User-Centric Implementation of a Parametric Audio Equalizer Using Digital Signal Processing on the Altera DE2-115

Mark Kubiak

James Ickes

University of Illinois

ECE 385 - Spring 2017

Abstract

This project successfully set out to implement a user-friendly and powerful parametric audio equalizer on the Altera DE2-115 Development Board. At a high level, the end user is able to arbitrarily define up to eight concurrent digital filters that take input from the Line In 3.5mm jack and modify the sound sent to the Line Out jack. These audio-focused filters are created by the user, who enters the qualities of a filter as prompted by a VGA display. The display also relays the types of filters implemented and a logarithmic graph representing the modifications performed by the digital filter array. The digital signal processing is performed by two arrays of eight digital filters (one per channel) connected in series. Each filter is implemented as a standard "biquad" IIR filter, which takes 5 arbitrary coefficients to modify the input signal. Peak EQ, low shelf, high shelf, low pass, and high pass filters were all implemented for this project. The end result is a powerful parametric audio equalizer that is applicable for room correction, headphone/speaker tuning, digital crossovers, or simply changing the sound signature to match a user's preferences.

Introduction

The goal of this project was to create a user-oriented parametric equalizer that modifies line-level analog signals routed through the Altera DE2-115 FPGA board. At a high level, the end user provides a set of up to eight audio filters: peak EQ, low pass, high pass, low shelf, and/or high shelf. The user is prompted by a VGA display, where he can also see his filters and a graphical representation of how the combined filters are changing the sound. This audio-focused project has dozens of uses: it can be used for manual room correction, for correcting irregularities in the frequency response of headphones or speakers, or even just for changing the sound signature to fit the end user's preferences.

Audio Overview

All audio work for this project, including all signal processing, stems from audio_controller.sv and its corresponding state machine. At the highest level, the audio controller initializes the Wolfson

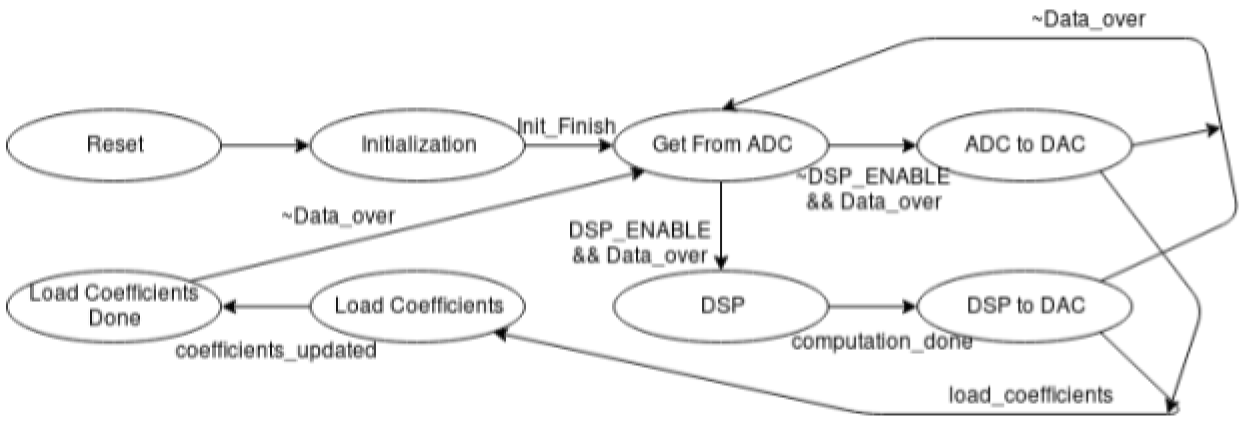


Figure 1: Audio Controller State Machine

WM8731 audio chip, then goes into a loop of getting a 16-bit sample from the analog to digital converter (ADC), processing it if necessary, and then sending it to the digital to analog converter (DAC) for playback. The state machine for audio_controller is shown in Figure 1.

Audio Interface with Wolfson WM8731

The Wolfson WM8731 audio driver was used in this project due to its inclusion on the Altera DE2-115 development board. Working VHDL code to utilise this chip was written by past student Koushik Roy and distributed by the instructors of this course. This module both initializes the chip using the I2C protocol and manages the transmission of integer audio samples from the ADC and to the DAC.

The first section of the code handles initializing the chip using the I2C control protocol. The device address for this audio chip is 0x35, and it contains 10 9-bit registers that control the functionality of the WM8731. Control signals are then sent to deactivate the outputs, set the ADC to 0db input gain, set the DAC to 0db output gain, disable filters, enable master mode and DSP mode, set the ADC and DAC to 16-bit integer samples, set the sampling frequency to 48khz, and then reactivate the outputs. The "master mode" control signal allows the chip to export its clocks to the FPGA fabric, increasing audio quality. Once this is done, the INIT_FINISH signal is set to high, indicating the WM8731 is ready to be used for audio.

The next section of the code handles sending out samples from the ADC, taking input samples for the DAC, and managing control signals to relay timing to the user. Samples are sent using DSP mode, which sends the left then right channels MSB first to/from the DAC/ADC, respectively. The timing diagram in Figure 2 clearly shows this. The DACLRC/ADCLRC clocks, defined by the WM8731 in

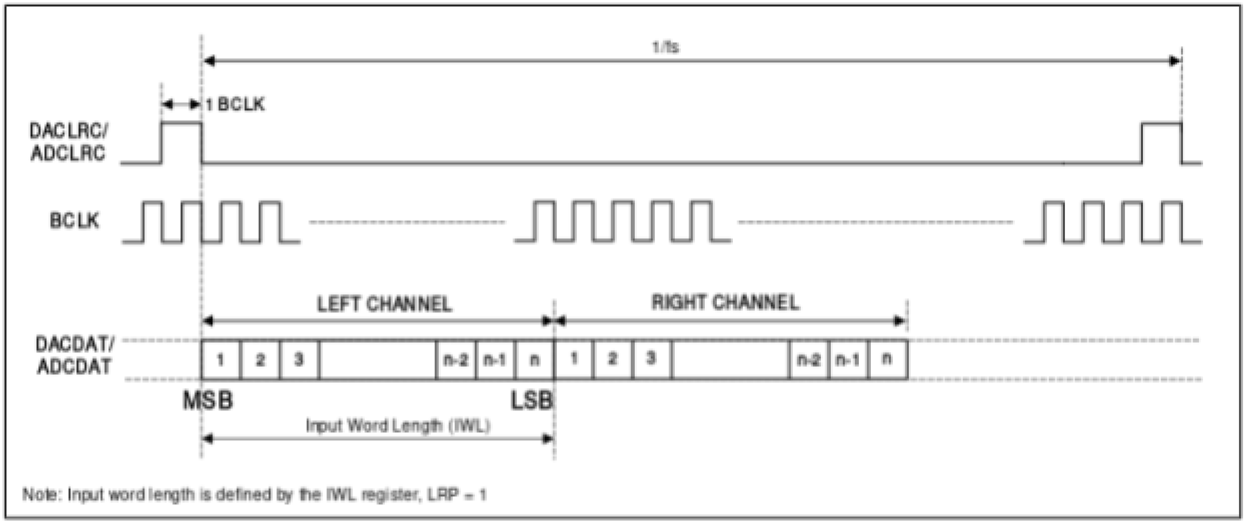


Figure 2: DSP Mode Timing and Bit Transmission

master mode, indicate when a new sample is to be sent or received. The provided code handles these clocks and acts as an interface to send/receive samples. This is abstracted in the provided module by outputting a 32-bit ADCDATA and taking in 16-bit LDATA and RDATA. The adc_full and data_over control signals are sent when the ADC and DAC, respectively, have processed the data. These control signals are used effectively by the Audio Controller state machine to send and receive audio.

Digital Signal Processing with Biquad Filters

If DSP is enabled using SW0, the 16-bit 2's complement audio sample will be sent to a chain of eight biquad filters. These 8 filters can be arbitrarily defined to implement any of 5 types of filters: peak EQ, low pass, high pass, low shelf, or high shelf. These filters form a chain, where the first filter sends its output to the second, the second to the third, and so on until all eight filters have modified the original sample in some way. This allows the user to implement multiple filters at the same time and have them all affect the final sound.

The actual modification of the audio sample is performed by individual biquad filters. Biquad filters are common IIR DSP filters that, using just five coefficients, can implement any one of dozens of DSP-related filters. Figure 3 shows a diagram for a typical biquad filter. Z^{-1} indicates an older sample, IE x[n-1] or y[n-1]. The incoming sample, old input samples, and old output samples are multiplied by the coefficients b0, b1, b2, a1, and a2, summed, and then outputted as y[n]. This method can implement a wide range of DSP filters based on the values of the coefficients, including all of the 5 types specified above.

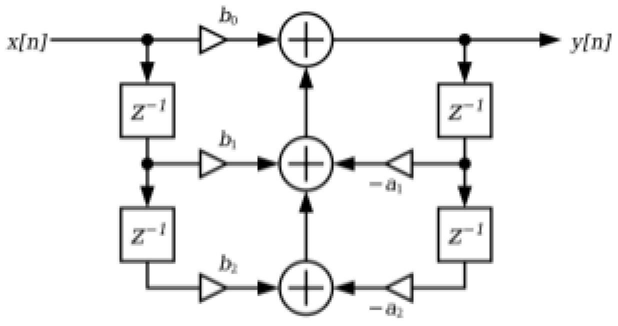


Figure 3: Biquad Filter

Calculating these coefficients was done in software. The user would enter the type of filter, the center frequency, Q factor, and, if applicable, the gain. Using these values, the 5 coefficients would be calculated and transmitted to software. All 5 types of filters share these common variables:

$A = 10^{G/40}$ $\omega_0 = \frac{2\pi F}{Fs}$ $\alpha = \frac{sin(\omega_0)}{2Q}$

The other calculations vary based on the type of filter. For example, the peak EQ filter is calculated by the following:

$\alpha_0 = 1 + \frac{\alpha}{A}$ $b_0 = \frac{1+\alpha A}{\alpha_0}$ $b_1 = \frac{-2cos(\omega_0)}{\alpha_0}$

$b_2 = \frac{1-\alpha A}{\alpha_0}$ $a_1 = -\frac{-2cos(\omega_0)}{\alpha_0}$ $a_2 = -\frac{1-\frac{\alpha}{A}}{\alpha_0}$

These equations were derived from a biquad calculation spreadsheet provided publicly by miniDSP Ltd. The full equations used and a link to the aforementioned spreadsheet are listed in Appendix D. The various equations were used on the software side to calculate b0, b1, b2, a1, and a2 as floating-point values. However, implementing floating-point arithmetic on the hardware side is not natively or easily supported. Thus, using 4.14 fixed-point multiplication (4 integer bits, 14 fraction bits) was decided upon for the following reasons:

1. The Cyclone IV includes over 500 native 9x9 (or 18x18) high-speed hardware multipliers. 18-bit length maximized the precision with minimal resource usage.
2. Floating-point multiplication is significantly more expensive than fixed point multiplication and would not offer significant gains in sound quality

3. All coefficients fall well within the range from -8 to +7, lending well to a fixed-point representation

The C code converts the floating-point coefficients to 4.14 fixed point representation right before they are sent to the hardware side. To use only a single 18x18 (two 9x9) hardware multiplier per biquad multiplication (5 per filter), the input sample is converted to 16.2 fixed point representation and multiplied with the 4.14 fixed point representation for the coefficient, resulting in a sample represented as 20.16 fixed point.

The actual operation of the SystemVerilog implementation of the biquad filter is relatively straightforward. When the biquad module receives a control signal indicating that a new sample has been pushed, the filter pushes the old coefficients down the stack. IE y1 becomes y2, x0 becomes x1, the input sample becomes x0, etc. The multiplication is performed by the multiplier modules, whose outputs are summed into a preliminary result. The result is rounded and checked for overflow, then output from the module as the least significant 16 bits of the integer part. Once this calculation is done, the biquad outputs a signal to indicate that its computation is complete and the Audio Controller state machine outputs the modified sample.

USB Keyboard Input

The Altera DE2-115 uses the Cypress EZ-OTG (CY7C67200) USB controller chip, which acts as a USB host for the keyboard. The EZ-OTG has four HPI registers: hpi data, hpi mailbox, hpi address, and hpi status. The registers are set using our code to give commands to the Cypress chip using the functions UsbWrite(), UsbRead(), IO_write(), and IO_read().

IO_read: This function reads from the USB IO by storing the address to be read into the otg_hpi_address register, then enabling reading by setting otg_hpi_r to 0 (since it is active low). A temporary variable is then created to hold this value so that otg_hpi_r can be reset to 1. The temporary variable is returned.

IO_write: This function writes to the USB IO by storing the data to be written and the address to be written to in otg_hpi_data and otg_hpi_address respectively. The otg_hpi_w signal to 0 so the data is written, and then back to 1. Nothing is returned.

UsbRead: A higher level function that makes use of the IO_read and IO_write functions to read the data at a particular address in the Cypress USB controller. The value is returned.

UsbWrite: A higher level function that makes use of the IO_write functions to write data into a specific address onto the Cypress USB controller. Nothing is returned.

Using these functions and the Cypress EZ-OTG chip, our program can communicate with the USB OTG chip to initialize a USB keyboard and read values from it.

Keycode Parsing and Processing

The input handling section of the software side is split into three main parts: the keycode user input, conversion to a string, and processing the inputs via state machine. The raw 16-bit keycode is received on the software side using the same code from lab8. This keycode is dependant on the number of keys pressed; if just one key is pressed then the keycode begins with 0x00. However, if two keys are pressed, the more recent one is placed into the 8 most significant bits. The C code takes this into account when checking for keycodes to send a single 8-bit keycode to handle_keycode(). This process loops infinitely to handle all input while the C code is running.

The 8-bit keycode determined by the aforementioned loop is sent to the handle_keycode() function. Since the keycodes do not line up with their respective ASCII values, handle_keycode() acts as a lookup table to convert keycode to character, then essentially builds a "string" (character array) of the characters represented by the pressed keys. When a number, '.', or '-' is pressed, the character array has an element added to the back of the queue. The function also handles the "E" press to correctly initialize the state machine. Once the user is done entering a string, they will press Enter, which calls the the sm_process_str(char* str) function, resetting the character array and progressing the state machine for valid inputs as shown in Figure 4.

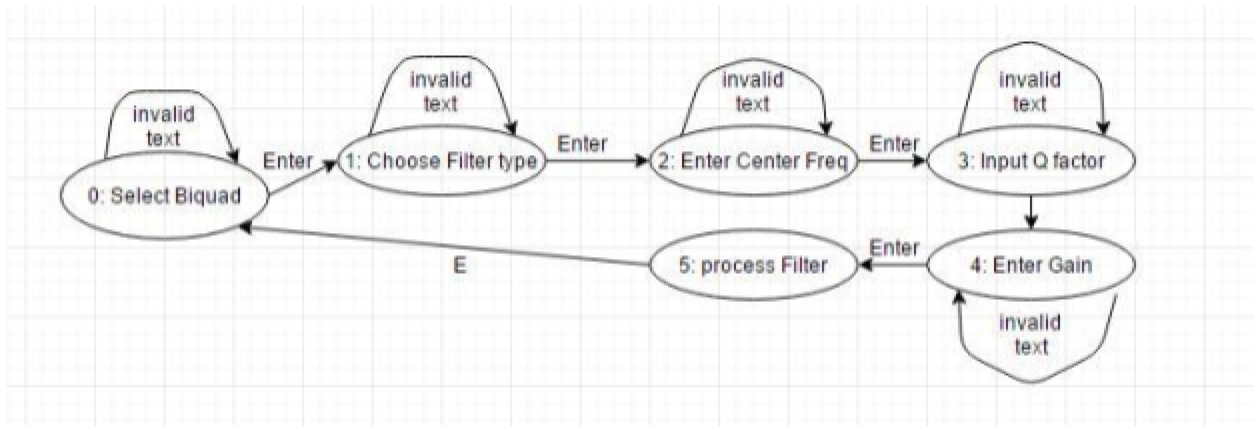


Figure 4: Sm_process_str(char* str) State Machine

There are six states in total. At the beginning of each state, the character array is converted into either floating point number or integer, depending on the state. The C functions atof() and atoi() are used to convert the character array to a floating point number or integer, respectively. The first five states change the value of to_hw_sig so that the hardware can display the correct prompts to the user. The first five states also will constantly loop back on themselves until they receive a valid input from the handle_keycode function for each of their values. State 0 must receive an integer 1-8, state 1 must receive an integer 1-5, state 2 must receive a number 20-20,000, state 3 must receive a number 0-20, and state 4 must receive a number -20 to 20. If this state does not receive a valid number, the user will be prompted to try again. If the conditions are met, the state will be incremented by 1. State 5 calls the process_filter() function, which does the vast majority of processing for both the coefficients and graph, as outlined elsewhere in this report.

SoC Communication: PIOs and IO Controller

Since the nature of this project requires significant communication between the software and hardware portions, many PIOs (Parallel I/O) were used for software to hardware transmission. Our final design consisted of thirteen PIO instances:

- **OTG_HPI_CS** - 1 bit output - Chip select for USB OTG chip
- **OTG_HPI_ADDRESS** - 2 bit output - Address for USB OTG chip
- **OTG_HPI_DATA** - 16 bit InOut - Data to or from the USB OTG chip

- **OTG_HPI_R** - 1 bit output - Read signal for USB OTG chip
- **OTG_HPI_W** - 1 bit output - Write signal for USB OTG chip
- **TO_HW_SIG** - 4 bit output - Generic control signal to hardware. Used for state machines and to select user message prompts.
- **TO_SW_SIG** - 2 bit input - Generic control signal to software. Used to indicate the status of the IO Controller state machine.
- **BIQUAD_INDEX** - 3 bit output - Index of biquad to load values into.
- **COEFFICIENT** - 18 bit output - Buffer for 4.14 fixed-point biquad coefficient and graph values to be placed into Graph RAM.
- **F_STRING** - 32 bit output - 4-character ASCII string to display as the center frequency of the current biquad.
- **G_STRING** - 32 bit output - 4-character ASCII string to display as the gain of the current biquad.
- **Q_STRING** - 32 bit output - 4-character ASCII string to display as the Q factor of the current biquad.
- **TYPE_INDEX** - 3 bit output - Index from 1 to 5 corresponding to the type of filter that was used for the current biquad.

These PIOs were handled almost entirely by the state machine defined in io_controller.sv. The state diagram for this state machine can be seen in Figure 5. It uses to_sw_signal to relay its current state

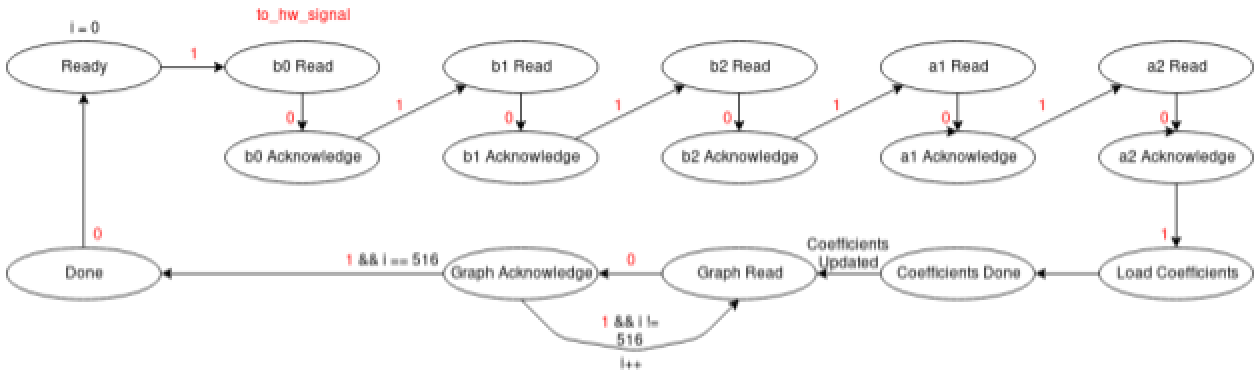


Figure 5: IO Controller State Machine

to the software side, and the C code sends back to_hw_signal so that the software and hardware portions stay in sync. For b0 - a2, the read state pulls in the 4.14 fixed point coefficient from the coefficient PIO and places it in a temporary register corresponding to that biquad coefficient. When the state machine reaches "Load Coefficients", all of these values are simultaneously exported to the biquad specified by the biquad_index PIO, immediately followed by a load_coefficients control signal. The "Coefficients Done" state then waits for the coefficients_updated signal, indicating that the biquad has received and updated its coefficients. This is immediately followed by over 500 cycles of "Graph Read" and "Graph Acknowledge". This is similar to loading coefficients in regards to the state machine, but the way it handles the data is significantly different. This section takes 8-bit values, placed into the 8 least significant bits of the "coefficient" PIO (space saving measure), and places them into the Graph RAM at address "i". This essentially fills the RAM with 8-bit graph values at addresses corresponding to the point's location on the X axis. After all values have been sent, the state machine is reset to the ready state and "i" reset to 0.

VGA Protocol and Drawing

Two modules are used to control the VGA output: vga_controller.sv and color_mapper.sv. We reused the same vga_controller we used for lab 8. The module outputs the value for vga horizontal sync, vertical sync, and clock. It uses counters to move left to right then top to bottom for each pixel. The color mapper module takes the current pixel from vga_controller and determines the color for that pixel, and outputs the VGA_R, VGA_G, VGA_B signals. The color mapper module is where the logic for every item displayed on the monitor is determined. There are several key components to the color_mapper module: static characters, strings, dynamic strings, graphing, and user prompts.

Static characters are drawn using the font_rom.sv module which contains data for the ASCII characters. Each character is stored as 16 by 8 bit words where each word contains the data for which pixels are on or off for that row (i.e 1 if the pixel is on, 0 if it is off). We created the module addr_compute.sv to take in the ascii value for a character and use it to compute the address in font_rom for that character. The address is passed to the font_rom which then passes the data for the row the current pixel is on. Then, the bit corresponding the current pixel is calculated by the equation "size - x_pixel + 8". This is done because the most significant bit corresponds to the left most pixel on the screen. This module also computes the signal sprite_on when the pixel is within the bounds of the character (in other words, if the current pixel is between the top-left corner, top-right corner, bottom-left corner, and bottom right-corner of where the character should be). Color_mapper passes the color black when sprite_on is high and when the data from font rom contains a 1 at the specific bit. Otherwise, it passes the color white.

To handle strings of characters, we also created the modules ascii_string, and ascii_string_2 to simplify writing out long strings of characters. These modules instantiate the addr_compute module four times and two times respectively. These modules are dynamic because they take an input string and a control signal that tells the string to "load" a new 2 or 4 character string to display on screen. This way, new values can be sent to these strings and they will update instantly.

To handle user prompts, the C code sends the to_hw_sig to determine in which user input state the system is in. The color_mapper module then uses if statements to determine the value of the 48 character signal prompt_ascii. This signal is then sent to the module ascii_string_48, which is similar to the other ascii_string modules but instantiates a significantly longer 48 character string.

Frequency Response Graph

As stated in the project proposal, an important goal for this project was to create a graphical representation of the combined frequency response, $H(\omega)$, that would be visible to the user on the VGA display. The creation and display of the graph can be split into four major parts:

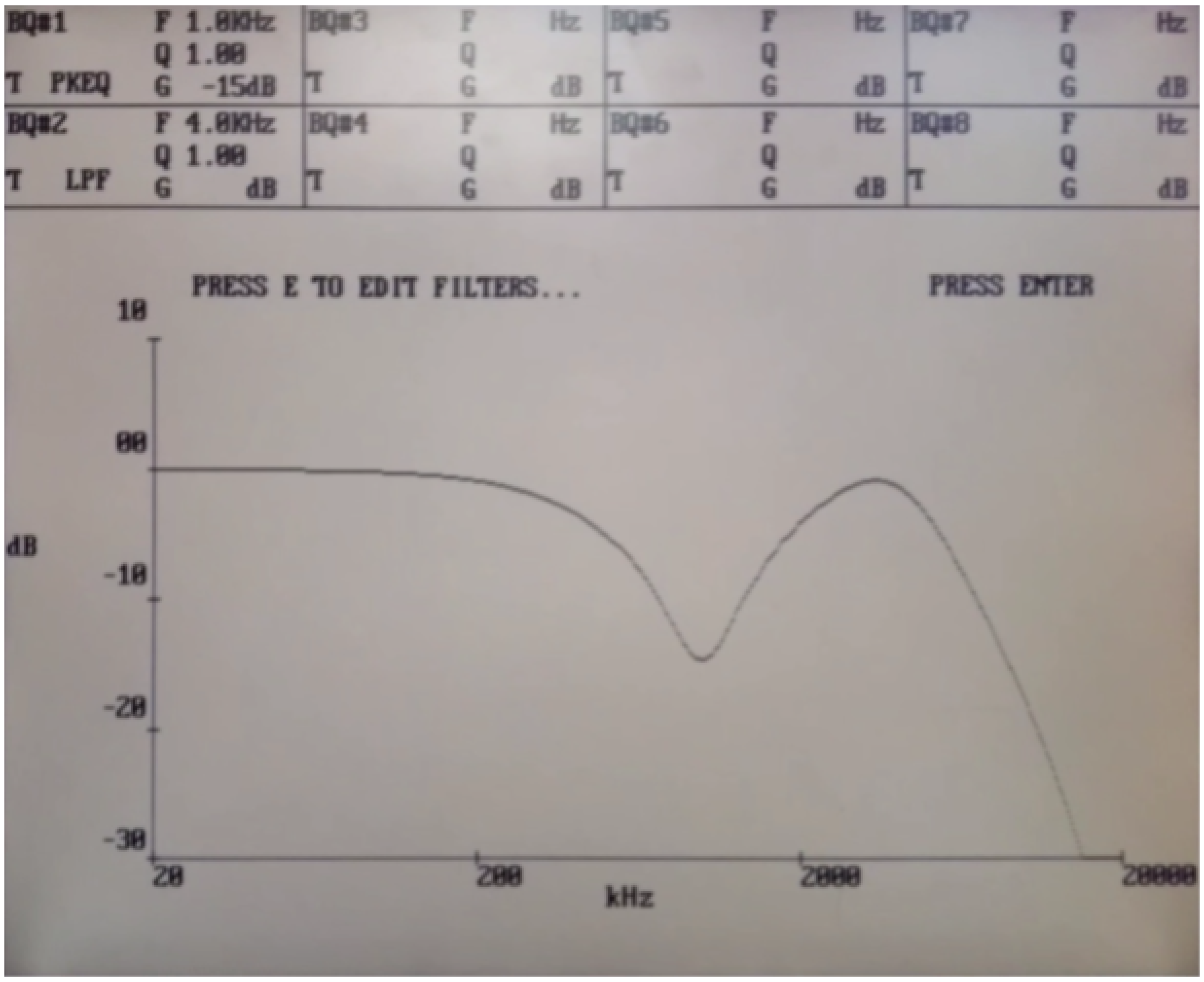
1. Calculation of the dB change for all filters
2. Calculating an integer "sample"
3. Transmitting and storing the sample
4. Displaying the sample in a graph on the VGA display

The equations used to find the dB change (in relation to 0db) per filter were derived from the aforementioned miniDSP spreadsheet. The derived equations for a frequency F are as follows:

$$\omega = \frac{2\pi F}{Fs} \qquad \varphi = 4sin^2(\tfrac{\omega}{2})$$

$$dB = 10log_{10}((b_0 + b_1 + b_2)^2 + \varphi(b_0 b_2 \varphi - (b_1(b_0 + b_2) + 4b_0 b_2)) - 10log_{10}((1 - a_1 - a_2)^2 + \varphi(- a_2\varphi - (- a_1(1 - a_2) - 4a_2))$$

This gives the change in decibels for each individual biquad filter. The total change at any frequency is calculated by summing the dB change for each individual filter into a total.



```
BQ#1      F 1.0KHz  BQ#3   F    Hz  BQ#5   F    Hz  BQ#7   F    Hz
          Q 1.00           Q               Q               Q
T  PKEQ   G  -15dB  T      G    dB  T      G    dB  T      G    dB
BQ#2      F 4.0KHz  BQ#4   F    Hz  BQ#6   F    Hz  BQ#8   F    Hz
          Q 1.00           Q               Q               Q
T   LPF   G     dB  T      G    dB  T      G    dB  T      G    dB

          PRESS E TO EDIT FILTERS...                PRESS ENTER
```

Figure 6: Example Graph Demonstrating PeakEQ and Low Pass Filters

The actual graph shown on the display was determined to be a logarithmic graph 516px wide and 256px tall. This is allows us to provide a logical X axis on the graph, from 20hz to 20khz, with every increase by a factor of 10 holding exactly 172px. On the Y axis, the height being a power of 2 allows us to hold the height of each point on the graph as an integer "sample" from 0 (min) to 255 (max). The graph was arbitrarily defined as being from -30db to +10db. A dB value of -30dB (min) translates to a sample of 0 and +10dB (max) translates to a sample of 255, making 0dB translate to 191. This conversion was done in software.

The next task is to display these integer samples and display them on the graph. These samples were sent to the hardware via PIO and stored in a two-port M9K on-chip RAM block with simultaneous read and write. The graph was defined as 516px wide and 256px tall. Thus, each address in memory mapped to a specific X "column", with the data at that address representing the height/Y location of the pixel in that column. Thus, the RAM was created with 1024 words X 8 bits = 8192 bit RAM. These values were transmitted sequentially by io_module.sv after the biquad coefficients were transmitted. To display the graph, Color_Mapper would determine whether the draw location was inside the graph area. If it was, it would lookup the RAM address corresponding to the DrawX value and compare it with a computed value for the current "height" of the graph. For example, if the current drawing pixel was at the graph location representing 0db at 200hz, the RAM would lookup the value at RAM address 172 (corresponds to 200hz). If the value returned from the RAM was 191 (corresponds to 0db), then the pixel would be black. If not, it would be white. This returned a log graph that accurately represents the enabled filters.

Conclusion

While very time-consuming and difficult to implement, all of the original goals set forth in the project proposal were met. USB keyboard input and VGA output worked flawlessly, including a dynamically generated graph of the frequency response of the biquad filters. The DSP section of the project worked perfectly; the filters provided audible changes and the signal processing did not impact audio quality in any noticeable manner. The final iteration of the project could absolutely be used for any of the cases presented in the introduction.

One small factor that we would have liked to improve for the final demonstration would have been an increase in speed for the graph generation. Graph calculations were done on the NIOSII/e, which does not support using the hardware multipliers provide by the Cyclone IV chip. An upgrade to the NIOSII/f would have drastically sped up graph generation due to its support for hardware multiplication and division. Unfortunately, the upgrade cost upwards of $500/year, and was out of the scope of this project. Transmission via PIO would also have not been feasible due to the fact that all multiplication for the graph generation would be in IEEE754 floating point representation, which is not a standard SystemVerilog type.

In conclusion, the final iteration of the project was a success. The audio portion of the project performed exactly as expected and the user interface made the project accessible to everyone. This was an excellent challenge and thoroughly tested our knowledge of SystemVerilog and SoC design, while still requiring us to learn new skills.

Appendix A: Block Diagram

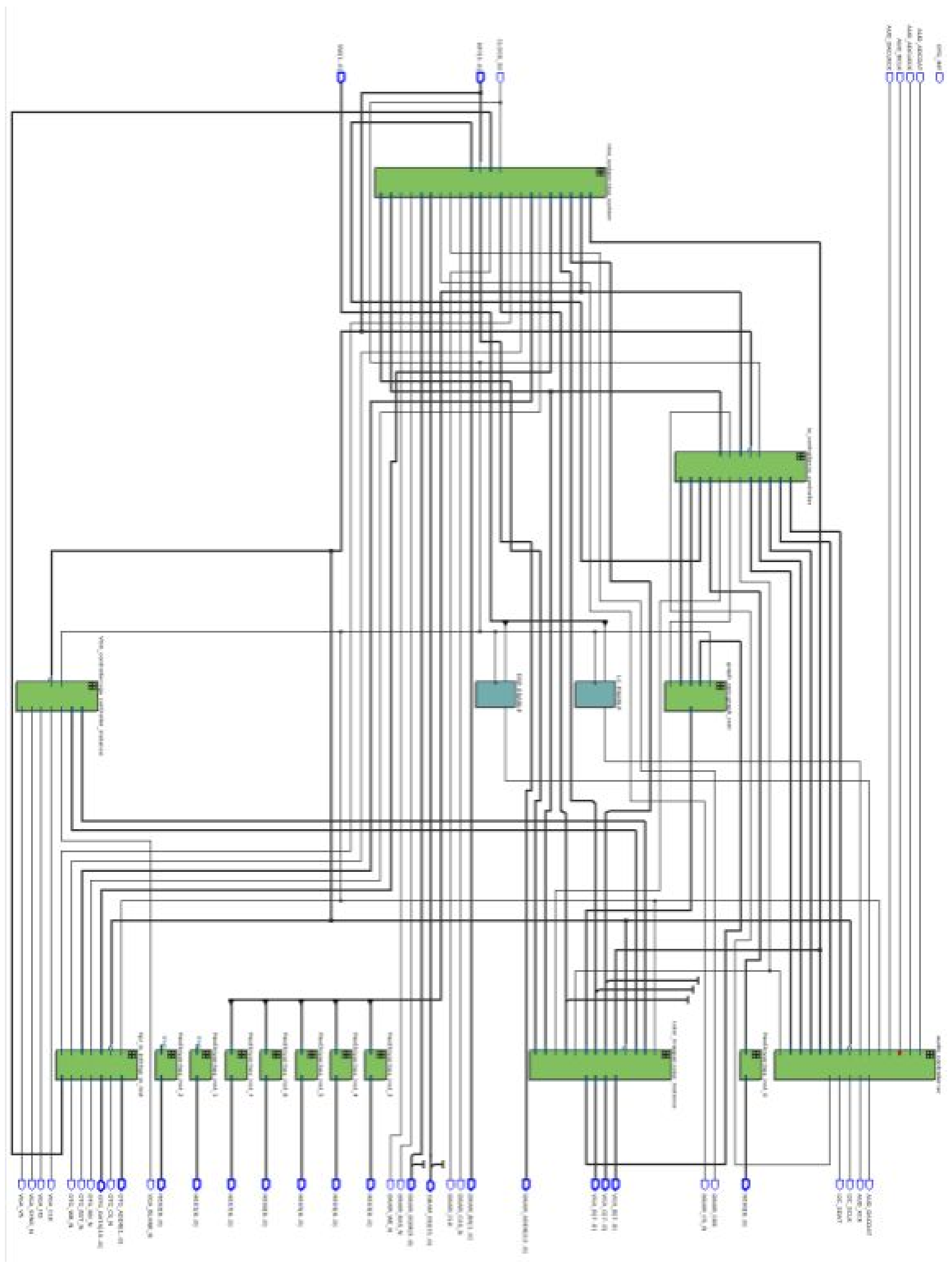


Figure 7: Top Level Block Diagram (Rotated to Fit Report)

Appendix B: Resource Usage Table

LUTS	11,213
DSP	320
Memory	52,224 bits
Flip-Flop	5,891
Frequency	55.07 MHz
Static Power	103.77 mW
Dynamic Power	209.46 mW
Total Power	448.33 mW

Appendix C: Module Descriptions

Module: Final_Project.sv
Inputs: CLOCK_50, [3:0] KEY, [1:0] SW, AUD_ADCDAT, AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK, OTG_INT
Outputs: AUD_DACDAT, AUD_XCK, I2C_SCLK, I2C_SDAT, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [6:0] HEX6, [6:0] HEX7, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK
Inout: [15:0] OTG_DATA, [31:0] DRAM_DQ
Description: Instantiates the signals and modules of the project
Purpose: Serves as the top-level module for the project.

Module: addr_compute.sv
Inputs: [10:0] shape_y, [10:0] shape_x, [9:0] DrawX, [9:0] DrawY, [7:0] ascii
Outputs: [10:0] sprite_addr, [10:0] u_shape_x, [10:0] u_shape_y, sprite_on
Descriptions: This is a module used by color_mapper to determine the pixels used in drawing a character.
Purpose: This is needed to draw the character for our display.

Module: ascii_string.sv
Inputs: Clk, Reset, [9:0] DrawX, [9:0] DrawY, [31:0] input_string_in, [10:0] shape_x, [10:0] shape_y, load_string

Outputs: sprite_on, [10:0] sprite_addr, [10:0] u_shape_x, [10:0] u_shape_y
Descriptions: This module instantiates four modules of addr_compute**.**
Purpose: This module is needed to make strings of four characters more easily.

Module: ascii_string_2.sv
Inputs: Clk, Reset, [9:0] DrawX, [9:0] DrawY, [15:0] input_string_in, [10:0] shape_x, [10:0] shape_y, load_string
Outputs: sprite_on, [10:0] sprite_addr, [10:0] u_shape_x, [10:0] u_shape_y
Descriptions: This module instantiates two modules of addr_compute.
Purpose: This module is needed to make strings of two characters more easily.

Module: ascii_string_48.sv
Inputs: Clk, Reset, [9:0] DrawX, [9:0] DrawY, [383:0] input_string_in, [10:0] shape_x, [10:0] shape_y
Outputs: sprite_on, [10:0] sprite_addr, [10:0] u_shape_x, [10:0] u_shape_y
Descriptions: This module instantiates 12 ascii_string modules in order to print out a 48 character string.
Purpose: This module is needed to print out the user prompts on the display.

Module: audio_controller.sv
Inputs: Clk, Reset, DSP_ENABLE, LC_ENABLE, load_coefficients, [2:0] biquad_index, [17:0] b0, [17:0] b1, [17:0] b2, [17:0] a1, [17:0] a2, AUD_ADCDAT, AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK
Outputs: AUD_DACDAT, AUD_XCK, I2C_SCLK, I2C_SDAT, coefficients_updated
Description: Instantiates the biquad chain modules and the audio interface. Runs a state machine that controls whether the modified audio signal is sent.
Purpose: This module passes data to the DSP section of the project.

Module: audio_interface.vhd
Inputs: [15:0] LDATA, [15:0] RDATA, clk, Reset, INIT, AUD_BCLK, AUD_ADCDAT, AUD_DACLRCK, AUD_ADCLRCK
Outputs: INIT_FINISH, adc_full, data_over, AUD_MCLK, AUD_DACDAT, I2C_SDAT, I2C_SCLK, [31:0] ADCDATA
Descriptions: Handles the input and outputs to the audio chip on the altera board.
Purpose: This module is needed for sending and receiving audio, a key part of the project.

Module: biquad.sv
Inputs: Clk, Reset, new_sample, new_coefficients, [15:0] sample_in, [17:0] b0_load, [17:0] b1_load, [17:0] b2_load, [17:0] a1_load, [17:0] a2_load
Outputs: [15:0] sample_out, computation_done, coefficients_updated
Descriptions: This module edits an audio sample by applying coefficients.
Purpose: This module is needed to change the audio signal, a key part of the project.

Module: biquad_chain.sv

Inputs: Clk, Reset, new_sample, push_coefficients, [2:0] biquad_index, [15:0] sample_in, [17:0] b0, [17:0] b1, [17:0] b2, [17:0] a1, [17:0] a2
Outputs: [15:0] sample_out, computation_done, coefficients_updated
Descriptions: Instantiates many biquad filters and chains them together.
Purpose: This is used to create to filtering chain effect and to control the coefficient loading logic.

Module: Color_Mapper.sv
Inputs: Clk ,Reset, load_coefficients, show_graph, [9:0] DrawX, [9:0] DrawY, [31:0] f_string, [31:0] g_string, [31:0] q_string, [2:0] type_index, [2:0] biquad_index, [3:0] to_hw_sig, [7:0] graph_value
Outputs: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, [9:0] graph_lookup_x
Descriptions: This module determines the color of each pixel. (see VGA protocol for more detail).
Purpose: This is needed for the display to function.

Module: font_rom.sv
Inputs: [10:0] addr
Outputs: [7:0] data
Descriptions: This module contains the information for all the ASCII sprites
Purpose: This is needed to draw characters for our display

Module: fp_multiplier.sv
Inputs: [15:0] sample_in, [17:0] fip_coefficient
Outputs: [35:0] mult_out
Descriptions: This module multiplies a fixed point coefficient with the sample.
Purpose: This module is needed to multiply the coefficients with the sample in the biquad filters.

Module: graph_ram.sv
Inputs: [ADDR_WIDTH-1:0] waddr, [ADDR_WIDTH-1:0] raddr, [ADDR_WIDTH-1:0] wdata, we, clk
Outputs: [WIDTH-1:0] q
Descriptions: This module is used to store the values representing the frequency response graph.
Purpose: This module is needed in the graph generation in order to continually display the points on the graph.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: Provides output to the hex display based on hexadecimal input.
Purpose: Lights the correct segments of the segmented display to properly display the provided hex code. Used primarily for debugging.

Module: hpi_to_intf.sv
Inputs: Clk, Reset, [1:0] from_sw_address, [15:0] from_software_data_out, from_sw_r, from_sw_w, from_sw_cs

Outputs: [15:0] from_sw_data_in, [1:0] OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N
Inouts: [15:0] OTG_DATA
Description: Manages the buffer between the software and the USB OTG chip on the DE2-115.
Purpose: Takes the data from the USB chip and synchronizes it with the rest of the board. Also manages the bidirectional OTG_DATA port so that data is only written to the port when write is active.

Module: io_controller.sv
Inputs: Clk, Reset, coefficients_updated, [3:0] to_hw_sig, [17:0] coefficient
Outputs: [1:0] to_sw_sig, load_graph_ram, [9:0] waddr, [7:0] wdata, [17:0] b0, [17:0] b1, [17:0] b2, [17:0] a1, [17:0] a2, load_coefficientsd, show_graph, [3:0] state_out
Descriptions: This state machine module coordinates the software and hardware signals and gives the hardware the calculated coefficients.
Purpose: This is needed for our C software and our SystemVerilog hardware to work together.

Module: mux_1_8.sv
Inputs: [2:0] select, data_in
Outputs: [7:0] data_out
Descriptions: This module routes data_in to various biquads.
Purpose: This module is used to control various signals being routed to individual biquads throughout the project.

Module: mux_8_1.sv
Inputs: [2:0] select, [7:0] data_in
Outputs: data_out
Descriptions: This module routes data_in to the data_out control signal.
Purpose: This is used to return the correct control signal from the chain of biquads throughout the project.

Module: VGA_controller.sv
Inputs: Clk, Reset
Outputs: VGA_HS, VGA_VS, VGA_CLK, VGA_BLANK_N, VGA_SYNC_N, [9:0] DrawX, [9:0] DrawY
Description: Handles the creation of the image using the correct control signals and VGA protocol.
Purpose: Creates and controls the VGA display signals. Outputs the current pixel for color_mapper.sv to draw and provides the horizontal and vertical sync pulses.

Appendix D: Coefficient Equations

All of the following equations were derived from miniDSP Ltd.'s "Biquad filter spreadsheet". This can be found free of charge on the company's website:

https://www.minidsp.com/applications/advanced-tools/advanced-biquad-programming

This list assumes the following:
Fs: Sampling Frequency = 48000hz
F: Center Frequency = 20hz - 20khz
G: Gain Factor (if applicable) = -20db to +20db
Q: Quality Factor = 0 to 20

Common equations:

$A = 10^{G/40}$ $\omega_0 = \frac{2\pi F}{Fs}$ $\alpha = \frac{sin(\omega_0)}{2Q}$

Peak EQ Filter:

$\alpha_0 = 1 + \frac{\alpha}{A}$ $b_0 = \frac{1+\alpha A}{\alpha_0}$ $b_1 = \frac{-2cos(\omega_0)}{\alpha_0}$

$b_2 = \frac{1-\alpha A}{\alpha_0}$ $a_1 =- \frac{-2cos(\omega_0)}{\alpha_0}$ $a_2 =- \frac{1-\frac{\alpha}{A}}{\alpha_0}$

Low Shelf Filter:

$\alpha_0 = (A+1) + (A-1)cos(\omega_0) + 2\sqrt{A}\alpha$

$b_0 = \frac{A((A+1)-(A-1)cos(\omega_0)+2\sqrt{A}\alpha)}{\alpha_0}$ $b_1 = \frac{2A((A-1)-(A+1)cos(\omega_0))}{\alpha_0}$

$b_2 = \frac{A((A+1)-(A-1)cos(\omega_0)-2\sqrt{A}\alpha)}{\alpha_0}$ $a_1 =- \frac{-2((A-1)+(A+1)cos(\omega_0))}{\alpha_0}$

$a_2 =- \frac{(A+1)+(A-1)cos(\omega_0)-2\sqrt{A}\alpha}{\alpha_0}$

High Shelf Filter:

$\alpha_0 = (A+1) - (A-1)cos(\omega_0) + 2\sqrt{A}\alpha$

$b_0 = \frac{A((A+1)+(A-1)cos(\omega_0)+2\sqrt{A}\alpha)}{\alpha_0}$ $b_1 = \frac{-2A((A-1)+(A+1)cos(\omega_0))}{\alpha_0}$

$b_2 = \frac{A((A+1)+(A-1)cos(\omega_0)-2\sqrt{A}\alpha)}{\alpha_0}$ $a_1 =- \frac{2((A-1)-(A+1)cos(\omega_0))}{\alpha_0}$

$a_2 =- \frac{(A+1)-(A-1)cos(\omega_0)-2\sqrt{A}\alpha}{\alpha_0}$

Low Pass Filter:

$\alpha_0 = 1 + \alpha$ $b_0 = \frac{1-cos(\omega_0)}{2\alpha_0}$ $b_1 = \frac{1-cos(\omega_0)}{\alpha_0}$

$b_2 = \frac{1-cos(\omega_0)}{2\alpha_0}$ $a_1 =- \frac{-2cos(\omega_0)}{\alpha_0}$ $a_2 =- \frac{1-\alpha}{\alpha_0}$

High Pass Filter:

$\alpha_0 = 1 + \alpha$ $b_0 = \frac{1+cos(\omega_0)}{2\alpha_0}$ $b_1 =- \frac{1+cos(\omega_0)}{\alpha_0}$

$b_2 = \frac{1+cos(\omega_0)}{2\alpha_0}$ $a_1 =- \frac{-2cos(\omega_0)}{\alpha_0}$ $a_2 =- \frac{1-\alpha}{\alpha_0}$

Frequency Response Graph:

$\omega = \frac{2\pi F}{Fs}$ $\varphi = 4sin^2(\frac{\omega}{2})$

$$dB = 10log_{10}((b_0 + b_1 + b_2)^2 + \varphi(b_0 b_2 \varphi - (b_1(b_0 + b_2) + 4b_0 b_2)) -$$
$$10log_{10}((1 - a_1 - a_2)^2 + \varphi(- a_2 \varphi - (- a_1(1 - a_2) - 4a_2))$$