to convert between analog and digital formats, cut or boost selected frequency ranges, remove unwanted noise, add effects, and obtain many other desired results [1]. The frequency range of audio signals is between 20 Hz to 20 kHz, which is the lower and upper bound that humans can hear.

An audio processor is a processor that is optimized to process sound. It often leverages ARM-based processor architecture and has components like analog-to-digital converters (ADCs), Digital-to-Analog Converters (DACs), multiple digital microphone inputs, hardware accelerators like FPGAs, and various interfaces. An audio processor is normally bundled with software or firmware designed to perform certain echo cancellation or noise reduction functions.

Article history:
Received (June 22, 2022), Review Result (August 26, 2021), Accepted (October 21, 2021)

Filters are considered the most basic circuit in any signal processing including audio signal processing. It removes unwanted noise, echo, and distortion, and allows the filtered data to pass through it. Typically, there are four types of filters: Low-pass Filter (LPF), High-pass Filter (HPF), Band-pass Filter (BPF), and Band-stop Filter (BSF). LPFs allow the frequencies below the selected cut-off frequency to pass and reject the frequencies above it. HPFs are the opposite of LPFs. HPFs pass the frequencies that are higher than the cut-off frequency and attenuate the frequencies that are lower than it. BPFs allow only the signals within a certain range of frequencies and reject all others. BPF circuits can be designed by combining the properties of low-pass and high-pass into a single filter. BSFs, also called band rejection filters or notch filters, are the opposite of BPFs. They remove frequencies in a specified frequency band and allow frequencies below the low cut-off point or above the high cut-off point to pass.

Field-programmable Gate Arrays (FPGAs) [2] are programmable logic devices that inherently contain parallel processing and pipelining features and allow flexible reconfigurable computing. FPGA has applications in many areas such as telecommunication systems [3], medical electronics [4], audio processing [5], image & video processing [6][7], and high-performance computing [8]. Combining an audio processor with an FPGA creates an ideal division of labor for many tasks in unique industrial communications and control applications.

For audio signal processing in practical applications such as acoustic detection, audio synthesis, music processing, noise cancellation, and speech recognition, researchers and engineers have conducted extensive research and experiments on FPGA based hardware accelerators due to their hardware- and software-level re-configuration and massively parallel processing capabilities. In [9], a low-complexity FPGA-based prototype was described to compute and visualize acoustic intensity images in real-time. In [10], an FPGA-based hardware platform was proposed to accelerate an audio tracking method. The accelerator was implemented in a Xilinx Virtex-5 device and the experimental results showed that it has achieved significant speedup compared with the software implementation of the tracking method. In [11], the authors presented an audio signal processing system based on FPGA, where FPGA is used as the high-speed signal processor to realize volume adjustment and audio effect control. In [12], an audio enhancement technique using digital FIR filters was implemented on FPGA to separate signal components based on their frequency contents so that the signal can be enhanced with the desired frequency components using a bandpass digital filter. In [13], a Wave Digital Filter (WDF) emulation system suitable for audio applications was implemented on an FPGA. The WDF emulations were validated by measuring the signals from the FPGA I/O modules with a data acquisition system.

In [14], an open FPGA platform was developed for the speech, hearing, and acoustics research communities to implement low-latency high-performance signal processing with deterministic latencies. A Simulink model was developed to implement audio processing and from it, the VHDL code was generated. The VHDL code was then implemented in FPGA which functions as a real-time signal processor. In [15], an open audio processing platform on Zync7000 FPGA was presented for collecting analog frequency signals through a microphone and sending a data set of frequencies and amplitudes to a UART interface. The platform was verified by a design on a real-time Automatic Music Transcription (AMT) system. In [16], an FPGA based embedded system was implemented for audio signal alignment, where a custom accelerator was designed for the Xilinx PYNQ board and the programmable logic (PL) of the Zynq SoC (system on chip) was used to accelerate the audio application. In [17], an audio signal processing implementation was presented on two FPGA development boards (Zybo and Zedboard). A Least Mean Square (LMS) adaptive filtering algorithm was implemented on FPGA and some performance measures were studied in both Zybo and Zedboard for comparison.

In this paper, we develop an FPGA based audio signal processing system on the Zedboard [18]. A Vivado [19] project and an SDK [20] application program are developed for hardware and software implementation. A C# GUI application is written via Visual Studio [21] and runs on a laptop for serial port communication with the Zedboard. In the SDK application development, The Matlab filter designer tool [22] is used for designing an FIR LPF and generating the filter coefficients file, which is used for the SDK programming. The filtering is designed by the convolution operation and implemented through C/C++ programming. A modified mean normalization algorithm is proposed to apply to the filtered data samples. Finally, we test the integrated system with a mixed music and interference tone input signal and provide the experiment results for verification.

The remainder of the paper is organized as follows: Section 2 describes the audio system and related theory for audio processing; Section 3 details the design and implementation of the proposed audio system including the external input and output circuit, the Zedboard hardware, and software (Vivado project and SDK application program), and a GUI application; Section 4 provides the experiment results; Finally, Section 5 concludes the paper.

2. System Description and Theory

The audio signal processing system to be implemented includes three parts: the external input and output circuit, the Zedboard involving both Processing System (PS) and Programmable Logic (PL) parts, and a Graphical User Interface (GUI) program running on a laptop, as shown in [Figure 1]. The input circuit receives a mixed audio and interference tone signal and amplifies it and sends it to the XADC port on the Zedboard. The Zedboard samples the mixed signal by Analog to Digital Converter (ADC) and processes it through a Finite Impulse Response (FIR) Low Pass Filter (LPF), which is implemented by the XADC IP wizard [23] on FPGA and the SDK application program. The processed signal is then sent out to the output circuit that contains a Digital to Analog Converter (DAC) ladder and an op-amp, which is connected to a speaker. A simple GUI program is developed for the user to control the signal processing through a Universal Asynchronous Receiver-Transmitter (UART) serial port.

The input audio plus interference signal from any source is first amplified through an op-amp circuit and then sent to the Audio Signal Processor (ASP) via the XADC interface on the Zedboard for processing. Note that the user can choose to add or not to add the noise signal to the audio signal for input.

The ASP is functionally comprised of the XADC IP and a Software Development Kit (SDK) application program. The XADC IP wizard is an analog mixed-signal module developed by Xilinx to configure the on-chip XADC to ADC block in 7 series FPGAs. The combination of the ADC with Programmable Logic (PL) enables a broad range of analog data acquisition and monitoring requirements. In Vivado, the XADC Wizard can be found under the IP Catalog. The wizard is a GUI that allows users to select the required block I/O and initialize the control registers for the required operation. The SDK application program is to implement an FIR digital filter. The filter samples are at 16 kHz (however, it is measured at 14 kHz in the experiment). The input signal sampling process is done through timer interrupt control in the system to be implemented by a Vivado project. The output of the filter is normalized before it is passed to the 10-bit DAC output circuit. The digital filter coefficients will be generated by the Matlab filter designer tool.

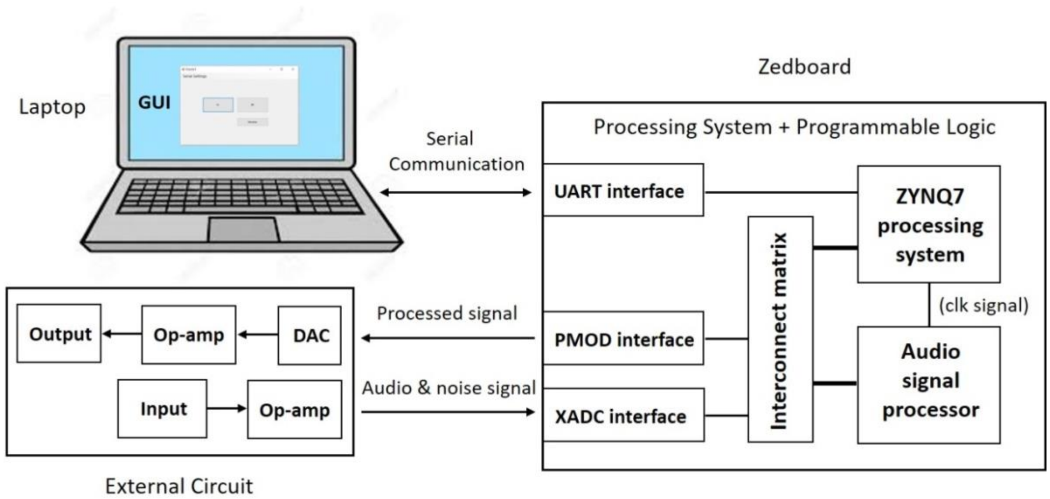

Figure 1. The block diagram of the proposed audio signal processing system

The filter designer tool is a quick way to design digital filters by setting filter performance specifications. It just needs to type filterDesigner at the Matlab command prompt for starting. When the digital LPF design is completed, we need to export the filter coefficients as signed 16-bit integers to a C Header file, which is stored in a local folder and will be used by the SDK application program.

Before writing the SDK application program, one needs to create a Vivado project by the IP Integrator in the Vivado Design Suite [19]. The Vivado project is used to support the hardware and software co-design. The detailed implementation procedure of the Vivado project will be addressed in the next section. Once the Vivado project is completed, one needs to export the hardware and launch the SDK. From the SDK window, one can start to write the filter application program, which involves two theoretical concepts about the sampled digital signal: convolution operation and normalization processing.

1. Convolution Operation

Convolution is the most important concept in Digital Signal Processing (DSP). Without convolution, the digital filter used in this system implementation would not be possible. Convolution in the time domain is multiplication in the frequency domain and vice versa. The standard mathematical expression for convolution is as follows [24]:

$$y[n] = h[n] * x[n] = \sum_{k=0}^{M-1} h[k]x[n-k] \tag{1}$$

where $x[n]$ is the input signal with N points from the index of 0 to N-1, $h[n]$ is called the impulse response of the digital filter system (also called the filter kernel) with M points from 0 to M-1, and $y[n]$ is the output signal with $N+M$-1 points from 0 to $N+M$-2. The index n determines which sample in the output signal is being calculated. The programming implementation of the convolution operation can be a loop that makes n run through each sample in the output signal. To calculate one of the output samples, the index k is used. When *k* runs through 0 to *M*-1, each sample in the impulse response *h*[*k*] is multiplied by the proper sample from the input signal *x*[*n*-*k*]. All these products are added to produce the output sample being calculated. A detailed implementation will be introduced in the next section.

2. Normalization Processing

Normalization is an important concept in the implementation of our audio processing system. Since the filter coefficients are fixed points and are saved as 16-bit integers, which means that the coefficients may be very large or very small within the range from -32768 to +32767. The processed samples will be sent to a 10-bit DAC output circuit. Hence, the desired output is within 0 to 1023. Commonly used normalizations in DSP are min-max normalization [25], mean normalization [26], and others [27][28]. Here we use modified mean normalization for the digitalized input. The corresponding formula is given as follows:

$$s_{norm} = \frac{s-\mu}{s_{max}-s_{min}} * 1023 + 512 \tag{2}$$

where s_{norm} represents the normalized data sample in a data set, s is the current data point to be normalized, μ is the mean of the data set, which has a minimum value s_{min} and maximum value s_{max}. This formula can be implemented in programming by the following algorithm:

Step 1: Calculate the mean μ of the data set including all the input samples.

Step 2: Search the data set to find the minimum value s_{min} and maximum value s_{max}.

Step 3: Calculate the normalized value s_{norm} for the current data point using the formula.

Step 4: Repeat the above step for the next data point until all data points are calculated.

The processed signal (or specifically, the data set) will be sent to the external output circuit through the PMOD interface on Zedboard. The output circuit includes an R-2R DAC ladder, an op-amp circuit, and a speaker.

A simple GUI panel is developed through Visual Studio for some controlling operations such as turning on the LPF, turning off the LPF, do normalization. The GUI is communicated with the Zedboard through a UART serial port.

3. System Design and Implementation

In this section, we discuss the detailed design and implementation issues for the audio signal processing system according to individual components: External input and output circuit, Zedboard hardware and software (Vivado project and SDK application program), and GUI.

3.1. External input and output circuit

[Figure 2] shows the input and output circuit diagrams. The input circuit is fed by a mixed audio and interference signal, which is amplified through the op-amp LM386 [29]. This is the input mixed signal that is usually adjusted to be very small to reduce the additive ambient noise.

It is worth noting that a voltage divider should be added at the output of the op-amp. The output of the LM386 can be close to 5 V (maximum), while the XADC interface on Zedboard requires a nominal analog input range from 0 V to 1 V (the maximum value is 1.5 V) [18]. Therefore, the voltage divider must be used, otherwise, the XADC interface can be damaged!

The amplified input biased signal by the voltage divider will be connected to the V_P terminal of the XADC interface, and the V_N terminal is connected to the ground, as shown in [Figure 3].

The processed data samples will be sent to the R-2R DAC ladder, which has ten terminals connecting to the PMOD interface. The Zedboard has five PMOD compatible headers. Four headers (JA to JD) interface to the PL-side of the Zynq-7000 AP SoC and connect to Bank 13 (3.3V) from the Zedboard Schematic [30]. One header, JE, connects to the PS-side on MIO pins [0, 9-15] in MIO Bank 500 (3.3V) [30]. Each PMOD interface includes eight users I/Os plus two 3.3V and ground signals, as shown in [Figure 4].

Our DAC has 10 bits so we need two PMOD headers. In the experiment, we assign 8 bits on the JC header and 2 bits on the JD header. The output of the DAC ladder becomes an analog signal, which is connected to the input of the output op-amp LM386 for amplification. The amplified signal is sent to the speaker. [Figure 5] shows the implementation of the input and output circuit on the breadboard.

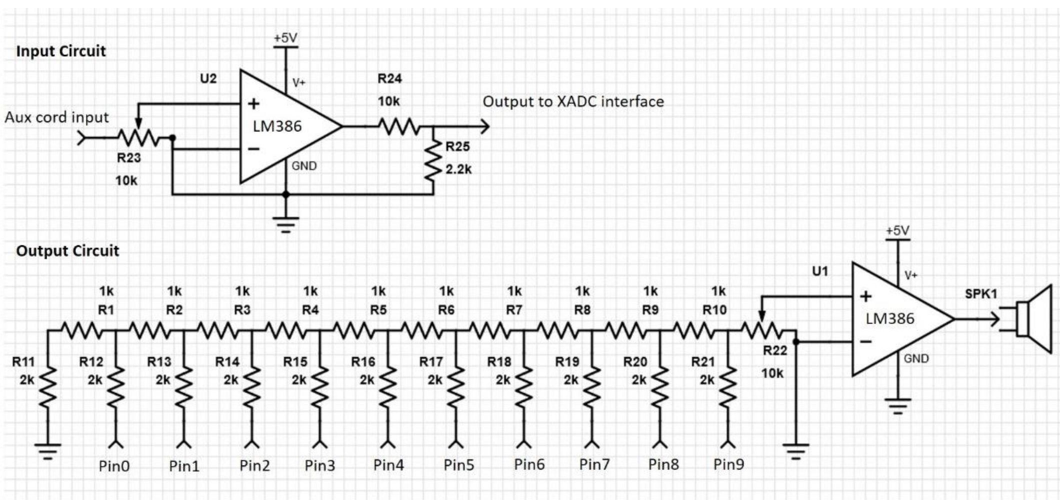

Figure 2. The external input and output circuit diagrams

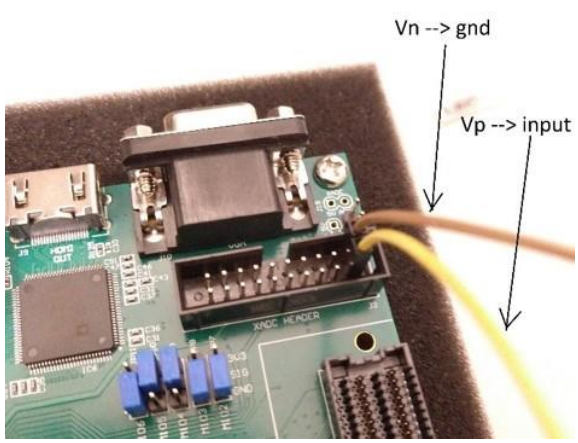

Figure 3. The XADC interface is to be connected to the voltage divider

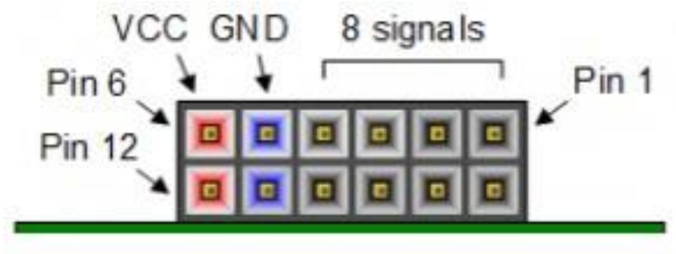

Figure 4. The PMOD interface is to be connected to the DAC ladder

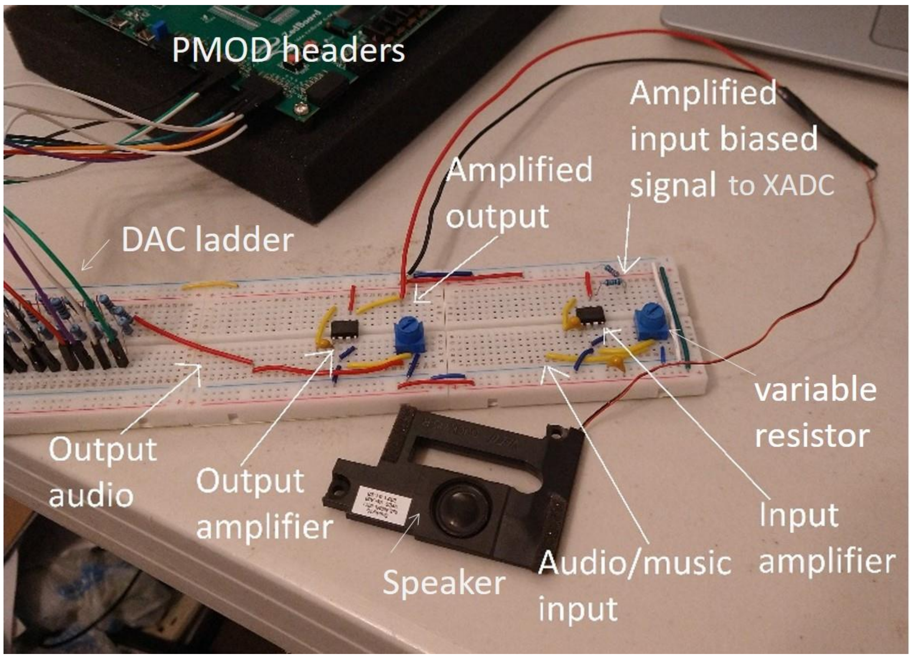

Figure 5. The implementation of the input and output circuit on the breadboard

3.2. Zedboard hardware and software

When the XADC interface receives the mixed audio signal from the external circuit, the Zedboard will take over the main work for the proposed system. This will involve the Zedboard hardware itself, a Vivado project that implements the FPGA design for the proposed system, and an SDK application program that can create a software environment to enable the interaction between the completed hardware architecture exported from the Vivado project and the physical components on the Zedboard.

1. Xilinx Vivado Block Design

Vivado [19] is an integrated hardware development tool for creating a Verilog or VHDL design and converting the design into a configuration file so that it can be programmed to the Xilinx FPGA. The hardware design allows one to specify which microprocessors, memory blocks, and other peripheral IPs to use, how the different IPs are interconnected, the memory map (i.e., addresses for memory mapped IO/peripherals), and how the different I/O signals map to actual pins on the FPGA and thus the development board (e.g., Zedboard). The output from Vivado is the FPGA configuration file that describes the hardware of the system.

For our proposed system, we first create a Vivado project select and select the Zedboard as our default board. Then Vivado projects select and select the Zedboard as our default board. Then we create a block design in the IP Integrator and add the necessary IPs into the design work environment. [Figure 6] shows the complete Vivado block design, which is composed of the ZYNQ7 Processing System and XADC Wizard IPs. Note that during the IP connection process, we have used the function of “Run Connection Automation” and the IP Integrator automatically instantiates two further IP blocks:

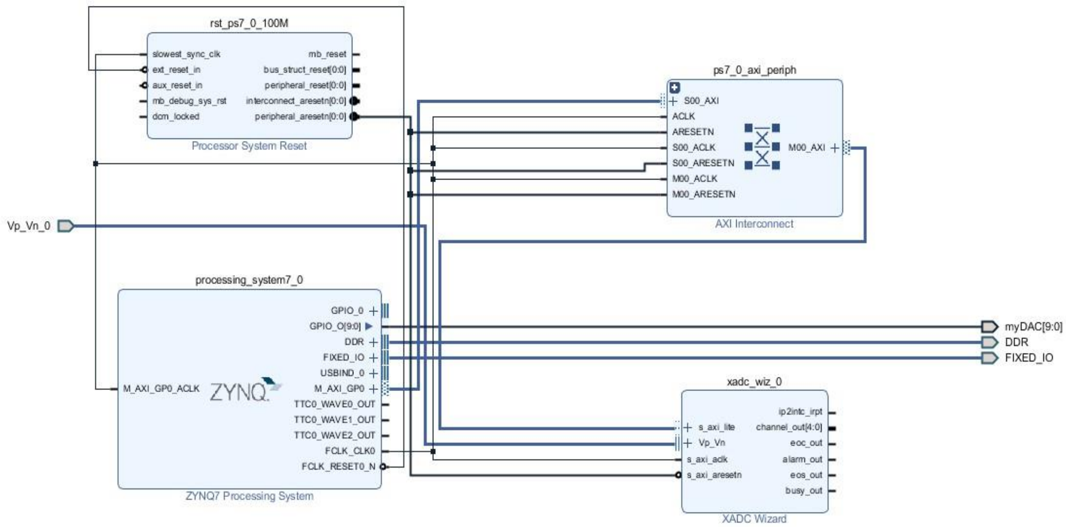

Figure 6. The complete block design diagram of the Vivado project for the proposed system

a) Processor System Reset IP - provides customized resets for an entire processing system, including the peripherals, interconnect, and the processor itself.

b) AXI Interconnect IP - provides an AXI interconnect for the system, allowing further IP and peripherals in the PL to communicate with the processing system.

In the following, we briefly introduce the IPs and their configurations for the proposed system.

- ZYNQ7 Processing System

The Zynq7 PS IP is the software interface around the Zynq-7000 Processing System, providing a logical connection between the PS and the PL while assisting the user to integrate custom and embedded IPs with the processing system using the Vivado IP integrator. The key features include enabling/disabling I/O peripherals, PL clocks and interrupts, and configurations of multiplexed IO (MIO), extended MIO (EMIO), DDR, and security.

How to configure the PS IP to fit into the proposed system?

Double click on the "ZYNQ7 Processing System" IP and navigate to the "MIO Configuration" page on the pop-out "Re-customize IP" window, then find the "EMIO GPIO (Width)" option under the "I/O Peripherals" and set it to be 10 (as our system uses a 10 bit DAC). Click OK to close the window. Now we can see a "GPIO_0[9:0]" port appears on the PS IP. Right-click on the port and select "Make External" to add an output terminal and change its name to "myDAC", as shown in Figure 6.

- AXI Interconnect IP

The AXI Interconnect IP connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. The AXI interfaces conform to the AMBA AXI version 4 specifications from ARM [x], including the AXI4-Lite control register interface subset. This IP uses well-defined master and slave interfaces that communicate via five different channels: Read address, Read data, Write address, Write data, and Write a response.

An AXI Interconnect manages the AXI transactions between AXI masters and AXI slaves, allowing multiple AXI masters to communicate with multiple AXI slaves. It can contain a few different digital components such as arbiters, decoders, multiplexers, protocol converters, register slices, and clock converters.

- XADC Wizard IP

The XADC Wizard IP [23] provides an easy way to configure the on-chip XADC analog to digital converter block in 7 series FPGAs to the user's desired mode of operation with a graphical interface. The graphical interface generates an HDL wrapper with all the needed configuration attribute settings, providing an easy way to integrate the XADC block into the HDL design.

The key features include easy configuration of required modes and parameters (ADC conversion rate, calibration settings, dynamic reconfiguration port (DRP) interface, etc.), simple interface for channel selection and configuration, ability to select/deselect alarm outputs and to set alarm limits for temperature and voltage levels, calculating all required parameter settings and register values from user inputs.

How to configure the XADC Wizard IP to fit into the proposed system?

Double-click on the XADC Wizard IP and the "Re-customize IP" window pops out, which includes five tabs: Basic, ADC Setup, Alarms, Single Channel, and Summary. In the "Basic" tab, keep all settings as default. However, in the "ADC Setup" tab and "Alarms" tab, uncheck all that is checked (One will find a few ports have disappeared from the IP icon on the work environment). In the "Single Channel" tab, make sure to select the channel as "Vp Vn". Finally, we can see a summary of all our settings in the "Summary" tab, as shown in [Figure 7].

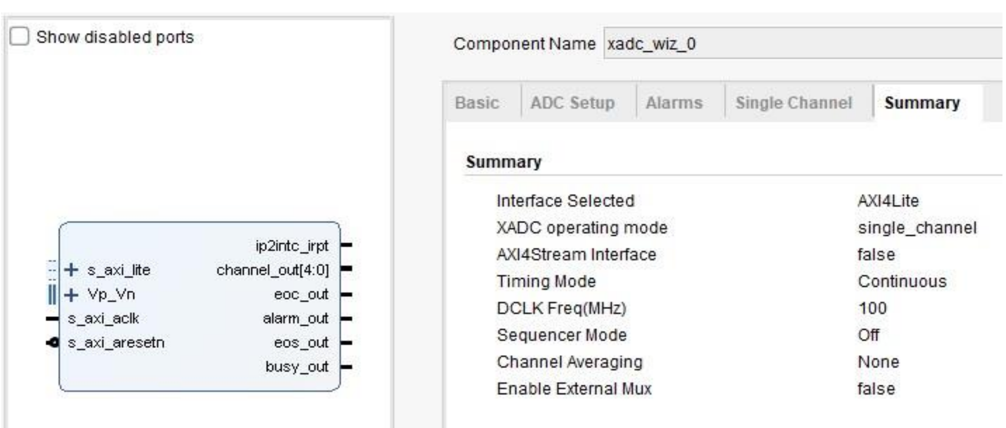

Figure 7. The configuration summary of the XADC Wizard IP

Click OK to close the "Re-customize IP" window and go back to the system block design diagram. Right-click on the Vp_Vn port of the XADC Wizard IP and select "Make External" to add an input terminal with the name "Vp_Vn_0", as shown in [Figure 6].

After the connection is done, click the validation checkbox to get a “Validation Successful” message. Then go to the "Sources" pane and right click on the block design file name and select "Create HDL Wrapper" to add the top-level HDL wrapper around the block design because a block design source cannot be synthesized directly.

Before running synthesis for the block design, we need to add a constraints file with the suffix XDC to the Vivado project. Our constraints file is displayed in Figure 8. Note that the name used in the constraints file must be the same as that name shown in the block design diagram (i.e., myDAC[9:0]) for mapping it to the real package pin name on the Zedboard (i.e., PMOD JC and JD pins).

Now it is ready to run synthesis, implementation, and generate the bitstream. A bitstream includes the description of the hardware logic, routing, and initial values for both registers and on-chip memory.

```
# The constraints file for the proposed system

# JC Pmod - Bank 13
set_property PACKAGE_PIN AB6 [get_ports {myDAC[1]}];  # "JC1_N"
set_property PACKAGE_PIN AB7 [get_ports {myDAC[0]}];  # "JC1_P"
set_property PACKAGE_PIN AA4 [get_ports {myDAC[3]}];  # "JC2_N"
set_property PACKAGE_PIN Y4  [get_ports {myDAC[2]}];  # "JC2_P"
set_property PACKAGE_PIN T6  [get_ports {myDAC[5]}];  # "JC3_N"
set_property PACKAGE_PIN R6  [get_ports {myDAC[4]}];  # "JC3_P"
set_property PACKAGE_PIN U4  [get_ports {myDAC[7]}];  # "JC4_N"
set_property PACKAGE_PIN T4  [get_ports {myDAC[6]}];  # "JC4_P"

# JD Pmod - Bank 13
set_property PACKAGE_PIN W7 [get_ports {myDAC[9]}];  # "JD1_N"
set_property PACKAGE_PIN V7 [get_ports {myDAC[8]}];  # "JD1_P“

# IOSTANDARD Constraints
# Note that the bank voltage for IO Bank 33 is fixed to 3.3V on ZedBoard.
set_property IOSTANDARD LVCMOS33 [get_ports -of_objects [get_iobanks 33]];
# Set the bank voltage for IO Bank 34 & 35 to 1.8V by default.
set_property IOSTANDARD LVCMOS18 [get_ports -of_objects [get_iobanks 34]];
set_property IOSTANDARD LVCMOS18 [get_ports -of_objects [get_iobanks 35]];
# Note that the bank voltage for IO Bank 13 is fixed to 3.3V on ZedBoard.
set_property IOSTANDARD LVCMOS33 [get_ports -of_objects [get_iobanks 13]];
```

Figure 8. The constraints file for the proposed system

2. Xilinx SDK Software Development

After the bitstream file is successfully generated, we can export the hardware and launch SDK [20]. The Vivado project’s new SDK workspace will pop out with a folder named design_1_wrapper_hw_platform_0 generated as shown in [Figure 9], which contains the bitstream and hardware spec exported from Vivado. Now it is ready to start our SDK application program.

First of all, we must create a new application project (i.e., my_true_adc) so we can start writing code to drive our hardware, which will generate two additional folders, as shown in Figure 9: my_true_adc and my_true_adc_bsp. The former one is the actual application folder, where our main program will be written here in C/C++ programming language. The latter is the Board Support Package (BSP), which is a collection of libraries and drivers that form the lowest level of the software application stack.

Next, we discuss our design and implementation in the main program (i.e., main.c). The main program implements the FIR LPF. It is sampling at 16 kHz (it is measured at 14 kHz in the experiment). Therefore, the sampling rate of the FIR LPF is designed using 14 kHz. We use Matlab Filter Designer to design the FIR LPF and generate the required filter coefficients for convolution calculation. In the programming, the sampling of the 16-bit ADC is done through timer interrupt control. The output of the filter is normalized before it is passed to the 10-bit DAC. The 10-bit DAC is controlled by using the two PMOD headers, i.e., the JC header connecting to the first 8 bits and the JD header connecting to the last 2 bits. The main program also communicates with the GUI via a serial port to turn on/off the filter and do normalization operations. The GUI implementation will be introduced in the next subsection.

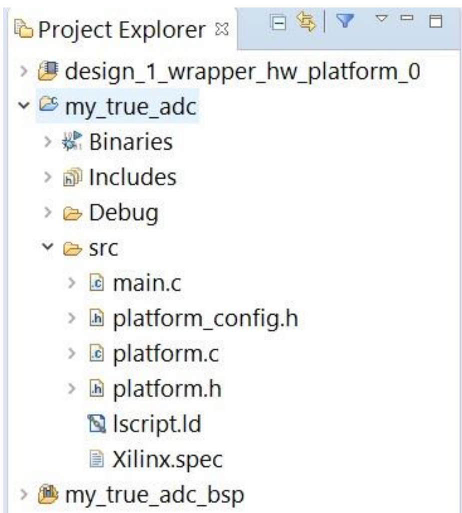

Figure 9. The launched SDK workspace for application development

- LPF Coefficients Generation

The FIR LPF is designed using the Matlab filter designer tool [22]. In our experiment, we use the window method (i.e., Blackman window) with 100 taps (however, when it is exported to a C header file, there will be 101 taps). The cutoff frequency is set to 6 kHz and the sampling frequency is 14 kHz. The LPF configuration is shown in Figure 10. The filter coefficients file is exported as a signed 16-bit integer and used by an array in the main program. The array will be used in the convolution calculation with the sampled input data.

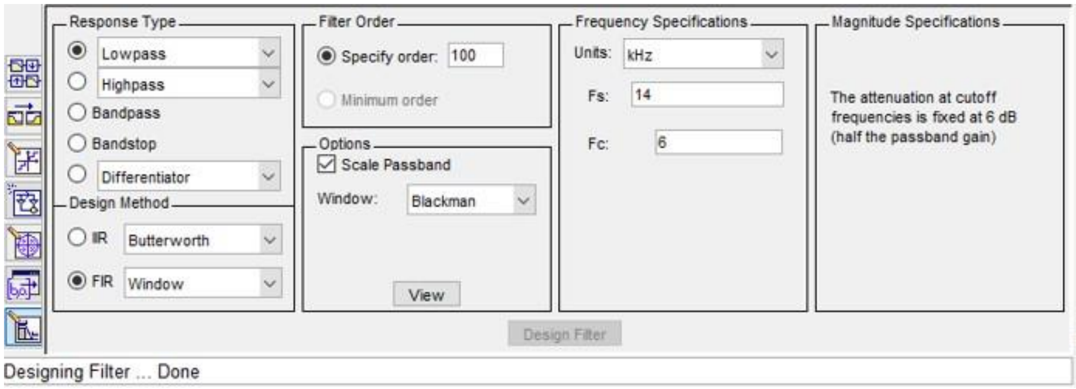

Figure 10. The FIR LPF design by the Matlab filter designer tool

- Programming Implementation for Convolution

The filter processing is implemented by the convolution operation. From Equation (1), the convolution algorithm is performed between two data sets (input data x[n] and impulse response h[n]); in our real-time application, a circular buffer of length 100 (the same length as the number of coefficients) is used to implement the convolution between the sampled data and the obtained coefficients by Matlab.

Since this circular buffer has a finite number of lengths, the main program will initialize a data array of size 101 with all zeros and start to sample the input signal and store the sampled data in the array, which is combined with the coefficients array for convolution calculation. Note that for every sampled data in the circular buffer, the convolution is performed to produce filtered output data. It will continue until all the sampled data in the circular array are finished. After that, the next new sampled data will replace the oldest sample to start a new round of operation.

The specific code implementation for the convolution operation is done by a timer Interrupt Service Routine (ISR), as shown in [Figure 11]. When the ISR starts, the program will first read the ADC module. By default, the variable counter holds the value of 0. For every sampled data read, there needs an accumulator to loop 101 times for the multiplication between the sampled data and the filter coefficients. It also checks if the variable current_position is 0. If it is 0 then the program assigns it to start at 100; otherwise, it will decrement until it reaches 0. Outside the for-loop, the variable counter counts in ascending order. Once it is larger than 100, it will be reset to 0.

```c
//The timer handler for convolution
void tmrHandler()
{
    sum = 0;
    input_samples[counter] = XSysMon_GetAdcData(&adcMon, XSM_CH_VPVN);  //read ADC and store in array
    current_position = counter;    //capture 'counter' (outer) and save as 'current_position' (inner)
    for(s8 i = filter_length_minus_1; i >= 0; i--)     //perform the convolution
    {
        sum = sum + (s32)input_samples[current_position] * (s32)coef[i];   //multiply and accumulate
        if(current_position == 0)
            current_position = filter_length_minus_1;    //if it reaches zero then start at 100
        else
            current_position--;                          //decrement inner counter
    }

        //[ Some lines not related to convolution are omitted ]

    counter++;        //increment the outer counter once a full convolution took place
    if(counter > filter_length_minus_1)   //if the outer counter reaches 100 then reset to 0
    {
        counter = 0;
    }
        //[ Some lines not related to convolution are omitted ]
}
```

Figure 11. The code implementation for the convolution operation

- Programming Implementation for Normalization

The normalization algorithm is implemented based on Equation (2). The input is the filter output (which is to be normalized); the normalized output is sent to the DAC. A code snippet of the normalization is shown in [Figure 12].

```c
// The code snippet for normalization calculation
XGpioPs led;                   //GPIO instance
XScuTimer tI;                  //create instance for timer
u32 test_data[MAX_NUM_DATA];   // The array for new normalization normalization calculation
s32 sum = 0;
double norm_out;
u16 norm_out_to_dac;

if(norm_flag == 1)    // If normalization is requested from the C# GUI
   {
       test_data[test_counter++] = sum;  //Collect the output of LPF (NOT normalized)
       if(test_counter >= MAX_NUM_DATA)  //When enough data are collected
       {
           XScuTimer_DisableInterrupt(&tI);  //Disable the Timer interrupt
           norm_flag = 0;                    //reset the normalization flag

           //Variable initialization
           s32 s_max = 0;
           s32 s_min = 0;
           s32 temp_sum = 0;
           double diff = 0;
           double avg = 0;

           //Find minimum value, maximum value, average
           s_min = test_data[MIN_NUM_DATA];
           s_max = test_data[MIN_NUM_DATA];
           for(u16 i = MIN_NUM_DATA; i <= MAX_NUM_DATA; i++)
           {
               temp_sum = temp_sum + test_data[i];
               if(s_max < test_data[i])
                   s_max = test_data[i];
               else if(s_min > test_data[i])
                   s_min = test_data[i];
           }
           avg = ((double)temp_sum)/(MAX_NUM_DATA-MIN_NUM_DATA);
           diff = (double)s_max - (double)s_min;
           norm_out = (((double)sum - avg)/diff * 1023 + 512;   //do normalization
           norm_out_to_dac = (u16)norm_out;          //Convert it to u16 before sending to DAC
           XGpioPs_Write(&led, XGPIOPS_BANK2, norm_out_to_dac); //Write normalized output to DAC
       }
   }
```

Figure 12. A code snippet of the normalization

Besides the above key implementation, other code blocks are needed for the SDK application, such as UART serial port initialization, interrupt initialization, timer initialization, ADC initialization, etc. They are not discussed here due to space limitations. However, a flowchart of the overall SDK application program is provided in Figure 13.

After the SDK application program is done, we can program FPGA from the SDK menu to run the program on the hardware architecture designed earlier. This is emulated on the FPGA on the Zedboard, which means that the FPGA will be configured to make the software and hardware work. Specifically, we select “Program FPGA” from the Xilinx Tools menu to open a dialog box and make sure the bitstream file generated earlier appears there. Click the “Program” button to program FPGA. Before running the Launch on Hardware (GDB)”, we need to open the C# GUI application, which is implemented in the next subsection.

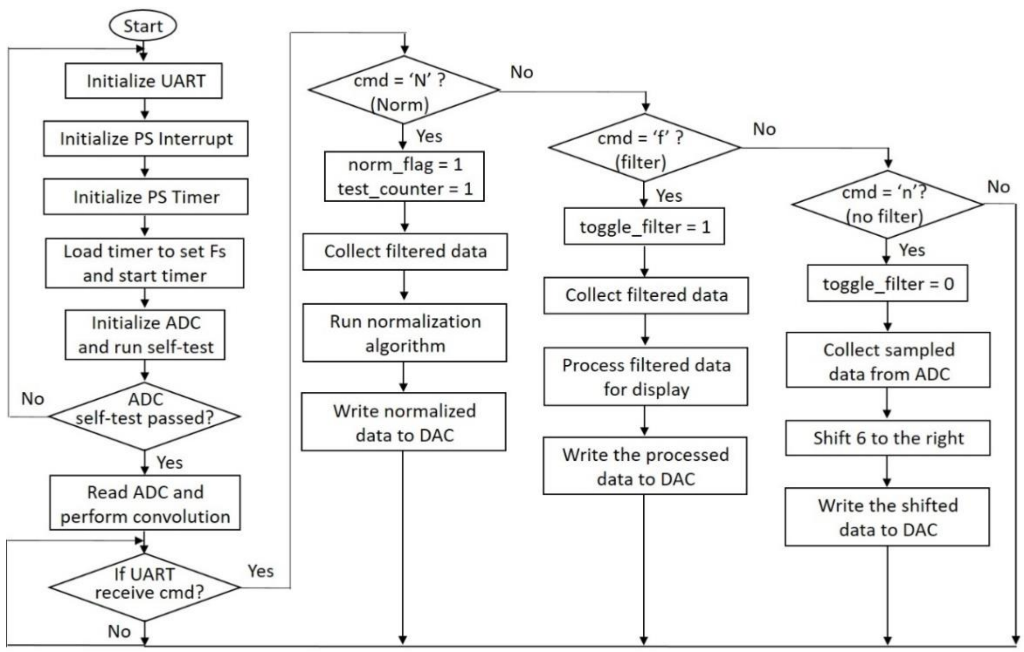

Figure 13. The flowchart of the SDK application program

3.3. GUI application

A simple C# GUI application is developed through Visual Studio [21] for the user to control the FIR LPF. It has three functions: Turn On LPF, Turn Off LPF, and Do Normalization. They are implemented by three buttons. When a button is clicked, a corresponding command character will be sent to the SDK application program. Specifically, the button “Turn On LPF” sends character “f”, the button “Turn Off LPF” sends character “n”, and the button “Do Normalization” sends character “N”. The GUI panel is shown in Figure 14, where the menu “Serial Port Settings” is used to trigger another form for serial port configuration.

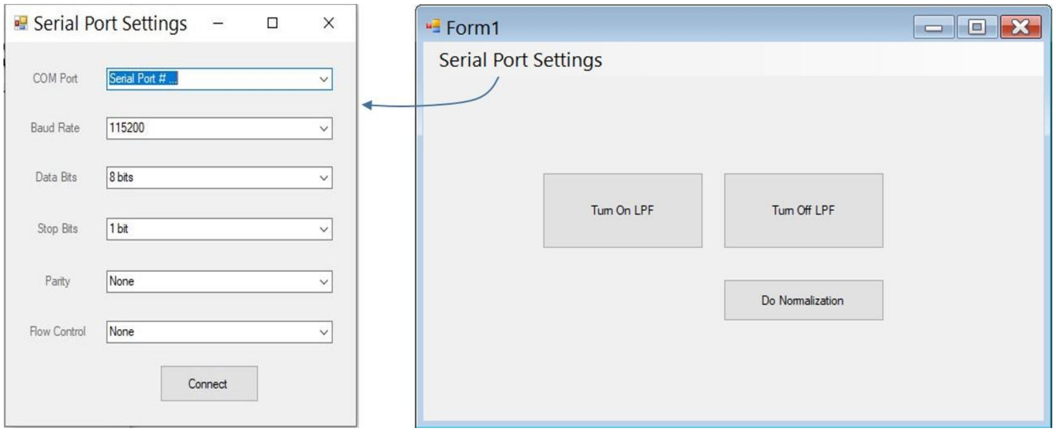

Figure 14. The C# GUI panel and the serial port configuration

4. Experimental Results

By putting it all together, we are now ready to run the whole audio processing system. This includes adding +5 V power to the external input and output circuit, programming FPGA and running the Launch on Hardware (GDB) from the SDK platform, running the C# GUI from the Visual Studio, and turning on a digital oscilloscope for signal measurement. The real experiment setting is shown in [Figure 15].

Our audio signal is randomly picked from music on Youtube, and the interference signal is picked from an online tone generator website (e.g., https://www.szynalski.com/tone-generator/). The music plus interference signal is fed into the input circuit from the laptop through an audio cable. The output signal is checked through a speaker. We can also use a digital oscilloscope with an FFT display function for watching the signal spectral characteristics. When running the C# GUI application, make sure to connect the serial port successfully through the "Serial Port Settings" menu. After running the Launch on Hardware (GDB), the system is ready to take commands from the GUI over the serial port. Some experiment results measured from the oscilloscope are studied as follows.

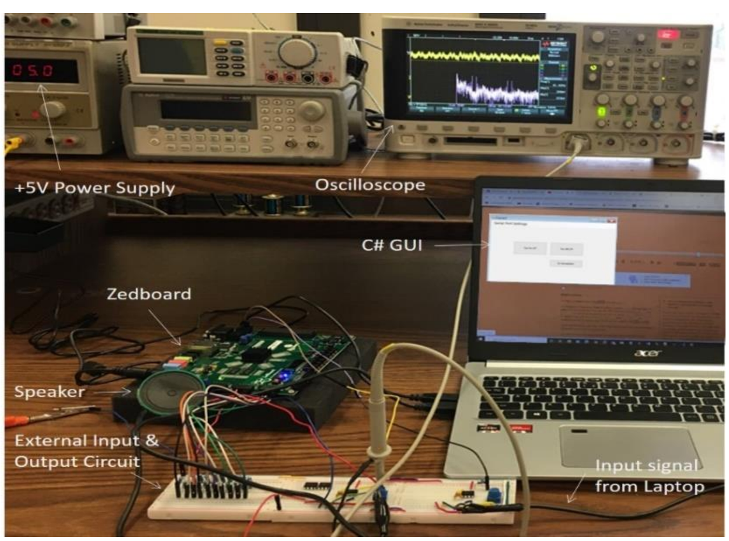

Figure 15. The audio signal processing experimental system

[Figure 16] shows an input mixed signal (music + interference tone) before the FIR LPF is performed. The tone frequency is set to 6.8 kHz as our LPF cutoff frequency is 6 kHz (we want to check the effectiveness of the designed LPF). In Figure 16, we also use the FFT function of the oscilloscope to watch the spectral characteristics of the input signal. It is seen the interference tone signal there. Then, we click the “Turn On LPF” and “Do Normalization” buttons to do filtering and normalization for the input signal. In [Figure 17], we can observe the normalized signal in the time domain and its spectral characteristics. The interference tone has been filtered.

Now we use another FIR LPF in the experiment. We first use the Matlab filter designer tool to design an FIR LPF with a cutoff frequency of 4 kHz and other parameters the same as before. Then we get the filter coefficients file and use it in the SDK main program. Figure 18 shows an input mixed signal (music + interference tone) in the time domain and its FFT display. At this time we set the interference tone to 5 kHz. After the input mixed signal is fed into the new LPF and the normalization is performed, we can see the normalized signal in the time domain and its spectral characteristics in [Figure 19]. As expected, the interference tone has also been filtered.

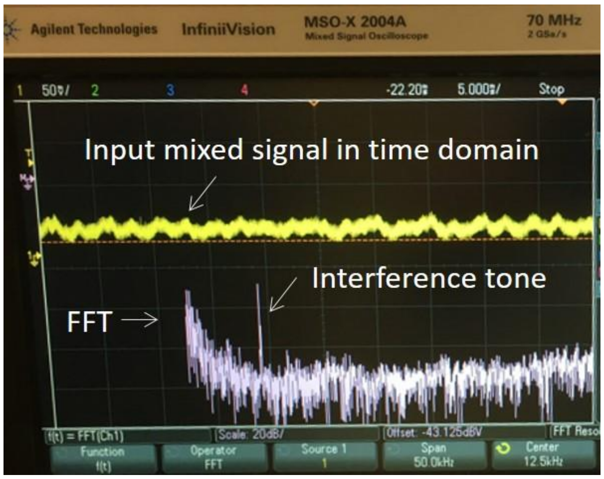

Figure 16. The audio signal processing experimental system

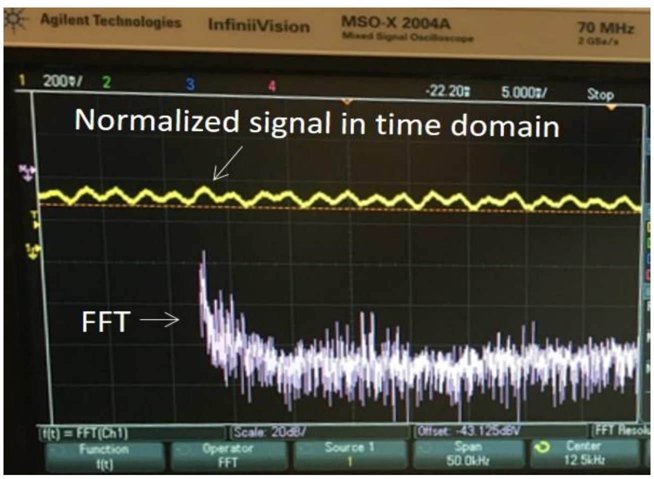

Figure 17. The audio signal processing experimental system

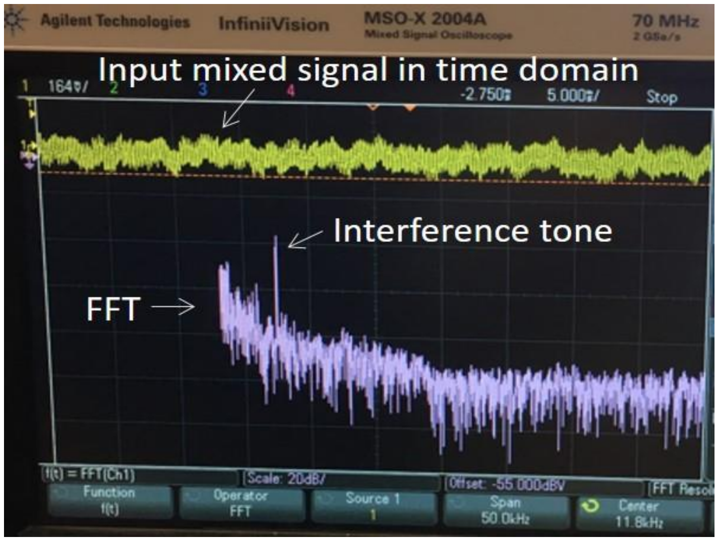

Figure 18. The audio signal processing experimental system

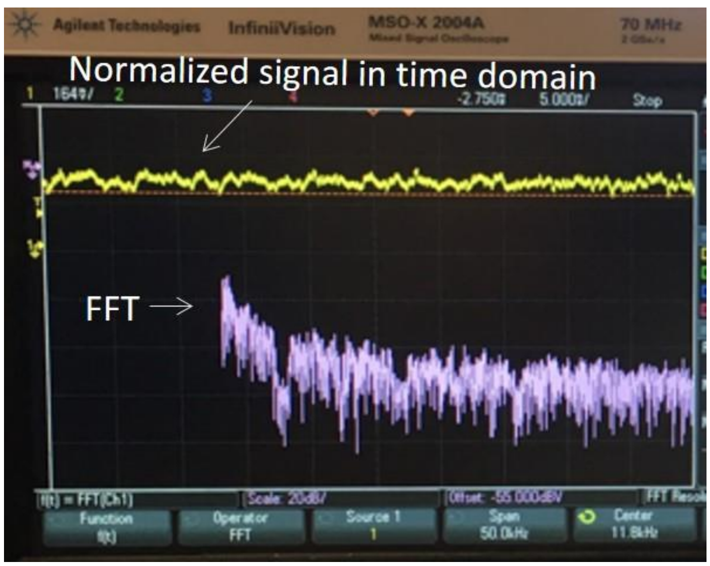

Figure 19. The audio signal processing experimental system

5. Conclusion

We developed an FPGA based audio signal processing system on the Zedboard. The system includes external input and output circuit created on the breadboard, a Vivado project for hardware implementation and an SDK application program for software implementation, and a C# GUI application for the serial port communication with the Zedboard. In the SDK application development, we used the Matlab filter designer tool for an FIR LPF design and generated the filter coefficients file for the SDK programming. The filtering was designed by the convolution operation and implemented through C/C++ programming. A modified mean normalization algorithm was proposed to apply to the filtered data samples. Finally, the integrated system was tested through a mixed music and interference tone signal as the input, ADC processing, LPF processing, and DAC processing to a speaker. We also used a digital oscilloscope to watch the input and output signals in the time domain and frequency domain. Experiment results verified the successful implementation of the audio signal processing system on FPGA (Zedboard). This paper provided hands-on experience in FPGA based embedded system design and implementation through Xilinx Vivado and SDK tools as well as C# GUI programming.

Acknowledgments

This work is supported in part by the Faculty Improvement Grant (No. 211208) from St. Cloud State University, MN, USA, and the Emerson Automation Solutions grant (No. 629528).

References

[1] W. L. G. Koontz, "Introduction to audio signal processing", RIT Press, **(2016)**

[2] S. Brown and J. Rose, "FPGA and CPLD architectures: A tutorial," IEEE Design and Test of Computers, vol.13, no.2, pp.42–57, **(1996)**

[3] P. Sikka, A. R. Asati, and C. Shekhar, "Speed optimal FPGA implementation of the encryption algorithms for telecom applications," Microprocessors and Microsystems, vol.79, (2020) DOI:10.1016/j.micpro.2020.103324

[4] P. Artem and S. Dmitry, "FPGA technologies in medical equipment: Electrical impedance tomography," East-West Design & Test Symposium (EWDTS 2013), 2013, pp.1-4, DOI: 10.1109/EWDTS.2013.6673157

[5] 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)**

[6] S. Tang, M. Sinare, and Y. Zheng, "Design, optimization and implementation of a DCT/IDCT based image processing system on FPGA," International Journal of Computer Applications in Technology (IJCAT), vol.67, no.4, pp.303-323, **(2021)**

[7] E. Morales and R. Herrera, "Video processing in real-time in FPGA," Proc. SPIE 10751, optics and photonics for information processing XII," 107510Z, September (2018), DOI: 10.1117/12.2322021

[8] M. C. Herbordt, T. Vancourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. Disabello, "Achieving high performance with FPGA-based computing," Computer, vol.40, no.3, pp.50–57, (2007) DOI:10.1109/MC.2007.79

[9] B. Zimmermann and C. Studer, "FPGA-based real-time acoustic camera prototype," Proceedings of 2010 IEEE International Symposium on Circuits and Systems, pp.1419-1419, (2010) DOI:10.1109/ISCAS.2010.5537301

[10] M. Psarakis, A. Pikrakis, and 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)**

[11] J. Zhang, X. Wang, and S. Zhang, "Audio signal processing based on FPGA," Advanced Materials Research, vol.1049-1050, pp.1759-1764, (2014), 10.4028/www.scientific.net/AMR.1049-1050.1759

[12] V. Elamaran, K. Abhiram, and N. B. P. Reddy, "FPGA implementation of audio enhancement using xilinx system generator," Journal of Applied Sciences, vol.14, pp.1972-1977, **(2014)**

[13] D. Hernandez, Y. L. Hsieh, and J. Huang, "Emulation of analog audio circuits on FPGA using wave digital filters," Proceedings of the 2nd International Conference on Communication and Information Processing (ICCIP 2016), pp.179–184, November **(2016)**

[14] R. Snider, C. Casebeer, Christopher, and R. Weber, "An open computational platform for low-latency real-time audio signal processing using field programmable gate arrays," The Journal of the Acoustical Society of America, vol.143, pp.1737-1737, (2018) DOI: 10.1121/1.5035667

[15] K. Vaca, M. M. Jefferies, and X. Yang, “An open audio processing platform with Zync FPGA,” 2019 IEEE International Symposium on Measurement and Control in Robotics (ISMCR), D1-2-1-D1-2-6, **(2019)**

[16] L. Stornaiuolo, M. Perini, M. D. Santambrogio and D. Sciuto, “FPGA-based embedded system implementation of audio signal alignment,” 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp.132-139, (2019) DOI:10.1109/IPDPSW.2019.00031

[17] A. K. Singh and R. K. Sharma, “Different FPGA implementations of audio processing using least mean square adaptive filtering: A comparative study,” 2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC), (2020), pp.980-985, DOI:10.1109/ICCMC48092.2020.ICCMC-000182

[18] Digilent, Inc, “ZedBoard hardware user’s guide,” Ver 2.2, January 2014. Available: http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[19] Xilinx, Inc, “Vivado design suite user guide: Using the Vivado IDE,” UG893, v2020.1, June 2020. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug893-vivado-ide.pdf

[20] Xilinx, Inc, “Getting started with Xilinx SDK,” v2016.2. Available: https://www.xilinx.com/html_docs/xilinx2016_2/SDK_Doc/index.html

[21] Microsoft Corp., Visual Studio 2019. Available: https://visualstudio.microsoft.com/

[22] G. Peck, “Signal processing in Matlab: Filter designer, logic analyzer, linear algebra and statistics,” CreateSpace Independent Publishing Platform, December **(2017)**

[23] Xilinx, Inc, “LogiCORE IP XADC Wizard v2.2 - User Guide,” UG772 July 25, 2012. Available: https://docs.xilinx.com/v/u/en-US/ug772_xadc_wiz

[24] S. W. Smith, “The scientist & engineer's guide to digital signal processing,” 1st Edition, California Technical Pub; January **(1997)**

[25] A. Celen, “Comparative analysis of normalization procedures in TOPSIS method: With an application to Turkish deposit banking market,” INFORMATICA, vol.25, no.2, pp.185–208, **(2014)**

[26] S. Galli, “Python feature engineering cookbook,” Packt Publishing, January **(2020)**

[27] S. Chakraborty and C-H. Yeh, “A simulation comparison of normalization procedures for TOPSIS,” Computing Industrial Engineering, vol.5, no.9, pp.1815–1820, **(2009)**

[28] A. Jahan and K. L. Edwards, “A state-of-the-art survey on the influence of normalization techniques in ranking: Improving the materials selection process in engineering design,” Materials & Design, vol.65, no.2015, pp.335–342, **(2014)**

[29] Texas Instruments Inc, “LM386 data sheet,” Revised May 2017. Available: https://www.ti.com/lit/ds/symlink/lm386.pdf

[30] Digilent, Inc, “Zedboard schematic (Rev. F.1),” Oct. 2020. Available: https://digilent.com/reference/_media/reference/programmable-logic/zedboard/zedboard-schematic-rev-e1-public.pdf

This page is empty by intention.