Implementations of Finite Impulse Response Filter

Will Wu
ESE-465: Digital Systems Laboratory
Washington University in St. Louis
willwu@wustl.edu

***Abstract*—This report introduces two different implementations of a Finite Impulse Response (FIR) low pass filter. One is implemented as a stand-alone C-program, while the other a hardware peripheral connected to a microprocessor. Through extensive testing and comparison, we find that hardware-implemented filters run much faster than their software counterparts. For hardware filters that experience software bottleneck, we implement optimization techniques that boost filter performance by a maximum of 57%.**

I. Introduction

Finite Impulse Response (FIR) filters are widely used in the electronics industry to treat and filter signals. In this paper, we present two implementations of the same FIR low pass filter. The first implementation takes the form of a C-Program running on a Microblaze® microprocessor, which is implemented on a Xilinx® Field Programmable Gate Array (FPGA) board. The second implementation takes the form of a hardware peripheral connected to the aforementioned microprocessor via an AXI4-Lite bus. Both implementations utilize the same underlying mathematics principle of a FIR filter.

This report is structured as follows: section II outlines the relevant mathematics behind a FIR filter; section III presents the results of testing, operation details and optimizing techniques, and section IV compares the testing results of two implementations and provide recommendations for implementing FIR filters.

II. Design

A. Mathematic Mechanics of a FIR Filter

An FIR filter is a linear and casual system. Given the system properties, the output signal $y[n]$ can be expressed as the convolution product of the input signal $x[n]$ and the filter impulse response $h[n]$. Note that all signals are in discrete time.

$$y[n] = x[n] * h[n] = \sum_{i=0}^{N} h[i]x[n-i] \tag{1}$$

where N signifies the order of the impulse response ($N+1$ signifies the number of impulse response terms). [1]

B. Data Structure and Representation

Given that the filter will be running on the Microblaze® microprocessor, which supports 32-bit data transferring at maximum, we choose to use the 1.15 signed decimal format to represent both the filter coefficients and the samples. The result from each multiplication operation in a convolution will take up two times the data size of the multipliers. Given this nature of binary/hexadecimal operations, two 16-bit multipliers ensure the multiplication product can be stored and transmitted inside the Microblaze®.

The forward and backward conversion from decimal format to 1.15 hex/binary format are carried out externally by either a python script or an Excel spreadsheet. Both the input signal and the filter coefficients are represented using the 1.15 format. The mechanics of the forward/backward conversion to 1.15 format exceed the scope of this report, and therefore are not covered. It is important to note that in the scope of the filter, regardless of the implementation format, all signals and filter coefficients are stored as 16-bit signed integers.

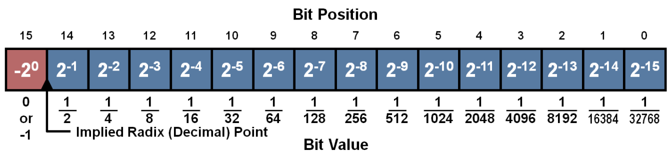


Fig. 1. Bit Values of 1.15 Format. Credit: Microchip Developer

We store the discrete-time output signals from our filter in a similar fashion to our filter coefficients and input samples - in the 1.15 format. A conversion from a 32-bit multiplication product to a 16-bit output is thus required. We apply the "round-and-truncate technique" for the conversion. The 32-bit product from two 16-bit number in 1.15-format follows a very similar bit-value layout to its multipliers. To round and truncate this number, we first discard the most significant bit (by a single left shift on a fixed-sized register) to convert the 32-bit product into a 1.31 format number. Before we discard the 16 least significant bits, we round our current number by adding half of the value of the least significant bit after rounding to the 32-bit number. For example, if we are discarding the least significant 16 bits, we add half of the value of bit 17 (which has a value of 2^16 in a 32-bit integer) to our output. After rounding the number, we discard the least significant 16 bits and interpret the most significant 16-bits as a number in 1.15 format. Note that this round-and-truncate operation is analogous to rounding and truncating a decimal number.

With every sample input fed into the system, the filter would carry out $N+1$ Multiply-And-Accumulate (MAC) operations to calculate the convolution product. Given the mathematical mechanics of the FIR filter, we choose to implement an circular buffer to hold input sample signals given the fixed number of MAC operations for each sample as shown in "Fig. 2". The write pointer of the circular buffer travels clockwise in accordance to the clockwise layout of the buffer, indicating the index of the latest sample. Given (1), we note that the first filter impulse response term is always multiplied by the latest sample, and the earliest sample received is multiplied by the last filter impulse response term. Hence, the read-pointer of the circular buffer travels in a counter-clockwise fashion, accessing samples received in a reverse chronological order.

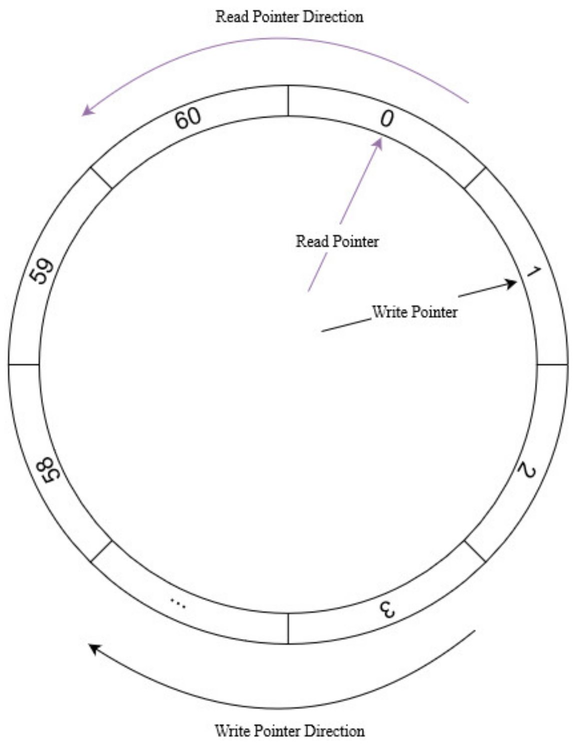


Fig. 2. Illustration of Circular Buffer Implementation

Although not pictured in "Fig. 2", both implementations also include a third, "filter impulse response" pointer, indicating the corresponding impulse response term for the sample. This pointer always starts at position 0. As the read pointer travels counter-clockwise by 1, the impulse response pointer travel clockwise (increase) by 1.

C. Software Filter Implementation

The software implementation of our FIR low pass filter takes the form of a C-program running on a Microblaze® microprocessor. The filter C-program is attached in the appendix of this report. The C programming language is a universal programming language, running on microprocessors or desktop processors alike. One relevant distinction of a Microblaze® microprocessor from other platforms lies in the data size of the processor. On a Microblaze® microprocessor, integer values are 32-bits, while *short* values are only 16-bits.

We designed the algorithm of our software implementation as follows:

Algorithm 1 FIR Filter Software Implementation

```
Require: Filter impulse response terms available to the program as $Filter$
INPUT: Input sample signal in discrete-time (as an array $Sample$)
OUTPUT: Filtered signal in discrete-time
 1: Initialization $Read = 0$, $Write = 0$, $Impulse = 0$
 2: Circular Buffer: Initialize an array $circ\_buff$ with the same number of elements as the number of impulse response terms
 3: while $Write \leq 60$ do
 4:     $circ\_buff[Write] = 0$
 5: end while
 6: $Write = 0$, $Result = 0$
 7: for $index = 0$, $index \leq Sample\ size$ do
 8:     $circ\_buff[Write] = Sample[index]$
 9:     Set $Read = Write$, $Impulse = 0$
10:     while $Filter \leq 60$ do
11:         $Result+ = Filter[Impulse] \times circ\_buff[Read]$
12:         $Impulse$++, $Read$–, if $Read = 0$, then set $Read = 60$
13:     end while
14:     Output $Result$ as the output discrete signal at corresponding time
15:     $Write$++, if $Write = 60$, then set $Write = 0$
16: end for
```

Note that $Result$ can either be presented as console text print outputs or collected in an array for later usage.

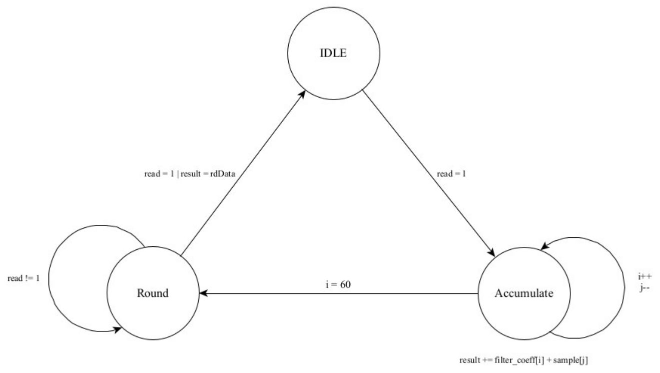

IDLE
read = 1 | result = rdData
read = 1
read != 1
Round
i = 60
Accumulate
i++
j--
result += filter_coeff[i] + sample[j]

Fig. 3. Logic Diagram of the finite-state Machine

D. Hardware Filter Implementation

The underlying mechanics of the hardware implementation of the FIR low pass filter are very similar to that of the

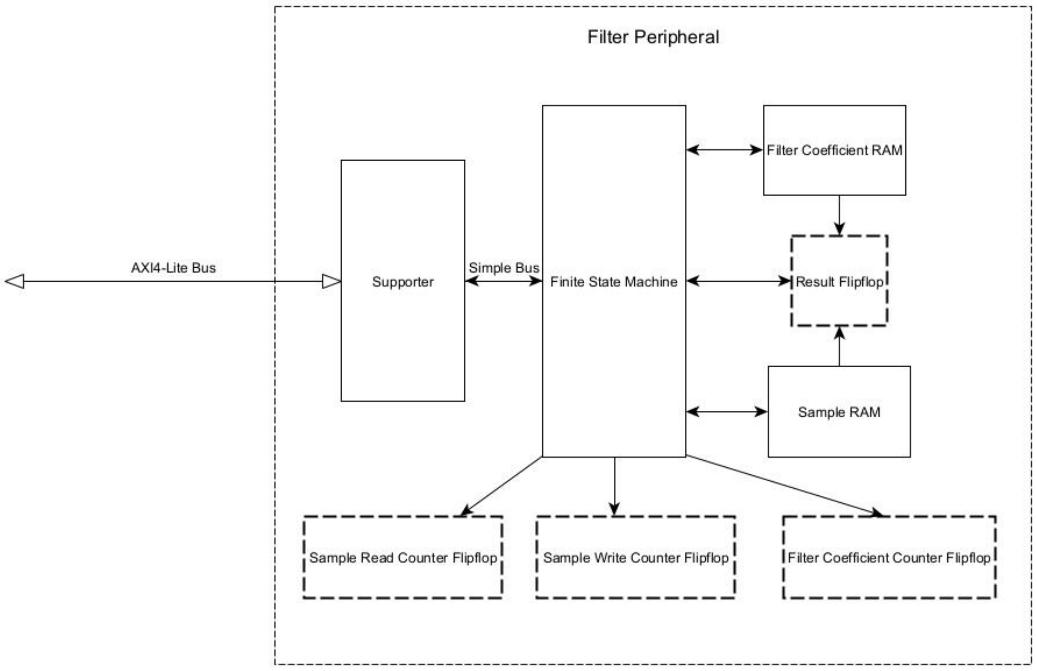


Fig. 4. Illustration of the Hardware Implementation

software implementation. The logic for conducting MAC operations are implemented as a Mealy machine in "Fig. 3".

The same circular buffer and read/write counters are implemented in the form of asynchronous-read Random Access Memories (RAM) and flip-flops (as shown in "Fig. 4)".

Note that since the filter peripheral does not contain dedicated memory to store results, the processed sample output must be extracted before the circular buffer accepts another sample. Therefore, the finite state machine will hang in state $round$ until the result is read/extracted by the Microblaze®.

III. Operation/Testing

Two sinusoidal test signals are used to test the performance of our FIR low-pass filter: one with frequency within the pass-band, and the other with frequency within the stop-band. Test signals are sampled at $50kHz$ for $0.02s$. Overall, 1000 sample points per test signal are generated. As shown in "Fig. 5", the $1kHz$ and $9kHz$ sinusoidal test signals both share a magnitude of 0.9. We set the maximum amplitude to 0.9 to avoid potential buffer overflow during MAC operations, given the maximum fraction value 1.15 format can represent is always less than one, as shown in (2).

$$N_{max} = 0 \cdot -2^0 + \sum_{i=1}^{15} 2^{-i} \approx 0.9999695 \qquad (2)$$

Test signals are discretized by an Excel spreadsheet and imported to both the software and hardware implementation via a C-header file. The details regarding signal import will be discussed in specific implementation subsections below. Filter impulse response terms are imported into both implementations in a similar fashion. We utilize a FIR filter design tool [2] provided by the instructor to generate the impulse response terms. Specifications regarding the filter are provided in Table I below.

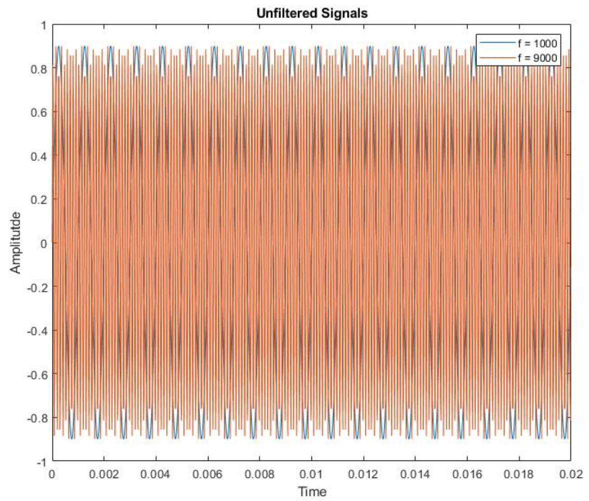


Fig. 5. Test Sinusoidal Signals

All testing and design are conducted using a Digilent Nexys™ 4 DDR development board. The development board houses a Xilinx™ Artix-7 FPGA chip, and has variable serial connections, peripherals, switches and displays. The Microblaze® processor is implemented using the onboard FPGA, and

TABLE I
FILTER SPECIFICATIONS

Filter Specifications	**Parameters**					
	f_{sample}	$f_{passband}$	$f_{stopband}$	Stopband Attenuation	Passband Ripple	Filter Order
Value	$50kHz$	$3.3kHz$	$6kHz$	$60dB$	$8.681mdB$	61

runs at a clock frequency of 30 Mhz.

A. Software Filter

a) Operation: Implemented according to the algorithm in section II-C, the FIR filter C-program imports two array header files that contain both the filter impulse response terms and the discretized input signals. The program, running on the Microblaze® processor, processes the input signals and records the output signals in an array. The user can choose to either pass the array on for further processing, or output the array via console text print.

b) Testing: For our low-pass filter implementation, we are mostly interested in the filtering/conversion rate of the filter implementation. The filtering of each discrete time sample is associated with a convolution operation - $N + 1$ MAC operations. Timing the MAC operations, therefore, can provide an indicative measurement of the performance of our FIR filter.

To time the MAC operations performed by our software filter, we deviate away from any software timing tools provided either by the compiler or the debugger. Instead, we choose to time the software externally using the circuit elements on the Nexys™4 DDR board that houses our Microblaze® processor.

Two PMOD pins are tied to two bits of memory associated with our Microblaze® processor. The C-program writes to these memory prior to any MAC operations, resulting in a $3.3V$ logic high output on the PMOD pins. After the processor complete all MAC operations, the program clears out the memory associated with the output pins, resulting in a logic low output. Using an Oscilloscope, we can easily calculate the amount of time elapsed between the logic high and logic low output, and subsequently time the program.

c) Testing Results: The filter program successfully retains the magnitude of the pass-band signal while attenuates that of the stop-band signal. "Fig. 6" shows the output signal. The results of hardware timing and calculated rates are presented below in Table II

TABLE II
SOFTWARE FILTER TIMING RESULTS

Timing Results	**Parameters**				
	Total # of MAC Ops	***Total Time***	***MAC/s***	***MAC/ Sample***	***Samples/ s***
Value	61000	$333.4ms$	182964	61	2999.4

We also compared the theoretical input lag to the actual filter input lag based on the input/output signals. Input lag results from partial convolution, where the number of impulse response terms exceed the number of total received samples. It can be calculated as shown below in (3).

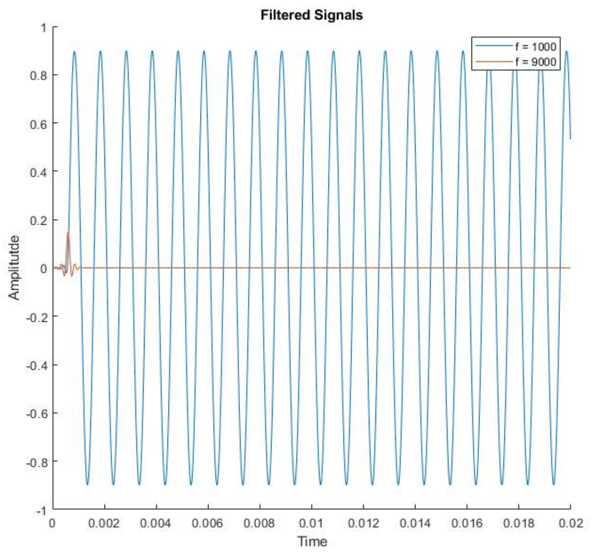


Fig. 6. Illustration of the Hardware Implementation

$$t_{lag} = \frac{\text{FIR Filter Order} \cdot T_{sample}}{2} = \frac{61}{2 \cdot 50kHz} = 0.61ms \tag{3}$$

Experimentally, the (time) distance between the first input signal peak and the first output signal peak of a pass-band signal yields the filter lag. As shown in "Fig. 7", the emperically measured lag is $0.6ms$.

B. Hardware Filter

a) Operation: Functionally, the hardware filter implementation is identical to its software counterpart. The hardware FIR filter is implemented as an AXI4-Lite peripheral using the same Xilinx® Field Programmable Gate Array (FPGA) board. The filter peripheral communicates with the Microblaze® processor using the AXI4-Lite bus. Based on the hardware and logic design given in section II-D, we constructed and realized the hardware implementation using our Nexys™4 DDR board. The Verilog description of the hardware peripheral is available in the appendix of this report.

To use this peripheral in an isolated fashion, an interface program running on the Microblaze® microprocessor is required. This program serves both as an filter interface that provides the impulse response terms and an Analog-to-Digital Converter (ADC), which provides input samples to the

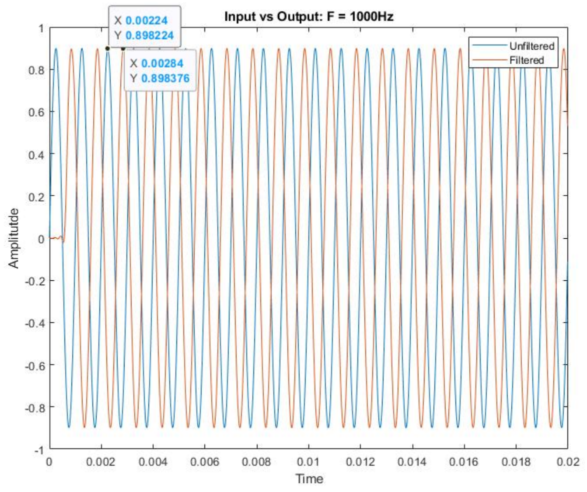


Fig. 7. Illustration of the Hardware Implementation

peripheral in discrete time. In addition, this program collects the output signals from the filter peripheral.

Analogous to how the software filter operates, the interface program access the peripheral first to write all filter impulse response terms to the dedicated, asynchronous-read RAM. The program then zeros out the sample-dedicated RAM to prepare for convolution operations. The interface program writes in discrete-time samples one at a time while extracting the processed output. After each sample-write operation, the program hangs the sample-writing process and continuously polls the peripheral until the filtered results is received.

b) Testing and Results: The filtered results obtained from the hardware peripheral are identical to those produced from the software implementation. However, the hardware filter peripheral is significantly faster at processing samples, as shown in Table III.

TABLE III
HARDWARE FILTER TIMING RESULTS

Timing Results	**Parameters**			
	Total # of MAC Ops	***Total Time***	***MAC/s***	***Samples/ s***
Value	61000	$4.8344ms$	12617905	206851

c) Software Bottleneck within Hardware Implementations: Given the layout of our finite state machine, when the filter remains in the "accumulate" state, every MAC operation takes exactly 1 clock cycle to complete. Given the $30MHz$ clock frequency of our Microblaze ®microprocessor, our maximum MAC/s rate should approach the Microblaze clock frequency, $30M$ MAC$/s$. In reality, our hardware filter is only able to achieve a maximum rate of $12M$ MAC$/s$.

Upon further examination, we discover that with every input sample, the finite state machine hangs in the "round" state and wait for the interface program to collect filter output data. Given our implementation, our interface program continuously polls the hardware filter until the MAC operations are complete. We hypothesize that reducing the software polling time - therefore reducing the finite state machine hang-time - would increase the hardware filter processing time as a result. Moreover, because our interface also serves as an ADC, increasing the "conversion rate" of our interface program can hypothetically boost the sample processing rate.

To reduce the software polling time and increase the sample-writing rate, we introduce loop unwinding, an optimization technique that sacrifices space complexity for time complexity. Programs written in C are translated to machine instructions in assembly by the compiler. During the translation process, complex branching commands are introduced. A loop-unwinded program conducts multiple loop operations in one sitting before checking the loop condition, thus avoiding extra branching commands. Given our finite state machine needs a set of amount of time to conduct MAC operations, we can safely loop-unwind our interface program and conduct a set number of polling operations without checking the loop condition that resets the finite state machine back to "idle" state. Overall, we see a speed increase in our filter performance after eliminating some branching commands from the interface program. Table IV shows the consequent MAC rate boosts from successive loop-unwinding.

TABLE IV
HARDWARE FILTER LOOP-UNWINDED

Timing Results	**Specifics**		
	Elasped Time	***MAC/s***	***Speed Increase*** (%)
Control	$4.8344ms$	12617905	NA
1x Polling -loop Unwinded	$4.1340ms$	14755684	16.94%
2x Polling -loop Unwinded	$3.8838ms$	15706267	24.48%
4x Polling -loop & 2x ADC -loop Unwinded	$3.0760ms$	19830949	57.17%

IV. DISCUSSIONS AND CONCLUSIONS

Though both implementations use identical logic and accomplish the same task, it is clear that the hardware implementation filters signals at a drastically faster rate. The software implementation can process signals around a rate of $3k$ MAC$/s$, while its hardware counterpart filters signals at around $207k$ MAC$/s$,This drastic speed difference could be attributed to the nature of software written in a high-level programming languages like C. The software implementation, on the surface level, carries out the same operations as the hardware implementation. Yet when the program is compiled into machine instructions, operations such as array-memory allocation and loop branching collectively consume much more time. Inversely, the hardware implementation eliminates memory allocation time by implementing dedicated storage, and cuts down on loop branching with a finite state machine, which constraints the maximum loop branching time to one clock cycle.

Given the widespread nature of FIR filters, it is strongly recommended for engineers to implement dedicated hardware filters. As demonstrated by this report, software implementations of FIR filters are simply too inefficient comparing to hardware implementations.

References

[1] A. V. Oppenheim, J. Buck, M. Daniel, A. S. Willsky, S. H. Nawab, and A. Singer, “Signals and Systems,” Prentice Hall, 2nd Edition, 1997.

[2] LabVIEW Filter Design Tool, ESE-465: Digital Systems Laboratory, Washington University in St. Louis, 2022

[3] When is loop unwinding effective?
https://stackoverflow.com/questions/190681/when-is-loop-unwinding-effective

For code, please see: https://github.com/WillWu88/FPGA_Project_Reports/tree/main/Code/Filter

Implementation of Dual-Purpose ADC/DAC Serial Peripheral Interface

Will Wu, Xueke Ye
ESE-465: Digital Systems Laboratory
Washington University in St. Louis

***Abstract*—To equip our Microblaze® microcontroller with diverse signal processing capabilities, we need to interface with analog and digital signal converters. In this report, we present the design and implementation of a peripheral for the Microblaze ® that utilizes the Serial Peripheral Interface (SPI) to communicate with signal converters. The peripheral we designed and implemented is dual-purpose, and therefore can be used to interface with both analog and digital signal converters.**

I. Introduction

Both analog and digital signals are widely used in the electronics industry. With their individual strength and weaknesses, analog and digital signals are each well-suited to transmit their subset of information. For example, analog signals possess higher information density, making them suitable to transmit video or audio signals. To receive and transmit both types of signals, conversion between analog and digital signals is required.

Dedicated circuits have been designed to carry out such conversions. An Analog-to-Digital Converter (ADC) samples analog signals at a set sampling rate and transcribes the samples collected into digital signals. A Digital-to-Analog Converter (DAC) performs the reverse function. To the conversion capabilities of these devices, our Microblaze ® Processor requires an interface to transmit/receive necessary signals to the external converters.

Developed by Motorola, the Serial Peripheral Interface(SPI) [2] offers a robust time-synced bidirectional communication interface. In our case, building a SPI peripheral for the Microblaze® processor would allow it to interface with the signal converters. Hence, in this report, we present our design of a dual-purpose SPI peripheral for the Microblaze® that can both interface with ADCs and DACs. Our report is structured as follows: section II outlines the logic design of the SPI peripheral; section III introduces testing methods as well as testing results, and section IV discusses the implication of our implementation.

II. Design

The design of our SPI peripheral largely depends on the type of ADC and DAC we choose. In order to interface with both devices using a unified design, we need to take timing characteristics of both devices into account. We choose to interface with the *LTC1865* ADC and the *LTC1654* DAC, both manufactured by Analog Devices. Both devices offer SPI-compatible serial I/O for data transfer and device configuration.

a) Timing: Consulting the device data sheets [3] [4], we summarize the important timing requirements into the following table:

TABLE I
Important ADC and DAC Timing Constraints

Minimum and Maximum Timing Requirements			
Label	Item	ADC	DAC
t_1	$CONV/CS$ Setup time	$60ns$ (min)	$15ns$ (min)
t_2	SDI setup time before SCK high	$15ns$ (min)	$45ns$ (min)
t_3	SDI hold time after SCK high	$15ns$ (min)	$0ns$
t_4	SCK high time	40% T_{SCK}	$20ns$ (min)
t_5	SCK low time	40% T_{SCK}	$20ns$ (min)
t_6	$CONV/CS$ low to valid SDO delay	$120ns$	N/A

Given the timing characteristics of both devies, we note that the ADC poses more timing restrictions than the DAC. Hence, we prioritize to accommodate for the ADC timing requirements in our SPI peripheral design. Given a set sampling rate of $25kHz$ and two channels for audio signal, we implement a ”switch” sampling mode. The ADC will, in total, sample at $50kHz$, but it samples each channel alternatively. Consequently, signals from each channel are sampled at $25kHz$.

Given the timing requirements of both devices, we design the SPI peripheral to generate the following timing signals (as shown in Fig .1) in order to drive the ADC/DAC with correct timing. Note that the Microblaze® clock, clk, is also shown in the graph.

We have two important considerations regarding the timing design. First, all minimum setup and hold times must meet requirements even in the worse case scenarios. Second, even though DAC operations require lengthier SDI and SCK outputs, our SPI timing design still should meet the timing requirements for both. Given our timing design, we clearly meet the formal requirement: SCK remains low for three clk clock cycles after $CONV/CS$ goes low, satisfying the $30ns$ minimum set up time. Given that each SCK cycle takes

Authors of this report can be reached at: willwu@wustl.edu, x.ye@wustl.edu

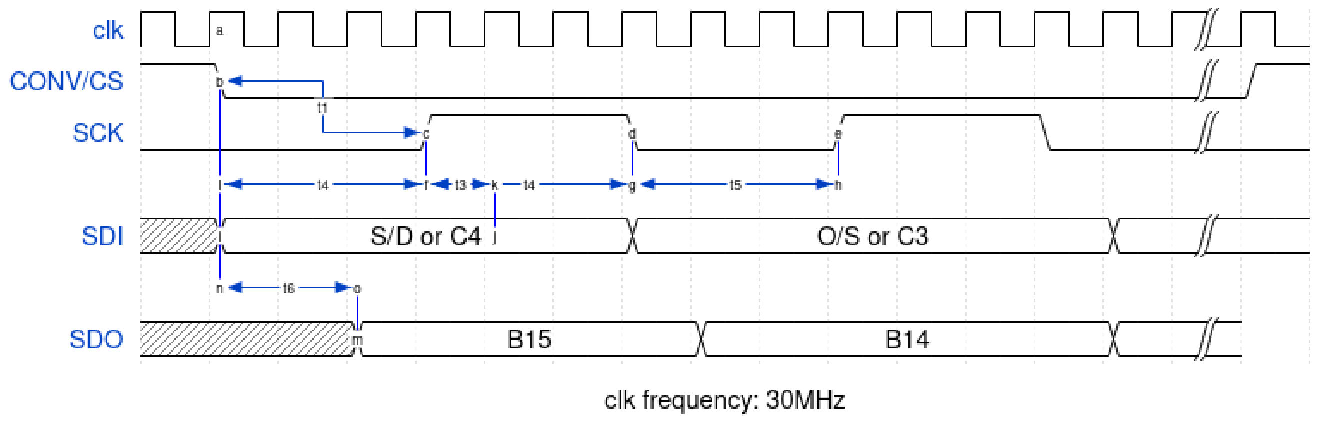


Fig. 1. SPI peripheral timing design

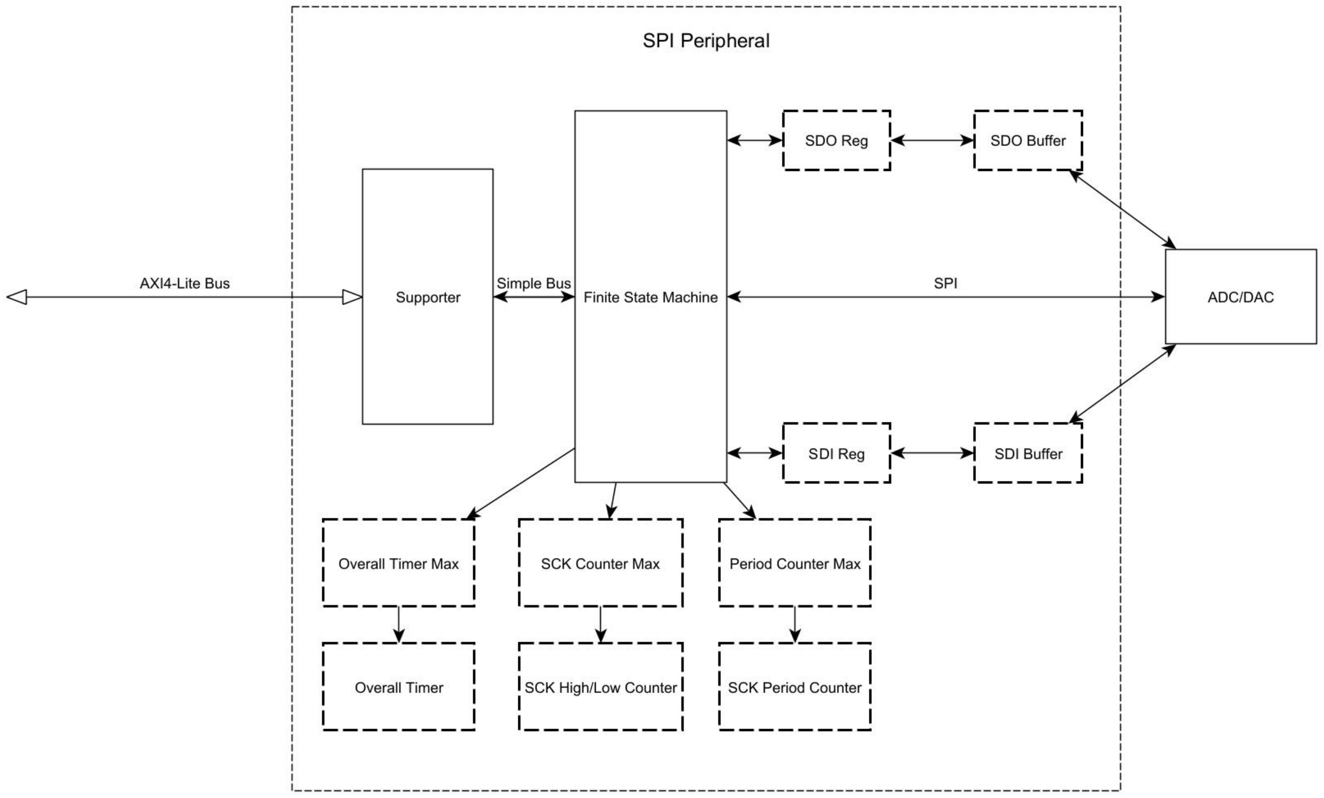


Fig. 2. Hardware architecture of the SPI peripheral

up $200ns$, and $CONV/CS$ must remain high for $3.2\mu s$, the timing design shown in Fig .1 still satisfies the sampling rate constraints, as shown by (1) and (2) below (given a $14ns$ delay):

$$\text{ADC: } T_{total} = 3200 + 16 \times 200 + 14 = 6414 < 10000(ns) \tag{1}$$

$$\text{DAC: } T_{total} = 3200 + 21 \times 200 + 14 = 8014 < 10000(ns) \tag{2}$$

b) Logic: The time requirement design provides our peripheral with timing constraints. Similar to other projects, we aim to use a finite state machien to represent the internal logic of our peripheral. Unlike previous projects, however, the transition between logical states are now affected by timing. In other words, previous finite state machines we designed stay in the same state until certain logical conditions are met. For our SPI implementation, the finite state machine stays in the same state for a certain period of time, which is constrained by our timing design. Timing in hardware, in reality, poses little challenge to implement. Since the Microblaze ® micro-

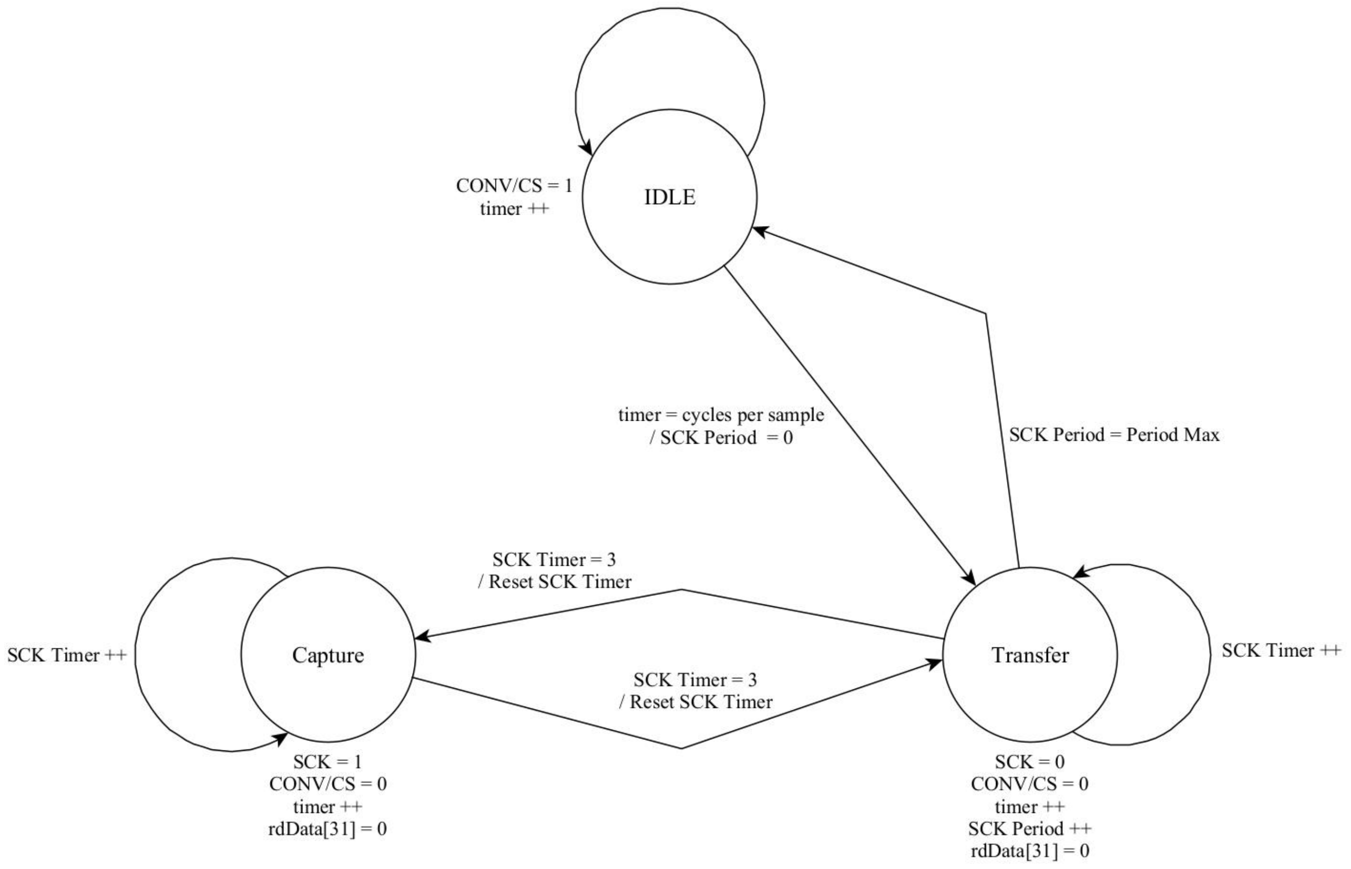


Fig. 3. Illustration of the finite state machine

controller runs on a set clock frequency, timing constraints can be represented as numbers of device clock periods. For example, since the Microblaze® runs at a clock frequency of $30MHz$, 3 complete clock cycles take up exactly $100ns$.

The finite state machine of the peripheral, designed based on the timing diagram and the DAC/ADC operation sequence [3] [4], is shown in Fig .3. A total of six timing registers, as shown in Fig .2, are tasked with timing the logical transitions used in the finite state machine that governs the SPI peripheral input/output signals. Three registers (that contain "max" in their labels) hold the period counts that represent the specific timing constraint, while the other three are used as counters. The timing constraint registers are modified by the Microblaze® processor, while the counter registers increase by 1 every clock cycle or by every SCK cycle. When the counter registers hit the maximum value indicated by the timing constraint registers, the finite state machine, as a result triggers state transitions.

We also implemented four memory registers to store SDI and SDO data. Out of the four registers, two are "buffer" memories while the rest are "program" memories. The separation between program registers and buffer memories is necessary to prevent potential conflict between the peripheral and the signal converter. The peripheral only updates its buffer registers when $CONV/CS$ outputs high, which implies that the interfaced converter is not actively waiting for the SDI input or actively outputting SDO.

III. Operation and Testing

a) Operation: The SPI peripheral is designed using Verilog, and implemented using a Xilinx Artix-7 FPGA, housed on a Digilent Nexys™ 4 DDR development board. We instantiate two SPI peripherals, one linked with the *LTC1865* ADC and the other with the *LTC1654* DAC. The peripherals are connected to the Microblaze® microprocessor with AXI4Lite buses.

We have written a C-program as the control program for the peripherals. The program serves three major functions. First, it configures the sampling frequency of the ADC/DAC peripheral by writing to the "Overall Timer Max" register. Given we sample at $50KHz$, and our clock runs at $30MHz$, we write in 600 as the maximum clock cycles per sample. Second, it configures the total period of peripheral SCK signal based on the type of converter connected: our ADC in use requires roughly 16 periods of SCK, while our DAC in use requires 24. Third and foremost, the C-program serves as a read/write interface. It reads the converted digital signals from the ADC while writes digital signals to the DAC to be converted. Note that all read/write operations follow a polling

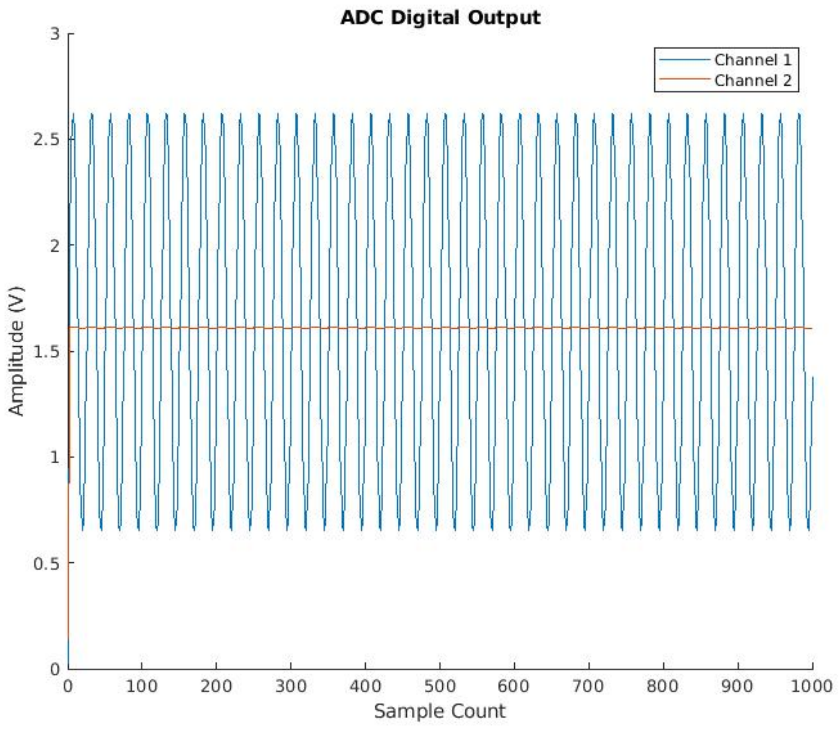
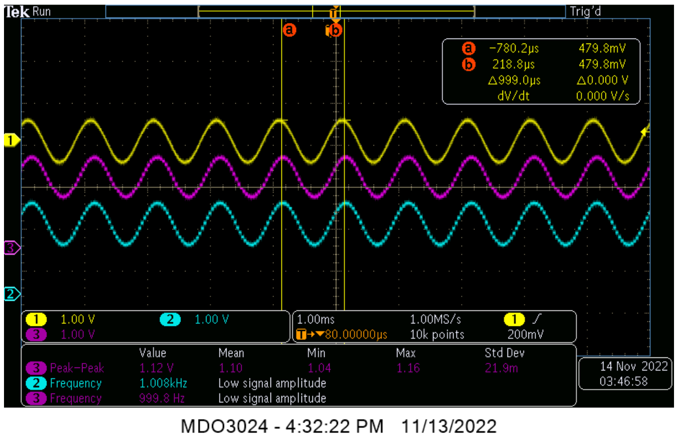


Fig. 4. First 1000 ADC samples

model that ensures only one new data is read/written to the peripheral within 1 sampling period. As the SPI peripheral provides a confirmation bit (bit 32 in "rdData") to indicate correct read/write timing, the program will continuously poll the peripheral for confirmation, but only reads/writes when it has confirmation. Serendipitously, the dual-purpose design of our SPI interface allows us to use only one control program. Additionally, it allows us to achieve a "loop-back mode", in which the program feeds sampled digital signals from the ADC into the DAC to replicate the original analog data. The source code to the C-program can be found in the appendix.

b) Testing: Following the board bring-up process, we examine the SPI signals first to ensure correct peripheral operation. Using an oscilloscope, we can capture the digital SPI signals between our SPI pheripheral and the converters. Fig. 6 (in the appendix) shows the captured SPI signals connected to an ADC. Note that the two cursors measures the $CONV/CS$ setup time before first SCK high. The setup time measures to $63.20ns$, which satisfies the minimum timing requirement of $30ns$. Fig. 7 subsequently shows correct channel switch during the ADC sampling cycle. Subsequent figures in the appendix, fig. 8 and fig. 9, demonstrate correct SPI signals and thus correct SPI operation when connected to a DAC.

ADC operations were tested first. Using a function generator, we feed in a sinusoidal signal with an amplitude of $1V$ and frequency of $1kHz$ into channel 1 of the ADC, while leaving channel 2 empty. We are expected to acquire a sine wave of the corresponding magnitude on channel 1, while a static signal on channel 2. Fig. 4 shows 1000 samples of the converted digital signal collected from channel 1 and channel 2. The signal frequency matches our expectation: $1kHz$. The signal magnitude of the sinusoidal signal also closely resembles the amplitude of the input signal: the samples yielded a peak-to-peak voltage of $2V$, which is equivalent to an amplitude of $1V$. Note that the magnitude of the output signals from channel 2 comes out to be $1.6V$, around half of the logical high voltage of the Nexys™ 4 DDR development board.



```
a  -780.2µs    479.8mV
b   218.8µs    479.8mV
   Δ999.0µs   Δ0.000 V
    dV/dt      0.000 V/s

1  1.00 V        2  1.00 V      1.00ms          1.00MS/s    1 ⨍
3  1.00 V                       T→▾80.00000µs   10k points  200mV

              Value      Mean                  Min    Max    Std Dev
3 Peak-Peak   1.12 V     1.10                  1.04   1.16   21.9m
2 Frequency   1.008kHz   Low signal amplitude
3 Frequency   999.8 Hz   Low signal amplitude

14 Nov 2022
03:46:58
```

MDO3024 - 4:32:22 PM 11/13/2022

Fig. 5. Loop back mode test result

DAC operations were subsequently tested by feeding a digital sinusoidal signal of 1000 samples into the SPI peripheral connected. The sinusoidal signal has an amplitude of $1V$ and frequency of $1kHz$. Using the "single trigger" mode on our mixed domain oscilloscope, we verify the correct operation of our DAC.

Lastly, continuous loop back mode is tested by simultaneously reading from both channels of the ADC and writing the converted digital signals to the ADC. We fed in a sinusoidal test signal of amplitude $1V$ and frequency $1kHz$ in both channels. Fig .5 shows the output signals (in blue and pink) from the DAC and the input reference (in yellow) into the ADC. We can consequently verify correct operation of the loop back mode.

IV. Discussion and Conclusion

This report presents the design, implementation and testing details of a dual-purpose SPI peripheral for the Microblaze® microprocessor. The peripheral can interface with both ADCs and DACs using the Serial Peripheral Interface. When instantiated correctly, the peripherals can operate in "loop-back" mode, where the output of the DAC replicates the input analog signal to the ADC.

Combined with our previous digital FIR filter, we now can process analog signals digitally. The SPI peripherals, in loop-back mode, enable us to filter and output analog signals in real time. Digital signal output from the ADC can be fed into our previously designed FIR filters, which attenuate signals in undesirable frequency bands. The SPI peripherals enable us to interface with signal converters, thus providing us with a solid base to construct a real-time digital audio equalizer.

References

[1] P. Dahker, "Introduction to SPI interface", Analog Devices, 2018
[2] Motorola, Inc, "SPI Block Guide V03.06", 2003
[3] Analog Devices, Inc, "LTC1864/LTC1865 Datasheet"
[4] Analog Devices, Inc, "LTC1654 Datasheet"

For code, please see: https://github.com/WillWu88/FPGA_Project_Reports/tree/main/Code/SPI

For more detailed graphs on the right, please see: https://github.com/WillWu88/FPGA_Project_Reports/tree/main/Figures/SPI

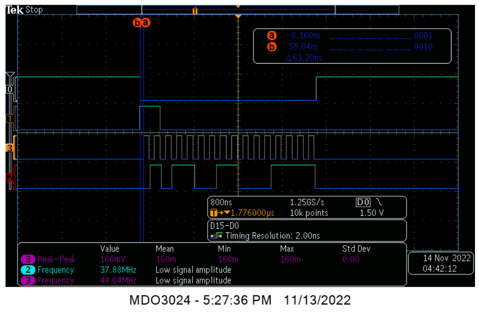


Fig. 6. SPI Signals from 1 ADC Transmission Period

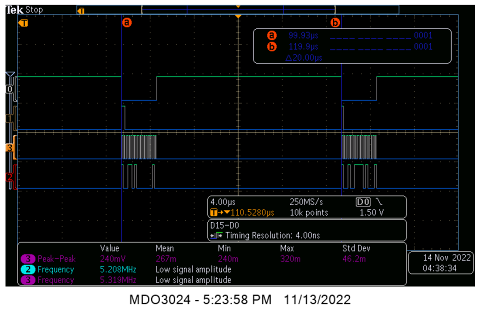


Fig. 7. SPI Signals from 2 ADC Transmission Period

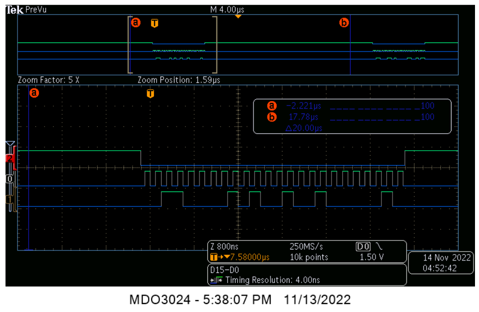


Fig. 8. SPI Signals from 1 DAC Transmission Period

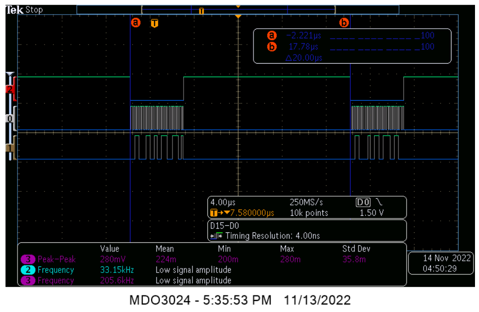


Fig. 9. SPI Signals from 2 DAC Transmission Period

The Audio Equalizer Axi4Lite Peripheral

Cassie Jeng
Washington University
Electrical & Systems Engineering
jeng.c@wustl.edu

Will Wu
Washington University
Electrical & Systems Engineering
willwu@wustl.edu

***Abstract*—Combining the signal conversion capabilities of Analog-Digital and Digital-Analog converters with the flexibility of digital Finite Impulse Response filters, an audio equalizer targets specified frequency bands and modifies their respective magnitude to achieve desired sound signatures. This report marks the culmination of our digital logic design series, as we combine our previous work on digital filters and signal converter interfaces into one functional equalizer device. This device processes audio samples with user-specified modification in real-time through a graphical interface.**

I. Introduction

An Audio Equalizer is a digital logic filter implemented by individually isolating bands of frequencies. Each frequency is either attenuated, amplified, or unaltered depending on its associated attenuation factor. When the frequency bands are recombined, the result is the original audio with different proportions of dominating frequencies. Audio equalizers are used in various hardware devices and used for effects such as bass-boosting or filtering high/low pitched background noise.

This audio equalizer design and implementation made use of all the individual components previously designed in Digital Systems Laboratory: an Axi4Lite Manager and Supporter, a Finite Impulse Response (FIR) Filter, and a Serial Peripheral Interface (SPI) module to interface with both an Analog to Digital Converter (ADC) and Digital to Analog Converter (DAC). Each module processes the input signal sequentially in order to output the attenuated version to connected speakers. The input signal flows from the ADC to the FIR Filter to the DAC. Signal control is managed by the equalizer Finite State Machine (FSM) to ensure correct timing specifications are met for all incorporated components, as well as that each frequency band is correctly multiplied by its appropriate attenuation factor without corrupting the data.

There are many design and implementation aspects to consider in constructing digital logic to combine several different pre-designed modules. Many of the modules are altered when adding them to the equalizer in order to support two channels, 13 bands, and all timing restrictions for set up and implementation. The overall equalizer FSM managed communication with all modules through the respective Axi4Lite Managers. This report outlines our equalizer design, and presents signal flow, individual components, and overall implementation in detail, as well as includes figures from operational testing through the scope and live audio.

II. Equalizer Design

Combining the digital signal processing capability of band-pass filters and the signal conversion capabilities of ADC and DAC, an audio equalizer processes analog sound signals and either boosts or attenuates signals in specified frequency bands. Given that we have successfully designed and implemented all necessary components of an equalizer in previous projects, our work will mainly focus on combining previously designed components and ensure the correct signal flow. The equalizer first collects a digital sample through the ADC SPI interface and then feeds the digital sample into the digital FIR filter for band separation and augmentation/attenuation. After proper processing, the equalizer feeds the processed sample into the DAC for analog audio output. Fig. 1 visualizes the entire signal flow described. While the FIR filter receives an update to expand its processing capabilities, the SPI module that interfaces with both the ADC and the DAC remains unchanged.

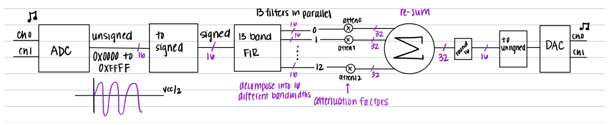


Fig. 1. Illustrated signal flow for the Audio Equalizer.

A. Equalizer Components

a) Signal Converters: The equalizer utilizes an ADC to convert analog signals into digital, processable signals for the FIR filter. A DAC is used to convert the processed digital signals back to analog signals for audio output. The SPI modules that interface with the DAC and the ADC are properly configured using our previous work to sample at $100kHz$ overall, while $50kHz$ on each channel. Note that the Nyquist Sampling Criteria requires analog signals to be sampled at twice the source frequency. Since we are sampling human-audible signals that range from $2kHz$ to $20kHz$, a $50kHz$ sampling frequency meets this sampling criteria.

Given our previous implementation of buffer memories in our SPI module, we significantly reduce the timing complexity of reading from or write to the signal converters. The equalizer module can read from or write to the signal converters anytime it sees fit, while the buffer memories keep the input/output data intact during the read/write operations.

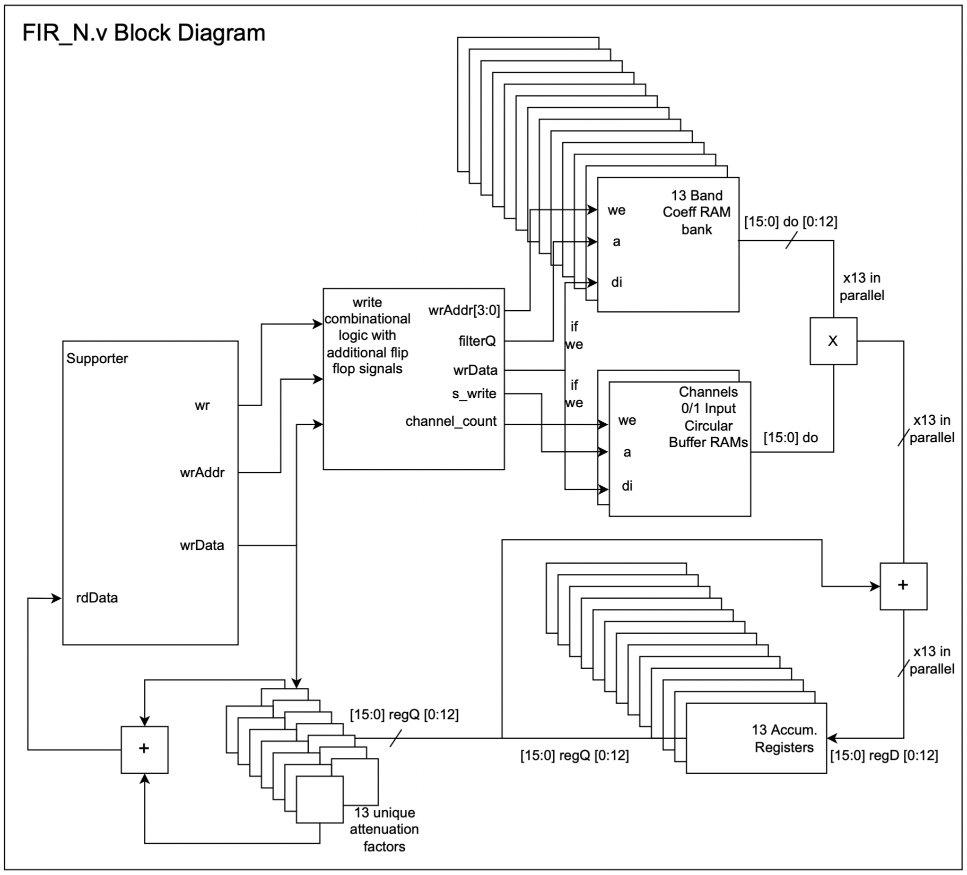


Fig. 2. Block diagram for implemented FIR_N module that uses 13 bands and 2 channels.

b) Finite Impulse Response Filter: Though our SPI module design remains unchanged, the FIR Filter design needs a significant expansion. An audio equalizer, by design, augments/attenuates specific frequency bands of a sound signal to achieve the desired audio effect. An equalizer can, for example, augment the "base" of a song by boosting the magnitude of the lower frequency bands of the audio signal.

To achieve band-specific attenuation/augmentation, we expand our previous, single-band FIR low-pass filter in two ways. First, we add in 12 additional band-pass filters that separate the audio signal into 13 frequency bands. The output signal, as a result, will be the sum of all outputs from all filters. Additional hardware such as extra storage units, rounding logic and summing logic are added as a result. Second, we add in additional write logic and storage to configure external attenuation constants. These attenuation constants target specific frequency bands and indicate if the magnitude within the band should whether be attenuated (with a less-than-one constant) or boosted (with a more-than-one constant). Since we wish to make our equalizer interactive, we equip the hardware with continuous read-write capabilities. The attenuation factors can be changed by a Microblaze® program at any given time. We again implement buffer memories to ease the timing constraint while ensure correct write timing. Fig 2 shows the expanded architecture of the FIR filter, and Fig 3 shows the updated logic design for the filter FSM.

B. Device Design

The 13-band audio equalizer combines two SPI modules and one FIR filter module. Fig. 4 shows the hardware architecture of the equalizer module. Given that we have designed these modules in our previous assignments as Axi4Lite peripherals, we instantiated Axi4Lite managers to interface with each peripheral through Axi4Lite buses that are internal to our module. This design simplifies the equalizer control path design, as the equalizer only needs to send or receive simple bus signals to the instantiated Axi4Lite managers, which manage complex signals and and relevant timing requirements. Without managers, the equalizer will have to mimic a myriad of Axi4Lite bus signals under additional timing requirements.

Serendipitously, instantiating filter module, SPI modules and their managers as separate entities enables the equalizer FSM to concurrently read or write from separate devices. Concurrent read or write between different modules reduces the signal transfer time required between each module, thus decreasing the timing complexity of the equalizer device logic.

C. Device Timing

Given that the equalizer module interfaces with three different components, it is crucial for the equalizer module to conform to timing requirements. Table I shows the timing requirements posed by each components.

Overall, four pieces of sequential operation post the most significant timing constraints. The overall, repeating operation

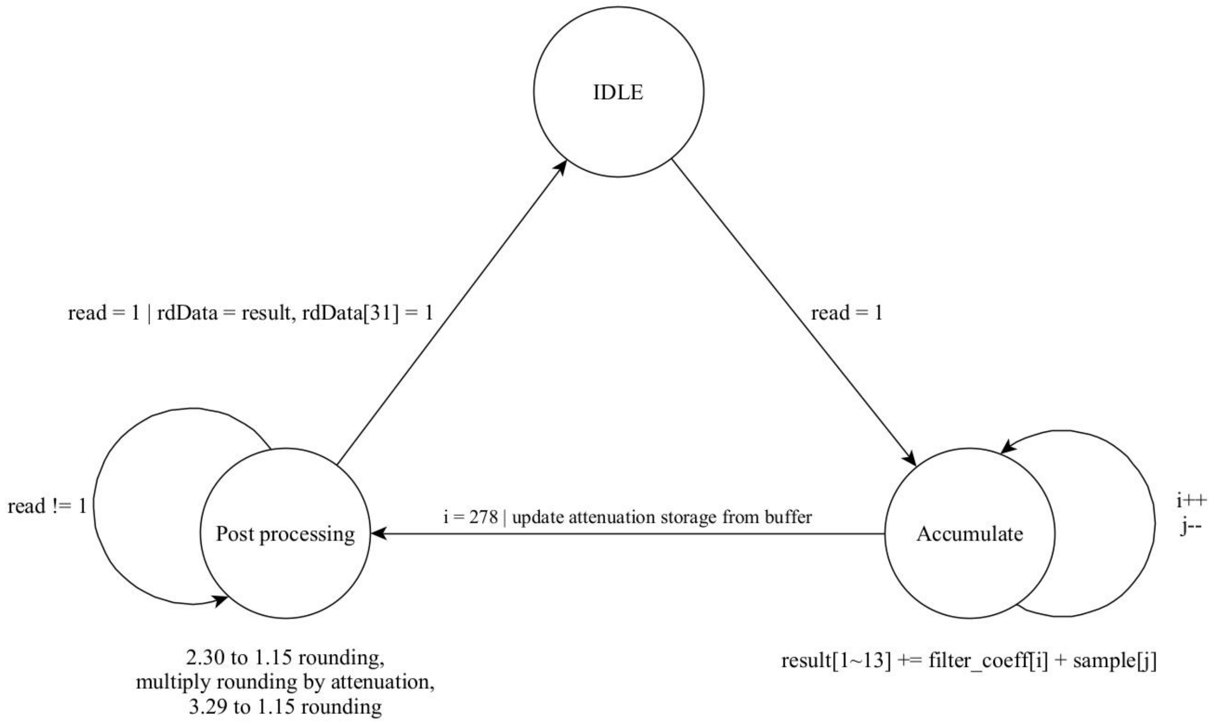

IDLE
read = 1 | rdData = result, rdData[31] = 1
read = 1
read != 1
Post processing
i = 278 | update attenuation storage from buffer
Accumulate
i++
j--
2.30 to 1.15 rounding,
multiply rounding by attenuation,
3.29 to 1.15 rounding
result[1~13] += filter_coeff[i] + sample[j]

Fig. 3. Bubble Diagram for Filter FSM .

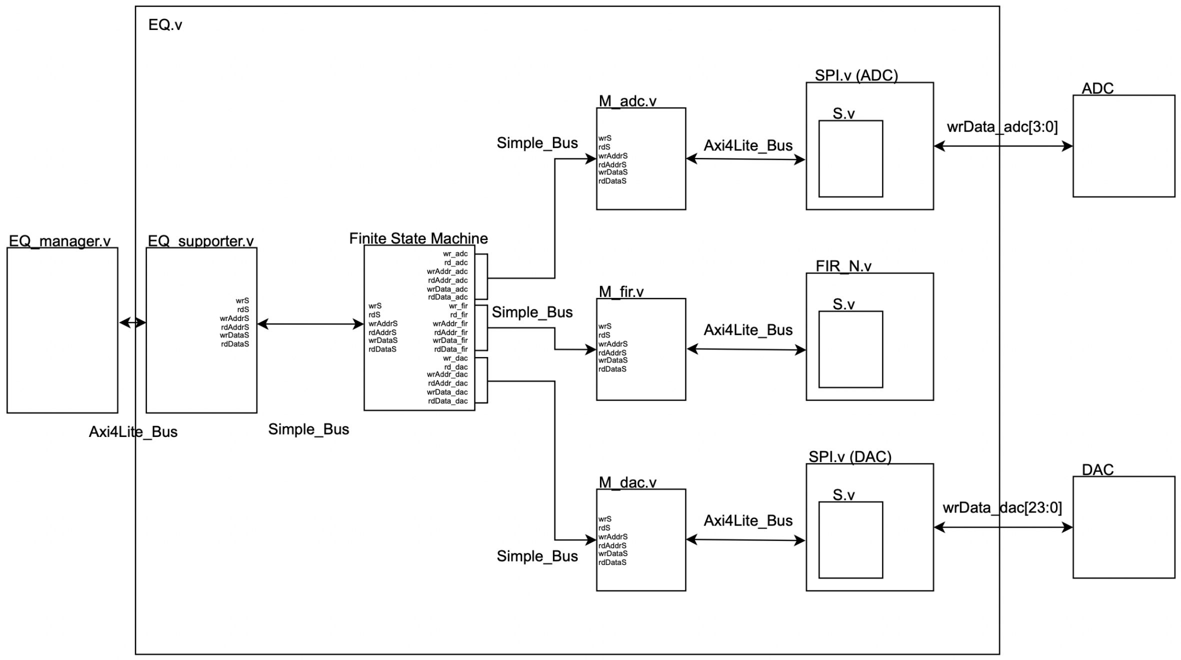


Fig. 4. Block diagram for entire module project with some abstraction conditional logic.

TABLE I
OVERALL TIMING CONSTRAINTS

Minimum and Maximum Timing Requirements			
Source	min/max	Value	Sequential/ Concurrent
Signal sampling rate/period	min	$100kHz$ 300 (clc cycle)	
Filter MAC	max	279 (clk cycle)	Sequential, must be after FIR write
Filter Read	max	2 (clk cycle)	Sequential, must happen after MAC
Filter Write	max	1 (clk cycle)	Sequential, must happen before MAC
DAC write	max	1 (clk cycle)	Sequential, must receive FIR result
ADC read	max	3 (clk cycle)	Sequential, starts processing
ADC/DAC config	min	2 (clk cycle) each	Sequential & concurrent, must happen before ADC read, can be concurrent to DAC/FIR read/write
FIR config	max	4185 (clk cycle)	Sequential & concurrent, must happen before FIR ops, can be concurrent to ADC/DAC read/write

sequence for sample processing can be described as: ADC Read →Filter Write →Filter MAC →Filter Read →DAC Write. These operations must happen in sequence, and must meet the 300 clock cycle ($100kHz$ sampling rate) constraint between each ADC reads. We do meet this timing requirement, given that all four operations combined takes up 286 clock cycles. We can further reduce the total time by concurrently reading from ADC and writing to Filter, as well as concurrently reading from Filter and writing to DAC. Though the SPI module provides DAC write confirmation, requiring such confirmation to move to the next sample processing cycle is not necessary, since sufficient time would have elapsed between two consecutive DAC writes. Based on these considerations, we design the following timing logic as described by the FSM in Fig. 5. The config stage, which contains all the necessary device configuration logic, concurrently configures the ADC, DAC and the filter to perform dual channel, 13-band, 279-term finite impulse response filtering at $100kHz$. After all configurations are done, the equalizer goes into the sampling loop, in which the equalizer sequentially receives, processes, and outputs audio signals.

III. Implementation and Operational Testing

All C programs and simulation results were culminated in a hardware implementation of the Microblaze processor using a Field Programmable Gate Array (FPGA). Results from the hardware were amply tested for correct frequency response, filter results, functionality with a windows application graphical user interface (GUI), and functionality with the serial port communication between the FPGA and GUI.

a) Verilog Hardware Implementation: To begin with the implementation process, we modify our previous FIR low-pass filter as outlined by section II-A. The expanded filter module along with proper SPI module and their respective Axi4Lite managers are then properly connected internally to the equalizer. The FSM that governs the inputs and outputs of the equalizer unit is subsequently implemented, along with appropriate read/write logic that receives data from the Microblaze processor.

Like our previous designs, the equalizer is implemented as an Axi4Lite peripheral connected to the Microblaze processor. The Verilog hardware description to each part can be found in the appendix.

b) Interface Code Implementation: The interface to our equalizer peripheral serves two main functions: device configuration and attenuation factor configuration. The device interface is implemented in C. The interface first writes in the filter coefficients sequentially to the equalizer while concurrently writes in SPI configuration data (maximum sampling rate, control word length, etc). After proper configuration, the interface goes into a continuous receiver loop that listens for attenuation factor input through serial communication. When the interface receives a full packet (13 2-byte attenuation factors), the interface passes the factors to the equalizer peripheral, and goes back to the receiver loop.

c) Filter Testing: The first aspect of testing for the equalizer was to ensure that the modified FIR module to include 2 channels and 13 bands was properly producing the output sine wave. To do this, a C program was used to implemented the same filer in software to numerically compare with the hardware simulation results. The plotted output 1kHz sine waves from both the C program and the hardware simulation are shown in Fig. 6. The two sine waves are identical numerically, as shown by difference calculation and Fig. 6.

The second aspect testing was the expected phase lag from the input to the output. Because the FIR filter module has 279 taps for this 13 band filter, the expected phase lag can be calculated using (1). From this, the expected phase lag is 0.00279 seconds. Experimental phase lag was calculated by plotting the input and output sine waves at 200Hz, as seen in Fig.7. The first peak of each was compared, finding the exact data points at which they occur. The experimental phase lag was 0.00278 seconds. There is probably error in the preciseness of finding the point at the peak of each sine wave that would leave to a phase lag slightly off from the expected value. However, this value is close enough to the expected lag to accept for functionality testing.

$$lag = \frac{279}{2 * 50,000\text{Hz}} \tag{1}$$

d) Finite State Machine Testing: After ensuring the functionality of the modified FIR filter module, the audio equalizer finite state machine was tested as the equalizer was constructed. Because of the abundance of moving parts with this implementation, and therefore several complex states, it was necessary to test each state as it was created in the equalizer. Each state connected a new peripheral, firstly the ADC, secondly the FIR filter, and lastly the DAC. With each new state, the output of the peripheral was compared to previous output from that same module. For the ADC, results were tested with the ADC Tester module provided for the SPI project. After adding the FIR state, the simulation results were

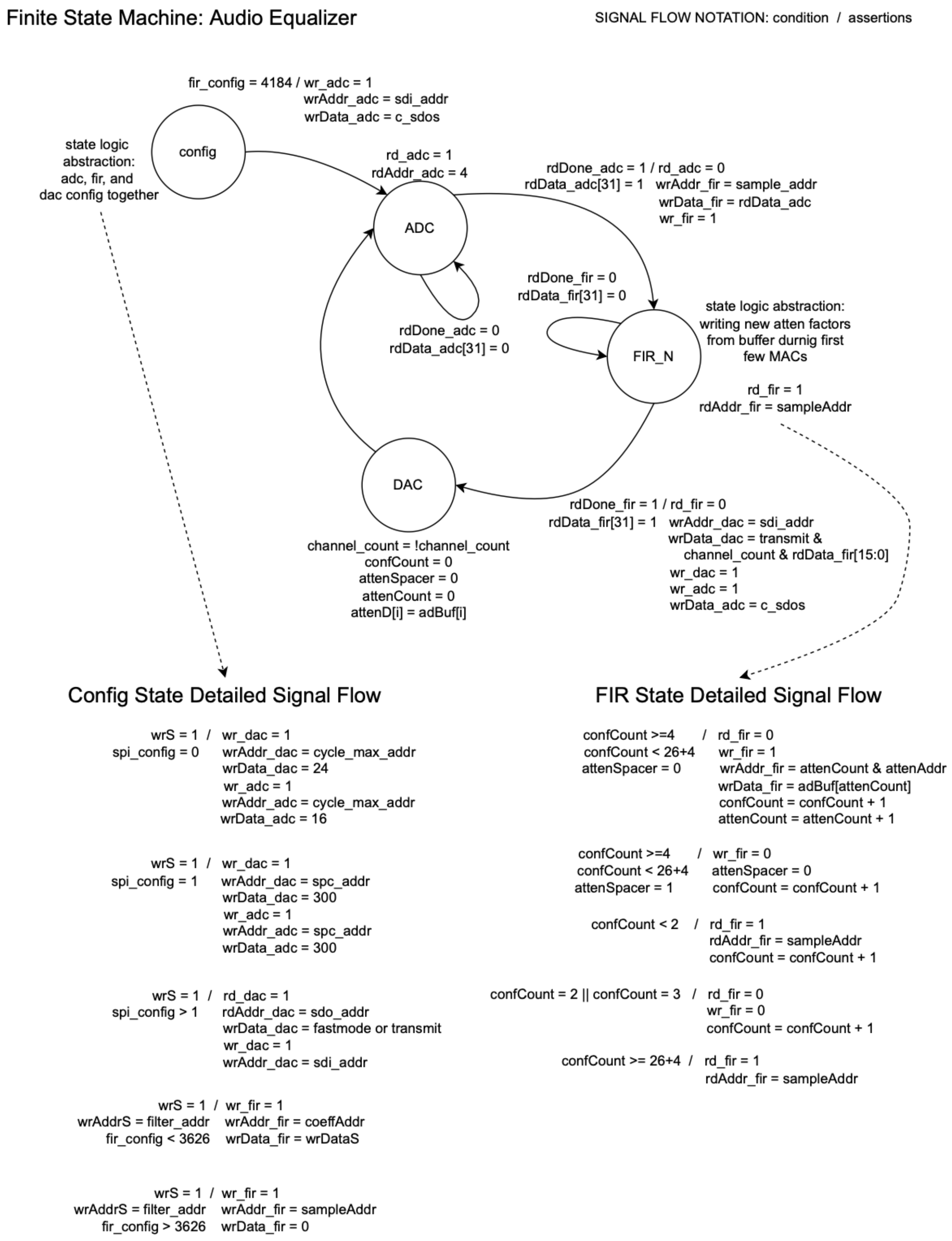

Finite State Machine: Audio Equalizer
SIGNAL FLOW NOTATION: condition / assertions
fir_config = 4184 / wr_adc = 1
wrAddr_adc = sdi_addr
wrData_adc = c_sdos
state logic abstraction: adc, fir, and dac config together
config
rd_adc = 1
rdAddr_adc = 4
ADC
rdDone_adc = 1 / rd_adc = 0
rdData_adc[31] = 1 wrAddr_fir = sample_addr
wrData_fir = rdData_adc
wr_fir = 1
rdDone_adc = 0
rdData_adc[31] = 0
rdDone_fir = 0
rdData_fir[31] = 0
FIR_N
state logic abstraction: writing new atten factors from buffer durnig first few MACs
rd_fir = 1
rdAddr_fir = sampleAddr
DAC
channel_count = !channel_count
confCount = 0
attenSpacer = 0
attenCount = 0
attenD[i] = adBuf[i]
rdDone_fir = 1 / rd_fir = 0
rdData_fir[31] = 1 wrAddr_dac = sdi_addr
wrData_dac = transmit & channel_count & rdData_fir[15:0]
wr_dac = 1
wr_adc = 1
wrData_adc = c_sdos
Config State Detailed Signal Flow
wrS = 1 / wr_dac = 1
spi_config = 0 wrAddr_dac = cycle_max_addr
wrData_dac = 24
wr_adc = 1
wrAddr_adc = cycle_max_addr
wrData_adc = 16
wrS = 1 / wr_dac = 1
spi_config = 1 wrAddr_dac = spc_addr
wrData_dac = 300
wr_adc = 1
wrAddr_adc = spc_addr
wrData_adc = 300
wrS = 1 / rd_dac = 1
spi_config > 1 rdAddr_dac = sdo_addr
wrData_dac = fastmode or transmit
wr_dac = 1
wrAddr_dac = sdi_addr
wrS = 1 / wr_fir = 1
wrAddrS = filter_addr wrAddr_fir = coeffAddr
fir_config < 3626 wrData_fir = wrDataS
wrS = 1 / wr_fir = 1
wrAddrS = filter_addr wrAddr_fir = sampleAddr
fir_config > 3626 wrData_fir = 0
FIR State Detailed Signal Flow
confCount >=4 / rd_fir = 0
confCount < 26+4 wr_fir = 1
attenSpacer = 0 wrAddr_fir = attenCount & attenAddr
wrData_fir = adBuf[attenCount]
confCount = confCount + 1
attenCount = attenCount + 1
confCount >=4 / wr_fir = 0
confCount < 26+4 attenSpacer = 0
attenSpacer = 1 confCount = confCount + 1
confCount < 2 / rd_fir = 1
rdAddr_fir = sampleAddr
confCount = confCount + 1
confCount = 2 || confCount = 3 / rd_fir = 0
wr_fir = 0
confCount = confCount + 1
confCount >= 26+4 / rd_fir = 1
rdAddr_fir = sampleAddr

Fig. 5. Finite State Machine showing Equalizer states and state transition logic implemented.

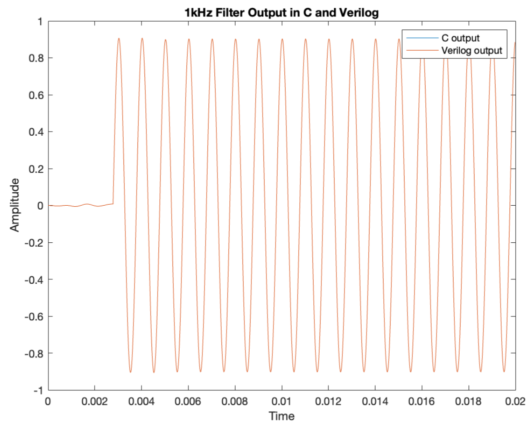


Fig. 6. C implementation and hardware simulation output sine waves.

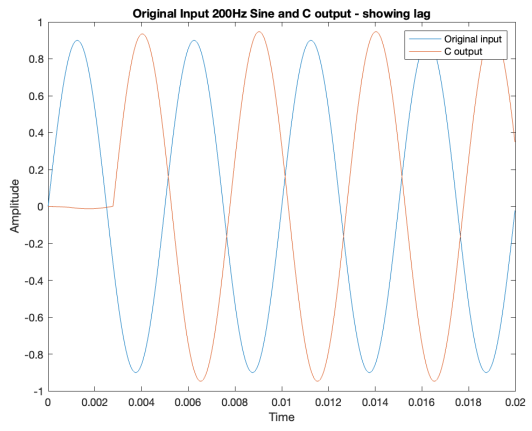


Fig. 7. Input/output sine waves at 200Hz showing phase lag calculation.

compared to previous testing results for just the FIR module modifications. For the DAC state, results were shown on the oscilloscope to ensure they matched as a time delayed version of the input from the function generator. This final state also served as one test for the whole equalizer since it also required the signal flowing through all implemented states to test the DAC. Additionally, for the DAC, the provided DAC Tester module from the SPI project was used to verify results.

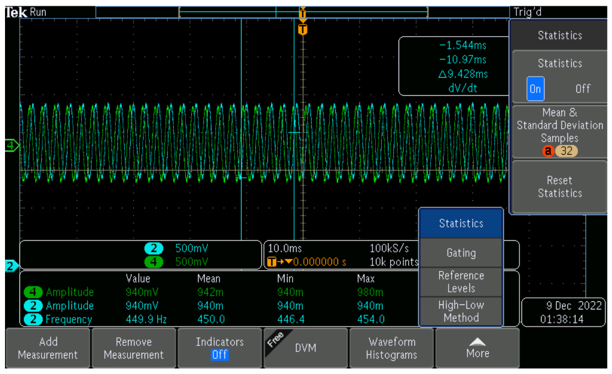
e) Frequency Response Testing: To ensure that the designed audio equalizer produced the correct frequency response from the Audio Equalizer Design Tool, two different comparison tests were performed. First, the natural output attenuation from the filter, without added attenuation factors, was compared against the expected attenuation. With all additional, band specific, attenuation factors set to 1, signifying no additional change, the output max value was compared to the input max value using (2). For this calculation, the filter produced 0.033599 dB of attenuation from the original signal, reasonably close to the expected value from the frequency response.

$$db = 20 \cdot \log_{10}(\frac{output}{input}) \tag{2}$$

Secondly, the function generator and oscilloscope were used to sweep the output frequencies. The FPGA was connected to the ADC and DAC peripherals with the function generator connected to the input of the ADC and the oscilloscope connected to the output of the DAC. Signals generated in the function generator were input to the audio equalizer through the ADC and displayed in comparison to a vertically shifted version of the original signal on the scope for verification. The mean amplitude value for the input and output sine waves were compared using (2). The table of expected dB attenuation based on the frequency response and collected dB attenuation during testing is shown below in II. An example image of the scope during this frequency response testing for 450Hz is shown in Fig. 8.

TABLE II
dB Attenuation in Frequency Response

Expected vs. Experimental Frequency Response		
Frequency	Expected dB	Experimental dB
200Hz	0.45	0.441
450Hz	-0.01	-0.018
1300Hz	0.25	0.254



−1.544ms
−10.97ms
Δ9.428ms
dV/dt

2 500mV
4 500mV
10.0ms 100kS/s
T→▼0.000000 s 10k points

	Value	Mean	Min	Max
4 Amplitude	940mV	942m	940m	980m
2 Amplitude	940mV	940m	940m	940m
2 Frequency	449.9 Hz	450.0	446.4	454.0

9 Dec 2022
01:38:14

Fig. 8. Input/output sine waves at 450Hz during frequency response testing.

The Filter design tool used for the initial filter coefficient creation was useful in comparing our results with the frequency response expected. An image of the magnitude and impulse responses of the designed 13 band filter is shown in Fig. 9, and an image of the time domain and frequency domain responses to the filter are shown in Fig. 10. Enlarging these images in the Filter Design Tool allowed comparison of the audio equalizer output to the correct frequency response for our Filter.

f) Attenuation Testing: Once the finite state machine and frequency response were verified, the filter module was modified again to add dynamic attenuation factors. This allowed for serial port communication to change the attenuation factors used for the filter calculations, but also needed testing. The first test of the attenuation factors used just the modified filter module and the test bench used to test only that module while originally modifying it. The output of the filter was plotted again the input 1kHz sine wave to verify proper attenuation and valid data. Fig. 11 shows this verification plot for the 1kHz sine wave.

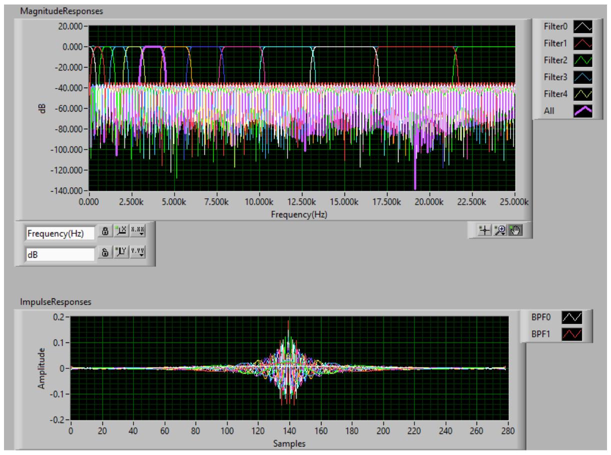


Fig. 9. Magnitude and Impulse responses from the 13 band filter.

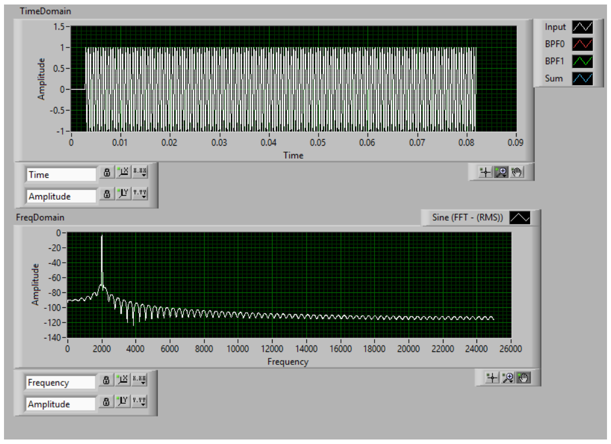


Fig. 10. Time and Frequency Domain responses from the 13 band filter.

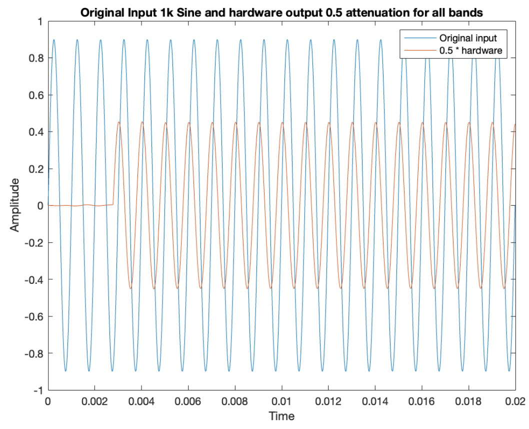


Fig. 11. 0.5 Attenuated output at 1.3kHz in filter simulation.

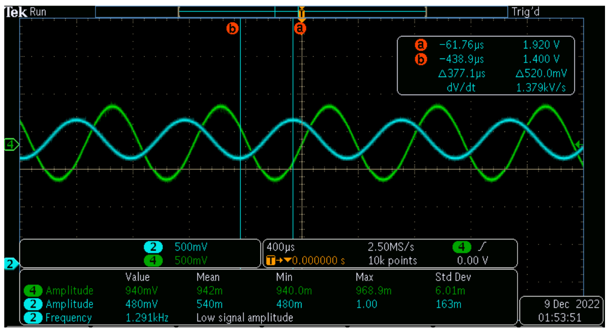

Tek Run
Trig'd
−61.76µs 1.920 V
−438.9µs 1.400 V
Δ377.1µs Δ520.0mV
dV/dt 1.379kV/s
500mV
500mV
400µs
2.50MS/s
10k points
0.00 V
Value Mean Min Max Std Dev
Amplitude 940mV 942m 940.0m 968.9m 6.01m
Amplitude 480mV 540m 480m 1.00 163m
Frequency 1.291kHz Low signal amplitude
9 Dec 2022
01:53:51

Fig. 12. 0.5 Attenuated output wave at 1.3kHz on Scope with hardware.

To test the attenuation factors on the hardware with the whole equalizer, the function generator and oscilloscope were also used to display the input and output sine waves, as shown in Fig. 12. Both qualitatively and quantitatively, the output was analyzed against the input magnitude to ensure proper attenuation. In Fig. 12, it can be seen that the output, in blue, is half the amplitude of the input, in green. Various combinations of the set of 13 attenuation factors were used while the frequency was swept on the function generator. This allowed verification of attenuation at all of the 13 different frequency bands.

g) Windows Application: The final testing for the audio equalizer was to implement a GUI as a windows application to allow a user to interact with the equalizer while playing audio through the ADC. This primarily involved designing a front end and back end for the windows application to interact with the FPGA and peripherals through serial port communication. For our GUI, the application was designed using Python and its Tkinter framework. The backend of this application involved establishing a serial port communication connection to the Microblaze processor. The application then wrote the attenuation factors to be stored in the buffer for the filter to use in the filter state. Attenuation factors are written to the FPGA when the send button is clicked, sending the values of all 13 sliders.

UART serial communication only allowed us to send at most 16 bytes across the port in a single write operation. Because each attenuation factor is 2 bytes (4 hex nibbles), we were only able to send 8 attenuation factors at one time. To solve this, our application sends two consecutive writes and the module only finishes reading, allowing the attenuation factors to be used, once it has received 32 bytes.

The sliders in the application range from 0 to 100, where 0 represents attenuating the band entirely, therefore returning 0% of the original signal, and 100 represents not attenuating the signal at all, therefore returning 100% of the original signal. The GUI, with all sliders set at 100% is shown in Fig. 13. When the application is opened, or when the Reset Sliders button is pressed, all sliders default to 50, representing 50% attenuation and therefore half the volume for all 13 frequency bands.

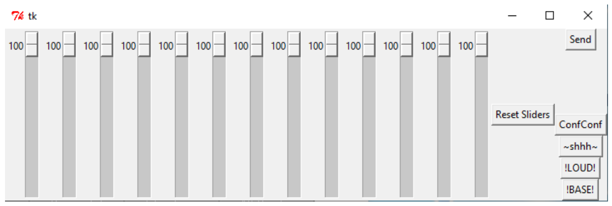


Fig. 13. Windows Application with all attenuation factors at 100% (no attenuation).

The buttons on the side add additional functionality such as presets for the sliders. The “shhh” button sets all the sliders to 0, attenuating all the frequency bands to 0% output when written to the equalizer by hitting the send button in the upper right corner in Fig. 13. The “!LOUD!” button sets all the sliders to 100. The “!BASE!” button sets the sliders to a custom preset that amplifies the base frequencies in the input audio, as shown in Fig. 14.

A special extra feature was implemented to add to the functionality of the windows application. The “ConfConf” button accesses a secret mode that adds a new set of 13 sliders, one for each frequency band. Because the attenuation factors are designed to be from 0.0 to 1.0 inclusive, they must be implemented in 2.14 format. However, this also means that the attenuation factors can technically range from 0.0 to 1.99 inclusive. The ConfConf extra sliders allow a range from 0 to 190, instead of stopping at 100, and allowing the user to amplify the input signal at certain bands to 190% of the original volume. This secret mode is shown in Fig. 15. The EXTRA button, similar to the !LOUD! button in the normal mode, sets all sliders to 190%. The EXTRA BASE button, similar to the !BASE! button in normal mode, sets sliders to the customized preset for base frequency amplification, as shown in Fig. 15.

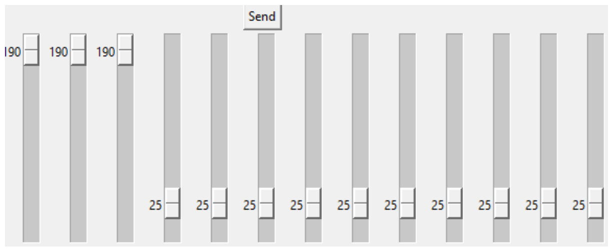


Fig. 14. Base preset for sliders to amplify all base frequencies and attenuate others.

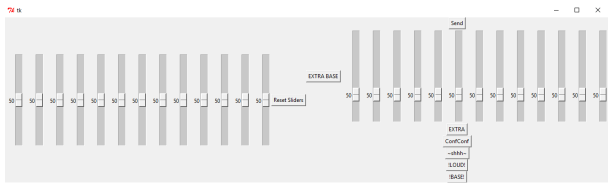


Fig. 15. Additional Sliders in Secret Mode of Application for 190% amplification capability.

Testing of the GUI included playing a song from a laptop through the FPGA and output from speakers. Audibly, it was verified that changing the sliders affected the sound of the output audio signal based on which attenuation factors were applied to which frequency bands. For example, changing all attenuation factors to 50% caused the song output to play half as loud, changing all factors to be 0% muted the song entirely, and using the custom base preset amplified the base in the song and attenuated the vocals.

IV. Discussion/Conclusion

The Audio Equalizer combined various modules designed throughout the semester into a culminating hardware module and windows application GUI. This allows users to interact with the designed module to send specified audio through an ADC, FIR filter, and DAC to observe the resulting audio with 13 different frequency bands. Each band has the capability to be attenuated or amplified based on an attenuation factor between 0.0 and 1.0 inclusive, dynamically passed through the windows application sliders. The design and implementation of this module showed all of the timing requirements and design restrictions necessary when incorporating several different modules together to interact with one another. Each was modified, even if it was just slightly, to ensure compatibility for the overall audio equalizer.

A single finite state machine controlled the data flow between all modules, ensure configuration in the first of four states, and then smooth, sequential transitions between states while processing the input audio signals. Every module was double buffered to allow for writing to the module at any point, and preventing timing conflicts and data corruption between states.

The overall project underwent ample testing at every step of design and implementation to ensure that the frequency response matched the designed 13 band filter, that all the components worked together without affecting the input signal, and that the audio played back with the appropriate attenuation factors. The final implementation is capable of playing a song through a speaker that dynamically attenuates chosen bands to amplify different frequencies in the song, such as base boasting to focus on the base frequencies of the song. The affect on each band can be visually seen by sweeping the frequencies in the function generator and watching the scope as each frequency is affected by its associated factor.

Future work for this project includes a real time feature for the windows application and pipelining for the equalizer finite state machine. Real time updating from the windows application would allow us to dynamically change the attenuation factors without having to hit the send button after adjusting the gains on the GUI sliders. The application would send the new values to the equalizer’s attenuation factor buffer as an event listener when the sliders move. Additionally, pipelining the FSM for the whole equalizer would allow the module to be constantly working in all states, increasing the speed of processing. Currently, our FSM iterates through all three states after configuration, processing in each one individually before moving to the next. Pipelining would allow each state, and subsequently each peripheral, to be constantly processing either a current or upcoming signal.

Authors and Affiliations

The authors of this report, Will Wu and Cassie Jeng, can be reached at willwu@wustl.edu and jeng.c@wustl.edu, respectively.

References

[1] 1st ed., vol. 2013, Xilinx, 2013, Vivado Design Suite User Guide.
[2] Linear Technology, ”Dual 14-Bit Rail-to-Rail DAC in 16-Lead SSOP Package,” LTC1654. Linear Technology Corporation: Milpitas, CA.
[3] Linear Technology, ”μPower, 3V, 16-bit, 150ksps 1- and 2-Channel ADCs in MSOP,” LTC1864L. Linear Technology Corporation: Milpitas, CA.

For code, please see: https://github.com/WillWu88/FPGA_Project_Reports/tree/main/Code/EQ