Real-Time Audio Processor

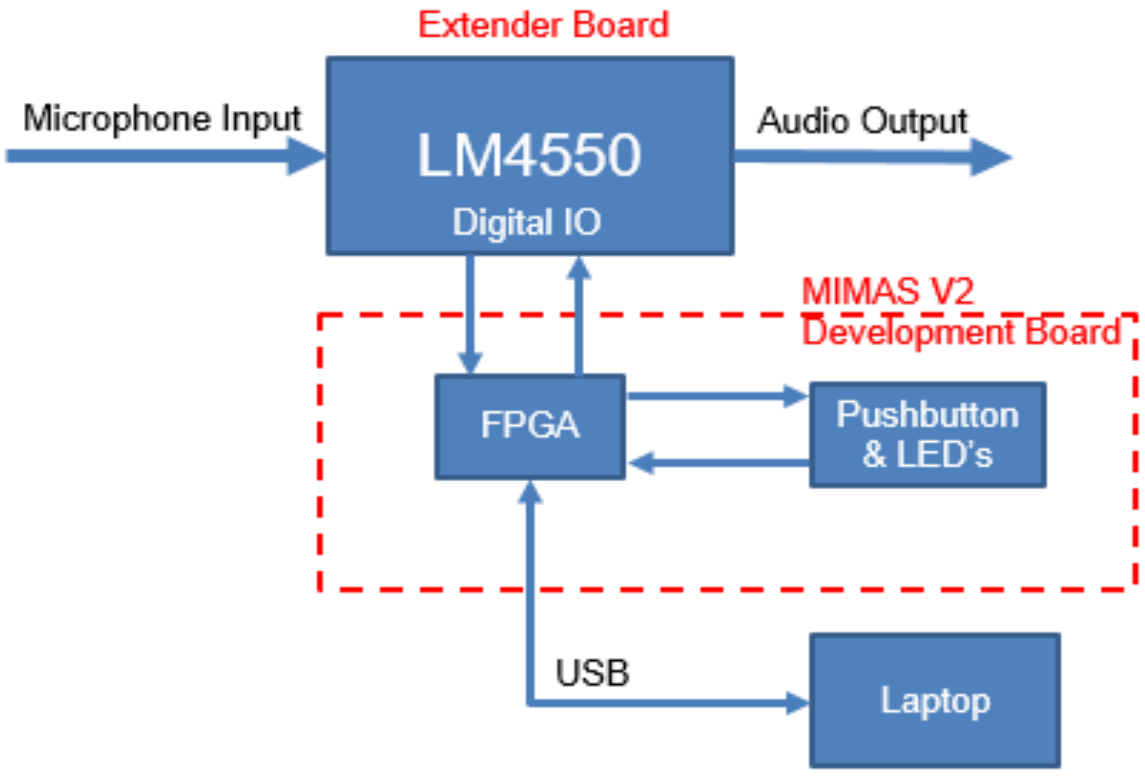
Robert Sikora

EEO441 Final Report

Stony Brook University – Spring 2017

Project Advisor – Dr. David Westerfeld

Course Instructor – Dr. Tatiana Tchoubar



Abstract

A prototype real-time audio processor was implemented using a commercial development board from Numato Lab. The MimasV2 development board is based on a Spartan-6 field programmable gate array (FPGA). The development board also includes push-buttons, switches, seven-segment displays and LED's and four 12-pin extender ports. Also used in this project, is an AC97 extender board that includes an LM4550 for ADC and DAC input and output of audio. Verilog code from a lab class at MIT was used as a starting point. This code had been written for an MIT development board that used a Virtex2 FPGA and an LM4550. Verilog modules were written to use push buttons in pairs to set the audio volume, filter selection and oscillator frequency. Verilog modules were also written to implement 30 tap FIR filters for low-pass, band-pass and high-pass functions. A delay module was written to implement a simple single-delay echo, single-delay reverberation and multiple-delay reverberation filters. The maximum register array size is 4k for a maximum delay of 85 ms. There is an oscillator module in the MIT code that generates a 750 Hz oscillator. This code was modified so that the frequency was made variable and could be controlled by a pair of push-buttons. This oscillator is then used for Ring Modulation effect where the input audio is multiplied by this oscillator. For simplicity only single channel 8-bit data was transferred to and from the LM4550. Each of these filters was tested either by looking at the effect on the spectrum of a periodic pulse, or by time-domain response to a single pulse using an oscilloscope. All the filters functioned roughly as expected.

Table of Contents

Section 1: Goals and Impacts

1.0) Project Goal:

The goal of this project is to build a programmable real-time audio processor using a commercially available field programmable gate array (FPGA) development board. Filters are written in Verilog and loaded into the FPGA. The result of the project is the Verilog Code that is needed to use a development board for audio processing. Anyone wishing to duplicate this project would need to purchase an FPGA development board, a 7.5V power supply and an LM4550 extender board. This project will include a set of instructions on how to connect the development board, extender board and power supply. It will also include the UCF file that connects the FPGA to the hardware on the development and extender boards. Verilog code written in the execution of this project provides various filters which can be copied and edited to make other filters.

1.1) Description of hardware

The development board is a MimasV2 from Numato Labs that includes a Spartan-6 FPGA and a number of push buttons, switches, seven-segment displays and LED's. Attached to the development board is an extender board that includes an LM4550 for audio input and output. DC power is provided by a single power supply (7.5V) to the development and extender boards.

1.2) Source of Verilog Code

There is a lab from MIT, available on the web, that laid out instructions for writing the Verilog code used to make a digital audio recorder and a digital filter [1]. The development board used for this lab was made by the people at MIT and it used a Virtex2 FPGA and a few other components including an LM4550 for audio input and output. The MIT code was modified for this project to suit the newer Spartan-6 FPGA and development board layout.

The UCF file of the MIT project was adapted for the Spartan-6 and the extender board of the Numato Lab hardware.

1.3) Carbon footprint of Board

To determine the carbon footprint of this project, the following data was used: A round trip from NYC to Europe or San Francisco creates two to three tons of carbon dioxide per person [2]. That's a distance of roughly 6,000 miles. The Numato Lab board is manufactured and shipped from India. The distance from Mumbai, India to New York is about 12,500 miles. For this distance, it's roughly 5 tons for the total trip. That's 10,000 lbs CO2 per person. Divide this by the average weight of a human being which is about 170 lbs. That's roughly 60lbs of CO2 per lb of cargo. The weight of the FPGA and the LM4550 combined is half a pound, making the carbon footprint 30 lbs.

1.4) Environmental Impact

Question: Does manufacturing aboard in India generate more or less pollution than the same board generated in the United States?

The Government of India Ministry of Environment and Forests categorized the industry sectors into four different categories: red, orange, green and white with red being most polluting and white being least polluting. Electronic and electrical was put into the white category. On the other hand, the e-waste disposal is a major problem and a major source of pollution, put in the red category [3]. So, as long as the board isn't thrown away, there shouldn't be an issue regarding waste pollution. The impact of a PC board manufacturer in India or in the US are probably similar and of low environmental impact.

If the board from India is compared to an American made board from Digilent (Washington), both Numato Labs and Digilent are getting their components from similar sources. If there is a difference in the pollution generated by the manufacturer of the board, it's probably with the manufacturer of the PCB itself, board layout and etching, population and soldering [4].

1.5) Economic Impact

The Numato Labs Development Board is designed and manufactured in Bangalore, India, as opposed to for example the Digilent Board which is designed and manufactured in Pullman, Washington. So, my project is supporting a company in India rather than one in the United States. The decision to buy hardware from Numato Labs was mostly based on the cost of the board. Although the Digilent Board has numerous additional features, such as onboard Ethernet connection, etc. the cost is not worth the additional features for this project.

On the other hand, buying from a company India might be preferable to buying from a company from China for example. “By comparison, India offers a stable democracy and low wages. Even skilled factory workers here in Pune, sometimes called the Detroit of India, earn about $300 a month, half of Chinese wages” [5].

1.6) Social Impact

The hardware was selected to ensure affordability. This project is primarily firmware can easily be duplicated by other students and hobbyists. Students can hear the result of applying audio filters. There is also potential for an internet community of users on a website, such as GitHub, to be duplicated and or improved on.

Section 2: Background

2.1) Survey

Digital filters are found in just about every commercial audio device from car stereos to flat screen TV's. Understanding how digital filters work is important for any student.

- Commercial Hardware

  The hardware is a Xilinx Spartan-6 FPGA on a development board from Numato Lab along with an extender board with an LM4550 also from Numato Lab. The cost of the development board was $50 and the Extender Board was $30. Digilent has a development board that also uses a Spartan-6 FPGA and an LM4550. The cost is $490. The Digilent board was used in a similar senior project at CalPoly [6].

- What other undergraduates are doing

  There are many students who took on the task of audio processing in their senior projects. Many students seem to be using commercial boards, most often the board by Digilent. I haven't come across any senior projects using an FPGA Board from Numato Lab, which is less expensive than the Digilent by far.

  There are many universities that include programming an FPGA as part of a lab class. These includes MIT [1], SUNY Buffalo [7], and Kansas University [8]. As part of the lab, students are asked to make a digital filter or a simple recording device. They also typically have a development board that was made specifically by the staff at the university rather than a commercial board.

- Who else is working on audio digital filters

  In sound and reproduction, equalization is used to alter the frequency response of an audio system using filters. Most home audio equipment use filters to make bass and treble adjustments. For example, the TEAC Equalizer, has ten different frequency bands. The volume can be adjusted by +/- 12dB [9]. Another example is the Yahama EMX312 Power Mixer, used for live performance. It has seven different equalizer bands where the volume can be adjusted by +/- 12 dB. In addition, it has 16 different digital effects including reverb, echo, chorus, flanger, phaser, and distortion [10].

Fig. 2.1.1: Teac Equalizer (EQA-220) [9]

Control Range: ± 12dB
Frequency Bands: 30, 60, 125, 250, 500, 1K, 2K, 4K, 8K, 16KHz

Fig. 2.1.2: Live music amplifier YAMAHA emx312 [10]

7-band graphic EQs for main and monitor power channels: 125Hz, 250Hz, 500 Hz, 1kHz, 2kHz, 4kHz, 8 kHz, +12/-12 dB

2.2) Project Planning

- Understand hardware connections and specifications.

  The FPGA development board requires a USB connector to transfer the compiled Verilog code to the FPGA. The LM4550 on the extender board requires 3.3 V for the digital section of the chip and requires a separate supply of at least 4.5V for the analog section of the chip. There are separate voltage regulators for each of these voltages. But the regulator has a voltage drop of 3 V. Therefore a 7.5 V supply is needed to get the correct voltage. A selectable voltage power supply was used and 7.5 V was selected.

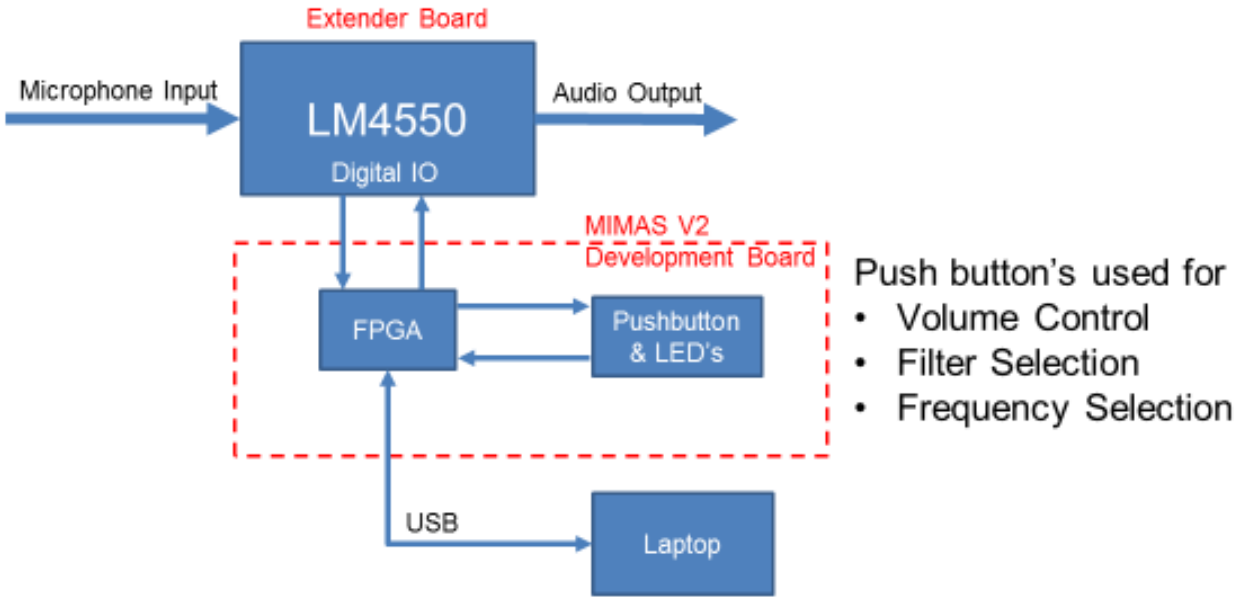


Fig. 2.2.1: Block diagram of the MimasV2 and the connections to the computer.

- Use Xilinx 14.7 ISE

  The Xilinx 14.7 ISE programming environment was used in a Digital Logic class EEO219. After that, it was a matter of looking at the code from MIT and understanding how its program worked. Some time was spent using the Xilinx 14.7 simulation tools and running a test bench to understand exactly what the code was doing. Numato Lab provides an application for the MimasV2 that allows the Verilog binary code to be downloaded into the FPGA.

- Find documents published by Xilinx.
  Xilinx, has a user's guide and has lots of documentation on how to use an FPGA and things that are tailored specifically to what the Spartan-6 can do [11].

- Adapt the code for the hardware.

  The MIT code used a Virtex2 FPGA and a 27MHz Clock. The connection to the audio chip was conFig.ured differently in the UCF file. So the UCF file of the MIT project was modified and adapted for the Spartan-6 and the extender board. The MIT code did most of the hard work of communicating with the LM4550 once the clock and pins had been redefined.

- Create filters

  With the FPGA connected to the audio extender board, and the code modified to get the digital audio to pass through the FPGA, it is possible to begin creating digital filters. The outline of the filter was suggested by the MIT Lab used for this project, namely a finite impulse response filter (FIR) with 31 coefficients. Making use of the fact that the audio sampling rate is 48kHz and the clock rate of the FPGA is 100MHz, there are about 2000 clock cycles to do filter calculations.

  An FIR filter requires creating an array of previous samples. Each previous sample is multiplied by a coefficient to produce the output. In the Echo Filter, most of the previous samples are ignored except for a sample after a fixed (long) delay. A previous sample is added to the current sample. For example, an array of 1000 samples will give a delay of 20ms. In this project, the maximum delay is 4095 samples. The Reverberation Filter also requires an array of large array of previous samples. In the Ring Modulation filter, the incoming audio is multiplied by a sine wave. The effects here become more distorted compared to the other filters because the output contains sum and difference frequencies.

  I used a pair of push buttons to change the value of a register called filtnum to select the desired filter. A 7-segment display is used to display the filter number.

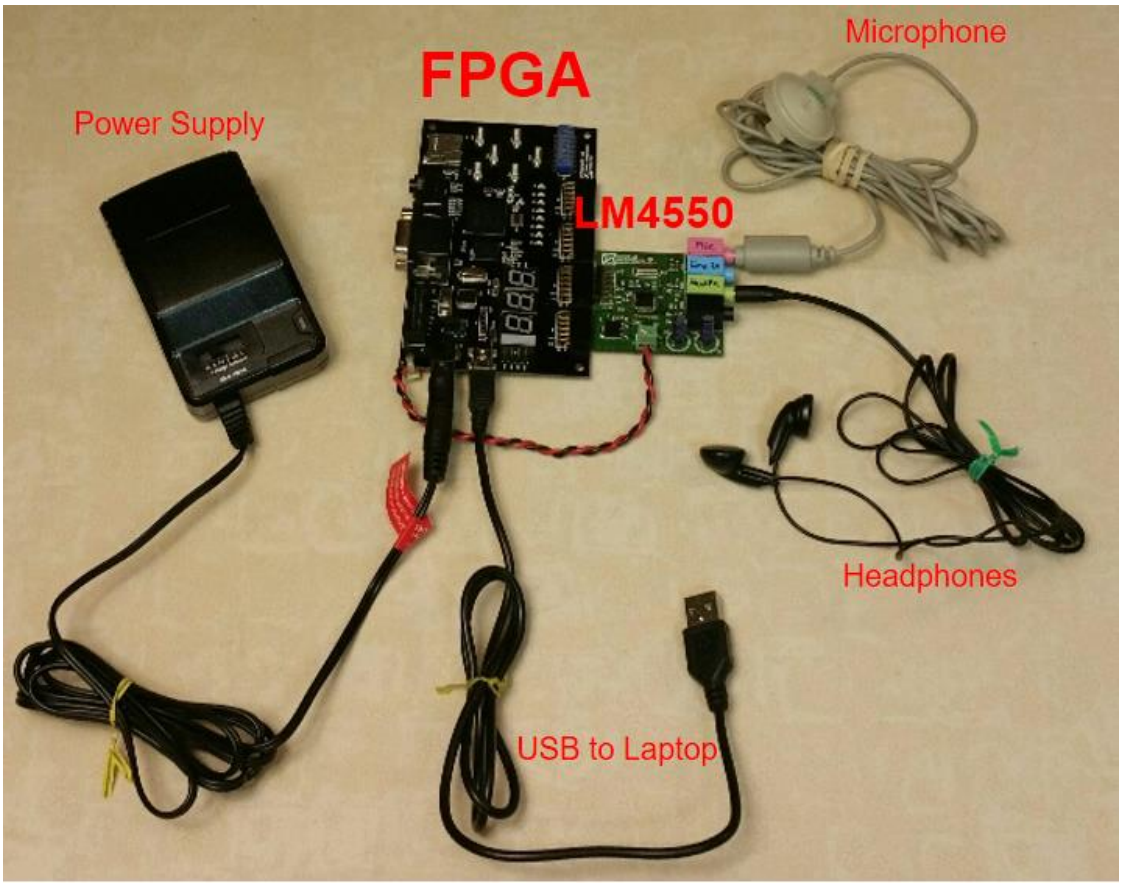


Fig. 2.2.2: The project hardware.

Section 3: System Design

3.1) Design Constraints:

The hardware included in the project should be capable of audio processing. The project needs analog to digital conversion as well as digital to analog conversion, each at 48kHz. It should be able take samples and do a fair number of calculations on the digital samples before getting sent back out into analog. Digital operations include multiplication and addition. An FPGA can do multiplication, addition and other operations each in a single clock cycle and the clock frequency can be hundreds of MHz. So an FPGA with appropriate ADC and DAC seems to be a good choice. Also, in order to do digital filtering, the hardware would have to have either access to external memory or sufficient internal memory.

It would also be helpful to choose an FPGA that already has a development board to avoid having to lay out and build a new board, which would have been a separate project in itself. It is also important to choose an FPGA that has a freely available programming environment to avoid license fees. The Xilinx 14.7 ISE is an FPGA programming environment that is used in the Digital Logic class EEO219. So the FPGA should be selected so that it is compatible with Xilinx 14.7 ISE.

In surveying the development boards available, the Numato Lab MimasV2 seemed suitable because it uses a Xilinx Spartan-6. The MimasV2 development board has connected to the Spartan-6 switches, buttons, LED's, etc. and it has an audio extender board. The Spartan-6 has a clock rate of 100MHz and registers that can be used as memory. The development board also has connections for an additional external memory. The AC97 Audio Extender Board uses an LM4550 which is compatible with AC97 Codec Standard.

3.2) Designs Considered

There were some considerations of which development board to use, such as the MimasV2 that was $50 and the Digilent Atlys that was $500, both of which use a Spartan-6. While the Digilent had more capabilities such as an Ethernet Port and more seven segment LED's on it, they weren't necessary for this project. The MimasV2 with the analog extender board met all the requirements for the project. Most of the design was working with firmware.

3.3) Final Design

The firmware was based on code written for a development board at MIT where for example they would select between straight-through audio and filtered audio by pushing a single button to toggle between the two states. Since the plan for this project includes seven or more filters in this design, pairs of buttons would be used for selecting the active filter.

Volume up/down buttons were already written in the MIT code, so this technique was adapted to make the filter selector. On the MimasV2 there is a total of six push buttons. They are used in pairs, two of volume up/down, two for filter selection and two for frequency selection of the ring modulator oscillator.

Only one channel of audio is used so that the left and right outputs are identical. Although, the audio extender board is capable of 20-bit input and output, only 8-bit data is used.

In the final design, the following filters have been implemented:

- Straight through
- FIR Low Pass Filter (30 coefficients)
- FIR Band-Pass
- FIR High-Pass
- Echo, repeating input audio after a delay
- Reverberation with a single delay
- Reverberation with multiple delays
- Ring Modulation with an adjustable frequency sine wave.

In creating the **FIR filters**, the MATLAB/ Octave program was used to generate the coefficients using the function fir1(). The function freqz() generates the plots such as in Fig. 3.1 -3.3. The coefficients for the FIR filters are multiplied by 1024 so that the FPGA can use integer arithmetic. The results are shown in Table 3.1.

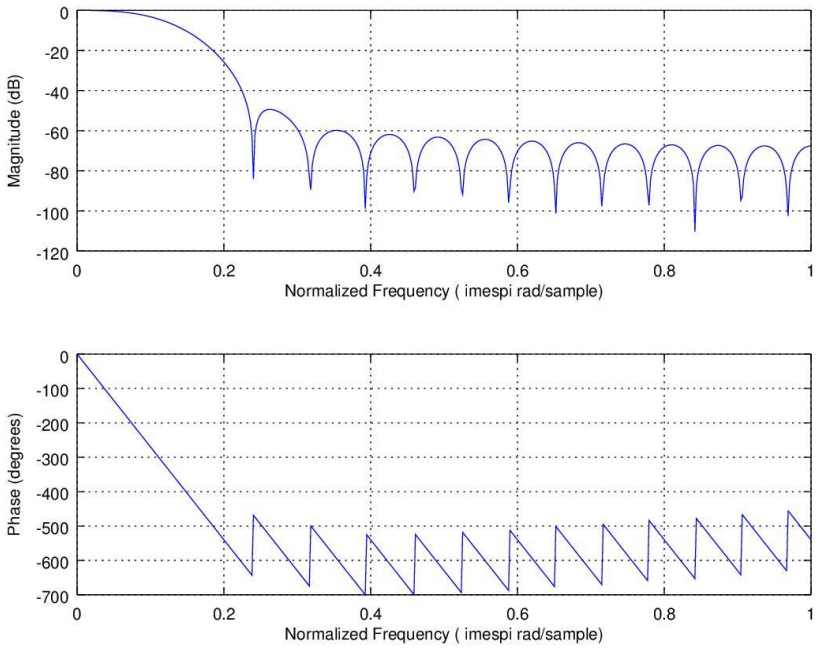


Fig. 3.1: FIR Low-Pass Filter Frequency Response. fir1 (30, 0.125). The frequency 1 corresponds to the Nyquist frequency. Plot generated in Octave.

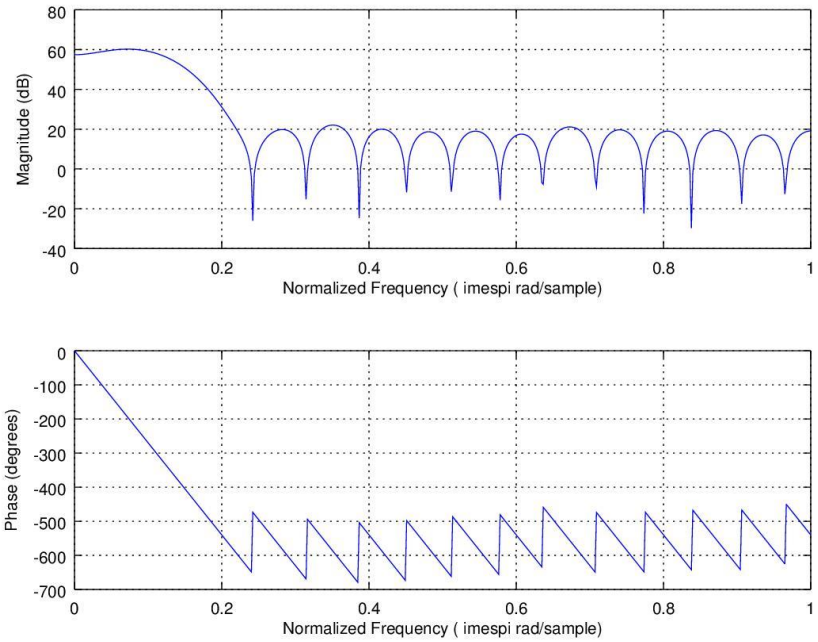


Fig. 3.2: FIR Band-Pass Filter Frequency Response. fir1 (30, [0.05,0.1]). The frequency 1 corresponds to the Nyquist frequency. Plot generated in Octave.

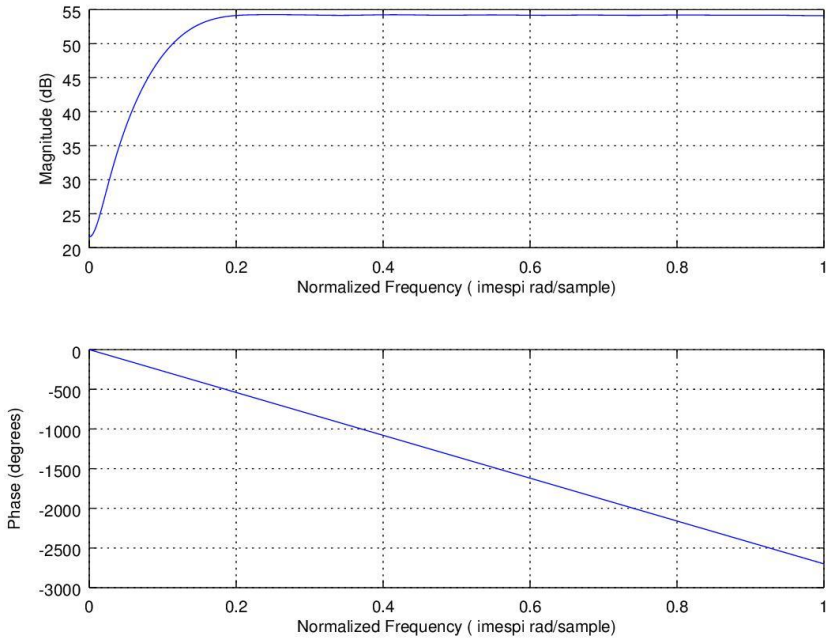


Fig. 3.3: FIR High-Pass Filter Frequency Response. fir1 (30, 0.1, ”high”). The frequency 1 corresponds to the Nyquist frequency. Plot generated in Octave.

Coefficient #	low-pass filter fir1(30,0.125) x1024	band-pass filter fir1(30,[0.05,0.1]) x1024	high-pass filter fir1(30,0.1,'high') x512
1	-1	-8	1
2	-2	-9	1
3	-3	-13	1
4	-5	-18	1
5	-6	-22	1
6	-7	-25	0
7	-5	-24	-2
8	0	-17	-6
9	11	-4	-11
10	27	15	-18
11	47	39	-25
12	69	65	-33
13	91	90	-40
14	110	112	-46
15	122	126	-49
16	127	131	462
17	122	126	-49
18	110	112	-46
19	91	90	-40
20	69	65	-33
21	47	39	-25
22	27	15	-18
23	11	-4	-11
24	0	-17	-6
25	-5	-24	-2
26	-7	-25	0
27	-6	-22	1
28	-5	-18	1
29	-3	-13	1
30	-2	-9	1
31	-1	-8	1

Table 3.1: Shows the 31 coefficients for low-pass, band-pass and high-pass filters using the Octave function fir1().

The echo filter requires a long memory so that a time delay is possible. The *input* data samples are stored in a 4 k memory block so that the current sample can be combined with a sample from the past. More than one delay can be used. Any delay can be choosen up 4,095 sample periods long. At a 48 kHz sample rate, the total possible delay is about 85ms.

Reverberation is similar to the echo except that the *output* samples are stored in the memory block so that they can be combined with the present input sample.

In **Ring Modulation**, the incoming samples are multiplied by a sine wave at frequency ω_0. For any incoming frequency ω_n, the result of the multiplication will be the following two frequencies: ω_0 +/- ω_n.

$$sin(\omega_n t) * sin(\omega_0 t) = \frac{1}{2} * (\cos[(\omega_0 - \omega_n)t] - \cos[(\omega_0 + \omega_n)t])$$

To make the modulating sine wave ω_0, the code from MIT has a look-up table that contains 64 values making a sine wave at a fixed frequency. For this project, the frequency was made variable and controlled using the push buttons.

In implementing the echo and reverberation, there were unexpected limitations in the size of the RAM that was available. The compiler did not allow a size greater than 4K.

Section 4: System Implementation and Testing

4.1) Implementation Problems:

- Power Supply

  The first problem discovered was that there was no output from the extender board. Examining the different pins between the extender board and the development board, it was found that the bit-clock from the LM4550 was not present. It was then realized that a separate power supply connection was needed for the analog portion of the LM4550. The LM4550 had separate digital and analog power connections. The digital power is 3.3 V and the required analog voltage is 5 V. Next it was realized that on the extender board, there is a LM7805 5 V regulator. This voltage regulator needs about 7.5 V at its input to produce a 5 V output. Once the 7.5 V power supply was connected, the bit-clock was present and there was finally a sound output.

- Filter

  To get the filter to work, two things need to be controlled: an index into a 32-element register array with samples arriving at 48 kHz and in between each sample, calculations need to be executed based on the information in the array. So a test bench was made to understand how the filter was going to work. The ISE Xilinx 14.7 simulation lets you see what is happening on each clock cycle. This was very helpful.

  It turns out that when the test filter was moved to the main Verilog file and implemented in hardware, there was a bug in how this was done where the index to the array incremented many times between each audio sample. This gave a noisy and distorted output. This was fixed by ensuring that the filter output would output a value and the index would increment exactly once every sample.

  When an 8-bit register array is declared in Verilog, the compiler does not allow an array size greater than 4,096. The data sheet for the Spartan-6 claims that it has 32 dual-port block RAM's each with 2000 8-bit memories. This discrepancy was not resolved since the 4K was enough to get some sort of noticeable form of reverberation.

  Incidentally, I must have installed the ISE 14.7 Xilinx Development program incorrectly because when I would run the program and try to open another file, the program would crash. It seemed that it would crash whenever it would look for a different file on the system. I just worked around this by starting the program by opening the file that I was

working on. Since I was only working on one file that contained all of the modules, this didn't pose as too much of an issue.

4.2) Final Implementation

- Hardware Setup

  Connect the extender board to the development board through the 12-pin connection at P7. Obtain a power supply that can deliver 7.5 V that is used to power the system with a center positive 2.1mm barrel connector. On the bottom of the MimasV2 board, solder two wires, one to ground, the other to the power-supply voltage connected on J1. Route these two wires to the EXT VIN of the extender board with positive going to the positive terminal and ground going to the negative terminal. The microphone is connected to the pink jack of the extender board and the headphones to the green jack.

  For programming, the development board is connected to the computer using a USB cable. There is a switch on the development board SW7. For programming, this switch should be positioned towards the USB connector.

- Programming

  HDL code was used to synthesize and implement the filters using the ISE 14.7 program from Xilinx. In the "generate programming file" command, the creation of a binary configuration file needs to be the selected option. Once this .bin file has been created, it is loaded into the FPGA flash memory using a program provided Numato Labs called the MimasV2 Configuration Tool. The MimasV2 User Manual has detailed instructions on how this is done.

- Verilog Code

  1. Communication with the LM4550 Extender Board

     There is code available on the web for a development board that had been made for the 6.111 Lab at MIT. The Verilog code for this board has a lot of very useful modules that were adapted for this project. The most important of these were the modules that established communication between the FPGA and the LM4550. Adapting this code for the Numato Lab hardware required only a few modifications, mostly in changing the clock and the pins used on the FPGA. Most of this work was done by modifying the User Constraints File (UCF). The UCF file was modified assuming that the P7 connector on the MimasV2 board would be used and

by matching the functionality of the LM4550 to connect to the appropriate pins of the FPGA. There are five signals that need to be specified the UCF file as seen in the table below.

Name New	Name Original	"LOC"	Pin on MimasV2	Pin on Expansion	Function
ac97_sdata_out	"IO_P7[6]"	V8	2	1	SDO
	"IO_P7[7]"	U8	1	2	-
ac97_bit_clock	"IO_P7[4]"	T8	4	3	BIT-CLK
ac97_sdata_in	"IO_P7[5]"	R8	3	4	SDI
ac97_synch	"IO_P7[2]"	T5	6	5	SYNC
audio_reset_b	"IO_P7[3]"	R5	5	6	RESET
	"IO_P7[0]"	V9	8	7	-
	"IO_P7[1]"	T9	7	8	-
		NA	10	9	GND
		NA	9	10	GND
		NA	12	11	VCC3V3
		NA	11	12	VCC3V3

Table 4.1: Table of signals connecting to the LM4550 and their names in the UCF File.

Communication with the LM4550 is bidirectional—there are separate serial-in and serial-out lines, as well as a common bit clock. With a sample rate 48 kHz, there are 256 bits transferred for each sample. The frequency of the bit-clock is 256 times 48 kHz which is 12.288 MHz. The bit stream contains 20 bits of data plus numerous commands that are used to configure the LM4550. For example, whether to use the microphone or line input, whether to turn on or off 12 dB of gain to the microphone input. The datasheet for the LM4550 has a table showing which bits to use for setting various parameters.

There is an MIT Lab module called *lab4audio* that creates instances of the *ac97* and *ac97commands* modules and extracts from them, 8 bits of audio to and from the LM4550. The MIT Lab used only 8 bits of data because their lab involved making a digital recorder. For this project, 8-bit data from the *lab4audio* module is used because it takes less space for the memory operations that are involved in this project. The *lab4audio* module in return is contained within the top-level module *lab4*. All the inputs and outputs of the FPGA go through the *lab4* module. Within this top-level module is an instance of the *lab4audio* module described previously. There are also instances of *debounce* modules that are used to connect to push-buttons on the board used for controlling the volume, filter selection, and setting the frequency selection of an oscillator.

2. Top Level Module lab4()

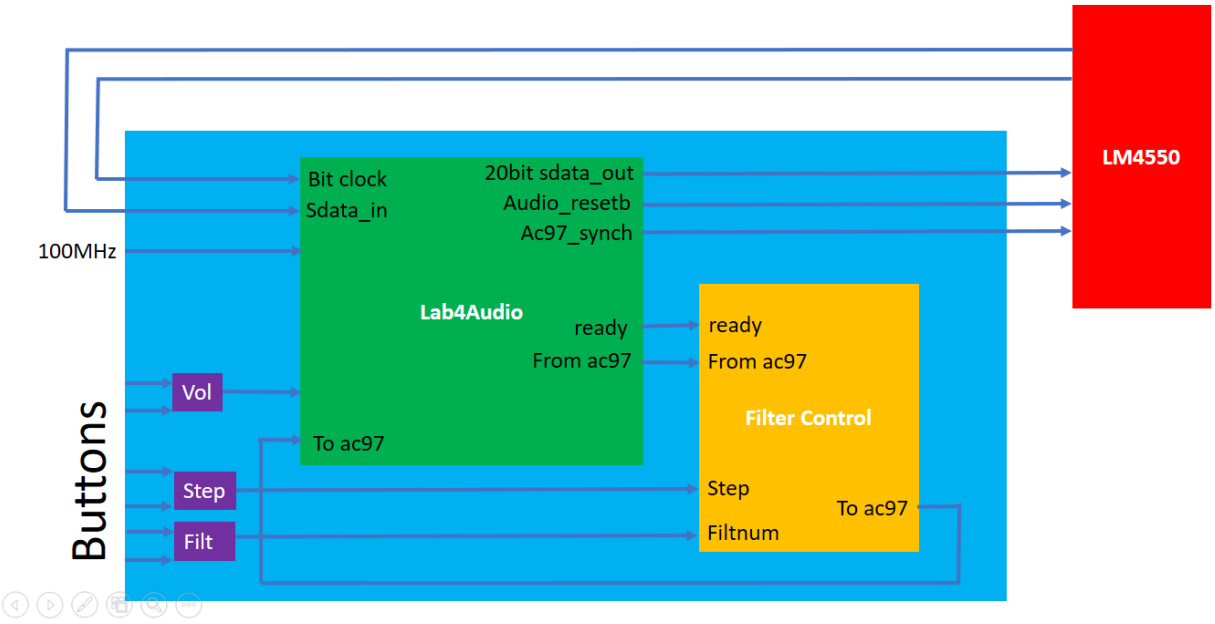


Fig. 4.2.1: Block Diagram of the top-level module.

Contained in the top-level module is an instance of the *FilterControl* module. This is the module that contains most of the work involved in this project. The top-level module also contains the *debounce* circuit. This is shown in Figure 4.2.1. The code to realize this functionality is given below.

```
//////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes, 6.111 staff
//
// Modified by Robert Sikora May 2017
// to be compatible with Numato Lab MimasV2
//
//////////////////////////////////////////////////////////////////////////////

module lab4(
  // Remove comment from any signals you use in your design!

  // AC97
  output wire /*beep,*/ audio_reset_b, ac97_synch, ac97_sdata_out,
  input wire ac97_bit_clock, ac97_sdata_in,

  // BUTTONS, SWITCHES, LEDS
  input wire vol_button_down,
```

```
  input wire vol_button_up,
  input wire filt_down,
  input wire filt_up,
  input wire step_button_down,
  input wire step_button_up,
  output wire [7:0] led,
  output wire [7:0] SevenSegment, //FD
  output wire [2:0] SevenSegmentEnable,
  output wire [7:1] IO_P9,

  // CLOCKS
  input wire CLK_100MHz
);
   //////////////////////////////////////////////////////////////////////////
   //
   // Reset Generation
   //
   // A shift register primitive is used to generate an active-high reset
   // signal that remains high for 16 clock cycles after configuration finishes
   // and the FPGA's internal clocks begin toggling.
   //
   //////////////////////////////////////////////////////////////////////////
   wire reset;
   SRL16 #(.INIT(16'hFFFF)) reset_sr(.D(1'b0), .CLK(CLK_100MHz), .Q(reset),
                                     .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));

   wire [7:0] from_ac97_data, to_ac97_data;
   wire ready;

   // AC97 driver
   lab4audio a(CLK_100MHz, reset, volume, from_ac97_data, to_ac97_data, ready,
               audio_reset_b, ac97_sdata_out, ac97_sdata_in,
               ac97_synch, ac97_bit_clock);

   // record module
   FilterControlModule r(.clock(CLK_100MHz), .reset(reset), .ready(ready),
               .step(step), .filtnum(filtnum),
               .from_ac97_data(from_ac97_data),.to_ac97_data(to_ac97_data));


   //Send signals to the P9 connector

   assign IO_P9[7] = ac97_bit_clock; // Pin 1
   assign IO_P9[6] = audio_reset_b;  // Pin 2
   assign IO_P9[5] = ac97_sdata_out; // Pin 3
   assign IO_P9[4]= ac97_sdata_in; // Pin 4
   assign IO_P9[3] = ac97_synch; // Pin 5
   assign IO_P9[2] = reset;                          // pin 6
   assign IO_P9[1] = CLK_100MHz;                     // pin 7

   // allow user to adjust volume
   wire vup,vdown;
   reg old_vup,old_vdown;
   debounce bup(.reset(reset),.clock(CLK_100MHz),.noisy(~vol_button_up),.clean(vup));
   debounce bdown(.reset(reset),.clock(CLK_100MHz),.noisy(~vol_button_down),.clean(vdown));
   reg [4:0] volume;
   always @ (posedge CLK_100MHz) begin
     if (reset) volume <= 5'd20;
     else begin
        if (vup & ~old_vup & volume != 5'd31) volume <= volume+5'd1;
        if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-5'd1;
     end
     old_vup <= vup;
     old_vdown <= vdown;
   end
```

```
   // allow user to change the filter number
   wire fup,fdown;
   reg old_fup,old_fdown;
   debounce filtup(.reset(reset),.clock(CLK_100MHz),.noisy(~filt_up),.clean(fup));
   debounce filtdown(.reset(reset),.clock(CLK_100MHz),.noisy(~filt_down),.clean(fdown));
   reg [3:0] filtnum;
   always @ (posedge CLK_100MHz) begin
     if (reset) filtnum <= 4'd0;
     else begin
        if (fup & ~old_fup & filtnum != 4'd15) filtnum <= filtnum+4'h1;
        if (fdown & ~old_fdown & filtnum != 4'd0) filtnum <= filtnum-4'h1;
     end
     old_fup <= fup;
     old_fdown <= fdown;
   end

           FilterDisplay
fdisplay(.filtnum(filtnum),.SevenSegment(SevenSegment),.SevenSegmentEnable(SevenSegmentEnable));

//   allow user to select step (frequency) up/down using buttons 4 (up) and 3 (down)
   wire sup,sdown;
   reg old_sup,old_sdown;

   debounce sbup(.reset(reset),.clock(CLK_100MHz),.noisy(~step_button_up),.clean(sup));
   debounce sbdown(.reset(reset),.clock(CLK_100MHz),.noisy(~step_button_down),.clean(sdown));

   reg [15:0] step;
   always @ (posedge CLK_100MHz) begin

     if (reset)
        step <= 16'h0400;

     else begin
        if (sup & ~old_sup & step != 16'hFFFF)
           step <= step + 16'h0F;
        if (sdown & ~old_sdown & step != 16'h0)
           step <= step - 16'h0F;
     end

     old_sup <= sup;
     old_sdown <= sdown;

   end
endmodule
```

3. Filter Selection: FilterControlModule ()

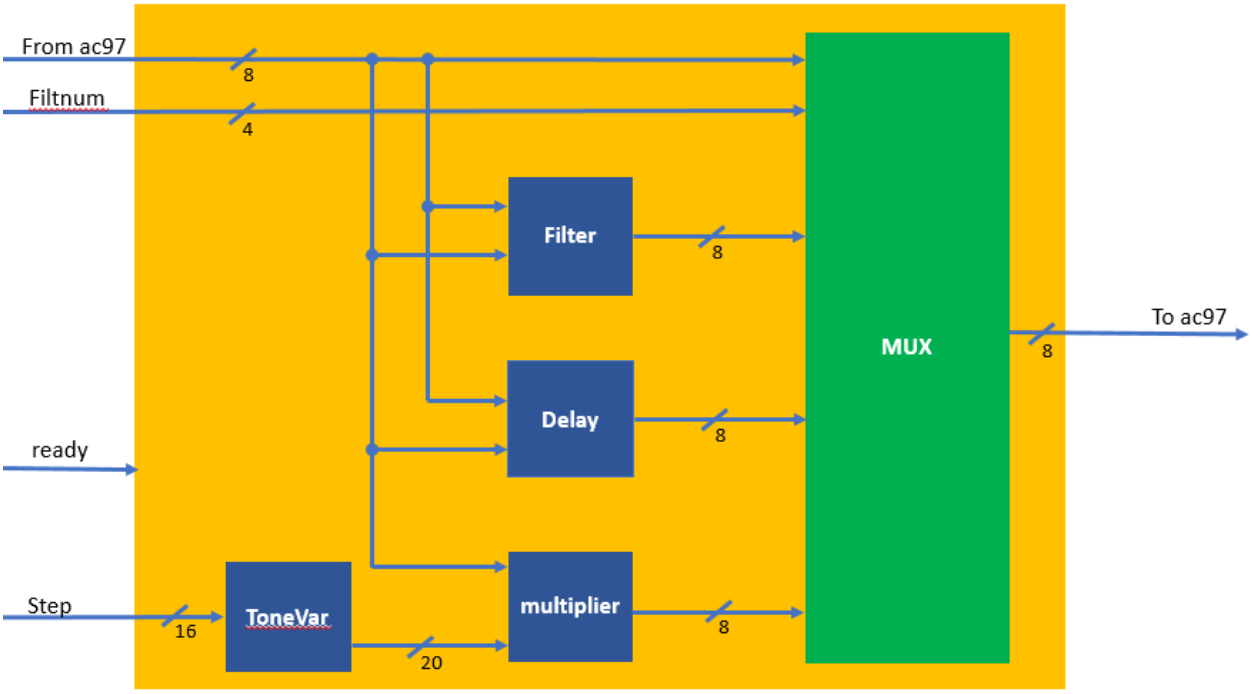


Figure 4.2.2: Block Diagram of *FilterControlModule()*. Ready, Reset and Clock are connected to all the modules within the *FilterControlModule*, but is not shown in this diagram.

The *FilterControlModule* creates instance of the following modules: *fir31 module*, *DelayModule*, *ToneVarModule*, and *MultiplierModule*. The controlling variable filtnum is used in a case statement to determine which module output will be sent to the headphone (to_AC97). Filtnum is also used by the *fir31 module* and *DelayModule* to modify their behavior, as will be described in later subsections. The variable Step is sent to the ToneVar module to determine what frequency sine wave is output by that module. The code for the *FilterControlModule* is given below.

```
//////////////////////////////////////////////////////////////////////////////////
//
// Filter Control Module
//
// Created by Robert Sikora May 2017
// to be compatible with Numato Lab MimasV2
//
//////////////////////////////////////////////////////////////////////////////////

module FilterControlModule(
  input wire clock,               // 27mhz system clock
  input wire reset,                 // 1 to reset to initial state
  input wire ready,                 // 1 when AC97 data is available
  input wire [15:0]step,              // 1 for playback, 0 for record
  input wire [3:0] filtnum,         // 1 when using low-pass filter
  input wire [7:0] from_ac97_data, // 8-bit PCM data from mic
```

```
  output reg [7:0] to_ac97_data    // 8-bit PCM data to headphone
);
   // test: playback 750hz tone, or loopback using incoming data
   wire [19:0] tone;
   tone750hz xxx(.clock(clock),.ready(ready),.pcm_data(tone));

   wire [17:0] filtered;
   fir31 fir(.clock(clock),.reset(reset),.ready(ready),.x(from_ac97_data),.y(filtered),
             .filtnum(filtnum));

   wire [19:0] tone_2;
   ToneVarModule tone02(.clock(clock),.ready(ready),.step(step),.pcm_data(tone_2));

   wire signed [15:0] mult;
   MultiplierModule mul(.clock(clock), .ready(ready), .x(from_ac97_data),.y(tone_2[19:12]),
                        .prod (mult));

   wire signed [17:0] delayed;
   DelayModule del(.clock(clock),.reset(reset),.ready(ready),.x(from_ac97_data), .y(delayed),
                   .filtnum(filtnum));

   always @ (posedge clock) begin
      if (ready) begin
         case (filtnum[3:0])
            4'h0:  to_ac97_data <= from_ac97_data;
            4'h1:  to_ac97_data <= filtered[17:10]; // Low Pass #1
            4'h2:  to_ac97_data <= filtered[17:10]; // Band Pass #2
            4'h3:  to_ac97_data <= filtered[17:10]; // High Pass #3
            4'h4:  to_ac97_data <= tone[19:12];
            4'h5:  to_ac97_data <= tone_2[19:12];
            4'h6:  to_ac97_data <= delayed[17:10];   // echo with filtnum = 6
            4'h7:  to_ac97_data <= delayed[17:10];   // reverb single delay with filtnum = 7
            4'h8:  to_ac97_data <= delayed[17:10];   // reverb 4 delays with filtnum = 8
            4'h9:  to_ac97_data <= mult[15:8];
           default: to_ac97_data <= from_ac97_data;
         endcase

      end
   end
endmodule
```

4. Filter Implementation

The *fir31 module* implements a 30th order FIR filter as described in Section 2. The sample array contains the current sample, plus 31 previous samples. As each audio sample is ready, the variable offset is incremented to give a new location in the array. Since the offset is five bits, it will cycle repeatedly from 0 to 31, but it only increments once on each sample. When the new data is ready, the variable index is initialized to 0 and on each clock cycle (100 MHz), index is incremented so that it takes values from 0 to 30. With each value of index, a coefficient is obtained from the *coeffs31* module and is multiplied times the relevant previous sample and the result is added to the accumulator. When all of these calculations are done, the result is loaded into the output register output_reg. The code for obtaining the coefficient was part of the original MIT code. This code was modified so that depending on the value of filtnum, a different set of coefficients would be used. The code for *fir31 module*, is given below.

```
//////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- fir31 module
//
// For Labkit Revision 004
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes, 6.111 staff
//
// Modified by Robert Sikora May 2017
// to make a functioning filter.
//
// 31-tap FIR filter, 8-bit signed data, 10-bit signed coefficients.
// ready is asserted whenever there is a new sample on the X input,
// the Y output should also be sampled at the same time.  Assumes at
// least 32 clocks between ready assertions.  Note that since the
// coefficients have been scaled by 2**10, so has the output (it's
// expanded from 8 bits to 18 bits).  To get an 8-bit result from the
// filter just divide by 2**10, ie, use Y[17:10].
//
//////////////////////////////////////////////////////////////////////////////

module fir31(
  input wire clock,reset,ready,
  input wire signed [7:0] x,
  output wire signed [17:0] y,
  input wire [3:0] filtnum
);

  reg [4:0] index;
  reg [4:0] offset;
  reg signed [7:0] sample [0:31];
  reg signed [17:0] accum;
  reg signed [17:0] output_reg;
  wire signed [9:0] coeff;
  reg done;

  assign y = output_reg;

  initial begin

     index = 5'h0;
     accum <= 18'd0;
     output_reg <= 5'h0;

  end

  always @(posedge clock) begin   // could this increment more than once on each "ready" ?
     if (ready) begin
        offset <= offset + 5'h1;
     end
  end

  coeffs31 coe(filtnum, index,coeff);

  always @(posedge clock) begin
    if(ready) begin
       sample[offset] <= x;

       index  <= 0;
       accum  <= 0;
       done   <= 0;
    end
```

```
    if (!ready && (index < 30) ) begin   // calculate when ready is low. (< 30)
       accum <= accum + coeff * sample[offset-index];
       index <= index + 5'h1;
    end
    if (!done && (index == 30) ) begin   // when finished, update output (== 30)
       output_reg <= accum;   // update ouput register, once.
       done <= 1'b1;
    end
  end

endmodule
```

5. DelayModule()

   The delay module uses the ideas of the *fir31 module* -- that is, keeping some number of previous samples for use in calculations. There are two major differences, one is that the array is much larger (4096 elements) and that not all the previous samples are used, only selected previous samples. The delay module was written so that the same sample array could be used to generate a simple echo or a reverberation effect, depending on the value of filtnum.

   For the echo, the present sample is summed with the sample 4000 audio samples ago to give an output that includes the delay of 4000*20.83us = 83 milliseconds. The selection of which previous delay to use implemented through a module called *delay4k*. For the echo effect, only the first value from *delay4k* is used.

   To accomplish reverberation, rather than putting the input samples into the array, the output samples are put into the array. For example, with reverberation of a single delay of 4000, a portion of a given sample will repeat every 4000 audio samples. Filtnum determines whether to use the *DelayModule* to make an echo or a reverberation. The code for the delay module and for the *delay4k* module are given below.

```
//////////////////////////////////////////////////////////////////////////////////
//                        DelayModule()
//
// Created by Robert Sikora May 2017
//
// Stores audio input samples in an 8 bit x 4k long array and will output the
// sum of the set of samples[m] returned by an instance of the module delay4k.
// For example, this could be the most recent sample plus the sample from
// 4000 samples ago. This would give an echo from 1/48kHz * 4000 seconds ago.
// Or about 80 ms.
//////////////////////////////////////////////////////////////////////////////////

module DelayModule(
   input wire clock,reset,ready,
   input wire signed [7:0] x,
   output wire signed [17:0] y,
   input wire [3:0] filtnum
);

reg [11:0] offset;   // offset to location of current audio data in circular buffer.
reg [3:0]   index;    // index to case statement
```

```
reg         done;     //
reg signed [7:0]  sample[0:4096];
reg signed [17:0] accum;
reg signed [17:0] output_reg;
wire       [11:0] delay;
reg        [7:0]  size;
reg signed [9:0]  revamp;

assign y    = output_reg;   // output_reg;

initial begin
   offset    <= 12'h0;
   index     <= 4'h0;
   accum     <= 18'd0;
   output_reg <= 18'd0;
   sample[0] <= 0;
end

// This module demonstrates both a single echo (filtnum = 6)
// and reverberation (filtnum = 7).
// For echo, use only the first value of delay (size = 1)and increase the
// amplitude of that echo so that it is very distinct (revamp = 1FF).
// For reverberation, four delays are used. Since the samples from
// these different delays are all added, the scale factor needs
// to be smaller, otherwise the amplitude could go beyond a valid range. (revamp = 0AF)
//

always @(posedge clock) begin   // could this increment more than once on each "ready" ?
   if (ready) begin
      offset <= offset + 12'h1;// Increment the offset once on each ready (when new audio sample)
         if (filtnum == 6) begin    // Echo
            size   <= 8'd1;
            revamp <= 10'sh1FF;
         end
         else if (filtnum == 7) begin // Reverberation single delay
            size   <= 8'd1;
            revamp <= 10'sh1FF;
         end
         else begin                   // Reverberation multiple delays
            size <= 8'd4;
            revamp <= 10'sh0DF;
         end
      end
   end

   delay4k del(index,delay);

   always @(posedge clock) begin
      if(ready) begin
         index  <= 0;
         accum  <= 10'sh1FF * x;   // 1FF put the current sample into the accumulator
         done   <= 0;
      end
      if (!ready && (index < size) ) begin   // calculate when ready is low. (< 2)
         accum <= accum + revamp * sample[offset - delay];
         index <= index + 4'd1;
      end
      if (!done && (index == size) ) begin         // when finished, update output (== 2)
         if(filtnum == 6 ) begin
            sample[offset] <=   x;               // for echo
         end
         else begin
            sample[offset] <=   accum[17:10];     // for reverberation
         end
         output_reg     <=  accum;
         done <= 1;
```

```
      end
   end
endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// Created by Robert Sikora May 2017
// Return the delay that is used by DelayModule
//
//////////////////////////////////////////////////////////////////////////////////

module delay4k(
  input wire [3:0] index,
  output reg [11:0] delay
);
  // tools will turn this into a 16 x 12bit ROM

  always @(index)
     case (index[3:0])
           4'd0:  delay =  12'd4079;
           4'd1:  delay =  12'd3469;
           4'd2:  delay =  12'd2579;
           4'd3:  delay =  12'd1823;
           4'd4:  delay =  12'd1;
           4'd5:  delay =  12'd1;
           4'd6:  delay =  12'd1;
           4'd7:  delay =  12'd1;
           4'd8:  delay =  12'd1;
           4'd9:  delay =  12'd1;
           4'd10: delay =  12'd1;
           4'd11: delay =  12'd1;
           4'd12: delay =  12'd1;
           4'd13: delay =  12'd1;
           4'd14: delay =  12'd1;
           4'd15: delay =  12'd1;
           default: delay = 12'd1;
      endcase
   end
endmodule
```

6. ToneVarModule

   There is a lookup table of 64 20-bit numbers that describe a sine wave. A 16-bit variable step is used to increment the register index to find the relevant sample from this lookup table. The lowest frequency possible for the sine wave output is the 48 kHz sample rate divided by 64 k which is less than a hertz (~0.75 Hz). Although index is 16 bits, only the most significant 5 bits of index are used to determine the location in the lookup table. The code for the *ToneVar* Module is given below, it was adapted from MIT Lab which had a fixed frequency of 750 Hz.

```
//////////////////////////////////////////////////////////////////////////////////
//
// ToneVarModule()
//
// based on tone750hz module
// For Labkit Revision 004
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes, 6.111 staff
//
// Modified by Robert Sikora May 2017
```

```
//
// generate PCM data for a variable frequency sine wave (assuming f(ready) = 48khz)
// Look up table of 64 values of 20 bits.
// The input value of step (16 bits) determines the frequency.
// Only the upper 6 bits of step are used giving 64 possible values for the location
// in the look up table.
// So a value of step = 1024 would require 64 increments of index for a full cycle.
// The output frequency would be 48 kHz/64 = 750 Hz.
// If step is 1, the frequency would be 1024 times slower or about 0.7 Hz
//////////////////////////////////////////////////////////////////////////////////
module ToneVarModule (
  input wire clock,
  input wire ready,
  input wire [15:0] step,
  output reg [19:0] pcm_data
);
   reg [15:0] index;

   initial begin
      index <= 8'h00;
      // synthesis attribute init of index is "00";
      pcm_data <= 20'h00000;
      // synthesis attribute init of pcm_data is "00000";
   end

   always @(posedge clock) begin
      if (ready) index <= index + step;
   end

   // one cycle of a sinewave in 64 20-bit samples
   always @(index) begin
      case (index[15:10])
        6'h00: pcm_data <= 20'h00000;
        6'h01: pcm_data <= 20'h0C8BD;
        6'h02: pcm_data <= 20'h18F8B;
        6'h03: pcm_data <= 20'h25280;
        6'h04: pcm_data <= 20'h30FBC;
        6'h05: pcm_data <= 20'h3C56B;
        6'h06: pcm_data <= 20'h471CE;
        6'h07: pcm_data <= 20'h5133C;
        6'h08: pcm_data <= 20'h5A827;
        6'h09: pcm_data <= 20'h62F20;
        6'h0A: pcm_data <= 20'h6A6D9;
        6'h0B: pcm_data <= 20'h70E2C;
        6'h0C: pcm_data <= 20'h7641A;
        6'h0D: pcm_data <= 20'h7A7D0;
        6'h0E: pcm_data <= 20'h7D8A5;
        6'h0F: pcm_data <= 20'h7F623;
        6'h10: pcm_data <= 20'h7FFFF;
        6'h11: pcm_data <= 20'h7F623;
        6'h12: pcm_data <= 20'h7D8A5;
        6'h13: pcm_data <= 20'h7A7D0;
        6'h14: pcm_data <= 20'h7641A;
        6'h15: pcm_data <= 20'h70E2C;
        6'h16: pcm_data <= 20'h6A6D9;
        6'h17: pcm_data <= 20'h62F20;
        6'h18: pcm_data <= 20'h5A827;
        6'h19: pcm_data <= 20'h5133C;
        6'h1A: pcm_data <= 20'h471CE;
        6'h1B: pcm_data <= 20'h3C56B;
        6'h1C: pcm_data <= 20'h30FBC;
        6'h1D: pcm_data <= 20'h25280;
        6'h1E: pcm_data <= 20'h18F8B;
        6'h1F: pcm_data <= 20'h0C8BD;
        6'h20: pcm_data <= 20'h00000;
        6'h21: pcm_data <= 20'hF3743;
```

```
        6'h22: pcm_data <= 20'hE7075;
        6'h23: pcm_data <= 20'hDAD80;
        6'h24: pcm_data <= 20'hCF044;
        6'h25: pcm_data <= 20'hC3A95;
        6'h26: pcm_data <= 20'hB8E32;
        6'h27: pcm_data <= 20'hAECC4;
        6'h28: pcm_data <= 20'hA57D9;
        6'h29: pcm_data <= 20'h9D0E0;
        6'h2A: pcm_data <= 20'h95927;
        6'h2B: pcm_data <= 20'h8F1D4;
        6'h2C: pcm_data <= 20'h89BE6;
        6'h2D: pcm_data <= 20'h85830;
        6'h2E: pcm_data <= 20'h8275B;
        6'h2F: pcm_data <= 20'h809DD;
        6'h30: pcm_data <= 20'h80000;
        6'h31: pcm_data <= 20'h809DD;
        6'h32: pcm_data <= 20'h8275B;
        6'h33: pcm_data <= 20'h85830;
        6'h34: pcm_data <= 20'h89BE6;
        6'h35: pcm_data <= 20'h8F1D4;
        6'h36: pcm_data <= 20'h95927;
        6'h37: pcm_data <= 20'h9D0E0;
        6'h38: pcm_data <= 20'hA57D9;
        6'h39: pcm_data <= 20'hAECC4;
        6'h3A: pcm_data <= 20'hB8E32;
        6'h3B: pcm_data <= 20'hC3A95;
        6'h3C: pcm_data <= 20'hCF044;
        6'h3D: pcm_data <= 20'hDAD80;
        6'h3E: pcm_data <= 20'hE7075;
        6'h3F: pcm_data <= 20'hF3743;
      endcase // case(index[5:0])
   end // always @ (index)
endmodule
```

7. Ring Modulation: MultiplierModule()

   Finally, is the *MultiplierModule*. This is by far the simplest module of the project but it gave the most interesting sound output. This module takes two 8-bit input values and multiplies them together to generate a 16-bit output. As described in Section 2, this module was used to multiply the input audio and the sine wave generated by the *ToneVar* module, generating sum and difference frequencies in the output. The code for this module is given below.

```
////////////////////////////////////////////
//      MultiplierModule()
//
//  Multiply the two 8-bit input signals x and y
//  The output is a 16 bit signal called prod
//
//  Created by Robert Sikora May 2017
//
////////////////////////////////////////////////////////////////////////////////

module MultiplierModule(
   input wire clock,
      input wire ready,
      input wire signed [7:0] x,
      input wire signed [7:0] y,
      output reg signed [15:0] prod
```

```
);

   always @(posedge clock) begin
      if(ready)
         prod <= x * y  ;
      end
   end
endmodule
```

4.3) Testing

Initial testing was done with a National Instruments (NI) MYDAQ to measure the output of the audio processor, especially in the frequency domain. The audio output connector of the MYDAQ was used to supply a periodic pulse to the microphone input and the headphone output of the development board was connected to the audio input connector of the MYDAQ.

The spectrum of a periodic pulse is a series of harmonics that repeat at the pulse repetition frequency with an envelope that is determined by the pulse width. The Fourier components of a periodic pulse are given by following equation where T is the width of the pulse train and T1 is the width of the pulse:

$$A_k = \frac{2}{\pi k} \sin\left(\frac{\pi k T_1}{T}\right)$$

The coefficients will go to zero whenever the argument of the sine is a multiple of pi. If the ratio of T1/T is 1/100, then the zero will appear at multiples of k = 100.

To generate a pulse, the MYDAQ arbitrary waveform generator was used. An arbitrary waveform with 100 pts was created where 1 is high with an amplitude of 0.1 V and the other 99 are low. This was output with an update frequency of 40 kHz giving a rectangular pulse with a repetition frequency of 400 Hz. The spectrum was measured by connecting the MYDAQ output to its input and looking at a range from 0 to 40 kHz confirming that the first zero is at about 40 kHz. This should be a reasonable spectrum for testing the FPGA circuit since it covers more than the audio range of 0 to 20 kHz. Spectra and pulse generators are shown in Figures 4.3.1-3.

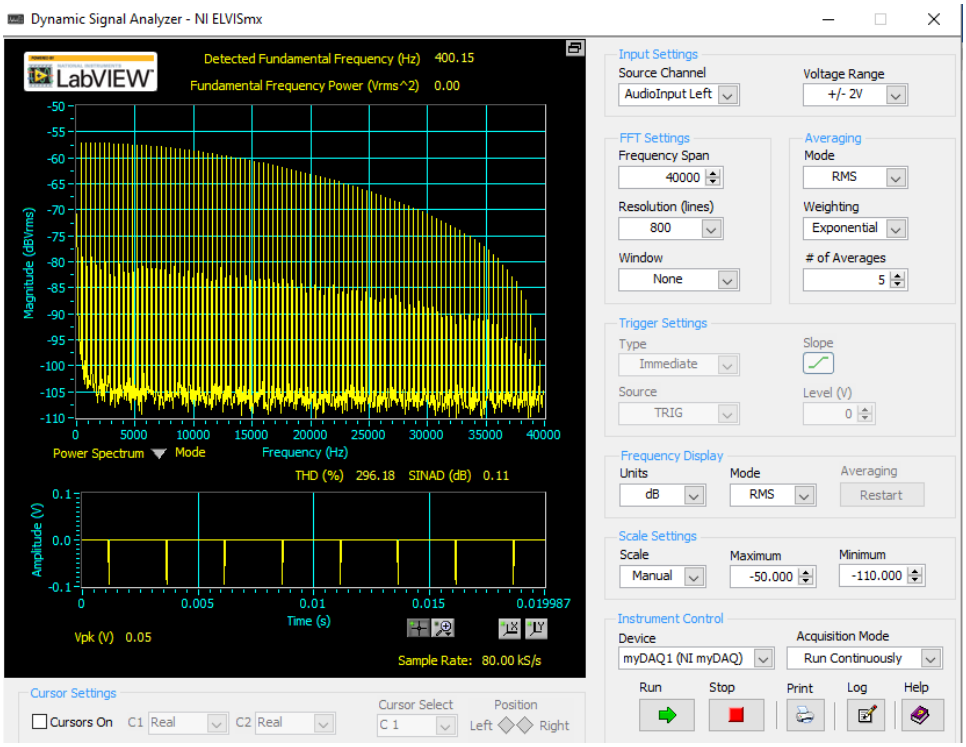


Fig. 4.3.1: MYDAQ Dynamic Signal Analyzer Spectrum of the output of the MYDAQ Arbitrary Waveform Generator.

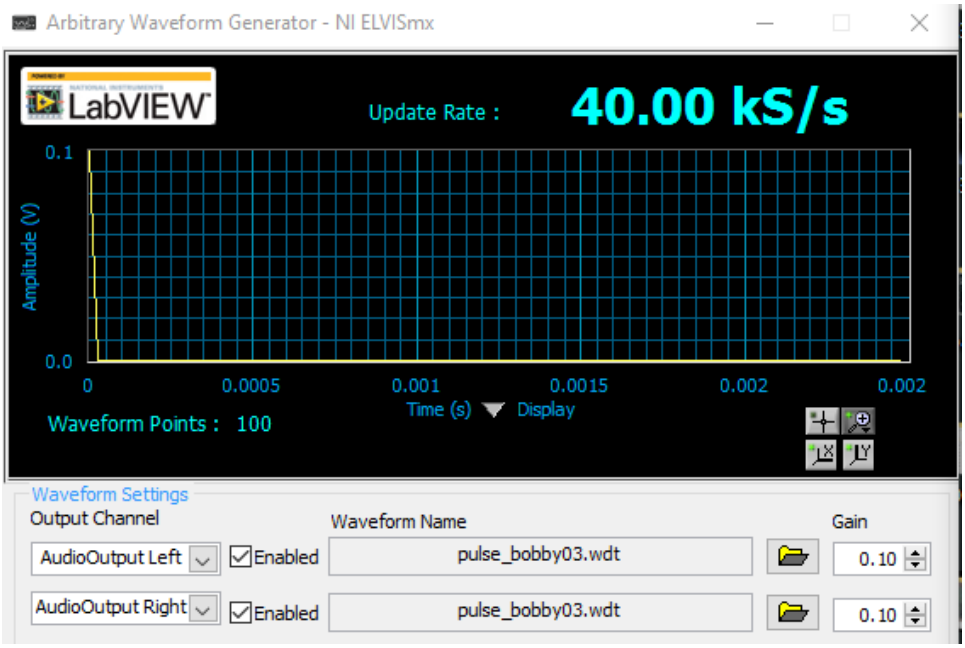


Fig. 4.3.2: Arbitrary Waveform Generator configured to output 400 Hz pulses as described in the text.

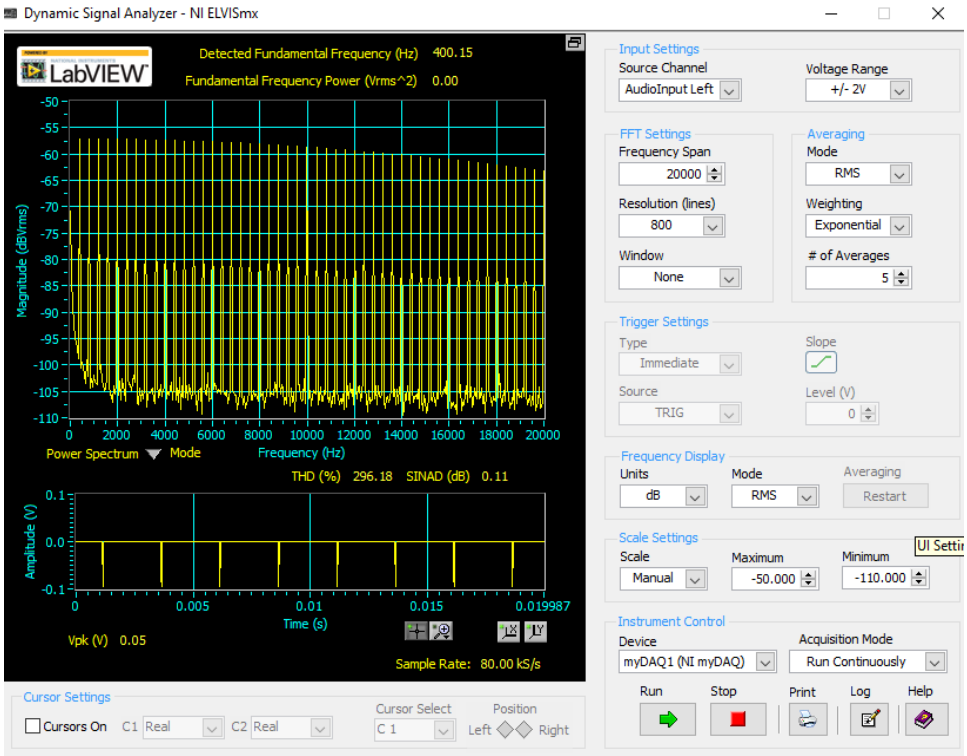


Fig. 4.3.3: MYDAQ Dynamic Signal Analyzer with an adjusted audio range set to 0-20 kHz.

When the test pulse is routed through the FPGA, the spectrum looks similar to the signal routed directly to the MYDAQ, except that the height of the harmonics above the noise floor is less. Changing the volume moves both the noise floor and the harmonic amplitudes by the same amount. The volume was fixed at 8 steps (out of 32) for all the measurements that follow. I also noticed that the sign of the pulse is inverted after it goes through the FPGA. The spectrum is shown in Fig. 4.3.4 below.

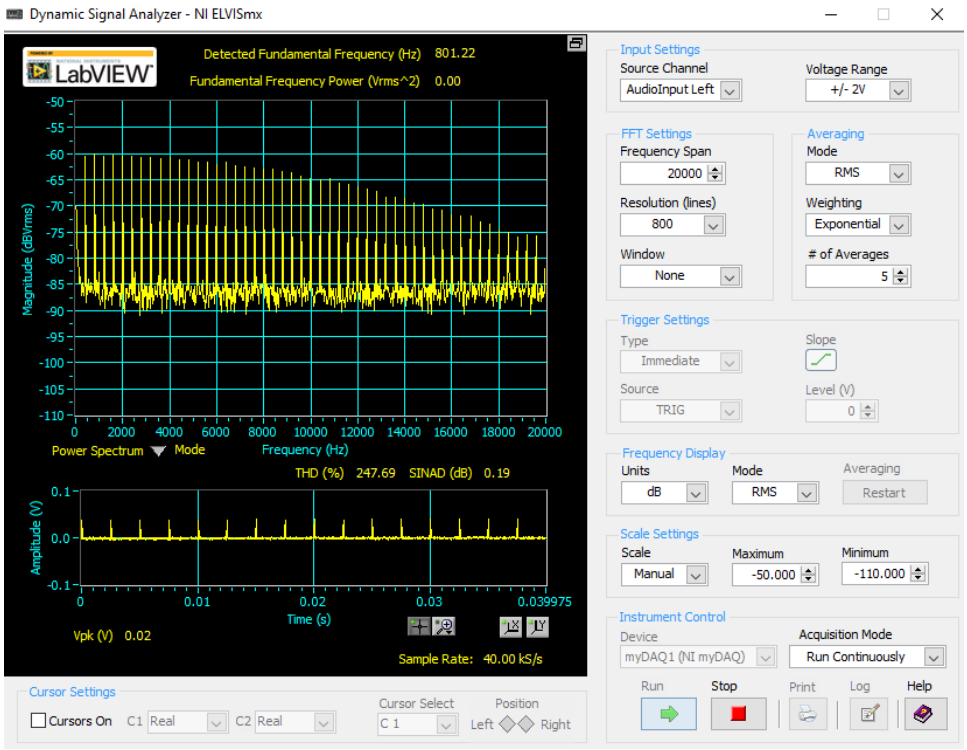


Fig. 4.3.4: Spectrum of audio sent through the FPGA with no audio filtering, with same pulse as before. Volume control is set to 8.

- **Low-Pass, Band-Pass and High-Pass Filters**

Using the pulse output as described above, sending the signal into the FPGA with the output volume set to 8, three filters were tested to see the effect that they had on the spectrum: low-pass in Fig. 4.3.5, band-pass in Fig. 4.3.6 and high-pass in Fig. 4.3.7. The coefficients for these filters are given in Table 1 of Section 3.

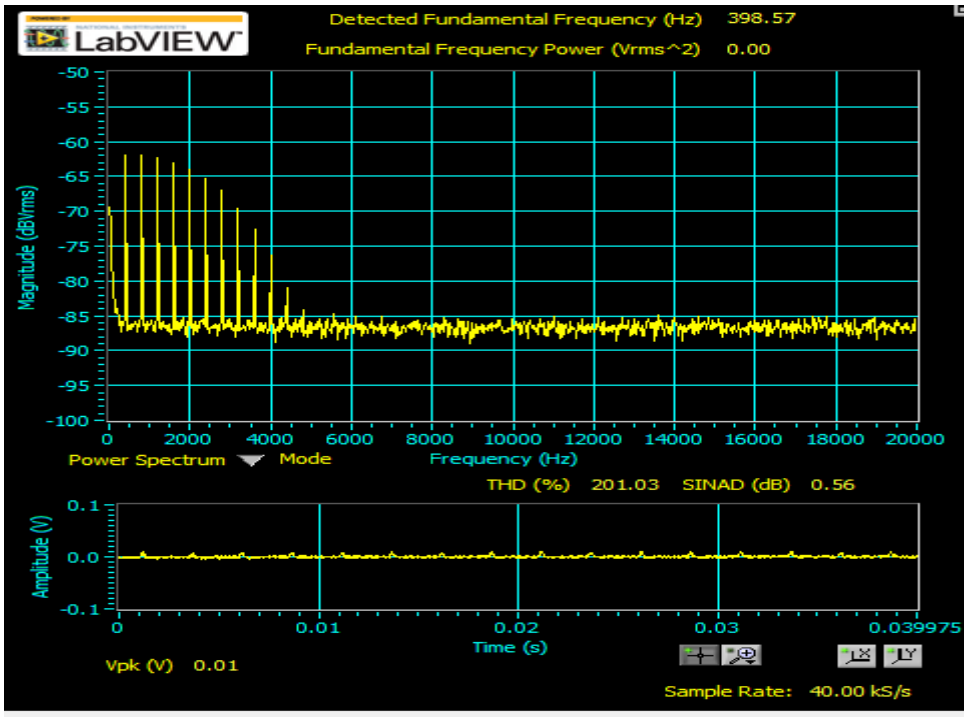


Fig. 4.3.5: Spectrum with the Low-Pass Filter at a volume setting of 8. Octave equation fir1(30,0.125) was used to obtain the coefficients.

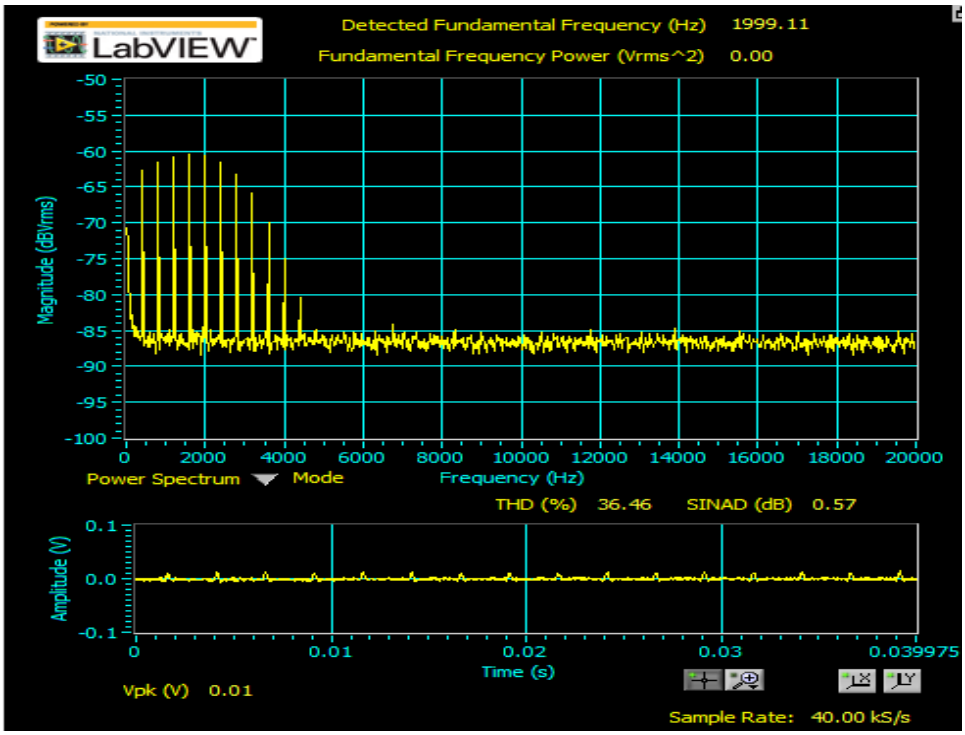


Fig. 4.3.6: Spectrum with the band-pass Filter at a volume setting of 8. Octave equation fir1(30,[0.05,0.1]) was used to obtain the coefficients.

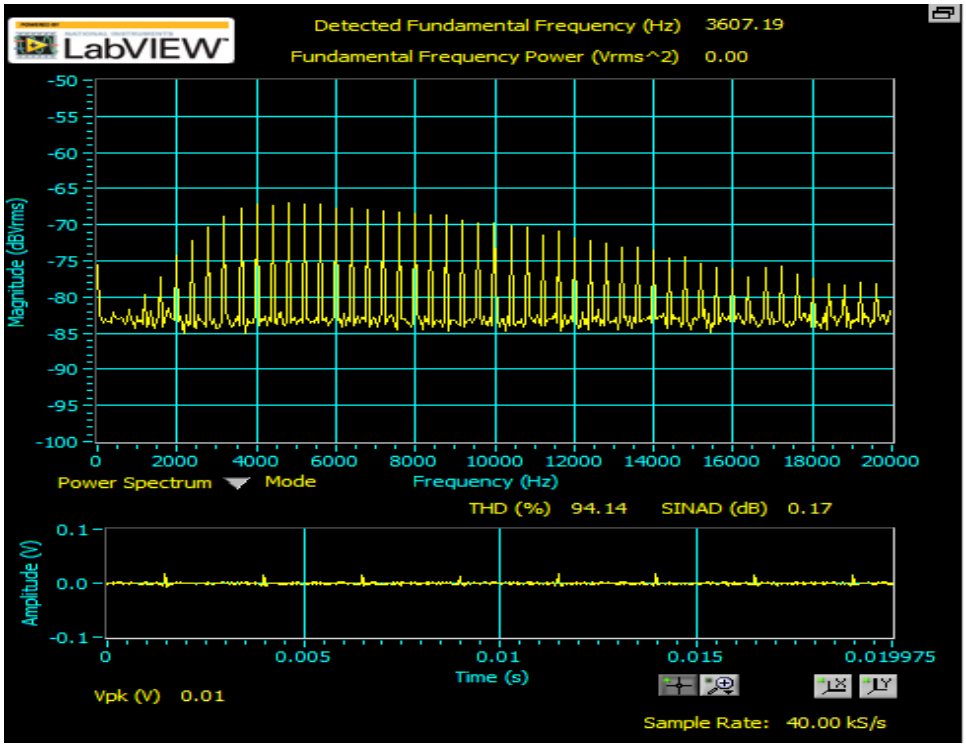


Fig. 4.3.7: Spectrum with the High-Pass Filter at a volume setting of 8. Octave Equation fir1(30,0.25,"high") was used to obtain the coefficients. Note: When generating integer coefficients, the values obtained from fir1() were scaled by 512 rather than 1024 that was used for the other filters.

- **Echo and Reverberation**

Testing of the echo filter was done in the time-domain. Pulses were used at a reduced repetition rate and the output of the FPGA was recorded using the Oscilloscope of the MYDAQ. Fig. 4.3.12 shows the output of the FPGA in response to a single pulse without filtering. The distortion that appears after the pulse is probably due to the AC coupling at the microphone input. Fig. 4.3.13 shows the response with echo filter which uses a single delay of 4000 samples so that the echo is 83 ms after the initial pulse.

To avoid the distortion produced by a unipolar pulse, the signal generator was made to generate a shorter pulse that was bipolar. This gave an improved response as can be seen in Fig. 4.3.14 which shows a straight-through response without filtering. Notice the responses are cleaner because it avoids the problem of the lack of DC response of the AC coupled microphone input. Fig. 4.3.15 shows the effect of the echo filter with a single delay of about 4000 and the echo appears at about 85 ms after the pulse. Fig. 4.3.16 shows the effect of the reverberation filter when a single delay of about 4000 samples is used. Finally, fig. 4.3.17 shows the response of the reverberation filter with multiple delays. Since this effect has multiple delays and thus more complicated, it's not as easy to see if the filter is doing as expected. But to the ear it sounds like reverberation.

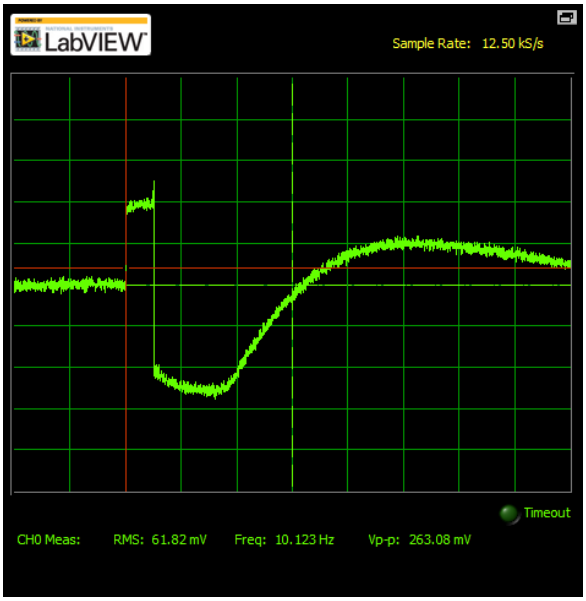


Fig. 4.3.12: Oscilloscope waveform of a single unipolar pulse without FPGA filter.

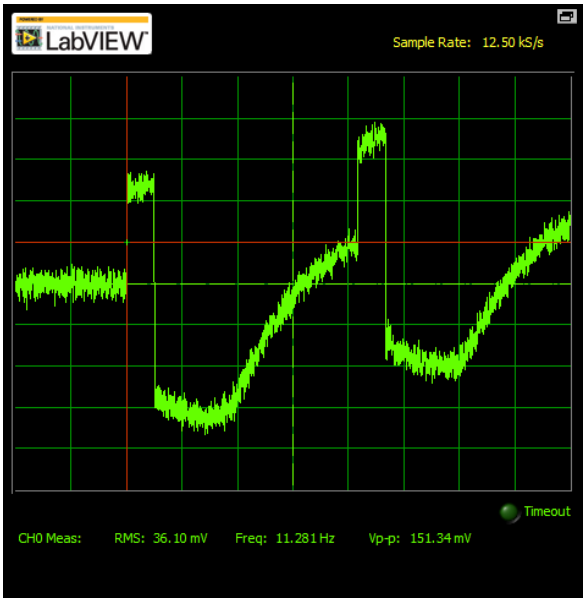


Fig. 4.3.13: Waveform of the Echo filter with a single unipolar pulse.

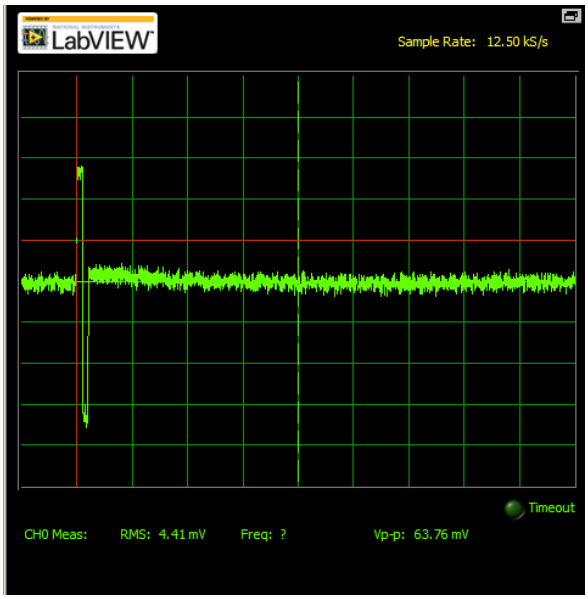


Fig. 4.3.14: Waveform without FPGA filter using a bipolar pulse.

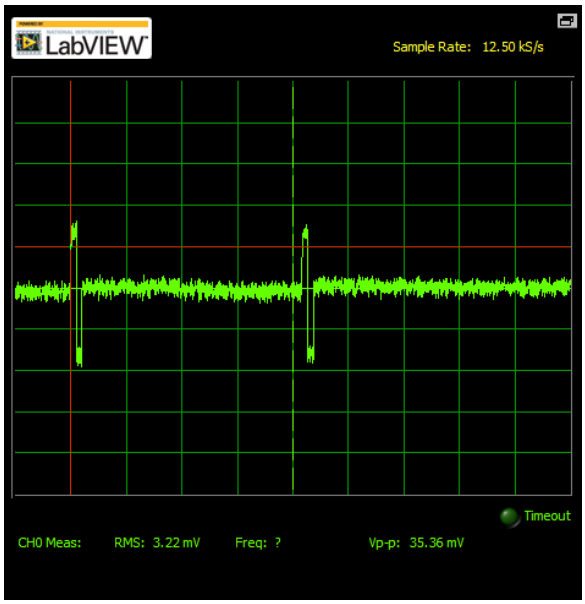


Fig. 4.3.15: Response of the Echo filter to a bipolar pulse showing a 83ms delay.

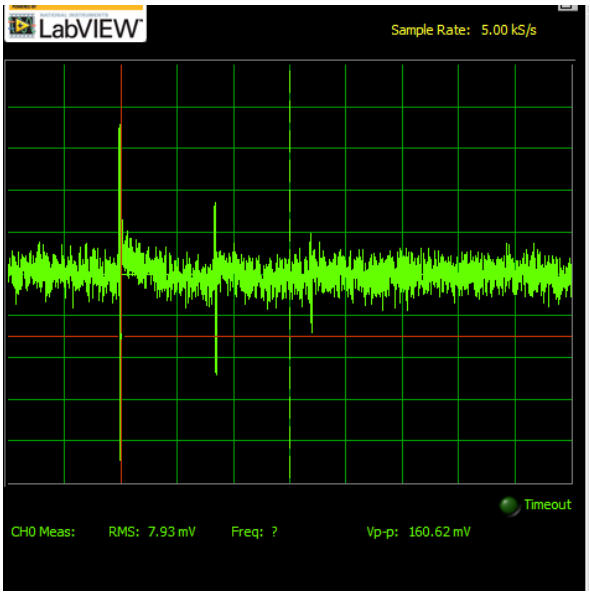


Fig. 4.3.16: Response of the Reverberation filter with a single delay to a bipolar pulse.

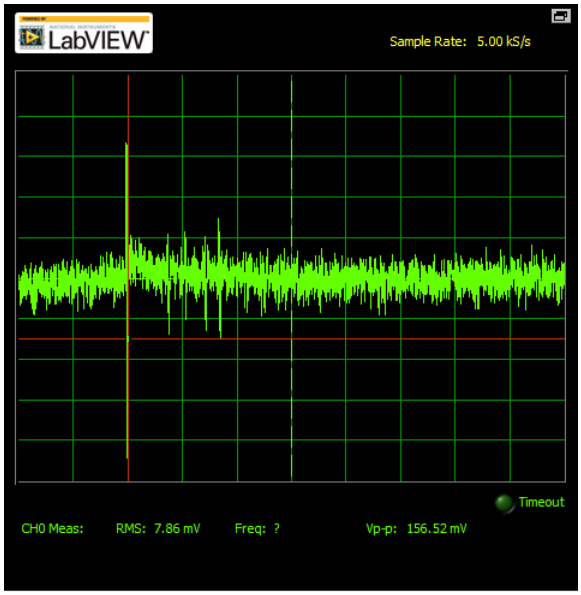


Fig. 4.3.17: Response of the Reverberation filter with multiple delays to a bipolar pulse.

- **ToneVar Module**

The FPGA generates a sine wave using the ToneVar module as described in Section 4.2.6, where a 64-element look-up table is used. The default output frequency is 750 Hz, which occurs when the 64-element look-up table is output at the 48 kHz sampling rate (48 kHz /64 = 750 Hz). The natural index for the look-up table would be 6 bits, but in this implementation, an index of 16 bits is used and only selecting the upper 6 bits of that index to select the location in the look-up table. In this way, the frequency can be changed by less than a hertz. However, this method generates other frequencies in addition to the desired frequency. The spectrum of the sine wave output at 750 Hz is shown in Fig. 4.3.8.

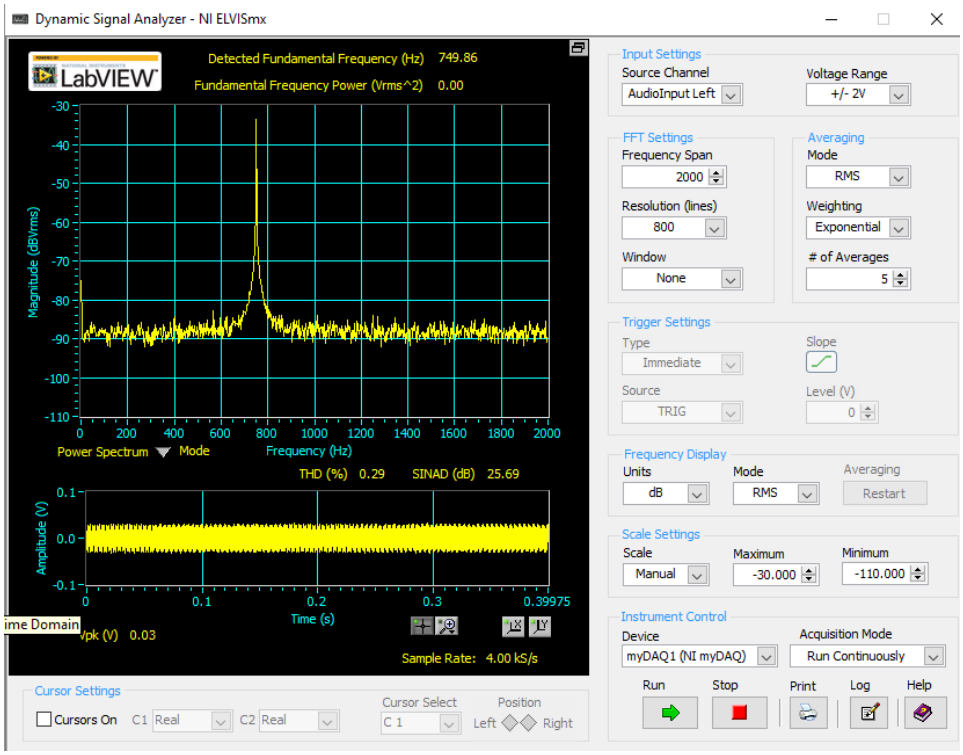


Fig. 4.3.8: Spectrum of the 750 Hz sine wave (ToneVar) generated by the FPGA, measured by the MYDAQ.

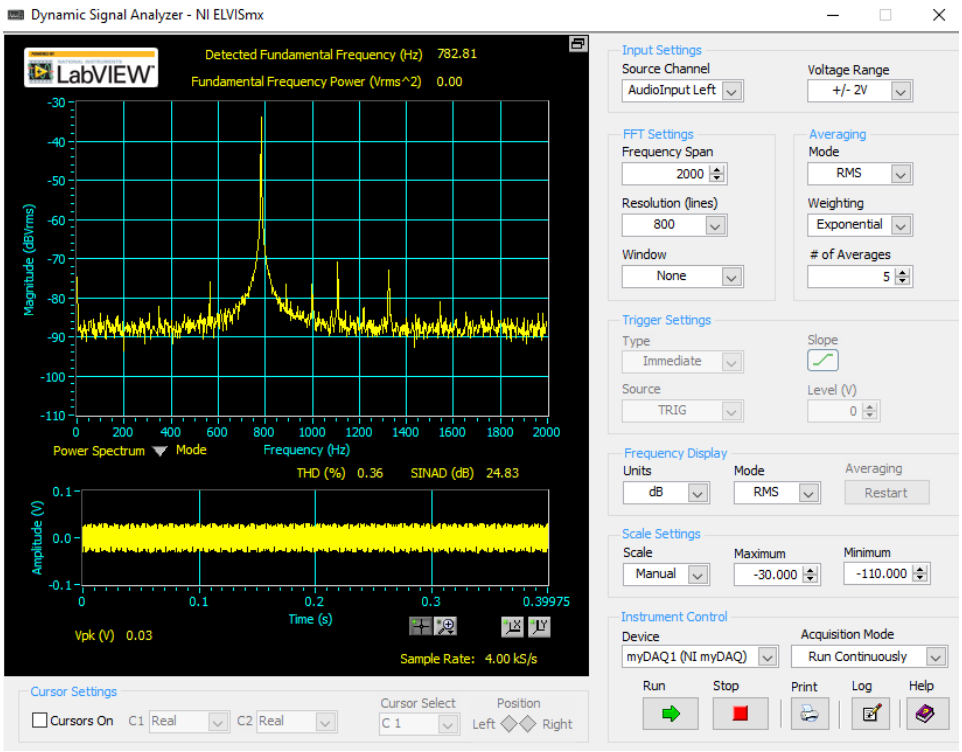


Fig. 4.3.9: Spectrum with a frequency of the sine wave set to 783 Hz, some additional frequencies are seen in the spectrum.

- **Ring Modulation**

The Ring Modulation uses the sine wave generated by the FPGA and multiplies it times the microphone input. The result is sum and difference frequencies as described in Section 4.2.7. The simplest test of Ring Modulation would be to send a sine wave to the microphone input at a single frequency. The function generator was set to output a sine wave at 200 Hz with an amplitude of 0.01 V. The setup is shown in fig. 4.3.10.

The spectrum of the Ring Modulation in Fig. 4.3.11 shows two frequencies which are 200 Hz above and below the 750 Hz modulation frequency. The 750 Hz signal is also visible in the spectrum. Ideally this would not be the case, but it must be due to a small DC offset in the input data to *MultiplierModule*().

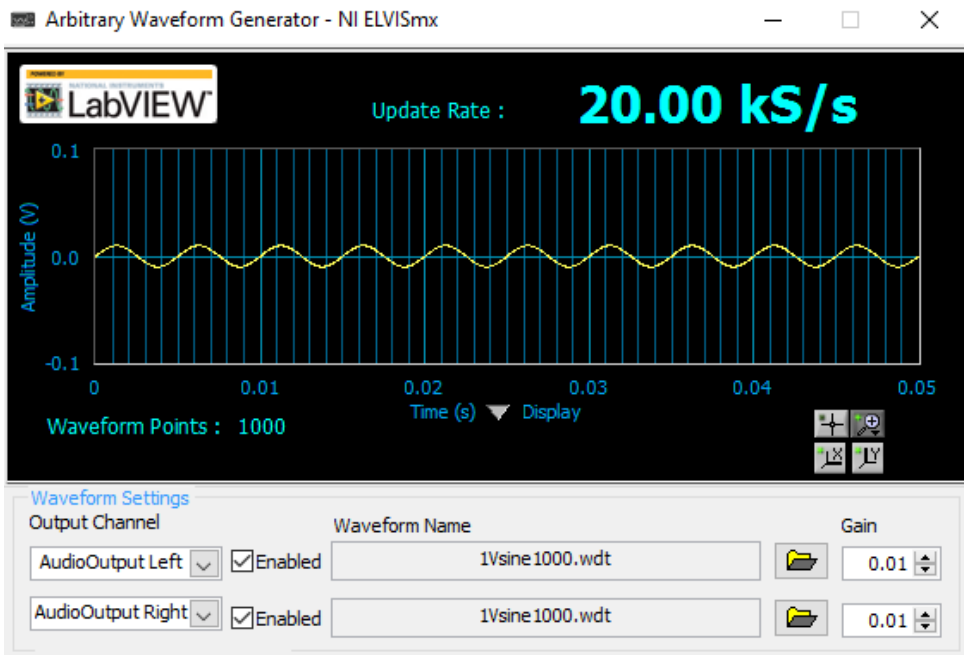


Fig. 4.3.10: MYDAQ Function generator setup used for the Ring Modulation Test.

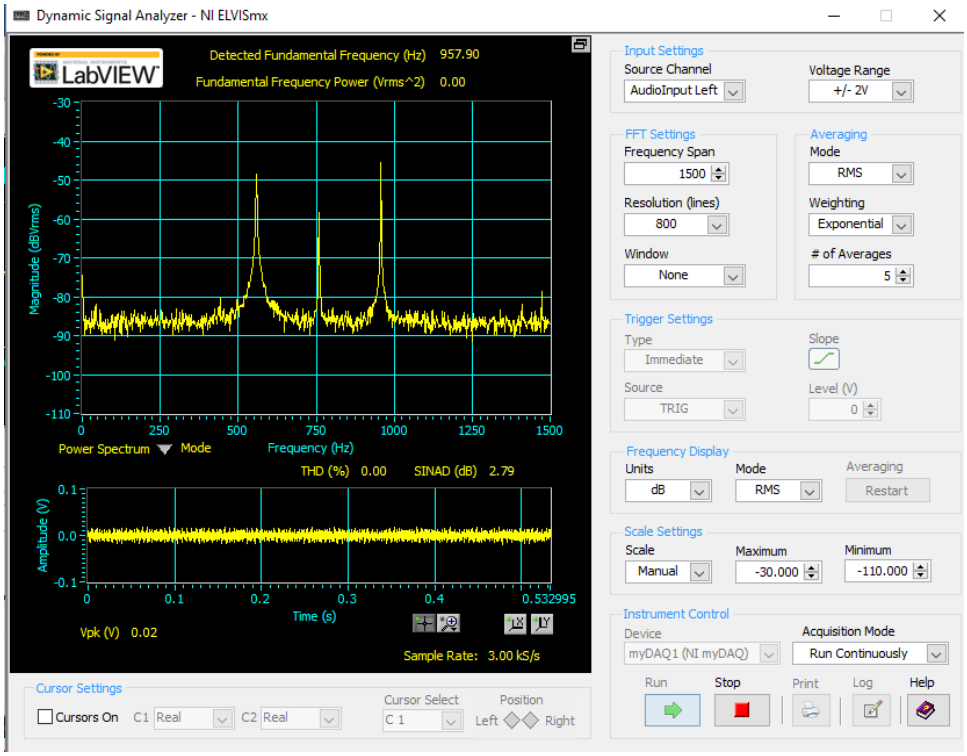


Fig. 4.3.11: Spectrum of the Ring Modulation with a frequency generated inside the FPGA of 750 Hz and this is multiplied by a sine wave from the MYDAQ of 200 Hz going into the microphone input. Its amplitude is 0.01.

- **Testing Summary**

The low-pass, band-pass and high-pass filters behave as expected. When creating the coefficients for these filters, one must be careful that the signed numbers should not exceed the number of bits available, including the sign bit. When using ToneVar, extra frequencies will appear if the tone is not 750 Hz. In Ring Modulation, the product of two sine waves is their sum and difference frequencies, however, the modulating frequency is also visible in the spectrum. The echo filter works as expected and the testing was improved by using a shorter bipolar pulse. With the reverberation filter, it was hard to see whether it was performing as expected but it could be heard that the filter was working.

Section 5: Results and Discussions

5.1) Result Analysis

Analog audio can be passed to and received from this device and the FPGA can manipulate the data using various filters. For simplicity, only a single channel of 8-bit audio was used. The filters matched the simulation provided by Octave for the low-Pass, band-Pass and high-Pass filters. The performance of the delay used in the echo and reverberation modules, was checked and seems to be performing properly. The Ring Modulation effect when tested with a single sine wave also performs as expected.

There are a few improvements that could be made to this project. Implementing 16-bit stereo processing overall, better use of block memory or development board memory for making longer delays possible, making the reverberation module more flexible with separate gains for each delay component, improve the purity of the sine wave produced by the *ToneVar* module and establishing control of the device with a GUI since all the buttons on the development board are already in use.

5.2) Multi-disciplinary Issues

Aside from a few power supply issues at the very beginning of the project, most of the actual work involved FPGA programming. Throughout the project, I've been getting advice from my father, who is an RF engineer at Cornell University. At the beginning of the project, neither of us knew anything about FPGA programming. His RF experience led to the idea of including the Ring Modulation module, which is a common technique referred to as mixing by the RF community and Ring Modulation by musicians.

One of the challenges of this project was making the translation between filter design and Verilog code. The fir1 function in Octave (or MATLAB) provided the filter coefficients that were needed to generate the various filters in Verilog. At this point, the coefficients were entered directly into the code of filter coefficient module rather than being downloaded from the computer. Establishing communication of data or instructions between the computer and the FPGA was beyond the scope of this project.

5.3) Professional/ Ethical Issues

The primary professional/ethical issue of this project concerns the Verilog code. The code developed for this project was based on code written for the 6.111 Lab class at MIT, which contains several modules. Some of these modules were used as found with very little modifications needed to make them function. A good example is the use of the *ac97* and the *ac97commands* modules. Here, the modifications needed were the changes in the clock rate and modifications to the UCF file to make the module compatible with the MimasV2 development board. At the other extreme are modules that I wrote completely from the beginning which were not based on any module in the MIT Code, like the *DelayModule*. There are several functions in a gray area between these two extremes like the *FilterControlModule*, which was originally a selection between filtered and unfiltered audio. I modified this to allow many filters to be selected. At this point, it would barely be recognizable when compared with the original code. Another example that is closer to the original MIT code is the coeffs31 module, which originally had coefficients for a single filter, but I changed it so that several different sets of coefficients could be chosen using a case statement.

If I am to make the Verilog code for this project publicly available, I should give credit to the original authors of the MIT Code. Nowhere in the code is copyright mentioned. Publishing modified code would not be a violation of copyright law. But the code does include the statement that it was written by Nathan Ickes, 6.111 MIT Staff. So, at the very least, this statement should be left in the code. On the other hand, I would like to give myself credit for the code that I have written. For the moment since all the code is in a single file, I'm leaving at the top of the file, an acknowledgement that the code was written by MIT with considerable modifications made by Robert Sikora. Eventually, I should probably go through the code, module by module and include one or both names in the comments at the beginning of the module as appropriate.

5.4) Impact of Project on Society and Contemporary Issues

As stated in the goals of this project, this device is meant to be a starting point that can be used by other students or hobbyists. This could be as simple as a demonstration of audio filters where the code can be edited to include different coefficients for different filters or a completely new module can be developed to extend the capabilities of this audio processor. For example, a real-time Autotune function where a voice is made to sound at a specific pitch or pitches in a scale that is different from the input audio. As another example, this project has a module that produces sine output, this could be expanded to include multiple sine waves and other synthesis algorithms for producing sound in addition to filtering incoming audio. These things could be done either in a lab setting as part of a class or as a collaboration effort where people can share the modules that they have produced or modified.

Section 6: Summary and Conclusion

A Real-time Audio Processor was implemented using a MimasV2 development board from Numato Lab along with an extender board also made by Numato Lab. The development board was designed around a Spartan-6 FPGA and also had push-buttons and a seven segment LED that were used for the project. The extender board features an LM4550 which was used for ADC and DAC so that input from a microphone could be digitized and passed through the FPGA, then sent to the headphones.

Verilog Code was adapted from an MIT board with similar hardware to handle the communication with the extender board. The majority of the project was Verilog code that was written to implement audio modification modules. These modules included 30 element FIR audio filters for low-pass, band-pass and high-pass. Code was also written to produce an echo with a maximum delay of 85 ms and a reverberation filter that could make use of one or more delays. Verilog code was modified to produce an oscillator with an adjustable frequency. A multiplier module would multiply the incoming audio times the sine wave generator. This gives the effect of ring modulation.

The LM4550 can digitize a stereo input with at least 18-bit resolution. However, for simplicity only a single channel was used and only 8-bit data was used for the implementation of this project.

This development board is a more economical alternative to some of the other development boards on the market like the Digilent Atlys.

Acknowledgements

I would like to thank my father, John Sikora, for helping me with troubleshooting the Verilog code and with proofreading this report and for providing general guidance throughout the execution project.

References:

[1] Gelb, B., Hom, G., Terman, C., & Valys, A. (2008). MIT, *6.111 Lab#4*. Retrieved from http://web.mit.edu/6.111/www/f2008/handouts/labs/lab4.html

[2] Rosenthal, Elizabeth. (2013, Jan 26). *Your Biggest Carbon Sin May Be Air Travel*. Retrieved from http://www.nytimes.com/2013/01/27/sunday-review/the-biggest-carbon-sin-air-travel.html?rref=collection%2Fbyline%2Felisabeth-rosenthal&action=click&contentCollection=undefined&region=stream&module=stream_unit&version=latest&contentPlacement=58&pgtype=collection

[3] Government of India Ministry of Environment and Forests. (2016, Mar 5). *Environment Ministry releases new categorization of industries*. Retrieved from http://pib.nic.in/newsite/PrintRelease.aspx?relid=137373

[4] United States Environmental Protection Agency. (1990). *Guides to Pollution Prevention: The Printed Circuit Board Manufacturing Industry*. Retrieved from https://archive.epa.gov/sectors/web/pdf/01050.pdf

[5] Bradsher, Keith. (2015, Oct 14). *India's Manufacturing Sector Courts the World, but Pitfalls Remain.* Retrieved from https://www.nytimes.com/2015/10/15/business/international/indias-manufacturing-sector-courts-the-world-but-pitfalls-remain.html?_r=0

[6] Giardina, Anthony. (2012). Digital Graphic Equalizer Implemented Using an FPGA. Retrieved from http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1155&context=eesp

[7] Ababei, Cristinel. (2012). Lab 7: SUNY Buffalo, *Interfacing FPGA Spartan-6 with Ac'97 Codec*. Retrieved from http://www.dejazzer.com/ee478/labs/lab7_audio_ac97.pdf

[8] Chou, C., Evans, J. & Mohanakrishnan, S. Kansas University, *FPGA Implementation of Digital Filters*. Retrieved from http://www.ittc.ku.edu/Projects/FPGA/Digital_Filters.pdf

[9] Hifi Engine. *TEAC EQA-220*. Retrieved from https://www.hifiengine.com/manual_library/teac/eqa-220.shtml

[10] Musician's Friend. *Yahama EMX312SC Powered Mixer Product Detail*. Retrieved from http://www.musiciansfriend.com/pro-audio/yamaha-emx312sc-powered-mixer#productDetail

[11] Xilinx Spartan-6 Family. https://www.xilinx.com/products/silicon-devices/fpga/spartan-6.html

[12] LM4550 Datasheet National Semiconductor
http://web.mit.edu/6.111/www/f2008/handouts/labs/LM4550.pdf