# Web-Enabled Speaker and Equalizer

Final Project Report

December 9, 2016

E155

Josh Lam and Tommy Berrueta

Abstract

IoT devices are often hailed as the future of technology, where everything is connected. Our project adds to the growing list of IoT devices: a speaker and equalizer controlled from the internet. A website allows the user to use playback controls, select songs, and modify the equalization of the audio to their personal preference. Commands are sent to a Raspberry Pi which communicates with an FPGA via SPI. The FPGA filters the audio before sending it back to the Pi as an equalizer in the form of a filter bank with 7 FIR bandpass filters. The Pi then outputs PWM audio to a 4th order butterworth filter and then to an amplifier and set of speakers. However, in the actual implementation of the design, the filter bank did not work outside of simulation. Thus, the design was done using a single FIR filter rather than a full filter bank.

## Introduction

In an ever interconnected world, IoT devices look to enable devices previously independent to be able to be communicate and be controlled via the web. In particular for audio applications, IoT control could be hugely beneficial to remotely control speakers at an event or venue. This project describes the development of a web-enabled speaker and equalizer such that users can select songs via a web server and stream them through an equalizer where they are modified and finally output by speakers. A user could control the playback and equalization in real time and make sure the audio sounds as desired. The system is split up into three main partitions: Raspberry Pi, FPGA and external hardware. The Raspberry Pi hosts the web server where users can control the system. The FPGA hosts the hardware equalizer that is controlled by the settings on the web server. Finally, the external hardware consists of an 8x8 LED array to display current equalizer gains, a pair of speakers and a stereo amplifier to raise the level of the audio signals from the Pi to an audible level. Figure 1 shown below displays a top level view of the system architecture and communication between subsystems.

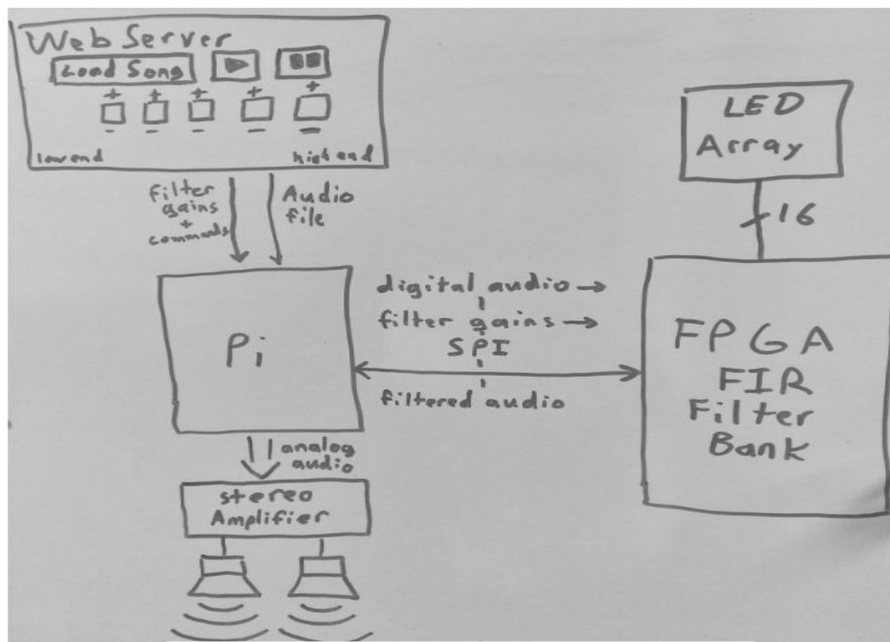



Figure 1: Top-level system diagram.

Schematics

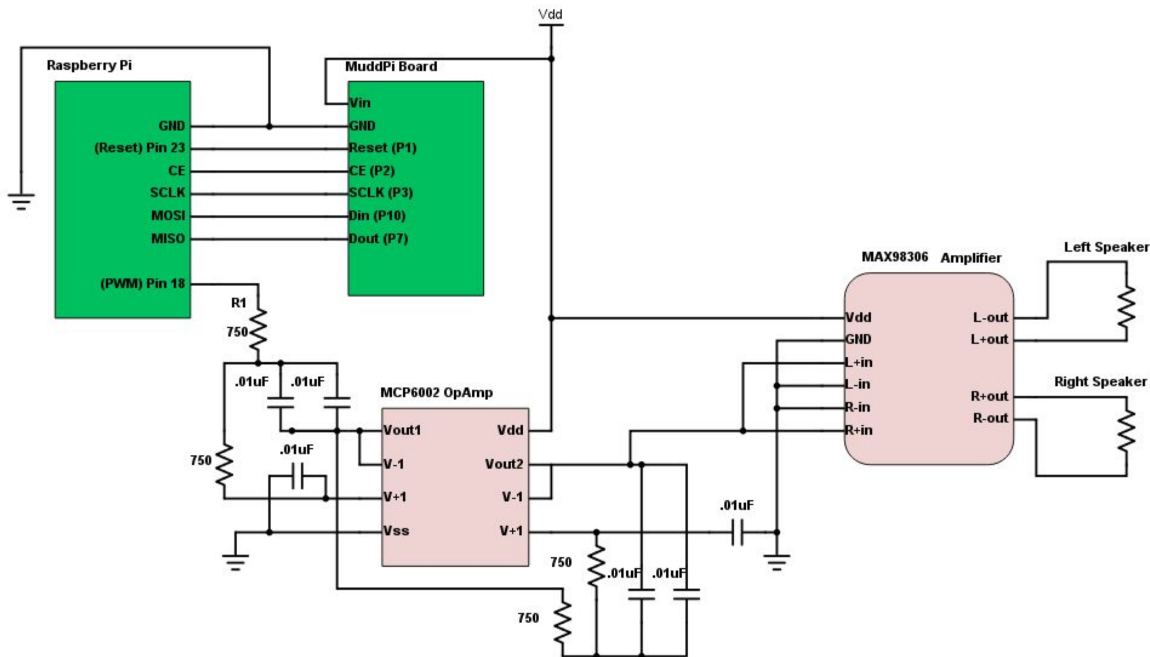



Figure 2: Breadboard schematic.

Shown above in figure 2 is our final Breadboarded Project. Of note, the LED array is not shown here since it was not completed. Running between the Pi and the FPGA board are 5 signal lines; the four SPI lines (channel enable, serial clock, MISO, and MOSI) and a reset line. Out of the Pi, pin 18 was the PWM output. To convert from PWM into an analog signal, the PWM wave was sent through a 4th order low pass filter in the form of two Butterworth filters with Sallen-Key topology with a cutoff frequency of 15kHz. The operational amplifier used was single-sided since the PWM wave was also single-sided. The output of these filters was wired to the left and right input of the stereo amplifier, with GND being wired to the negative inputs. Finally, the two speakers were wired to the outputs of the amplifier.

Raspberry Pi Design

There are two main aspects of the code implemented on the Raspberry Pi; the main loop which handles SPI and PWM communication, and the numerous CGI scripts that control the main loop through shared memory.

The Pi communicates with the FPGA and sends out audio through its SPI and PWM peripherals. All peripherals are accessed by writing to corresponding registers in the Pi's memory. These are mapped to using mmap. Much of this project's work was based off of EasyPIO.h (reference) which already initializes and sets up may of the Pi's peripherals, including GPIOs, timers and of course PWM and SPI. However, there were some slight modifications made to the script. The following sections detail those changes and how each peripheral is used.

1. GPIO:

EasyPIO has functions that write to the registers that change the pin modes and also read and write from the pins.

2. SPI:

The SPI peripheral is set up by setting the pin modes of pins 8, 9, 10, and 11 to ALT0. The SPI initialization allows the caller to set the SPI frequency and settings, which can be set to default operation by writing the settings bytes to 0. Sending and receiving is done through the SPI's FIFO register, After a value is written to the FIFO buffer, it is sent, and upon completion, the done bit of the Command and Status register is set to 1.

3. PWM:

To write to the PWM, pin 18 is set to mode ALT5. EasyPIO.h initializes the PWM peripheral with the maximum frequency of 25MHz. While it was originally planned to write directly to pins 40 and 45, which are connected directly to the audio jack, the Pi apparently does not grant access to them and in fact only pin 18 is allowed PWM functionality. To write to PWM, the PWM's DAT register is written to to determine the duty cycle of the PWM wave.

4. Timer:

The Pi has 4 internal timers running off of a 1 MHz clock. EasyPIO.h utilizes these by implementing delay functions on microsecond scales. To send the PWM samples out at the correct audio sampling frequency, these microsecond delays are used to wait for the length of the period of the frequency. This delay being microseconds is not completely accurate, resulting in slightly sped up or slowed down depending on what the delay is. Of note, it also take a non-zero amount of time to run the rest of the loop body, so the timer had to be experimentally tweaked to get the audio speed as close as possible to the actual rate.

These delay functions set one of the four system timers to be a microsecond offset from the current timer. Then, once the current timer reaches this value, it will set a corresponding flag in the system timer's command and status register. A while loop would wait for this condition to be reached, freezing the system until it is. Of the four timers, timers 2 and 0 are often used for other processes and are often inconsistent in performance when used for delays. Timers 3 and 1, on the other hand, perform consistently but are prone to failing resulting in the entire program being stuck in an endless loop. The team's best guess for why this is is that the system timer that fails is somehow set to a different value which the current timer never reaches. To counteract this, the delay function was modified to utilize all four system timers, and exit the while loop if any one of the four timers set a flag. In this way, the only way for the system to fail would be if all four timers failed at the same time. After empirical observation this was deemed to happen infrequently, as the code would have to play several songs in succession before finally failing, an extreme edge case.

Website and CGI Scripts

The website is at the center of the design, comprising of html action buttons that each trigger their own CGI executable scripts. A screenshot of the website is shown below, with descriptions of each button's function following. Each button redirects back to the project's webpage.

Playback Buttons

Choose Song

Filter ON/OFF

Bin 0 | Bin 1 | Bin 2 | Bin 3 | Bin 4 | Bin 5 | Bin 6

Lowest frequency----------------Highest Frequency

Figure 3: Screenshot of website.

1. Play Button:

The play button's script is where the main loop is written. The core functionality of the main loop is to take an audio sample, send it over SPI, receive a filtered sample, and send it through PWM. The main loop's settings, such as whether to play music or not or what song to play, all depend on bytes in shared memory. Shared memory is a C programming construct that allows different C scripts to communicate with each other by setting aside a block of memory that the scripts can all read and write to. In addition to redirecting back to the webpage, all scripts connect to the shared memory block using shmget. The outline of the shared memory is as follows in table 1 below:

| Byte number | Use |
| --- | --- |
| 0 | Unused |
| 1 | Status - MSB[Filter_On/-Off-, X, X, X, song_changed, locked/-open-, going/-stopped-, playing/-paused-]LSB |
| 2 | Song Number |
| 3-9 | Gains 0 to 6 |

Table 1: Shared memory mapping.

The first byte is the status byte, where each bit represents a different status setting. Upon pressing the play button, the two least significant bits are set to [going playing]. The code then checks the locked/-open- status bit to determine whether to begin the main loop. The locked/-open- bit ensures that only one instance of the main loop is running at any time, as the play button can still be pressed even if a main loop is already running. If no loop is running, the default setting is open, but as soon as a loop begins (play button pressed), the bit is changed to locked, and will not open until the stop or reset button is pressed.

Upon each loop iteration, the main code will check the status byte and respond accordingly. If the Filter_On bit is high, the loop will send received filtered audio to PWM. Otherwise it will send the unfiltered samples. If the song_changed bit is high, the loop will update the string holding the title of the song and re-read the wav file from memory to update the audio sample array. When changing the song, the code will also briefly set the reset pin to high to reset the FPGA. The next 4 bits are unused. The going/-stopped- bit and the playing/-paused- are distinct from one another since a paused song retains its previous sample location and the main loop still runs, but if the song is stopped, the previous sample location will not be held and the loop will exit. In the main loop, if a song is paused, the loop will continue to run (status still locked and going), but will not send SPI or PWM. The song number byte is where the current song number is stored and determines what song is playing. Lastly, the next 7 bytes are used to store the gain settings.

2. Pause, Stop and Reset Buttons:

The pause, stop and reset buttons affect the status byte of the previously described shared memory segment. The pause button changes the playing/-paused- bit to 0 (paused) if the going bit is high. The stop button changes the 3 least significant status bits to 000 [open stopped paused]. Lastly, the reset button changes the status byte and the song byte to all zeros and sets all the gain bits to sevens. The reset button also briefly writes the reset pin to high to reset the FPGA.

3. Song Change Buttons:

When one of the song buttons is pressed, it will not only flip the song_changed bit to high, but also change the value stored on this byte corresponding to the song number pressed. Currently, only 12 songs are available to play, but since there is a full byte of information, there could be up to 256 songs without the need for another byte of memory. These songs are stored as wav files in the same folder as the CGI scripts so the executables are able to read them.

4. Filter Toggle Button:

The filter on button changes the filter_on bit of the status byte. Specifically it does so by performing an XOR operation to toggle the bit from 1 to 0 or 0 to 1.

5. Gain Buttons:

If a plus button is pressed, the gain stored there will be incremented, whereas it will be decremented if a minus button is pressed. The values are capped to be between 0 and 15, with a default value of 7.

FPGA Design

The FPGA houses the equalizer in the form of a bank of bandpass FIR filters with settable gains. The hardware consists of several submodules all which serve the purpose of streaming and filtering audio through the system. The system takes in audio samples via an SPI module which handles communication between the FPGA and the Raspberry Pi. 16 bit words are sent and received simultaneously as audio is streamed and filtered. In addition to the 16 bit audio sample, a 16 bit sample containing the equalizer gains follows. These two alternating samples are sorted by a finite state machine that sends samples into the FIFO buffer to be accessed by the FIR filters, and the gains to the FIR filters themselves to be used in the calculation. The FIFO buffer is a circular buffer that stores a certain window of samples. This buffer is used to carry out the convolution in the FIR filters. The FIR filter bank consists of a "master" FIR filter, and 6 slaves. The master filter is the only one that request samples from the FIFO buffer and then passes on those samples to all of the slave filters. This is done such that there only needs to be a single request to the memory at a time. The FIR filter topology itself is a fully pipelined multiply-accumulate architecture. The filter uses a single multiplier running a convolution in a loop between the audio samples and the filter coefficients. The filter coefficients themselves are pre-loaded into memory utilizing a memory initialization file that loads the designed filter into read only memory. The modules are described in more detail in the further sections:

1. SPI:

The FPGA's SPI module implements a simple SPI slave that listens for 16-bit words. The module will take MOSI, CS, and SCLK as inputs connected from external pins. It will combine the received bits into 16 bit words and output them as q. It also takes internal input d to be sent

out as MISO. Another important logic element is the signal that keeps track of when a full word has been completely received. This "done" signal goes on to the parsing FSM which routes gains and samples to appropriate subsystems.

2. Parsing FSM:

The parsing FSM groups counts every other "done" signal from the SPI module and sends the equalizer gains onto the FIR filter bank to wait until it gets used, and the audio samples onto the FIFO buffer to fill up the convolution window.

3. FIFO Buffer:

The FIFO buffer stores and updates a certain window of input audio samples such that the FIR filters can request samples from the window in order to carry out a convolution. As new samples come in, the values themselves do not move and a pointer instead moves, making all reading and writing relative to the position of said pointer. The FIFO buffer takes in an address requested by the master FIR filter from the bank and will output the requested value.

4. Coefficient Memory:

The coefficient memory is a read only memory that is initialized at synthesis with the values of all the coefficients required to design the different FIR filters. There will be as many memories as there are filters given that the coefficients themselves are what define the filters, otherwise they are just convolution modules. These values will be determined ahead of time by designing the filters in MATLAB and converted to the appropriate bit depth.

5. FIR Filter Bank:

The FIR filter bank implements 7 FIR filters in parallel that will each carry out a convolution between a buffer of audio samples and the coefficients stored in memory. The resulting signals will be multiplied by the equalizer gains and then added together to create an aggregate filtered signal to be sent back to the Pi via SPI. A master filter will send address requests to the fifo modules and send the samples to the rest in order to avoid issues with communicating with a single memory. However, every filter will communicate with its own coefficient memory since these requests are independent of one another. The received coefficients and samples will be used in a pipelined multiply-accumulate topology to implement the convolution and output the final value to the SPI communication module.

6. LED Array Logic:

The LED array logic module would use 16 pins from FPGA to display the equalizer gains as lit bars on the array. In order to display multiple bars simultaneously on the array, time-multiplexing of the pins pulling the LEDs high and low would have to be implemented.

However, the LED array was not in the final version of the project and thus was not written in code.

Results

While a fully integrated deliverable was achieved in the end, our final project did not achieve all deliverables outlined in our project proposal. On the Raspberry Pi side, almost all systems are robustly implemented. The output audio quality is very good considering it is PWM audio, with only a small hiss of noise. The shared memory seamlessly lets different buttons interact with the main loop that plays audio. However, running the main loop for an extended amount of time makes the Pi's microcontroller chip to heat up. Furthermore, the timer peripheral that controls the sample output rate is not absolutely reliable, nor exactly accurate, as there is a possibility of timer failure, and the speed is a best estimation of the correct sample rate. However, the possibility of failure has been observed to be infrequent and the audio sounds close enough to the accurate despite the limitations of the timer peripheral.

On the FPGA side, a fully functional FIR filter module was successfully integrated with FIFO and SPI modules. This worked in both simulation and reality. However, in extending the full setup to include a full filter bank with 7 filters each with their own gains, we obtained a working simulation, but the code did not work in reality. The team suspects this was due to timing complications extending from an asynchronous design. Complications that a single FIR filter did not have issues with, but when expanded caused the full FIR filter bank to fail. A timing diagram showing the full FIR bank working in simulation is attached in the appendix. Finally, the 8x8 LED array was not able to be implemented due to time constraints. In total, instead of a web enabled speaker with a 7-bin EQ, our final delivered project was a web-enabled speaker with a toggle-able high pass filter. All Raspberry Pi functionality was achieved as desired and most FPGA functionality was as well. However, while simulation showed the system to function, reality did not prove to be as amenable.

Bibliography

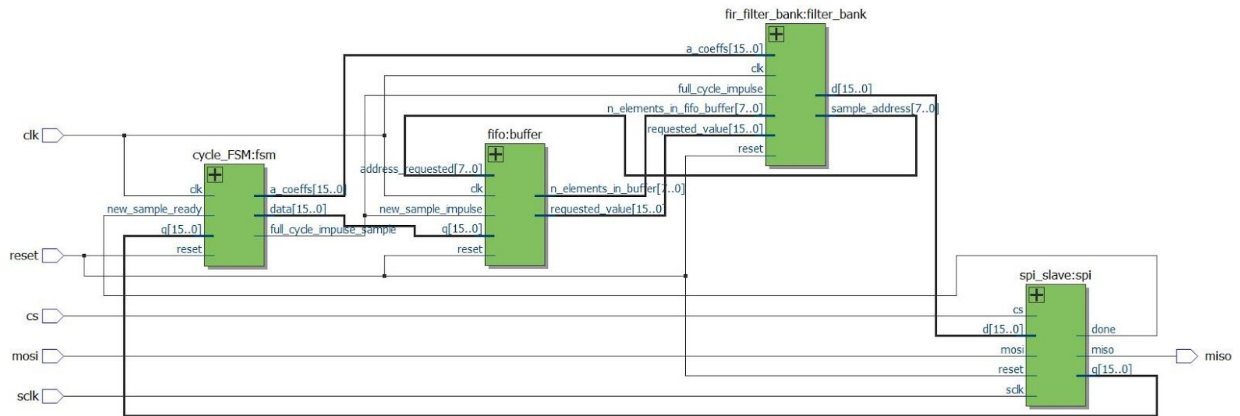

1. Butterworth Sallen Key Topology - https://en.wikipedia.org/wiki/Sallen%E2%80%93Key_topology
2. Adafruit Amplifier Usage - https://learn.adafruit.com/stereo-3-7w-class-d-audio-amplifier/inputs-and-outputs
3. Shared memory Example - https://users.cs.cf.ac.uk/Dave.Marshall/C/node27.html
4. Raspberry Pi Peripherals - https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf
5. Digital Design and Computer Architecture Ch 9 - http://pages.hmc.edu/harris/class/e155/09_Ch%2009_online.pdf
6. EasyPIO.h - http://pages.hmc.edu/harris/class/e155/EasyPIO.h


Parts List

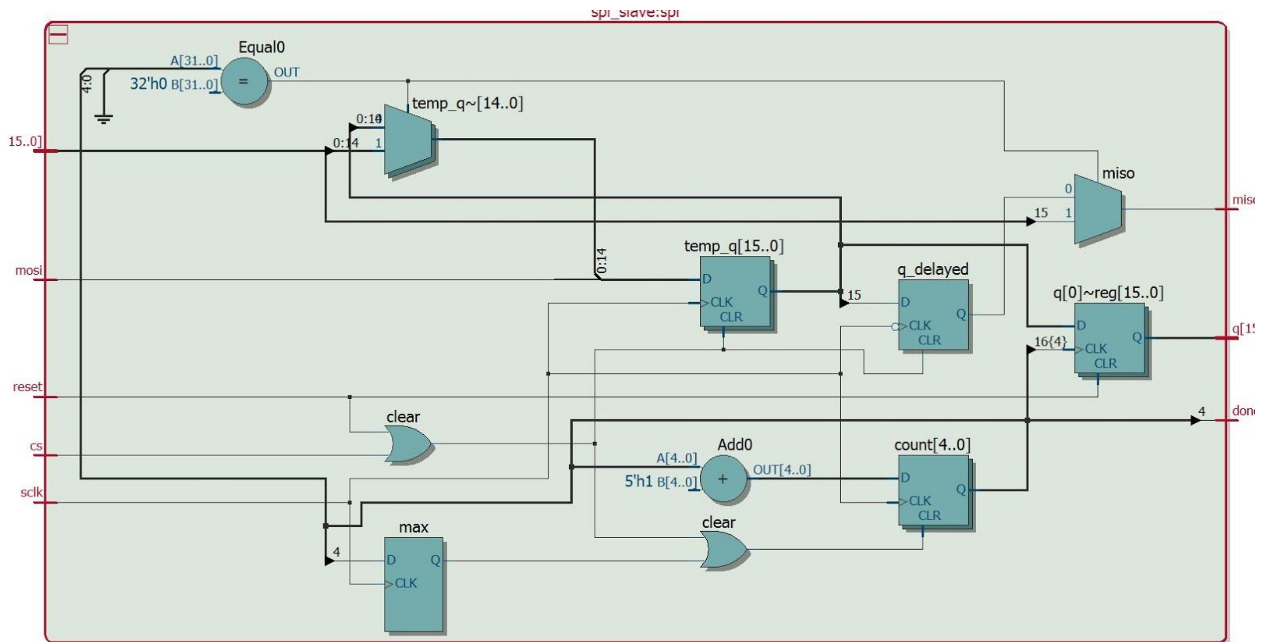
| Part | Source | Vendor Part # | Price |
|---|---|---|---|
| MCP6002 Op Amp | stockroom | | |
| Stereo Amplifier MAX98306 | Adafruit | 987 | $8.95 |
| (2x) 3” Speakers | Adafruit | 1314 | $3.90 |
| 8x8 LED Array | Adafruit | 455 | $3.95 |

<u>Appendix</u>

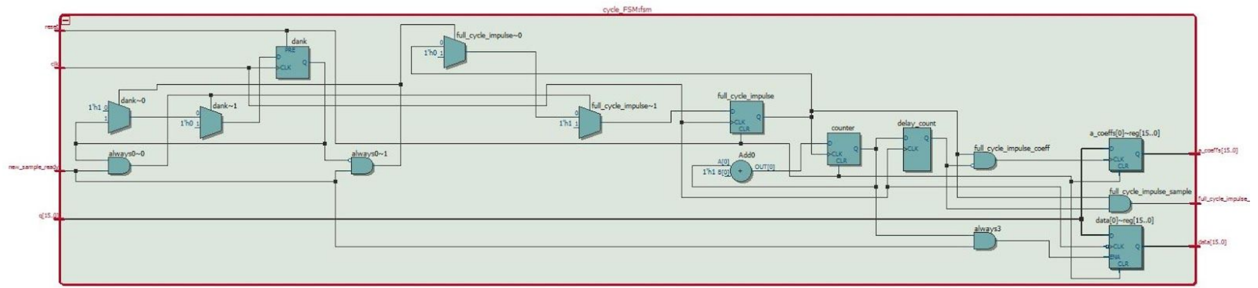
Top Module:



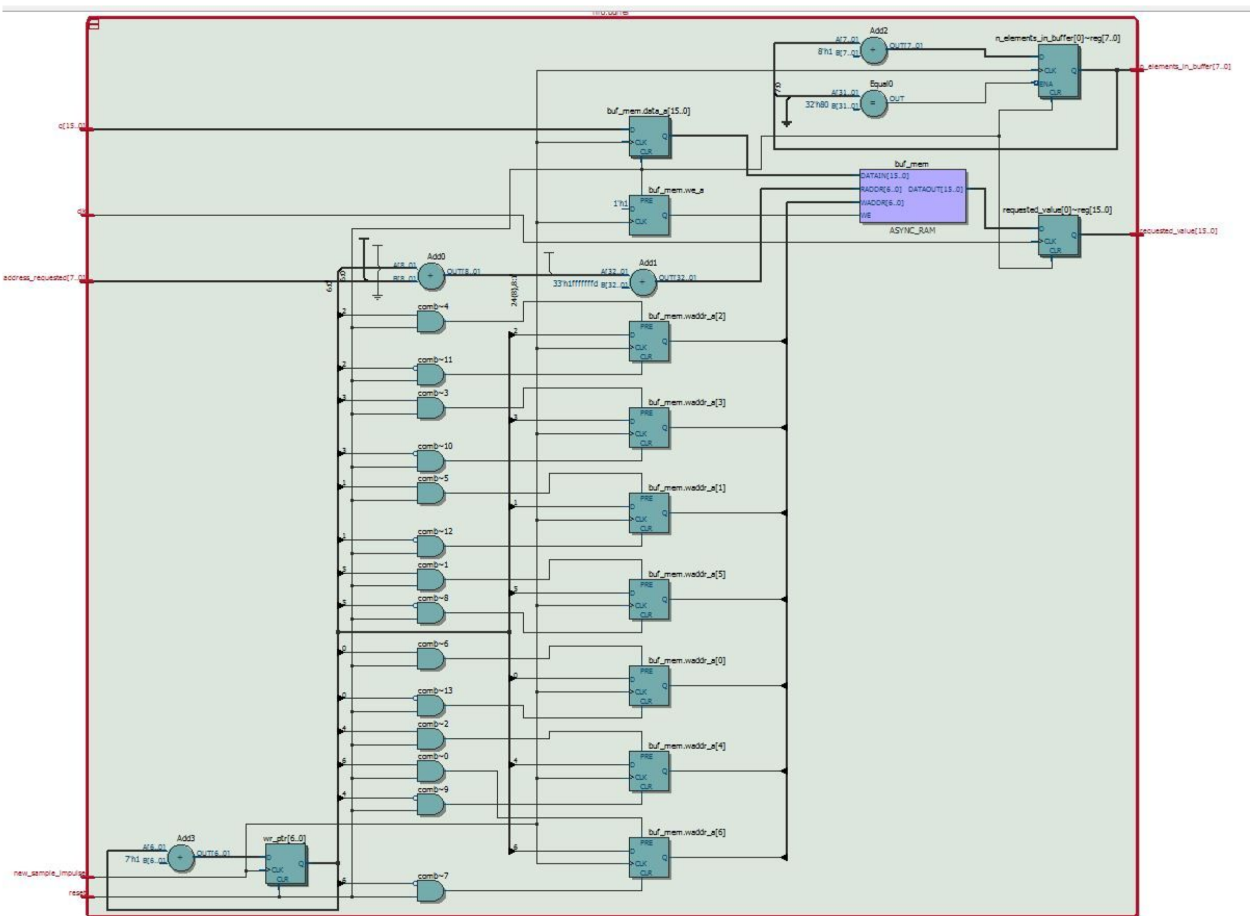

SPI Module:



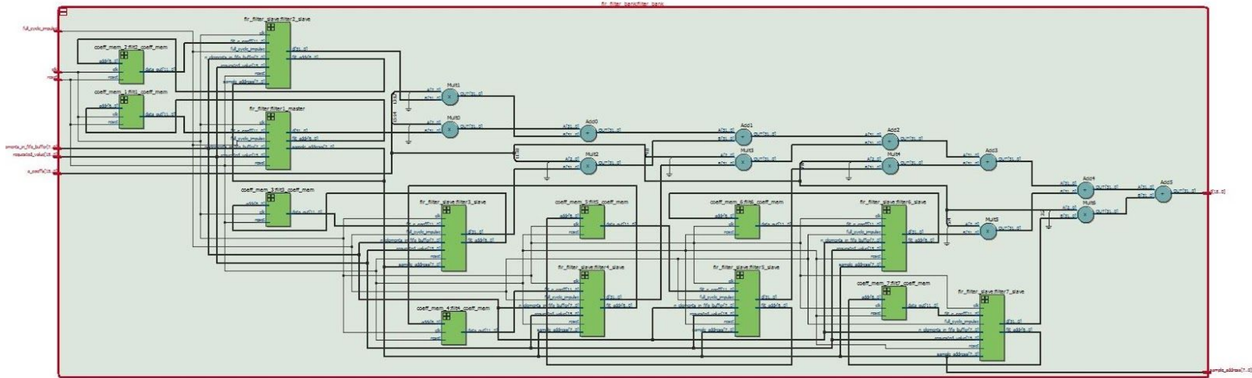
FSM:



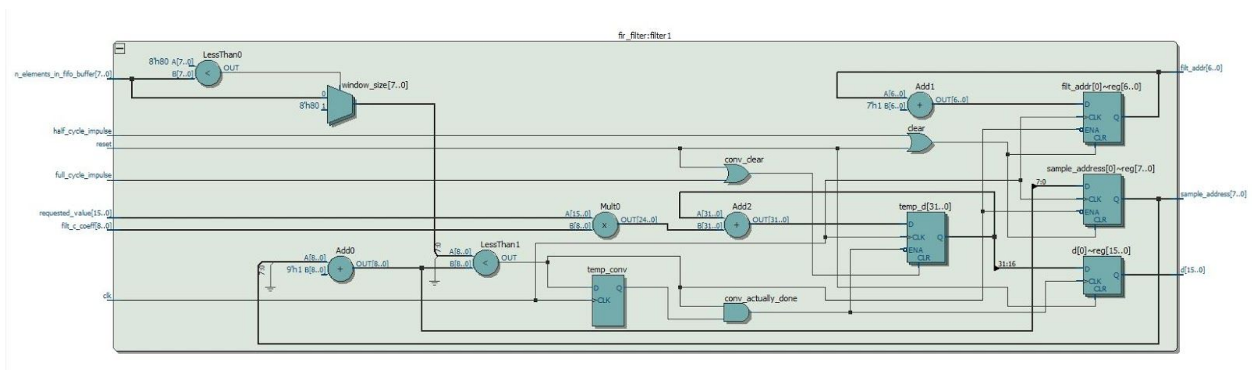

FIFO Buffer:



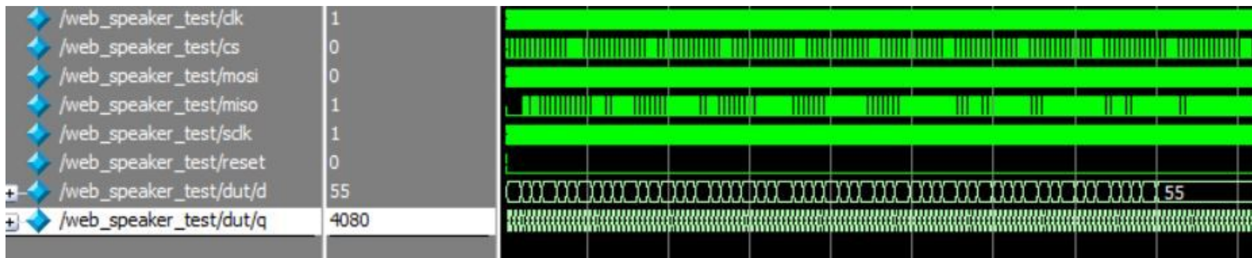
Filter Bank:



FIR Filter




Filter Bank Simulation Steady State Results:




Expected 55 = 0x37 = (0x0ff0 (input) * 0x80 (window size) * 0x7 (number of filters)) >> 16
Given filter tested was a unity gain, and equalizer values were all set to unity.