ESE465: Audio Equalizer

By Nathan Jarvis & Ethan Snow

1. Abstract

An audio equalizer is a tool for adjusting the amplitudes of individual frequency bands within a recording. They are commonly used to process music recordings. This report describes the construction and implementation of an audio equalizer using an FPGA to allow real-time processing and attenuation of 13 separate frequency bands. The final design uses each of the previous projects for this class, and culminated in a successful demonstration.

2. Introduction

a. Sampling Sound

When sampling an analog signal to obtain a digital signal, the sampling frequency must be at least twice as high as the highest frequency in the signal (the Nyquist frequency), or aliasing will result. Humans can hear frequencies up to approximately 22 kHz, so an audio equalizer must have a sampling frequency of at least 44 kHz. Any sound fed into the equalizer must be prefiltered to remove any frequency components above 22 kHz, or else this sound may alias into the audible range. The songs played through the equalizer from phones and computers have already been prefiltered, so this is not a concern for our design .

This particular equalizer samples at 50 kHz per channel. Because stereo audio has two channels, the equalizer has an aggregate sampling rate of 100 kHz. Both channels pass through digital filters that isolate out 13 different frequency bands. The equalizer applies a user-determined attenuation between 1 and 0 (full volume and no volume) to each band before reassembling and outputting the sound in real-time.

The equalizer uses a 30 MHz clock. For an aggregate sampling rate of 100 kHz, a sample must be collected from the ADC (analog-to-digital converter), run through the filter, attenuated, reconstructed, and put back out to the DAC (digital-to-analog converter) every 300 clock cycles. The filters each have 279 coefficients, and the filter module performs 1 MAC (multiply and accumulate) operation per clock cycle. This means that the sample spends 279 clock cycles in the filter, and all the remaining steps must happen within the remaining 21 clock cycles. Embedded software would not be able to guarantee completion of the transfers between modules quickly enough, so this project required the creation of a controller module to handle all the transitions. Embedded software was used for PC communication and initialization of the hardware. Figure 2.a.1 shows the circuit schematic of the ADC board, and Figure 2.a.2 shows the circuits schematic of the DAC board that were used for sampling and playing back the sound. Professor Richter provided both schematics on the class website.

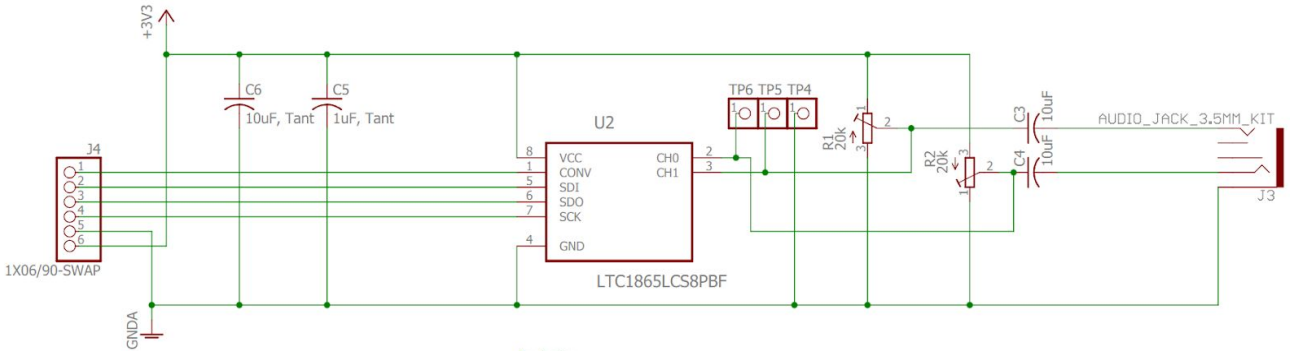

Figure 2.a.1 Circuit Schematic for the ADC Board

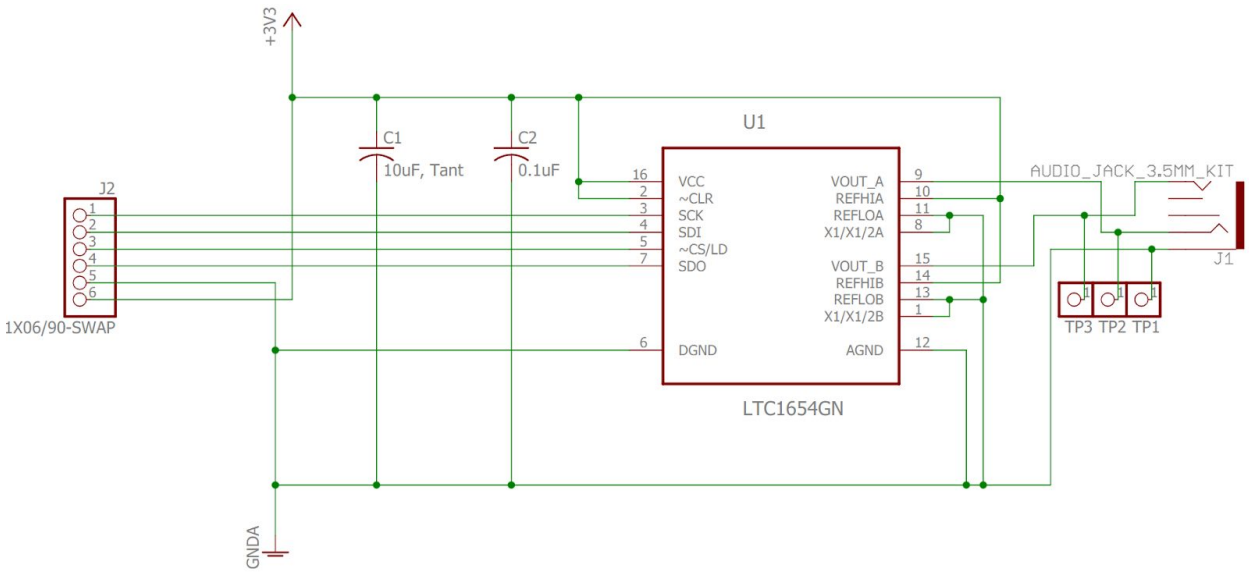

Figure 2.a.2 Circuit Schematic for the DAC Board

The 100 kHz aggregate sampling rate and real-time filter computations for each of the 13 filters place high performance demands on the equalizer. The 13 separate filters, however, naturally suggest a parallel approach. The high performance requirement, real-time requirement, and parallel nature of the task make an FPGA the preferred approach.

b. Filtering and Attenuation

Of the 13 filters, the first is a low-pass filter, the middle 11 are band-pass filters, and the last is a high-pass filter. Each one selects a range of frequencies from the sound, which it passes through with minimal attenuation or amplification while attenuating all other frequencies. Figure 2.b.1 shows the frequency responses of the 13 filters as well as their combined frequency response.

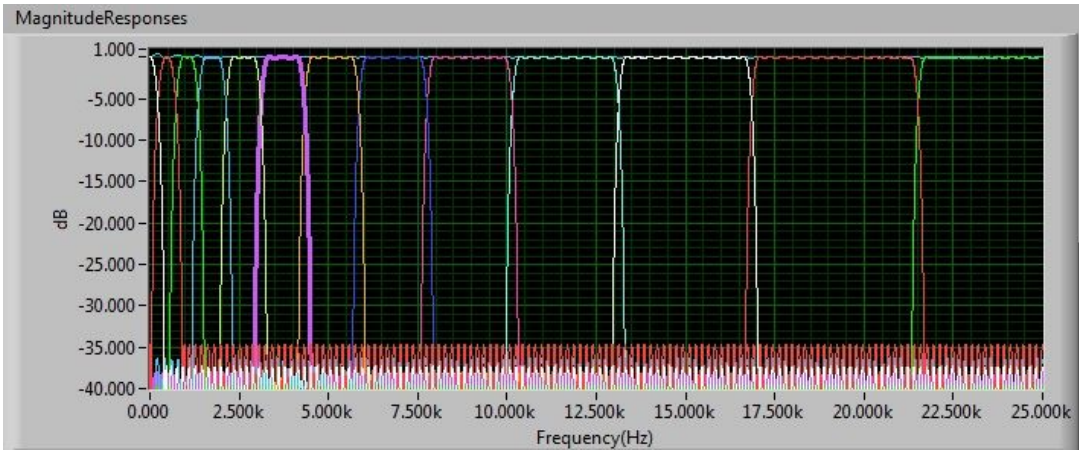

Figure 2.b.1 The Frequency Responses of the 13 Filters

Multiplying the frequency spectrum of a song by one of these frequency responses does not change the amplitudes of the frequencies in the pass-band by very much, but it cancels out all the other frequencies. Any attenuation applied to the resulting signal will effectively apply only to the frequencies in the pass-band. Because multiplication in the frequency domain corresponds to convolution in the time domain, the signal is filtered by convolving the incoming samples with the time-domain representations of the filters. An in-browser LabView tool provided by Professor Richter on the course website was used to generate the coefficients, or "taps," representing the filters in the time domain.

The equalizer uses finite impulse-response (FIR) filters. IIR (infinite impulse-response) filters typically require slightly fewer computations, but they distort the phase. In this application, the filter outputs must be linear-phase to avoid distorting the sound. While IIR filters can be made linear-phase, they then require more computations than FIR filters, so FIR filters are used.

c. Numerical Representations

All of the numbers used by the equalizer are in fixed-point format, since this is a simple way of handling decimal numbers. The taps and outputs are represented with one number in the units place and 15 numbers after the radix point (f1.15). The attenuations values are represented in the format f2.14 so that they can reach the value of 1. The f2.14 representation has a weight of -2 in the most significant bit and +1 in the next most significant. The remaining bits are powers of ½. We can then represent an exact value of 1 in f2.14 as 0x4000. When numbers with f2.14 and f1.15 formats are multiplied, the result is f3.29. This number can be converted to f1.15 by selecting the bits 29 through 15 around the same radix point.

d. Serial Communication

Aside from the hardware, this project also required the creation of a Windows application to allow the user to select the attenuation for each frequency band. LabVIEW was used to create the application. It needed to convert slider values that the user could choose into ASCII (American Standard Code for Information Interchange) representations that were then written out to the serial port. Embedded software on the FPGA side receives the attenuation values and places them in the attenuator module, a set of registers available to the filter. Once the filtering convolution and attenuation multiplication are complete, the results are simply added to reconstruct the sound.

e. Applications

An audio equalizer has several useful applications. It can be used to tweak the balance of different sounds and instruments to improve a recording. Some musicians and mixing artists also use equalizer for stylistic reasons. A musician might remove all the bass frequencies from a part of the song to create a certain artistic effect. Finally, the equalizer can remove high frequencies that most listeners cannot hear. Because the sound signal now contains less information, it can be further compressed for more efficient transmission and storage.

f. Abbreviations

Several abbreviations are used throughout this report. They are either defined in parentheses or mentioned in the list below:

MB: MicroBlaze processor

Mux: Multiplexer

TB/tb: Testbench

r: register

wr: write

rd: read

C: Controller

Atten: Attenuation

FSM: Finite state machine

RAM: Random-access memory

BD: Block diagram

I/O: In-out ports

GUI: Graphical User Interface

SPI: Serial peripheral interface

dB : Decibels

Eq_controller: equalizer controller module

MSB: most significant bit

SDO: Serial data out

SDI: Serial data in

SCK: Shift clock

CS_: Chip select (active low)

3. Design

a. Attenuator

The attenuator is a new module, designed for the equalizer. It consists of 13 registers and an Axi4Lite supporter. A block diagram of the attenuator is shown in Figure 3.a.1 below.

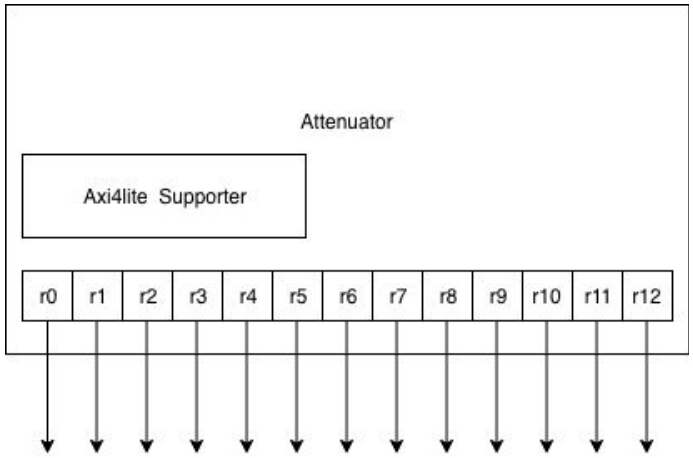

Figure 3.a.1 Block diagram of the attenuator module.

Each register is individually addressable by Axi4Lite supporter, and holds the f2.14 attenuation value for each of the 13 filters implemented. The initial value for each register was 1, or 0x4000 in f2.14, to meet the specifications for the lab assignment. The output of each register was made external to the attenuator module, as show by arrows in the figure, to be made available as an input to FIR filter module. This design was chosen so that the Microblaze could write attenuation values independently of the Eq_controller module. In the other designs we explored the idea of communicating new attenuation values to registers registers directly held in the FIR_filter. We ultimately decided against this design. The described design would require that values be sent in sync with the Eq_controller module. In our opinion, the synchronization required more overhead and testing than an independent module, and so so we created the attenuator module.

Internal Signals

wr: write signal from the Axi4Lite bus

wrAddr: 6 bit write address from the Axi4Lite bus

wrData: 32 bit data word from the Axi4Lite bus

rd: read signal from the Axi4Lite bus

rdAddr: read address from the Axi4lite bus

rdData: 32 bit data word from the Axi4Lite bus

External Signals

attenOutput#: The output reg assigned the value of register *n* at position *n-1* within the register array. These regs make the attenuation value external to the attenuator module for the FIR filter to use.

Registers

reg [15:0] attenD[0:12], attenQ[0:12]: An array of 13 different registers that are 16 bits wide. Each register holds the attenuation f2.14 value for filter *n* at position *n-1* within the array. Registers within the array are write addressable by the lower 4 bits of the wrAddr sign and its offset from 0. For example, wrAddr 0xA address filter 10 at position 9 in the array.

b. Parallel FIR_Filter

The parallel FIR filter used was based off the FIR filter designed in Lab 1 by Ethan Snow and Tom Wang. The original filter was designed as a single filter with a programmable number of taps and tap values. A detailed report on the implementation of the original FIR filter can be found in the Lab 1 report of Ethan Snow and Tom Wang. The original FIR Filter had to be modified for our equalizer because it required 13 filters on 2 channels and attenuation value input. This section will focus on the design changes to meet these new requirements.

One design solution would be to create a 26 instantiations of the existing FIR filter. This design was explicitly prohibited in the design specifications because it would have been ineffective for several reasons. First, there would be redundant tap values for each channel. Second, it would have taken considerable effort to ensure that these individual modules acted in sync with one another. For these reasons we decided to modify FIR filter module instead of using the original filter.

The first change to FIR filter was to add signed attenuation inputs. These values would come as outputs from the attenuator module and have the f2.14 weighting for each filter. In order to make addressing values easier, each input was assigned to a position within a signed reg array called attenInput. Each attenuation was positioned with the AttenInput array to match its numbering. This made it possible to multiply the attenuation values by the convolution results in a for-loop, making the code cleaner.

The next modification was to add block RAMs. We added 12 new filter block RAMs to used to store the tap values of each filter, 13 in total. We also added another RAM to store the the input values of the second channel. This gave us 15 RAMs in total, each 16 bits wide and 300 words deep. The required depth of our RAM was 279, the number of taps for each of our filters. The additional depth was added because the initialization of our RAM did not reach the highest position, and left it in an undefined state. The added size ensured that all elements of RAM used in FIR filter were initialized. The same goal could be accomplished by specifying a depth of 280, but 300 was chosen because it was a nice round number. In a design that had more memory pressure on the RAM, a depth of 280 would have saved space.

To program the tap values for each of the 13 tap RAM modules we used an address decode. The WR_TAPS address offset of 1 indicates a write to the tap memory.

The RAM module being addressed was decoded in the top 4 bits of wrData. This decode set the write enable high for that module, and captured wrData. The addressing within each RAM was taken care of internally by the module, as in the original Lab 1 implementation. The address was held in the aTaps register. The register, starting at zero, was incremented every time a tap was written. Once aTaps was equal to the number of taps within the module it was reset to an address of 0.

After programing the RAM tap values, we had to support dual channel input. Using an address decode, WR_INPUT_0 address would write to channel 0 input RAM, and WR_INPUT_1 would would write to the channel 1 input RAM. While writing to either channel, a register called select is set to 0 or 1 depending on the channel being written to. This select register is used in the convolution state of the FIR filter to determine which input RAM we read from, and perform the convolution on. The result register would hold the value of the convolution for the last input written. In an earlier design we simply switch channel on every other write to the filter. While this may have worked, in testing we revised the design to be addressable so that there was no synchronization conflicts with the dual channel support of Eq_controller.

After writing to either input, the FIR filter transferred to the convolution state: CONV. The convolution for each filter occurs in parallel with each other. The running sum of each MAC is held in the convSum array. The sum for filter *n* is held in at position *n-1* of the array. The filter remains in the CONV state as 13 MACS are performed every clock tick, until convCounter has reached the number of taps. On the last iteration of CONV step, we round the lowest bit up. We next had to weight the sums, accumulate the results, and round the final result. The rounding of individual filters and this final step could not be accomplished in the same clock tick. We added a new state called FINISH

for the purpose of completing these final operations. The revised FSM of FIR filter can be seen in Figure 3.b.1 below.

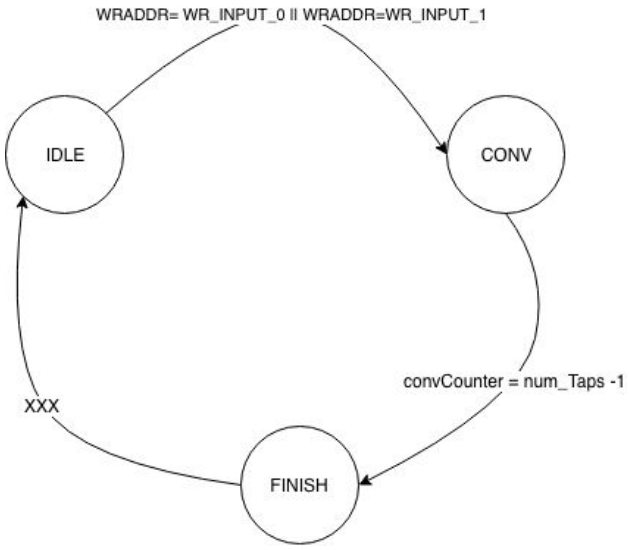

Figure 3.b.1 FSM of FIR Filter Module

In the FINISH state, each filter's convSum must be multiplied by its respective f2.14 attenuation value and added together with the other sums. The final sum be rounded and stored in the result register. In the tests of this state, we realized that the convSum register array was an unsigned register being multiplied with the signed f2.14 attenuation value array. This sign mismatch gave us the wrong multiplication results. To remedy this, we created a new signed register array called s_convSum, standing for signed convolution sum. This register array at position *n* was assigned bits [30:15] of register *n* in convSum array. Thus, convSum is a 32 bit value that is in f2.30. By assigning s_convSum 30:15 we retain the signed f1.15 value of the convolutional sums for each

filter. The signed convSum of each filter is then multiplied with the attenuation of that filter, summed into the input of the result register, and rounded. The read response of FIR filter had to take into account that the multiplication of the f1.15 s_convSums and f2.14 attenuations results in a f3.29 number. The required value formatting of the read response was f1.15. We change our read response so that the lower 16 bits of rdData were assigned bits 29:15 of the result register. The convDone signal is asserted in this state, not on the last iteration of CONV.

In addition, we added the convDone signal to the top bit of rdData in order to save a clock cycle. In the original design, a manager would read to a specific address to find the value of convDone. Once convDone was high, then the manager would have to read from a second address to actually get the value of convolution. By combining convDone and the convolution value into a single read response, the manager saves an entire read cycle.

ALL COMPONENTS MENTIONED ARE IN ADDITION TO THE ORIGINAL'S DESIGN UNLESS SPECIFIED

Internal Signals

attenInput*N*: Input attenuation values for each filter. There are 13 attenuation inputs labeled 0-12. These signed inputs come from the attenuator module.

inputsRam[]: Array of 2 RAMs, one for each input channel. This array of RAM repaces inputRam from the original module. This array of RAMs hold the input values for each channel, that are needed for the convolution.

tapsRam[]: Array of 13 Rams, one for each filter. This array of RAM replaces taps RAM. This RAM holds the tap values or each individual filter. The filter number matches the position of the position of its tap values, i.e. filter 0 has its

tap values stored in tapsRam[0]. Each filter is addressed by the upper four bits of wrData.

External Signals

reg signed [15:0] attenInput[0:12]: 16 bit wide signed regs array of size 13. The array holds the input values for each attenInput. The position in the array corresponds to the filter number. Values are assigned continuously. Array format is used so that each each attenuation value can be addressed using a for() loop.

reg [12:0] weTaps: Write enable signal for taps ram. The reg is 13 bits wide, each bit is connected to a single tapsRam. During a write to addr WR_TAPS, weTaps is set high so that wrData is captured as a tap value. The write enable is selected by top 4 bits of wrData.

reg [1:0] weInputs: Write enable signal array for channels input ram. The write enable for channel 0 corresponds to weInputs[0] and wrInputs[1] for channel 1. weInputs is set high in a write to either address WR_INPUT_0 or WR_INPUT_1, so that the wrData value is captured in the inputsRam.

Registers

reg [8:0] aInputsD[1:0], aInputsQ[1:0]: functions the exact same as aInputs register from the original module. There are simply two registers so that each channel can have its own address register.

reg [31:0] convSumQ[0:12], convSumD[0:12]: register to the the convolution sum. Functions the same was as convSum in the original FIR Filter module. Expand so that there are 13 registers for 13 filters.

reg signed [15:0] s_convSumQ[0:12]: 13, 16 bit wide signed register array. Assigned bits 30:15 of the convSum register array. Constant assigned so that the

the convSum[30:15] is assigned as the input to s_convSum register. This register is needed perform the signed multiplication with the attenuation values.

reg [0:0] selectD, selectQ: Holds the select value for which channel was last written. Used to select which inputTaps ram is selected to be used in the convolution.

Address Decode

NUM_TAPS (0): programs the number of taps for all filters

WR_TAPS (1): Write a new a new tap value to a filter. Addressing within the filter is taken care of internally. Filter selected by the top 4 bits of wrData

WR_INPUT_0 (2): writes a new input value to the input RAM of channel 0. Immediately begins convolution after doing so.

WR_INPUT_1 (3) :writes a new input value to the input RAM of channel 1. Immediately begins convolution after doing so.

c. SPI

The SPI peripheral was designed as a generic peripheral to facilitate communication between our MicroBlaze processor and both the LTC1865L (ADC) and LTC1654 (DAC). This peripheral was largely taken from Lab 2. The design of the SPI can be found in the SPI lab report of Nathan Jarvis and Lisette Torres. However, in testing Eq_controller we found a bug in the peripheral. In the original design, the valid bit was set high 2 clock ticks before the rdData was actually available. This did not appear in testing for the SPI peripheral because we issued individual read commands from MB. These read commands were spaced out enough so that they never presented a problem with reading valid values from the rdData register. In Eq_controller, we issue read

commands continuously. This means that we were reading ‘valid’ values before the SPI had actually asserted a valid value on the bus. This was fixed by adding a 2 clock cycle output delay to the valid signal. The delay was added using a new register named validRaw_D, validRaw_Q. The original valid register would act as an input to this register, and the rdData would have the Q output of the validRaw register in the top bit. This forced the assertion of the valid bit and a valid value on the bus on the same clock tick.

*****ALL COMPONENTS MENTIONED ARE IN ADDITION TO THE ORIGINAL’S DESIGN UNLESS SPECIFIED*****

Internal Signals

Same

External Signals

validRaw_Q, validRaw_D: One register bit register that holds the valid bit. The input of this register is valid_Q from the original SPI design. The output of this register is the top bit of rdData. This register guarantee a two clock delay from the assertion of valid_D, to the assertion of validRaw_Q. This delay synchronizes the the valid bit and a valid reading from the peripheral on the same clock tick.

Registers

Same

Address Decode

Same

d. Bus Multiplexer (Bus_Mux)

Bus_Mux takes in two Axi4Lite buses and outputs a single Axi4Lite bus. Bus_Mux is shown in figure 3.d.1 below.

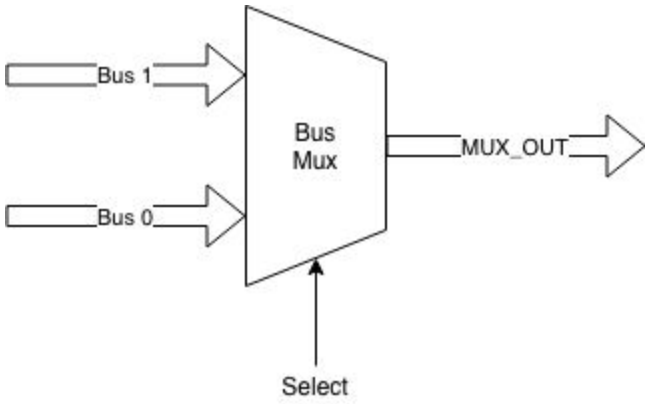

Figure 3.d.1 Diagram of the Bus_Mux module. MUX_OUT is ether BUS_0 or BUS_1

The output bus in one of the input buses selected by the select input signal. Select is a one bit input wire. If select is 0, then Bus 0 is output as the MUX_OUT bus. If select is 1, then Bus 1 is output as MUX_OUT bus. This module was used to select between input buses for the FIR filter module. The MB had one input bus and the Eq_controller had the other. Select was was the current state of the Eq_controller. If the Eq_controller was not in the INIT state, then select was 0 and MB bus was connected to FIR_filter. In any other state, select was set so that the Eq_controller was connected to FIR filter. This was designed so that we could easily program the tap values in FIR filter directly by the MB. These taps had to be programmed while the controller was in the INIT state and disabled. As an alternative we could have the FIR filter addressable from the controller's manager, using an address decode. We chose the former design. The main benefit of this design is cleaner C code for the address the filter tap values. The C code could use the

base address of the MB bus manager as opposed to an address decode. In a design that prioritizes efficiency, the later design would be chosen. The latter design reduces the I/O usage of the MB and makes overall design of the Eq_controller more straightforward.

Internal Signals

None

External Signals

S_XXX_XXX_0: input Axi4lite Supporter signals for Bus 0

S_XXX_XXX_1: input Axi4Lite Supporter signals for Bus 1

S_XXX_XXX_MUX_OUT: output Axi4lite supporter signals that is either Bus 0 or Bus 1 based upon the value of the input select.

Select: One bit input wire. If select is 0 then Bus 0 is output as MUX_OUT. If select is 1, then BUS 1 is output as MUX_OUT

Registers

None

Address Decode

None

e. LabView GUI & Serial Communication

The LabView GUI and Serial communication with the Microblaze were created so that our equalizer could be interactive. The GUI allows user input that set the attenuation values for each filter, while the serial communication actually transfers these new values to the MB so that they are written to the attenuator module. The LabVIEW GUI was based of the LabVIEW example Continous_Serial_Write_and _Read.vi. The example establishes a UART serial connection. We modified the example to fit our

needs. The block diagram of the final LabView program can be seen below in figure 3.e.1.

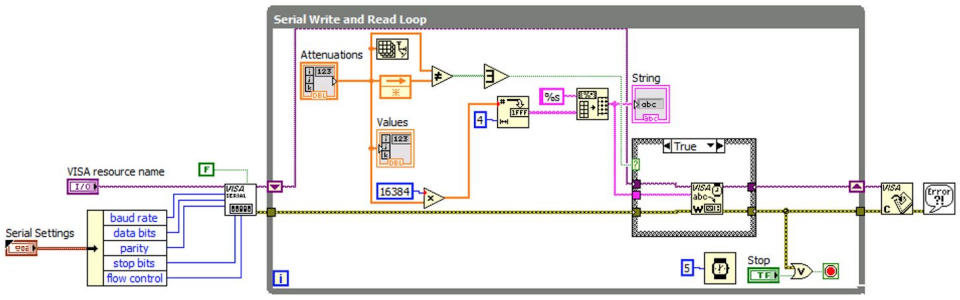

Figure 3.e.1 Block Diagram of LabView Program

The VISA blocks, and serial settings are all from the original Continous_Serial_Write_and _Read.vi program. These blocks establish the serial communication and write values to that channel. We removed the serial read VISA block from the program because we would only need to write need attenuation values in our program.

The program has an array of sliders that hold the attenuation values of each filter. Each of them can go from 0, no attenuation, to 1, max attenuation. These values are expressed as doubles in the 'values' box shown in figure 3.f.c. These double values must be converted to f2.14 before they can be sent to the MB. We did this be multiplying the slider values by 2^14, 16384, so that they the slider values would scale to f2.14. A slider value of 1 would equal 0x4000.These values were then converted to 4 digit hex strings. All of these strings were then concatenated in a longer string that was communicated to the MB. In our original design, we wished to only write the values of the sliders that had changed. This would have reduced latency in writing attenuation values. We decided

against this because we found in testing that the serial communication was not a noticeable bottleneck. This long string expresses the attenuation of every filter in hex and is shown in the String box of Figure 3.e.2. The write command for the string was originally triggered by a button. In our design, write is triggered based on a change in the slider values. The design change allowed people to change attenuation values as they moved the sliders. This was accomplished using a feedback node. The feedback node was used to compare the current value of the sliders with the previous. If the values differed then the write signal was sent high, and the new attenuation values were written.

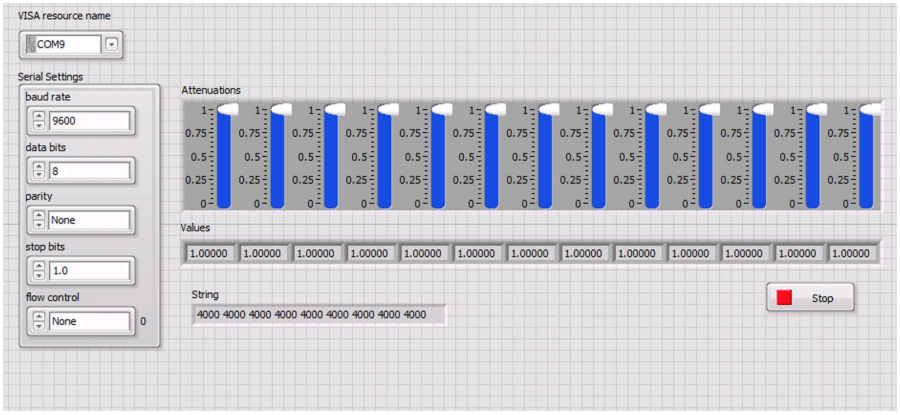

Figure 3.e.2 LabView GUI interface

The Serial communication on the MB side was based off the Vivado xuartlite_polled_example program. We used this program as a basis to establish a serial connection and to receive data. We made various modifications to the program for our application. After receiving the string of attenuations from LabView we had to convert the string to actual f2.14 numbers to be written to the attenuator. Due to a misunderstanding of the serial communication, we had the receiver buffer of our serial communication be of type char. Our impression was that only chars could be

communicated over the serial channel, and not decimal values. This assumption was incorrect but we were able to create a functional interface in spite of this. Under this assumption, we had to convert the hex characters of each attenuation into a f2.14 value. This conversion could have been avoided if we had left the receive buffer of type u8, like in the example. In future designs, this would be a great improvement. To perform the char conversion we created a function called hexadecimalToDecimal(). This function was based of a function on the geeksforgeeks website, and is cited at works cited section. This function was modified and developed by Nathan Jarvis and Kyle Cepeda for the equalizer application. The function parses through each character in the string and uses its ASCII offset to compute its decimal value. The decimal values are then shifted to the left by their byte position to obtain the decimal value of the entire string. This decimal value is then attenuation value sent to the attenuator module to modify the attenuation values for the FIR_filter. If we had kept the receive buffer as u8 then we could have received the bytes instead of chars. To convert to the f2.14 format would simply consist of concatenating to 2 bytes together. This would have been a much faster and cleaner solution than the conversion of chars.

f. Eq_controller

The equalizer controller or Eq_controller managed the flow of data between components of the equalizer. The controller consists of two SPI peripherals, an attenuator module, an FIR Filter module, 3 Axi4Lite bus modules, and one Axi4Lite supporter, and a Bus_Mux module. There is 1 SPI peripheral for the ADC, and 1 for the DAC. The FSM diagram of the controller is shown in figure 3.f.1

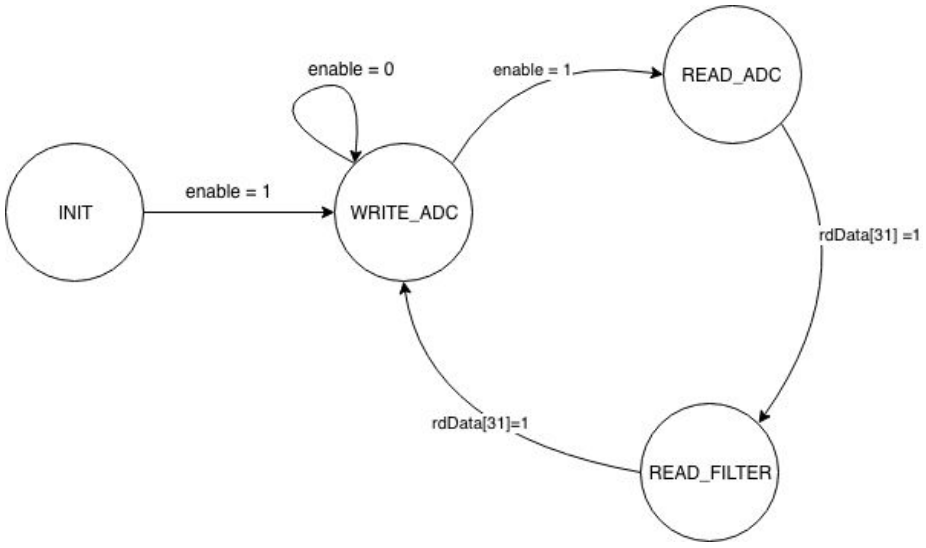

Figure 3.f.1 FSM of Eq_controller

The controller begins in the INIT state with the enable register set low. In this state the MB can write configurations and enable the SPI peripherals. These configurations are set by the MB using the CONTROLLER Bus. This functionality is implemented using an address decode and is only available while Eq_controlller is in the INIT state. The decision was made because all of the following address decodes do not need to be available outside of the initial set up of the controller modules. The addresses CONFIG_DAC and CONFIG_ADC each configure a SPI peripheral with the number of data bits and sample rate. When either of these address are written to, the wrData_CONTROLLER signal is passed to the configuration address of the corresponding SPI. The data is encoded with the lower 5 bits as the number of data bits and upper 15 specifying the number 30 MHz clock ticks per sample. The configuration of

a SPI module is elaborated in the Lab 2 report by Nathan Jarvis and Lisette Torres. In this design the MB would be programming these configurations from the C code, as in Lab2. As an alternative, we could have hard coded this configurations to be written by the controller. This was decided against because it was a less flexible design. If another ADC or DAC was chosen to interface with our SPI, then we would have to rebuild the entire project to change the configuration for the SPI. In the former design we would only have to change the C code on the MB to support the new configuration. The CONTROLLER bus supports an address decode for ENABLE_ADC and ENABLE. Any write to this address ignores wrData_CONTROLLER and writes the enable bit of the corresponding SPI high. With enable bit high, the SPI begins sampling. We chose to make the enable for each SPI as permanent to protect the user. There would be no need to disable the SPI modules individually once they have been enabled. This could cause complications with the Eq_conroller FSM waiting on a value from a disabled SPI, causing an infinite loop. We therefore went with the former design. The next address decode is DAC_MODE. DAC_MODE is used to configure the mode of the DAC into fast mode. This address decode pulls the wrData_CONTROLLER signal into the write buffer of the SPI. The data is then written out to the DAC itself, so that it is configured for fast mode. There is no need for this mode configuration on the ADC. It should be noted that the DAC must be enabled for the data to be written out. This is reflected in the C code for the equalizer. A more detailed explanation of DAC fast mode can be found in the Lab 2 report of Nathan Jarvis and Lisette Torres. The only other address decode is the enable bit address decode of the Eq_controller fsm, and it is available in all states of the FSM. This address decode and sets the enable bit of the Eq_controller FSM. The FSM remains in the INIT

state until the enable bit is set high. Once the bit is set HIGH the FSM transfers to the the WRITE_ADC state.

The FSM writes the channel select bits to the ADC for the channel being read. The channel is set by the fist two bits written out by the ADC. Bits 01 corresponds to channel 0 and 11 corresponds to channel 1. The ADC channel we are currently writing to is held in the cw_ADC register. The value of this register switches between the select of channel 0 and channel 1 after each convolution. The specifications for this select can be found in the LTC1654 ADC data sheet or the introduction section of the Lab 2 report of Nathan Jarvis and Lisette Torres. The FSM will stay in the WRITE_ADC state if the enable register is low. This is to add functionality for enabling and disabling the equalizer. We explored folding the WRITE_ADC state into the READ_FILTER state by writing the channel select in the final iteration of READ_FILTER. We decided against this design for several reasons. First, we should point out that it would not be possible to combine the READ_ADC and WRITE_ADC states because our implementation of the Axi4Lite bus does not support a simultaneous read and write on the same bus.The main reason to keep WR_ADC as a separate state was to cleanly support enable/disable functionality. If we were to have an enable/disable in state READ_FILTER then FSM would begin in the middle of read or write cycle. To avoid this garbage data, we created the WRITE_ADC state so that the enable/disable function would begin at the start of a sample every time.

After transfering from the WRITE_ADC state with enable high, the FSM enters the READ_ADC state. The READ_ADC state reads a value from the ADC SPI on the ADC manager. The SPI will have already been configured for sampling rate and the number of data bits in the INIT state of the FSM. The controller continuously sends the

read signal to the controller until a valid value is read. The SPI signifies a new valid value with a 1 in the 31st bit of the readData. In the same clock cycle that a valid value is read from the ADC, the value is pushed to filter inputs. Before it is pushed to the filter, the unsigned ADC value is converted to signed value for the FIR filter by flipping the MSB . The value is input to the inputRam of the channel from which it was read. This is accomplished by checking the value of cw_ADC and choosing the appropriate write address of FIR filter. The FSM reads from the filter in the READ_FILTER state until a valid convolution of the input value is read. A valid value is read from the filter when top bit in the read value is a 1. When a valid value is read several actions are performed in the same clock tick. First, the signed 16 bit filter value is converted to unsigned for the DAC by flipping the MSB. Next, the DAC control words are appended onto the front of the 16 bit value. The DAC control words are specify the DAC written to and action. Each word is 4 bits and the entire control signal is held held in the 16 bit wide cs_DAC register, using the lower 8 bits. The register specifies a load operation, 0101, and the channel to load data (DAC A:0000 or DAC B:0001). The DAC chosen is associated with a single channel of the ADC. The 24 bit data word is then written to the DAC SPI. In the same clock tick, the control words for both the DAC and ADC flip the ADC channels and DACs based off their current values. This is done in one step to ensure that one ADC channel is only used with one DAC, avoiding any channel crossover. The FSM then moves back to the WRITE_ADC state, where the new channel is written to the ADC. The cycle of reading values from the ADC, filtering them, and writing them to the ADC is continuous as long as the enable register is high. A block diagram of the Eq_controller is shown below in Figure 3.f.2.

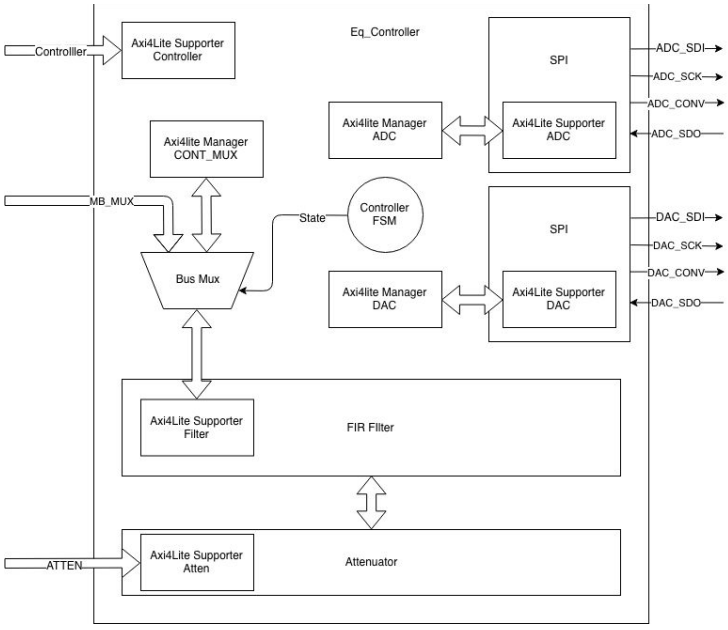

Figure 3.g.2 Block diagram of Eq_controller

From the block diagram it is clear there are several external signals and modules that have not been explained. The CONTROLLER bus signal has already been explained. The next bus is the ATTEN bus. The atten bus is a pass through bus that is connected to the attenuator module. As explained earlier, the attenuator was intended to directly addressable by the MB. While attenuator could be instantiated as a separate module, outside of Eq_conroller, this was decided against. Connecting the attenuator as internal module to the controller is much easier. We could write the structural verilog to make the exact connections we wanted. If we had the attenuator external to the controller we would be forced connect it through block design. We found that using the block design to connect components was much harder to debug and less intuitive than writing verilog.

The MB_MUX is an input to the Bus_Mux module. When the controller is n the INIT state the MB_MUX bus is connected to the FIR Filter module. In any other state it is disconnected. This was designed so that the MB could address the FIR filter directly

when writing tap values. In hindsight, a simple address decode within the controller would have been sufficient. Using an address decode we would not need the MB_MUX bus, or the Bus_Mux, reducing I/O utilization and LUT pressure. Once the FSM has left the INIT state, the controller has full control over the FIR filter.

The only other external signals are the SPI signals. The SPI signals are the SCK, SDO, SPI, and CONV/CS_ signals used by the SPI to interact with the the physical DAC and ADC hardware. These must be made external to the controller so that they can be connected to physical pins on the board. To mitigate confusion when assigning pins, the SPI signals are labeled for either the ADC or the DAC. The pin assignments are made in the .xdc file for the project, and are based off the schematic provided to us in Lab 2. This schematic can be found in the introduction.

Internal Signals

rdValue: reg used to hold values as they are passed between the ADC SPI, filter, and DAC SPI.

select: select signal for the Bus_Mux. The select is of value 1, selecting the controller bus, when the controller is not in the INIT state. It is 0 in the init state, selecting the MB_MUX bus.

SPI ADC simple bus

wr_ADC: write signal from the Axi4Lite bus

wrAddr_ADC: 6 bit write address from the Axi4Lite bus

wrData_ADC: 32 bit data word from the Axi4Lite bus

rd_ADC: read signal from the Axi4Lite bus

rdAddr_ADC: read address from the Axi4lite bus

rdData_ADC: 32 bit data word from the Axi4Lite bus

SPI DAC simple bus

wr_DAC: write signal from the Axi4Lite bus

wrAddr_DAC: 6 bit write address from the Axi4Lite bus

wrData_DAC: 32 bit data word from the Axi4Lite bus

rd_DAC: read signal from the Axi4Lite bus

rdAddr_DAC: read address from the Axi4lite bus

rdData_DAC: 32 bit data word from the Axi4Lite bus

Controller Supporter simple bus

wr_CONTROLLER: write signal from the Axi4Lite bus

wrAddr_CONTROLLER: 6 bit write address from the Axi4Lite bus

wrData_CONTROLLER: 32 bit data word from the Axi4Lite bus

rd_CONTROLLER: read signal from the Axi4Lite bus

rdAddr_CONTROLLER: read address from the Axi4lite bus

rdData_CONTROLLER: 32 bit data word from the Axi4Lite bus

Controller Manager to FIR Filter simple bus

*****note this bus is not connected when select signal is 1*****

wr_CONT_MUX: write signal from the Axi4Lite bus

wrAddr_CONT_MUX: 6 bit write address from the Axi4Lite bus

wrData_CONT_MUX: 32 bit data word from the Axi4Lite bus

rd_CONT_MUX: read signal from the Axi4Lite bus

rdAddr_CONT_MUX: read address from the Axi4lite bus

rdData_CONT_MUX: 32 bit data word from the Axi4Lite bus

External Signals

S_XXX_XXX_CONTROLLER: Axi4lite supporter signals for the controller

S_XXX_XXX_MB_MUX: Axi4lite supporter signals for the FIR filter. This bus is connected to the FIR FIlter only when select is 0.

S_XXX_XXX_ATTEN: Axi4lite supporter signals for the attenuator module.

ADC_SCK, ADC_SDI, ADC_SDO, ADC_CONV: Signals for the SPI peripheral to communicate with the ADC. This signals are assigned to pins on the actual board.

DAC_SCK, DAC_SDI, DAC_SDO, DAC_CONV: Signals for the SPI peripheral to communicate with the DAC. This signals are assigned to pins on the actual board.

Registers

enableD, enableQ: holds the enable status of the controller FSM. The controller will not transition from INIT or WRITE_ADC state if this register is low.

cw_DAC_D, cw_DAC_Q: 16 bit wide register that holds the control word for the DAC in the lower 8 bits. The configuration changes to the other DAC after each write to the filter.

cw_ADC_D, cw_ADC_Q: 16 bit wide register that holds the control word for the ADC in the lower 8 bits. The configuration changes to the other channel after each write to the filter.

nextState, state: holds the current state of the controller FSM.

Address Decode

CONFIG_ADC(0): Write a 32 bit word to the configuration address of the ADC SPI controller. The lower 5 bits include the number of data bits and the next 15 measure the number of clock cycles between samples.

CONFIG_DAC(1): Write a 32 bit word to the configuration address of the DAC SPI controller. The lower 5 bits include the number of data bits and the next 15 measure the number of clock cycles between samples.

ENABLE_ADC(2): Sets the the ADC SPI controller enable bit to high. Ignores wrData

ENABLE_DAC(3): Sets the the ADC SPI controller enable bit to high. Ignores wrData

ENABLE_EQUALIZER(4): Sets the enable register of the Eq_controller to the LSB of wrData

DAC_MODE(5): Writes a 32 bit word to the write buffer of the DAC SPI. The word specifies the configuration mode of the DAC. For more information on DAC modes refer to the Lab 2 report of Nathan Jarvis and Lisette Torres. The word will NOT be written out of the SPI if it is not enabled.

g. Equalizer

The goal of this project was to create an interactive audio equalizer. The block diagram for the equalizer is shown in figure 3.g.1.

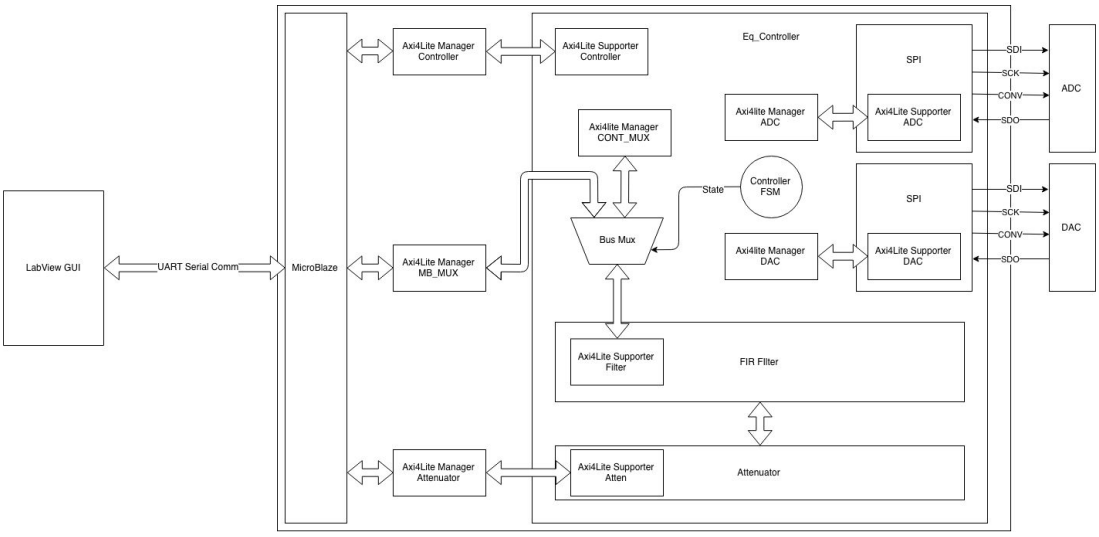

Figure 3.g.1 Block Diagram of Equalizer Design

The Equalizer is made of three main components: a LabVIEW GUI, the MB, and the Eq_controller. The majority of the equalizer's functionality is in the Eq_controller module. The Eq_controller module reads data from the ADC, passes it through an FIR Filter module, and writes it out to the DAC. The FIR Filter module implements 13 different FIR filters simultaneously. There is an attenuation input to the FIR module for each filter that weights the value of each filter in the final filter output. The attenuation values are held in registers in the attenuation module. These registers are addressable by the MB through an Axi4Lite bus. To implement the interactive feature of our equalizer, we had to create a way to write new values from a GUI to this attenuator module. This was accomplished by creating a UART_lite serial communication between the MB and the LabVIEW GUI. Attenuation values were written by the LabVIEW application to the MB, parsed by the MB, and then written to the attenuator module. This design implemented our audio equalizer. Any module or design improvements will be found in individual module sections.

4. Operation/Testing

a. Parallel FIR_Filter

The testing of the FIR filter module consisted of comparing the output of the FIR module to a C code simulation of the filter. This filter is not the original filter, but the modified filter that implements thirteen different filters. This module was developed without the attenuator module. The testbench block diagram for this testing can be found in the figure 4.a.1 below.

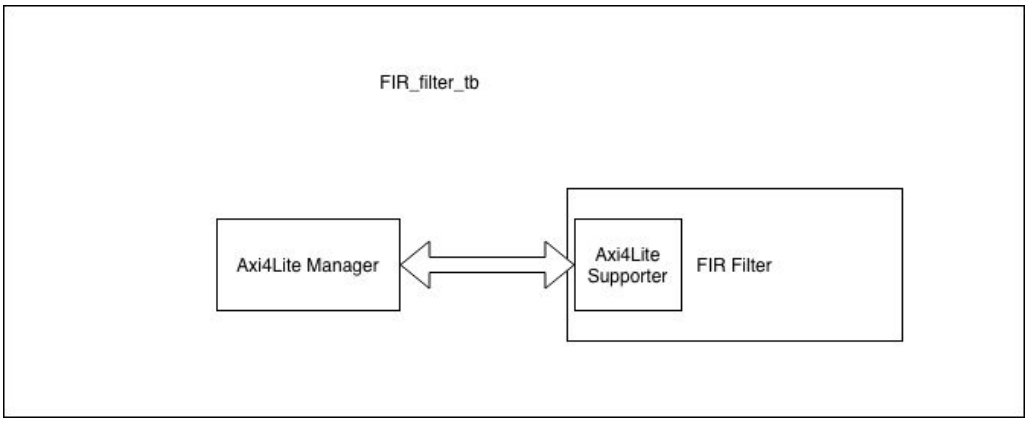

Figure 4.a.1 Block diagram of the FIR Filter TB

The C code simulation was based off the C code simulation of Ethan Snow from Lab 1. The C code was expanded to implement 13 filters instead of one. The output of this C code was verified by checking the gain at various frequencies. By comparing these gains to the gains web application predictions, we could confirm our C simulation as accurate. We compared the values at a valley and a peak of the filter, and found the values to very close at both points. From this we concluded the C simulation was

accurate. The comparison can be found in Output_verification.xlsx in the test_vectors tabs. The program can be found in 4_a_main.c

The hardware testbench programmed the tap values into the FIR filter as the initial set up, then spoon fed values of a 7 kHz sine wave, sampled at 50 kHz, to the FIR filter, and read out the convolution values. These results were compared to the C code simulation of the same filter. The FIR filter module was tweaked until the output of the FIR filter exactly matched the output of the C code. The exact match of hardware and software simulation confirmed that our hardware implementation was correct. The exact FIR module used in this TB is not available because it was used to create the attenuated filter. The TB itself is available in the as filter_tb.v

b. Parallel FIR_Filter w/ Attenuator

The attenuator module was tested as part of the FIR filter. Testing was performed using a modified version of the C code used in the FIR filter verification. The main for this C code can be found in 4_b_main.c. The main for this C code was tested with full attenuation. The resulting test vectors were the exact same as vectors without attenuation. We assumed that our C code was functioning correctly at this point and moved onto the hardware testbench. The testbench block diagram is shown in Figure 4.b.1 below.

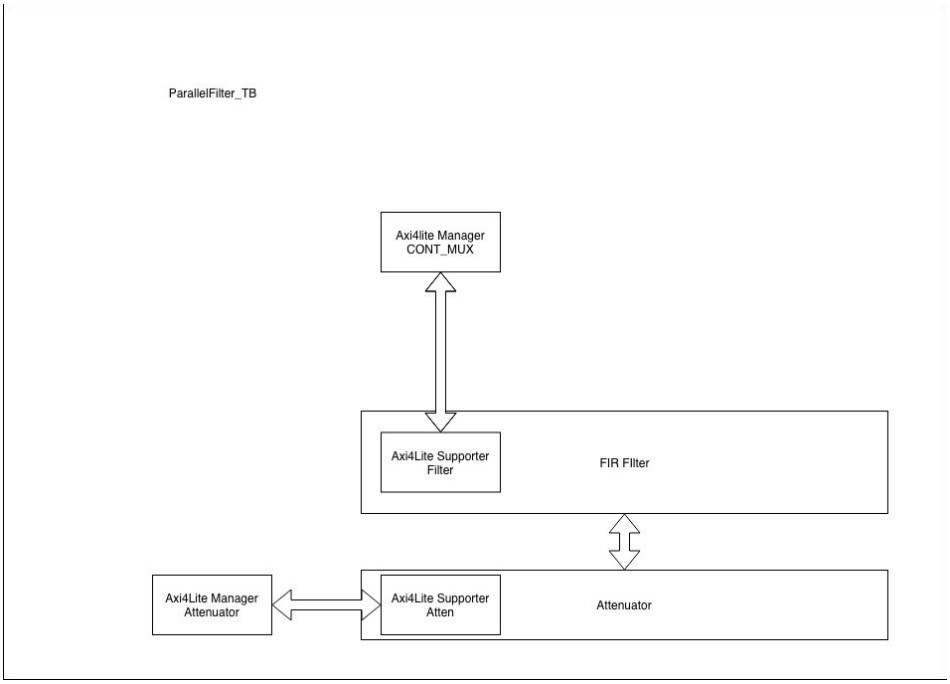

Figure 4.b.1 The Testbench for the Parallel Filter

In the the first test for this testbench we ran the exact same test as before. The reset values of the attenuator module are all 1 in f2.14. This meant the expected output would be the exact same as before. We tested and debugged the FIR_filter module & attenuator module until the output exactly matched the simulation. At this point we moved onto the second phase of testing in which we halved the attenuation values. We then reran the C simulation to create new test vectors. We modified the TB so that we wrote all the attenuations as half, and monitored the output. We modified the modules, until the output of testbench was the exact same as the C simulation. The testbench used can be found in filter_atten_tb.v file.

c. Bus Multiplexer (Bus_Mux)

The BUS_MUX was used to select between two separate busses at a single output. The BUS_MUX module was tested in simulation with a simple testbench, the structure of which is shown in Figure 4.c.1.

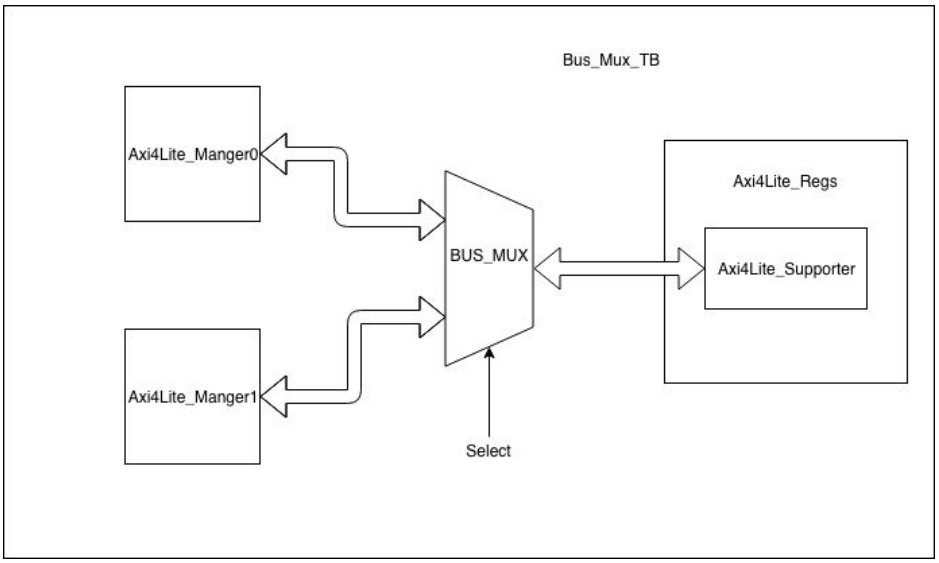

Figure 4.c.1 The Structure of the Bux Mux Testbench

The value 17 was sent to the WDATA AXI signal by the manager tied to select=0, and the value 16 was sent by the other. During two write cycles, both managers asserted write. Figure 4.c.2 shows the results. For the first write, since the select signal was 0, the value 17 was written and latched into the first register. For the second write, the select signal was set to 1, so the value 16 was written and latched into the register.

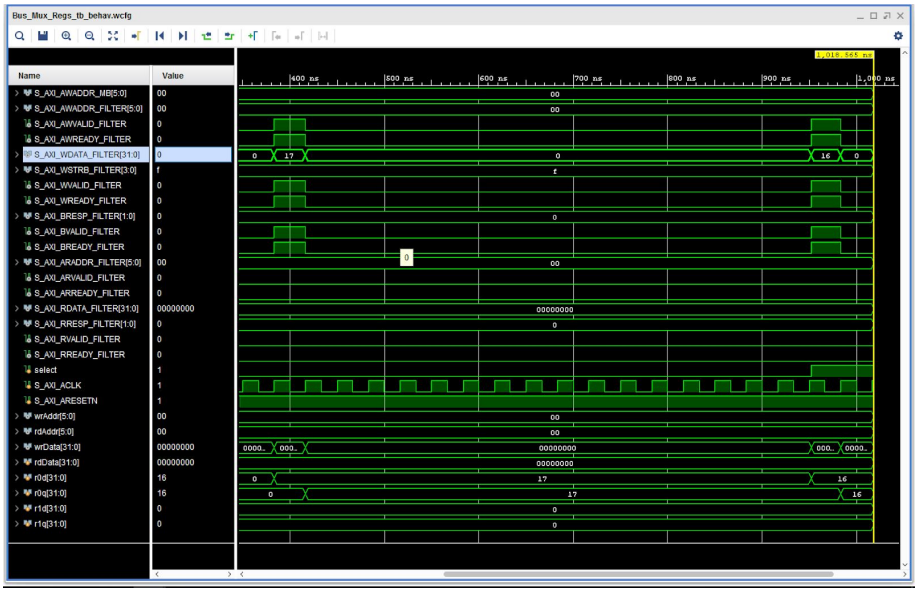

Figure 4.c.2 The Waveform Produced by the Bus_Mux Testbench

d. Eq_Controller

The overall controller module was tested using the ADC and DAC tester modules provided by professor Richter. Three Axi4Lite managers represented the MicroBlaze. They issued commands to initialize the SPI modules and enable the controller. Once the controller was enabled, the ADC tester fed in the same sine wave values that had been used for previous testing. The controller then processed the values with zero attenuation and wrote them out to the DAC tester. The outputs from the DAC tester were compared to the values from the filter module and the C simulation at various attenuation values. In the course of testing, it was discovered that changes needed to be made to the filter module. While the end results coming from the DAC tester did not match exactly, they were verified to within 0.0031. The difference probably came from some small difference in the way the values were calculated, either in the Verilog or in the C simulation. In either case, the difference did not appear to significantly impact the equalizer's function.

Output files were written from the DAC tester to provide easy access to the complete test vectors.

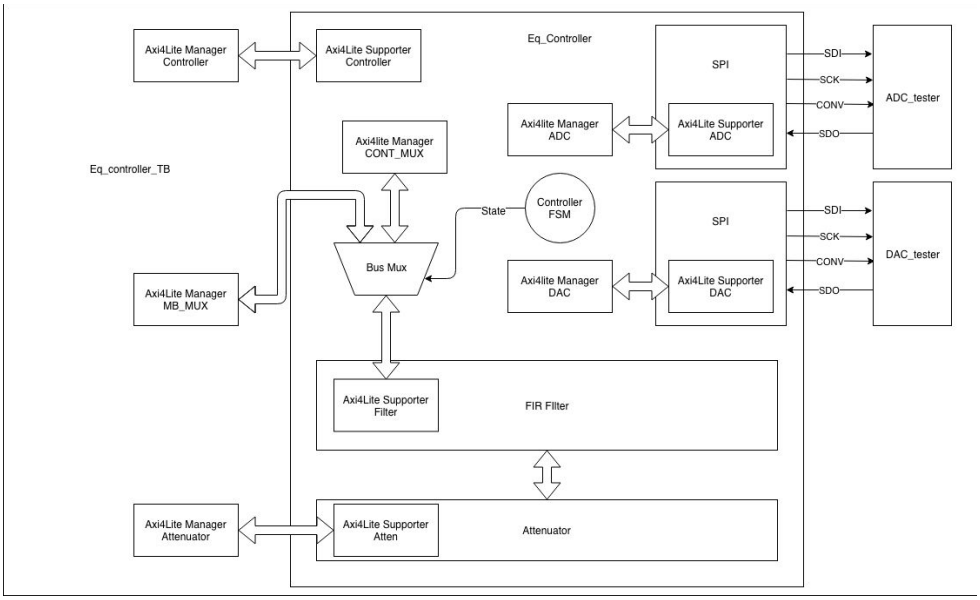

Figure 3.d.1 The Eq_controller block diagram

In addition to the output verification, we had to to verify that we meet timing. As mentioned in the introduction, the equalizer had to sample at 100 kHz. Based on the MB 30 MHz clock we could have, at maximum, 300 clock ticks between reading a sample from the ADC until we write it to the DAC in order to maintain this sampling rate. We verified that we had in fact met this requirement using simulation. In our simulation, we counted 282 clock ticks between read a value from the ADC_tester and writing a value to the DAC_tester.

e. LabView GUI & Serial Communication Test

A simple test was performed to verify correct serial communication between the LabView windows application and the embedded C code. The C code was run on the MicroBlaze processor through the Vivado SDK in debug mode so that the values of

variable would be visible. A string of 13 attenuation values was written to the serial port, and the C code hit a breakpoint once it detected enough bytes in the serial buffer and exited its read loop to process the values. Figure 4.e.1 shows a screenshot from the windows application with the test values displayed in a display box. Figure 4.e.2 shows the SDK in debug mode receiving the values correctly in the RecvBuffer array. The factors array is also shown, and demonstrates the correct conversion of the character received into f2.14 values.

Figure 4.e.1 A Screenshot of the Windows App Writing Test Values

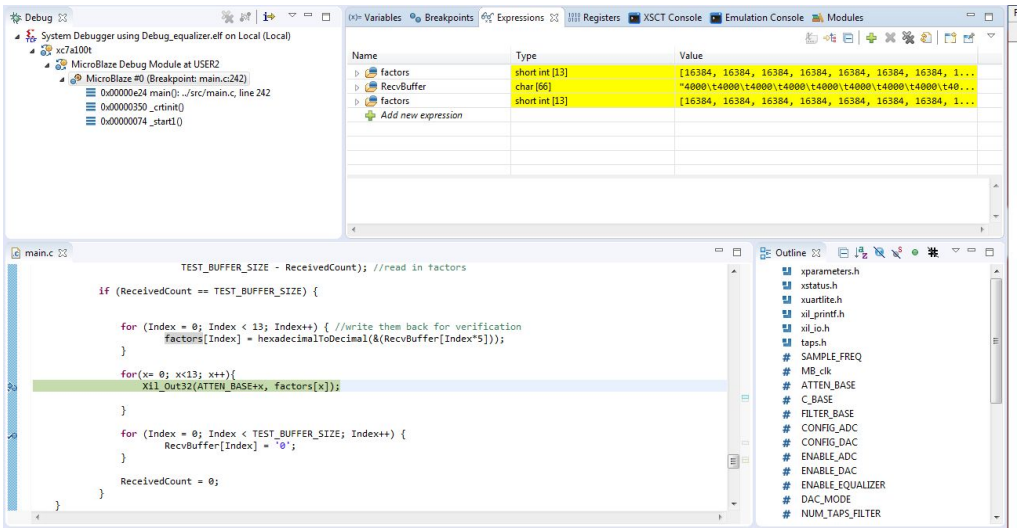

Figure 4.e.2 A Screenshot of the SDK in Debug Mode Displaying the Correct Receive Values

f. Equalizer Gain Verification & Audio Test

To verify in hardware that the device worked as expected, all attenuation values were first set to 1. Frequency values corresponding to peaks and troughs in the overall frequency response as determined by the LabView filter design application on the class website. The RMS output was recorded, since it remained relatively stable. The ratios of each RMS value to the next in dB should match the difference in the magnitude of the overall frequency response in dB at those frequencies. The table below shows the measurements taken and calculations done to verify the function of the equalizer. The fourth and last columns should ideally match. The values are close enough to demonstrate proper functioning. They do not match exactly due to nonideal conditions, such as noise. The ratios probably match the ideal differences less closely at higher frequencies because the difference becomes smaller and harder to distinguish from the noise.

Frequency (Hz)	Input (V)	RMS Output (V)	Ratio to Previous RMS Output in dB	Ideal Attenuation in dB	Difference from Previous Attenuation in dB
195	1	0.36		0.444	
450	1	0.34	-0.496471675	-0.092	-0.536
780	1	0.357	0.423785981	0.199	0.291
1.075k	1	0.336	-0.526578774	-0.008	-0.207

The final hardware verification test was to play music through our audio equalizer. We used a smartphone as an audio input to the equalizer, and a speaker as an output. Running the equalizer program, along with the windows app, we were able to listen to music playing from the phone on the speakers. Adjusting the equalizers attenations values using the app, we were able to manipulate the output sound. For example, by setting the lower three sliders to 0 in the app we were able to completely remove the lower frequency bass sounds from a song. This response, along with other input setups and their audio responses, confirmed that our equalizer functioned as intended.

5. Conclusion

Ultimately, the design worked successfully. It produced a reasonably good sound, and it allowed proper control of the attenuation values for each frequency band. The sound coming out of the equalizer did have some audible white noise in the background that was not in the original

recordings. The probably resulted from the quantization noise of the analog-to-digital and digit-to-analog converters. Sound quality could be improved by using an ADC and DAC with higher numbers of bits. Precision of the calculations in the equalizer could be improved by using more digits in the fixed-point numbers or by using floating-point numbers instead.

While our design was successful as a product, it could be improved in terms of design efficiency. For example, it used three Axi4Lite buses and the BUS_MUX. The functionality of the BUS_MUX, attenuation BUS, and MB_MUX bus could have all been combined into CONTROLLER. This could be accomplished with a simple address decode on the CONTROLLER bus. Using this designed would have reduced I/O usage as well as the amount of structural verilog.

Another possible area of improvement would be the GUI interface. If a slide is changed then all of the attenuation values are serially written to the MB. This design is inefficient as only one attenuation value needs to be updated. A more efficient design would write just the filter number and the new attenuation value to the MB.

A final idea for improvement would be to add a simple enable/disable button to the GUI. The Eq_controller already supports this function, but it was not implemented in the GUI. To add this function would require a single button on labview that writes a special 65 character long string. The string would have to be 65 characters long so that the serial read program on the MB continues past the point of receiving data. This special string would have to differentiate itself from an attenuation String. For example a string of 65 ‘F’ characters could be used as the special string. This would never appear as an attenuation string because we do not write negative f2.14 attenuation values. This special string could be parsed by the MB and send an enable/disable signal to the ENABLE_ EQUALIZER address on the CONTROLLER bus.

6. Acknowledgements/ Works Cited

Kyle Cepeda is acknowledged for his his help in developing the hexadecimalToDecimal() function.

Geeks for geeks code is acknowledged for providing a basis of the hexadecimalToDecimal() function. The source code can be found at the following URL:

https://www.geeksforgeeks.org/program-hexadecimal-decimal/