Custom Instrument and Audio Effects Processor Using NI myRIO

Eric Gubiani, Jeffrey Kirman, Jose Mastrangelo, and Justin Mendonca

***Abstract*—This paper presents a custom instrument capable of tone generation, variable tone quality, accelerometer based pitch and volume control, and an audio effects processor consisting of master volume, 3 band equalization, echo, and left-right balance. These applications were implemented on two separate National Instruments myRIO boards using labVIEW.**

***Index Terms*—tone generation, tone quality, accelerometer, 3 band eq, echo, myRIO, labVIEW, audio processing.**

I. Introduction

NEW types of musical instruments are constantly being created, attempting to break boundaries with new sounds and styles of playing music. Among those, electronic musical instruments have continuously grown and become increasingly popular in recent years. This project aims to create a musical instrument that creates electronic music and provide users with motion-controlled customizable tone, pitch, and volume. As well, it details a way to process the signal generated by the instrument with effects such as left-right balancing and echo.

II. Implementation of a Musical Instrument

The musical instrument is designed to be able to produce three separate notes with adjustable tone qualities and volumes. Frequencies for the notes are calculated in the microcontroller, and the signal generation and processing are implemented in the FPGA of the myRIO board. The FPGA is run at maximum clock speed, 40 MHz, to ensure operations are completed as quickly as possible. The accelerometer is used as a means for pitch and volume control. The way these units interact with each other can be seen in Fig. 1. All attributes that are to be varied by the user is done through the LabVIEW front panel seen in Fig. 2.

A. Frequency Generation

When generating the fundamental frequencies on the microprocessor, a button is used to control whether the board will be outputting chords or single notes (Chords), and a button used to control whether the notes being output are continuous or discrete (Slide). If the Slide button is pressed, we simply take the y-component of the accelerometer and manipulate it using arithmetic operations and send that frequency to be adjusted by the Octave slide on the front panel, to be discussed later. Alternatively, if the Slide button is not pressed, the y-component of the acceleration is sent to a custom made VI which discretizes the frequency into discrete notes.

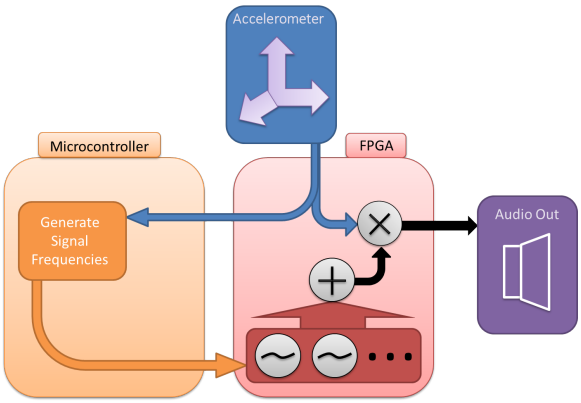

Fig. 1. Block diagram of the implementaion of the musical instrument.

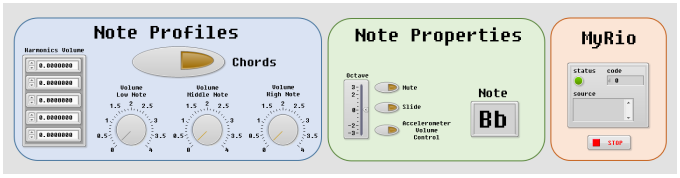

Fig. 2. Front panel for user input of the custom musical instrument.

This custom VI takes in the y-acceleration as input and outputs the discrete frequency representing a particular note and a string letter representation of the output frequency. The output frequency will be 1 of 11 notes of a pentatonic scale with the blues note added spanning over two octaves ($B_2 - D_3 - E_3 - G_3 - A_3 - B_3^b - B_3 - D_4 - E_4 - G_4 - A_4$) and is calculated using a sum of step functions. In order to obtain the frequency of a respective note, the entire range of accelerometer values is first divided into 12 ranges, where each range is assigned to a particular note. The input accelerometer values is subtracted by the upper limits of these ranges and the values will be used to shift the step functions of each note. Since the step functions now output values of 0 or 1 depending on the value of the accelerometer input, then if the accelerometer value is within a certain range, all step functions below and including the current range will output 1, while the ones connected to the higher notes will output 0. The value of the first note is directly multiplied by the value of its step function, but for subsequent notes, the difference between the frequencies of the current note and the previous note is multiplied by the value of its step function. The outputs

of the step functions are finally added together and we obtain an equation for the discrete notes, seen below.

$$f_{dis} = f_0 \cdot u\left(t - \left(A_y - \left(260 - (n-1) \cdot \frac{520}{n}\right)\right)\right) + \sum_{i=1,j=2}^{i=11,j=n-1} \Delta f_{i-1,i} \cdot u\left(t - \left(A_y - \left(260 - (n-1) \cdot \frac{520}{n}\right)\right)\right) \quad (1)$$

In equation 1, i represents the indexed frequencies of musical notes, j represents the index of the accelerometer ranges, and $n = 12$ represents the number of accelerometer ranges.

The string representation of the output frequency is obtained by retrieving a certain index in a pre-defined string array. This index is obtained by adding the values of each of the step functions.

An Octave slide is used to increase and decrease the octave of either the continuous or discrete notes. This is done using a scale by power of 2 function, since going up or down an octave is done by simply multiplying/dividing the frequency by 2.

Power chords were then generated by adding a note with its fifth, which is 1.5 times the the notes fundamental frequency [1]. The next octave (2x) of the note is also added in order to increase the fullness of the sound. Finally, the calculated fundamental frequencies are used to generate audio signals in the FPGA.

B. Signal Generation and Processing

The instrument creates a tone quality for each note by generating the fundamental frequency signal and combining it with the four subsequent harmonics. Since the operations required for signal generation and processing require fast performance, the signal generation and processing logic is designed on the onboard FPGA of the NI myRIO [2]. Fig. 3 shows the implementation of signal generation in LabVIEW. Sine wave generator blocks are used to produce a sine wave at a calculated frequency and a set sampling rate of 44 kHz, chosen due to its common usage for audio sampling [3]. To produce four subsequent harmonics, the calculated frequency is multiplied by integer factors, two through five, and the resultant frequencies are used as input to separate sine wave generator blocks. An array of fixed point numbers contain the gain level of each harmonic are converted into a cluster, de-bundled, and then multiplied to the output of their respective sine waves before they are summed together. This allows for variable tone quality by changing the volume of each harmonic. Multiplication is carried out using fixed point arithmetic for increased precision, requiring the 16 bit integer outputs of the sine wave generator to be converted to fixed point.

Three instances of the signal generation code are used to generate three separate notes. Similar to the harmonic volume each note is multiplied by a fixed point number gain to control the volume of each note. Chords are implemented by a summation of the three separate notes.

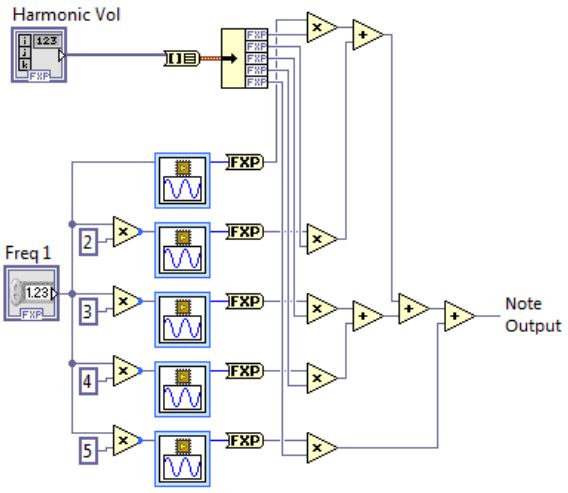

Fig. 3. LabVIEW code for generating the summation of a sine wave and four of its harmonics.

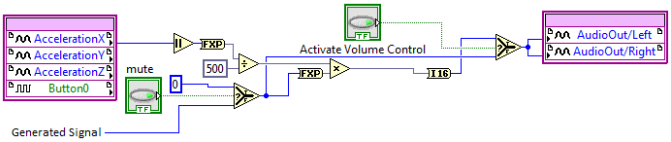

Fig. 4. LabVIEW code for changing the gain on the output signal using accelerometer data and controls.

Acceleration controlled volume level of the output signal is achieved through scaling the raw acceleration data from the native accelerometer to the myRIO, and multiplying the result with the output signal. Similar to signal generation, multiplication is carried out using fixed point numbers for precision. Fig. 4 shows the implementation of signal volume control in LabVIEW. A select block controlled by a control is used to switch between the signal at acceleration controlled volume and at maximum volume. Mute functionality is implemented similarly with a select block switching between the generated signal and no signal. The onboard button is also used to toggle the mute control when pressed and held, achieved by a select block and a not gate in the microcontroller code.

III. Audio Effects Processing Unit

The audio effects processing unit (AEPU) was required to have four functions: Left-right balance, 3-band equalization, echo, and master volume. These functions were implemented on a second myRIO board from the instrument. The entirety of the signal processing was done on the FPGA with the microprocessor responsible for supplying variouis parameters, most from user input via the front panel in labView shown in Figure 5. It was chosen to have all of the singnal processing allocated to the FPGA to attempt to have the highest processing speed possible to maintain the best sound quality. The FPGA was run at its maximum clock frequency of 40MHz, however the audio processing was done in a timed loop sampling at 20kHz, which is near maximum audible frequency for humans meaning the audio signal sound quality should be minimally affected. The

Fig. 5. LabView front panel UI for the effects processing unit

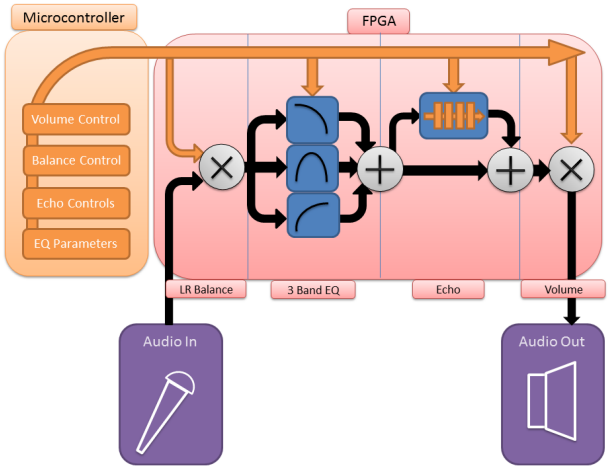

Fig. 6. Overall block diagram of the audio effects processing unit

overall block diagram of the system can be seen in Figure 6. As well, the implementation of the AEPU on the microprocessor can be found in Figure 7.

A. Left-Right Balance

The left-right balance effect which processes the audio signal on the FPGA unit, takes as input a fixed point value between -1 and 1 from the microprocessor obtained from the front panel control. The value passed in from the microprocessor, labeled Balance in Figure 8, is subtracted from 1 for the left channel and added to 1 for the right channel. The resulting values are multiplied by their corresponding audio signal paths to scale the signal accordingly. The result is that at -1, the right channel is completely cancelled and at 1, the left channel is completely canceled. The FPGA implemented block diagram can be seen on the left side of Figure 8.

B. 3-Band Equalizer

The next step in the signal path was the 3-band equalizer, which can be seen in Figure 8. Each channel was fed into three parallel filters which were either low-pass or high-pass filters. The cutoff frequency of each filter was controlled from the front panel of the microcontroller, where it was fed into a Butterworth Filter Coefficient subvi, and passed into the FPGA as a fixed point number. As well, the gain for each band is taken from the front panel of the microcontroller, converted to fixed point, and fed into the FPGA. The range

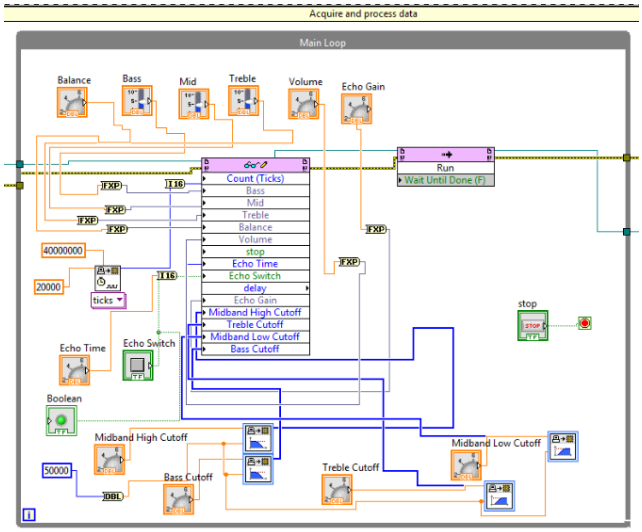

Fig. 7. Microprocessor implementation of parameter-supplying interface

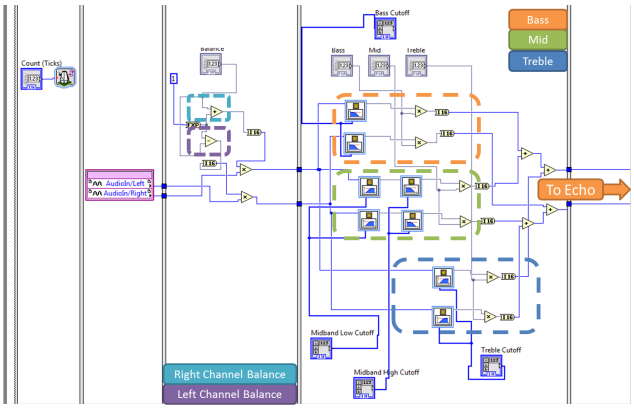

Fig. 8. labVIEW FPGA implementation of left-right balance and 3-band equalizer

of the coefficients is such that there can be slight overlap in the bands, while the range of the gain is from 0 to 8. The bass-band filter was implemented as a low-pass filter, as was the upper-limit of the mid-band. The lower-limit of the mid-band and the treble were implemented as high-pass filters. By using both a low-pass and a high-pass filter for the mid-range band, the combined effect is a band-pass filter. After each filter, the signals are converted back to 16-bit integers. At the end of the equalizer stage. the split-signals of each channel are recombined into one and fed into the echo stage.

C. Echo

Echo is implemented in entirely in the FPGA with the parameters Echo Time and Echo Gain passed to the FPGA from the microcontroller in the same fashion as the previously described functions. The corresponding FPGA implemented block diagram can be seen in Figure 9. To echo the signal, the samples must be written to memory and read a specific time later (the echo time) to be combined with the main signal path. The memory was implemented using a FIFO queue for each audio channel. The FPGA resources on the

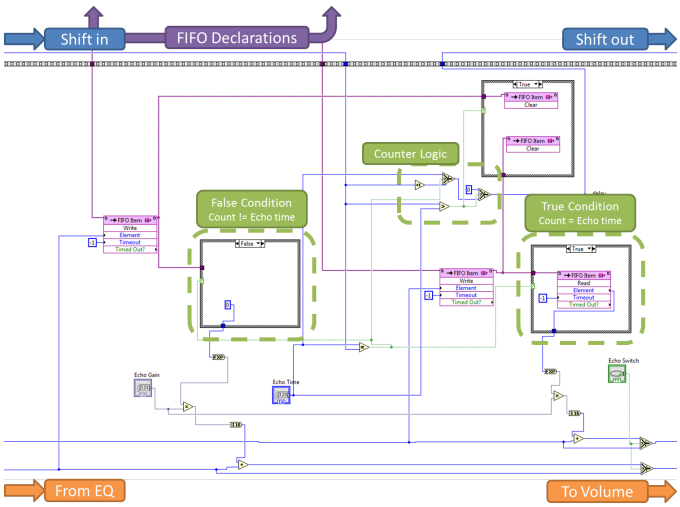

Fig. 9. labVIEW FPGA implementation of echo

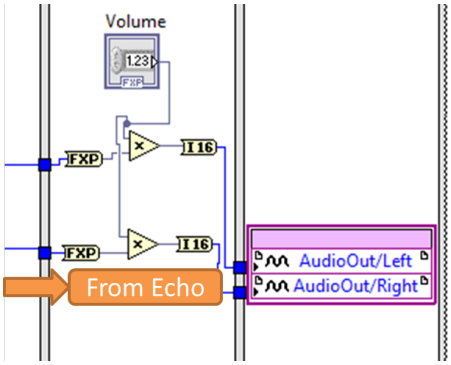

Fig. 10. labVIEW FPGA implementation of volume control

myRIO only allowed for a maximum of 31000 elements on each FIFO which set the upper limit on the delay time. The time was implemented using a counter initialized at 0. Each loop iteration the counter increments by 1 until the Echo Time is reached. When the echo time is reached, the counter is no longer incremented and the echo time is continuously passed to the shift register to ensure that the counter remains equal to the echo time. If the echo time is increased, the counter begins incrementing again. If the echo time is decreased below the counter value, the counter is reset to 0 and the FIFOs are cleared. Whenever the counter is equal to the echo time, the case structure is set to true which reads a sample every loop iteration. When the counter is not equal to the echo time, 0 is combined with the current signal. A fixed point value between 0 and 2 labelled Echo Gain in Figure 9 is also passed from the microprocessor and is multiplied by the echo signal before it is added back into the main signal path. This allows for either an attenuation of amplification of the echoed signal depending on user preferences.

D. Master Volume

The last step in the signal path of the AEPU was the master volume control shown in Figure 10. Its implementation was straightforward: a double was selected from the front-panel of the microcontroller between 0 and 2, converted to fixed point, and fed into the FPGA. Then, it was multpilied by the signal of both the left and right channel and converted back into a 16-bit integer. This results in either a gain or attenuation depending on what value was chosen.

IV. Discussion

A. Custom Instrument

The musical instrument unit was developed creating a basic instrument first and then adding features on using a modular approach. Initially implementing a basic sound generator was essential since testing required listening to the changes in the sounds produced by the instrument.

To test generating frequencies the board wad moved in the y-direction and the 11 discrete notes were compared to tones generated by a known source. The non-discretized pitch control was tested by listening for a continuous pitch change along the entire range of the accelerometer values. Chords were tested by differentiating between three different notes being played simultaneously using their seperate volume controls. Octave changes were tested by listening to the note and making sure that the pitch increased but the note remained constant.

Volume control using the accelerometer was first implemented by multiplying the output audio signal with the 16 bit integer value of the acceleration in the x-direction. The volume range was too large for stable control by the user, therefore fixed point conversion and scaling the raw acceleration data before multiplication with the output signal was implemented. A scaling factor of 50 was chosen since it produced an optimal volume range.

Tone quality was the final implemented feature to maximize FPGA usage. More sine wave generator blocks per note lead to a wider range of sounds; however, these blocks require a large amount of FPGA resources in LUTs. The optimal amount of signal generators was chosen through trial and error to be five per note, (resulting in fifteen in total for the entire FPGA), using 82Adding more sine wave generator blocks result in an overuse of resources despite there being 18

B. Audio Effects Processing Unit

The audio effects processing unit was developed one component at a time, working up from what was estimated as the easiest implementation to the most difficult. Volume and left-right balance were first to be implemented and little trouble was had in their development. Some adjustments were made later on regarding the amount of maximum gain to allow each stage to prevent distortion in the final output signal. 3-band equalization was next to be implemented and was also not much trouble as the filter VIs in LabView are very intuitive to use. Again there was some adjustments made regarding gain later on in development, as well as the addition of tunable filter range as an extra feature. The most difficult of the functions was echo. The first attempt consisted of using a large array (30000 elements) which would be used to store samples. The echoed sample would then be pulled out of

the array a designated number of elements from the front of the array. Unfortunately, even an array of 100 elements was enough to maximize the resources of the FPGA. Replacing the array with a FIFO queue alleviated this problem, and a counter was implemented to replace the index selection of the array method. This worked, however due to a timeout issue with the FIFO queues, the echo time was fixed at approximately 2 seconds until the an attempt was made to override the timeout, which fixed the issue. Finally, when the echo time was reduced below the current value of the counter, some issue with the FIFO was causing the FPGA to crash completely where it would then need to be rebooted. This issue was probably attributed to the FIFO queues reaching maximum capacity before a sample was read. This issue was fixed by clearing the FIFO every time this event occurred.

V. Conclusion

The result of this project was a musical instrument that has a robust amount of features and is fun to play. In hindsight, to reduce time spent on FPGA compilations, it would have been beneficial to fully design the FPGA code before implementation instead of an iteratively adding to the code. Also, indicators and waveform charts were an indispensable resource for debugging the code. Furthermore, other features such as vibrato, playback, and accelerometer stabilization were discussed but scrapped due to time constraints.

Acknowledgment

Thank you to the teaching assistant Michael Yuhas whose help was invaluable throughout this development process and to Prof. Meyer for the opportunity to work on such an engaging project.

References

[1] W. contributors, "Power chord," *Wikipedia, The Free Encyclopedia*, Feb. 2015. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Power_chord&oldid=647467594

[2] L. Adams, "Choosing the right architecture for real-time signal processing designs," *Texas Instruments*, Nov. 2002. [Online]. Available: http://www.ti.com/lit/wp/spra879/spra879.pdf

[3] W. contributors, "Sampling (signal processing)," *Wikipedia, The Free Encyclopedia*, Apr. 2015. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Sampling_(signal_processing)&oldid=656020839