Lab 07 Report: TUID FIR Digital Filter

Peter Mowen
peter.mowen@temple.edu

Summary

This lab explores Finite Impulse Response (FIR) filters in both MATLAB and C. A unique FIR filter is assigned to each student using his or her TUID number. Simulated sine waves are run through this filter. The Zybo and DA2Pmod are used to realize the filtered signal as a voltage.

Introduction

In this lab, the student will create an Finite Impulse Response (FIR) filter based on his or her TUID based on the following formula:

$$H(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + b_4 z^{-4} + b_5 z^{-5}$$

Where:

- b_0 is least significant digit (LSD) of the TUID number modulo 5 plus 1
- b_1 is second LSD of the TUID number modulo 4 plus 1
- b_2 is third LSD of the TUID number modulo 3 plus 1
- b_3 is fourth LSD of the TUID number modulo 2 plus 1
- b_4 is fourth LSD of the TUID number modulo 1 plus 1
- b_5 is fourth LSD of the TUID number plus 1

The code to generate these values in MATLAB is:

```
 6 -    TUID = [9 1 2 2 0 1 8 1 9];
 7 -    lsdPosition = length(TUID);
 8
 9 -    b = zeros(1,6); % holds coefficients based on TUID number.
10 -    fprintf("The coefficients for the TUID FIR filter are:\n");
11 -    for k = 1:length(b)
12 -        b(k) = mod(TUID(lsdPosition - (k - 1)),6-k) + 1;
13 -        fprintf("b%d = %d\t", k-1, b(k));
14 -    end
15 -    fprintf("\n\n");
```

Figure 1: MATLAB code to generate filter coefficients

The results of running this code are:

```
The coefficients for the TUID FIR filter are:
b0 = 5  b1 = 2  b2 = 3  b3 = 2  b4 = 1  b5 = 3
```

Figure 2: Filter coefficients generated in MATLAB

These coefficients were copied verified by hand and copied into the C code.

FIR filters generate an output based only on an input signal. This means that if the input signal is finite, then the output will also be finite. This is opposed to an Infinite Impulse Response filter; whose output depends on previous outputs. These filters will continue outputting numbers forever.

Discussion

Hardware Setup in Vivado HLx

Below is a screenshot of the block diagram from Vivado HLx:

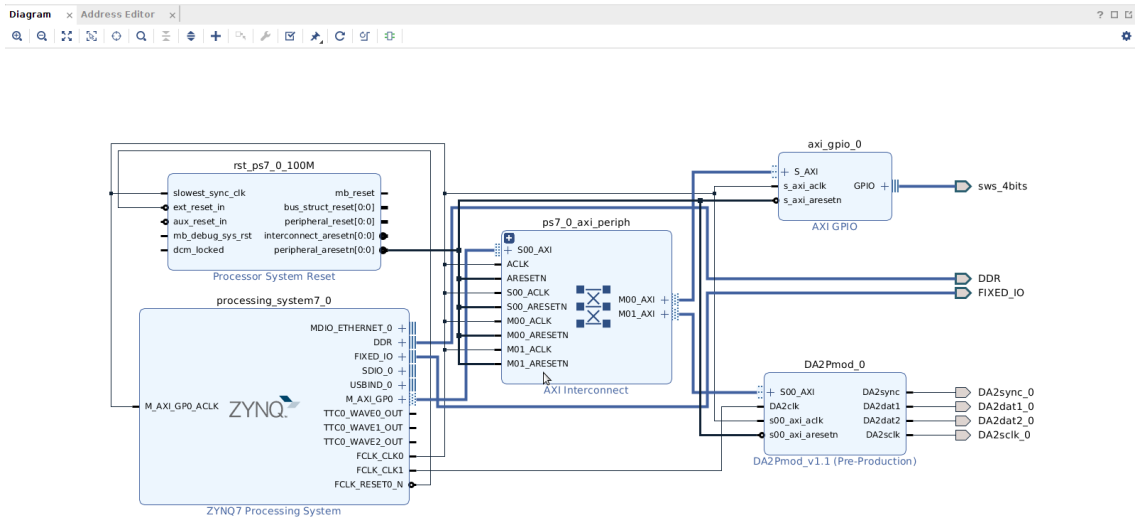

Figure 3: Vivado HLx Bloack Diagram

The switches will be used to select the sine wave to generate. The DA2Pmod will be used to output the filtered signal. Something to note is that the memory address of the switches had to be modified in order for them to work.

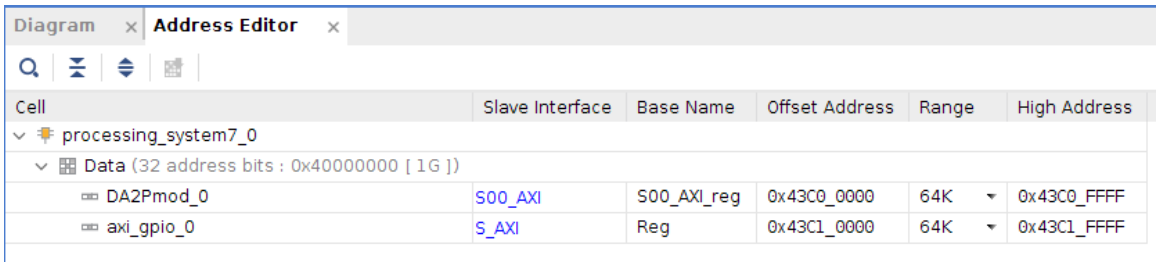

Figure 4: GPIO Addresses

The address 0x43C1_0000 was selected because it had been used in a previous lab without interfering with the other GPIO. The switches needed a new address because their original address conflicted with the address space the program was using to store the generated sine wave. A detailed discussion can be found in the Trouble Implementing Switches section.

Physical Hardware Setup

The physical hardware setup was almost identical to the previous lab, so a photo has been omitted. The main difference is the absence of the AD1Pmod chip. Otherwise, the hardware setup is the same.

MATLAB Frequency Response

MATLAB was used to facilitate the analysis of the TUID FIR filter. The following code was used to get vectors containing complex magnitudes for 4096 points between 0Hz and the Nyquist frequency and plot the results:

```
30 -    [Hz, frq] = freqz(numeratorOfFilterEquation, ...
31                        denominatorOfFilterEquation, 4096, fs);
32 -    magH = abs(Hz);
33 -    plot(frq, abs(Hz));
34 -    title("Frequency Response of TUID FIR");
35 -    xlabel("Frequency (Hz)");
36 -    ylabel("Gain (magnitude)");
```

Figure 5: Code to generate frequency response plot

The variable `Hz` holds the complex gain for the corresponding frequencies in the variable `frq`. The third argument to the `freqz` function tells it to test 4096 points and the Nyquist frequency. The Nyquist frequency is `fs/2`. The `frq` vector is equivalent to `linspace(0, fs/2, 4096)`. This vector will contain our fundamental frequency and its harmonics, as well as `fs/4` which means the `Hz` vector will contain the gains of interest. The magnitude of `Hz` is plotted against the `frq` to create the frequency response plot.

The following plot shows the frequency response of my TUID FIR filter:

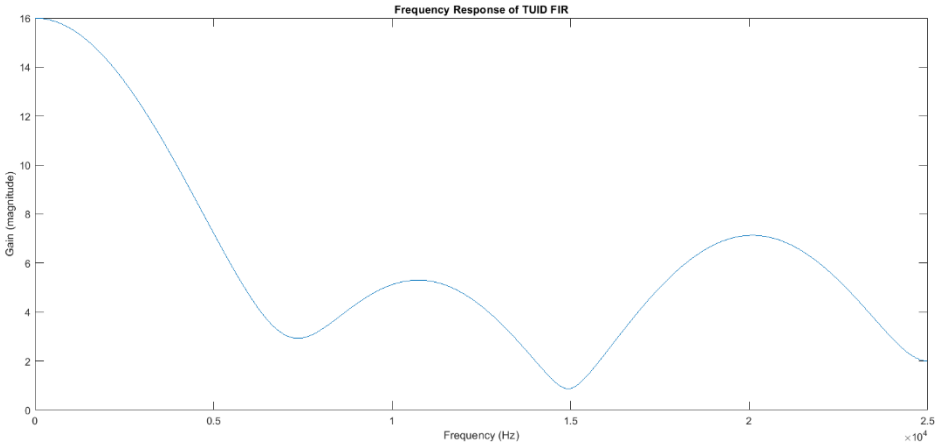

Figure 6: Frequency response of TUID FIR Filter

Note that for the low frequencies we will be feeding through this filter, the gain is roughly 16.

MATLAB was also used to generate the pole-zero plot for this FIR filter.

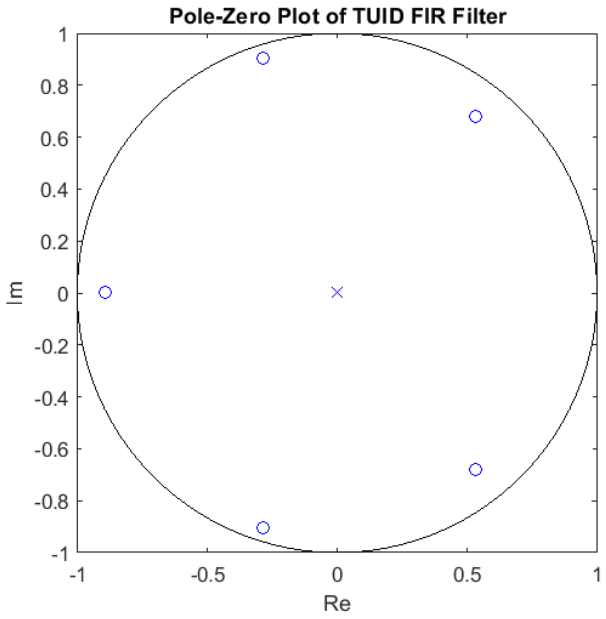

Figure 7: Pole-Zero plot for TUID FIR filter

The following code was used to calculate the gain at fs/4 "by hand" using the variables that were used to create the pole-zero diagram:

```
% Find gain at fs/4
fs = 50e3;
f = fs/4;

pointOnUnitCircle = [cos(2*pi*(f/fs)), sin(2*pi*(f/fs))];

numeratorRadicand =   (pointOnUnitCircle(1) + real(zr)).^2 ...
                    + (pointOnUnitCircle(2) + imag(zr)).^2;
numerator = prod( sqrt(numeratorRadicand));

denominatorRadicand =   (pointOnUnitCircle(1) + real(poles)).^2 ...
                      + (pointOnUnitCircle(2) + imag(poles)).^2;
denominator= prod( sqrt(denominatorRadicand));

calcGainAtOneQuarterFs = G*(numerator/denominator);
fprintf("The calculated gain at fs/4 is %0.2f\n", calcGainAtOneQuarterFs);

gainAtOneQuarterFsFromFreqz = magH(find(frq == fs/4));
fprintf("The gain at fs/4 from freqz is %0.2f\n", ...
            gainAtOneQuarterFsFromFreqz);
```

Figure 8: Code to find gain at `fs/4`

This gain was the printed to the command window. The gain at `fs/4` was also found in the gain vector from the `freqz` function. The result was also printed to command window. The results can be seen here:

```
The calculated gain at fs/4 is 4.24
The gain at fs/4 from freqz is 4.24
```

Figure 9: Gain at `fs/4`

Note that the results agree.

MATLAB: Simulated Sine Waves and FIR

The following code was used to generate a discrete-time sine wave with 4096 points, a fundamental frequency of 12.21 Hz, a sampling frequency of 50ksamples/sec, an amplitude of 2000, and a DC offset of 2048.

```
% Generate fundamental frequency sine wave
fundFreq = fs/numOfPoints; %Hz
discreteTime = 0:4095;
fundamentalSineWave =   floor(2000*sin(2*pi*(fundFreq/fs)*discreteTime))...
                      + 2048;
```

Figure 10: Code to generate fundamental frequency sine wave

The floor function was used to round each entry down to the nearest integer. This simulates the discrete nature of using an integer array to hold the sine wave C. The offset of 2048 was used to keep all the values positive since the DAC cannot handle negative values.

The following code generates the harmonic sine waves based off the fundamental frequency sine wave and stores the results in a matrix:

```
% place holder for matrix holding all sines
nthHarmonicSineWaves = zeros(maxHarmonic,4096);

% put the fundamental frequency sine wave in the 1st row
nthHarmonicSineWaves(1,1:end) = fundamentalSineWave;

for k = 2:maxHarmonic
    kthHarmonicSineWave = fundamentalSineWave(k:k:end);
    len = length(kthHarmonicSineWave);
    for m = 1:k
        %Create a sine wave wave of freqency k*f by taking the kth sample
        %   of the fundamental freqency sine wave and copying it for 4096
        %   samples
        nthHarmonicSineWaves(k, 1+(m-1)*len:m*len) = kthHarmonicSineWave;
    end
    subplot(2,4,k);
    plot(discreteTime, nthHarmonicSineWaves(k, 1:end));
    title(sprintf("Sine wave for f = %d*%0.2fHz",k,fundFreq));
    xlabel("Sample Number");
    ylabel("Sample Value");
    xlim([1 4096]);
    hold on;
end
```

Figure 11: Code to generate nth order harmonics based on fundamental frequency sine wave

To generate the sine wave for each kth order harmonic, the original sine wave is sampled starting at the kth sample, then taking each kth sample until the end. This ensures the `kthHarmonicSineWave` will be a multiple of 4096. This sine wave is then copied into the matrix and repeated so that its total length is 4096. The following plot shows the results:

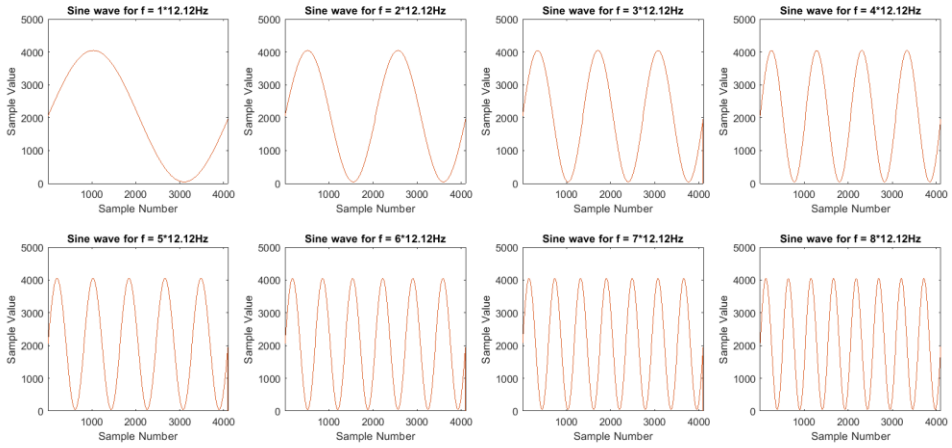

Figure 12: Unfiltered sine waves

Note that the number of periods per 4096 samples is equal to the order of the harmonic.

The following code simulates passing the sine waves through the FIR filter:

```
136     %------------ Task 5 : Plot nth Harmonic filtered Sine Waves --------------
137     % Run each harmonic through the filter
138 -   filteredSines = filter(numeratorOfFilterEquation, ...
139                         denominatorOfFilterEquation, nthHarmonicSineWaves,[],2);
140
141     % Plot all filtered harmonics, scaled appropriately
142 -   figure(figureNum); figureNum = figureNum+1;
143 -   for k = 1:maxHarmonic
144 -       subplot(2,4,k);
145 -       indexOfkthHarmonic = find(frq == k*fundFreq);
146 -       firGain = magH(indexOfkthHarmonic);
147 -       plot(discreteTime, (1/firGain)*filteredSines(k, 1:end));
148 -       xlim([1 numOfPoints]);
149 -       title(sprintf("Filtered wave for f = %d*%0.2fHz",k,fundFreq));
150 -       xlabel("Sample Number");
151 -       ylabel("Sample Value");
152 -   end
```

Figure 13: Code to filter sine waves

All the sine waves are filtered at once using the filter function. The results are then plotted together in a single figure. For each frequency, the filtered sine wave is divided by the gain in order to keep it in the range from 0 to 4095. From the frequency response curve, this gain is 16, which would overload our filter and cause nonsense to be sent out. The resulting plots can be seen here:

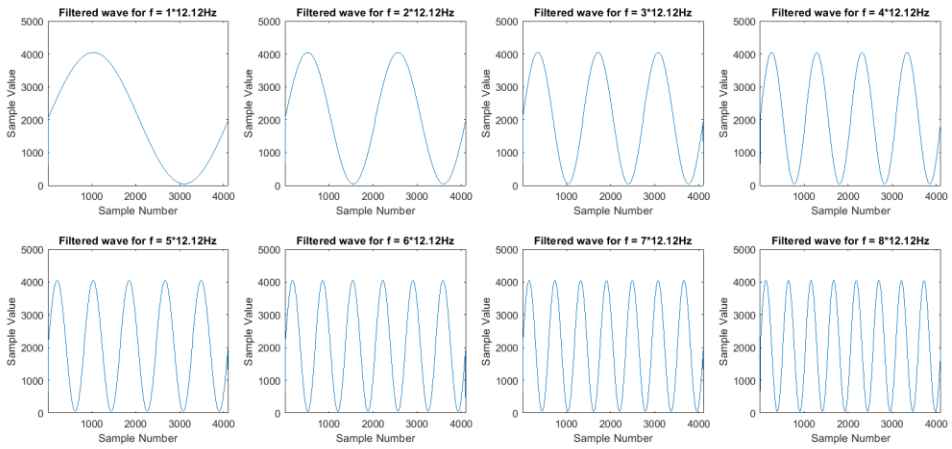

Figure 14: Filtered sine waves

Note that the filtered sine waves divided by the gain produce the original sine waves.

Finally, these filtered sine waves were converted into voltages using the following code:

```
%------- Task 6 : Plot nth Harmonic filtered Sine Waves in Voltz ----------
% Plot all filtered harmonics in voltzs, shifted down to match oscilloscope
%   output.

VREF = 3.3;
MAX_SAMPLE_VAL = 4095;
fundamentalPeriod = numOfPoints*ts;

figure(figureNum); figureNum = figureNum+1;
for k = 1:maxHarmonic
    subplot(2,4,k);
    indexOfkthHarmonic = find(frq == k*fundFreq);
    firGain = magH(indexOfkthHarmonic);
    filteredSineShiftedDown =   (1./firGain).*filteredSines(k, 1:end) ...
                              - ceil(MAX_SAMPLE_VAL/2);
    filteredSineInVoltz =    (VREF/MAX_SAMPLE_VAL)...
                           * filteredSineShiftedDown;

    time = 1000*linspace(0, fundamentalPeriod, 4096);
    plot(time, filteredSineInVoltz);
    xlim([0 max(time)]);
    title(sprintf("Filtered wave for f = %d*%0.2fHz",k,fundFreq));
    xlabel("Time (ms)");
    ylabel("Voltage");
end
```

Figure 15: Code to convert filtered sine waves to voltages

The filtered sine waves were shifted down because even though the DAC is outputting voltages between 0 and 3.3V, the oscilloscope in WaveForms graphs it without the DC offset. The simulated voltage outputs can be seen here:

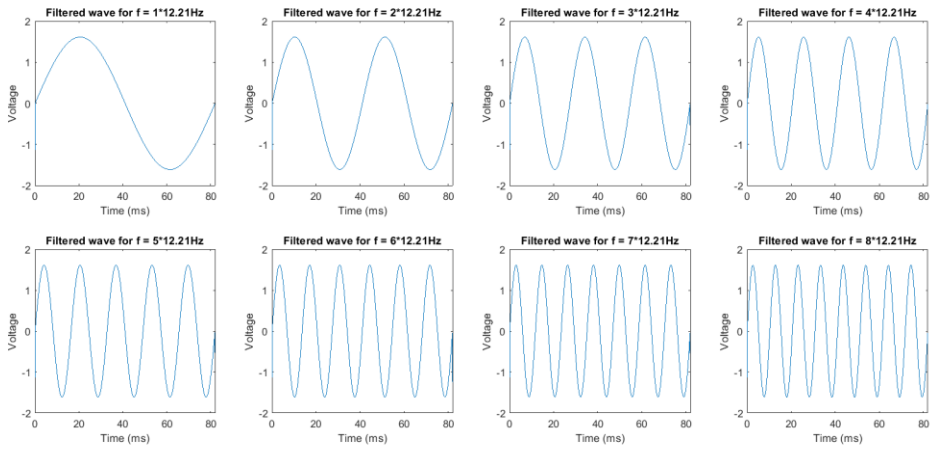

Figure 16: Simulated voltage outputs

FIR on Zybo

The following code generates the fundamental frequency sine wave in C:

```
52      // Generate sine wave
53      static int fundamentalFreqSine[NUM_OF_SAMPLES];
54      printf("Generating fundamental sine wave...\n");
55      for (int n = 0; n < NUM_OF_SAMPLES; n++)
56      {
57          fundamentalFreqSine[n] = (int)(2000*sin(2*pi*(F/FS)*n));
58      }
59      printf("Fundamental sine wave complete.\n");
```

Figure 17: C code to generate fundamental frequency sine wave

Because the `fundamentalFreqSine` is an integer array and the sine function is multiplied by 2000, each entry will take an integer value between -2000 and +2000. This range fits into a signed 12-bit number (11-bit number with 1-bit sign).

The following code runs the sine wave through the FIR filter and outputs it to the DAC:

```
74      int sample, nthHarmonic, m;
75      unsigned int switchValue;
76      int lastFiveSamples[] = {0,0,0,0,0,0};
77
78      // Initialize iterator for lastFiveSamples[m]
79      m = 0;
80
81      printf("Entering main while loop....\n");
82      while (1)
83      {
84          switchValue = XGpio_DiscreteRead(&SWInst, SW_CHANNEL) & 0xF;
85          nthHarmonic = (switchValue & 7) +1;
86
87          // Generate output based on selected harmonic
88          for (int sampleNum = 0; sampleNum < NUM_OF_SAMPLES; sampleNum++)
89          {   // Check to see if the sample number is divisible by the desired harmonic
90              if ( sampleNum % nthHarmonic == 0 )
91              {   // If it is...
92                  // Add the current sample to the list of last five samples
93                  lastFiveSamples[m] = fundamentalFreqSine[sampleNum];
94
95                  // Apply FIR filter
96                  sample = b[0]*lastFiveSamples[m]           + b[1]*lastFiveSamples[((m-1)+6)%6]
97                         + b[2]*lastFiveSamples[((m-2)+6)%6] + b[3]*lastFiveSamples[((m-3)+6)%6]
98                         + b[4]*lastFiveSamples[((m-4)+6)%6] + b[5]*lastFiveSamples[((m-5)+6)%6];
99
100                 // Divide by FIR gain as found in the MATLAB simulation to keep sample in -2048..+2047
101                 sample /= FIR_GAIN;
102
103                 // Add DC component to sample to keep it in 0..4095
104                 sample += 2048;
105
106                 // Increment iterator for the last five samples
107                 m++;
108
109                 // Keep m in 0..5
110                 m = m%6;
111
112                 // Write sample to DA2
113                 dacDataAvailable = writeSampleToDA2(sample, dacDataAvailable);
114
115                 // Wait so total time sample is on the wire is 20us
116                 usleep(18);
117             }
118         }
119     }
120 }
```

Figure 18: C code to generate filtered output

The integer array `lastFiveSamples` holds the previous five samples relative to the current sample depending on the specified frequency. It is initialized to all zeros because for the first pass, there are no previous samples. This array acts as the “push down” list, though modulo arithmetic is used to minimize rewriting the list.

At the top of the while loop, the switches are read in and used to set the order of the harmonic to be generated. The for loop runs from the beginning of the `fundamentalFreqSine` array to the end of it. The divisibility of each sample number by the desired harmonic is checked. If the sample number is divisible by the desired harmonic, it is added to the `lastFiveSamples` array.

The sample itself is generated using the filter formula. The current sample is the m^{th} sample of `lastFiveSamples`. Based on the value of m, the value of m-1, m-2, … m-5 is calculated using modulo arithmetic. The calculated sample is divided by the gain from the MATLAB simulation. Since all the frequencies of interest had a gain of 16, one gain variable was defined. This division keeps the sample value in the desired output range.

Next, 2048 is added to the sample; 2048 is the midpoint of the DAC's output range. This DC offset ensures out sample is a value between 0 and 4095. Once the sample has been calculated, the iterator into the `lastFiveSamples` array is incremented, then the modulus is taken to keep it in the right range. Finally, the sample is written to the DAC and the program calls `usleep(18)` to wait 18 µs. Waiting 18 µs ensures the sample is on the wire for a total of around 20 µs, which is the time for one sample given in the lab manual.

Below is a screenshot of the chip select signal for DA2Pmod without `usleep(18)`:

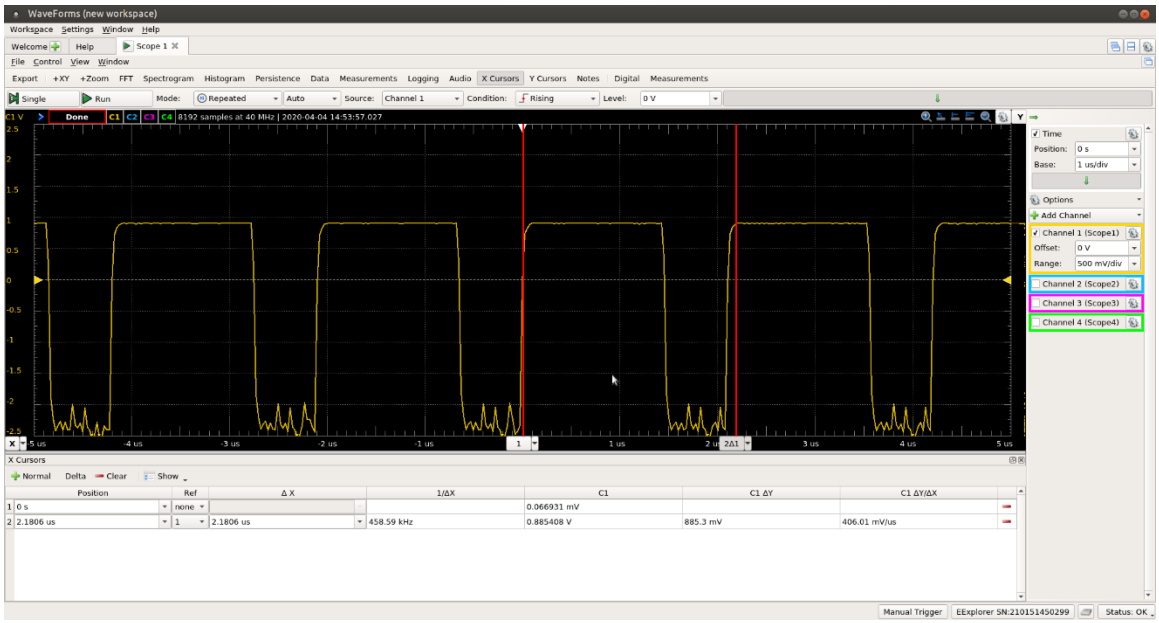

Figure 19: DA2Pmod Chip select signal without `usleep`

Note that the time that one sample is on the wire is about 2 µs. This means that to make a sample last for 20 µs, the code will have to wait for 18 µs.

Zybo FIR Filter Results

The following is a table summarizing the measured frequency compared to the desired frequency of each harmonic:

Table 1: Desired Frequency vs Measured Frequency

Desired Frequency (Hz)	Measured Frequency (Hz)	Percent Error (%)
12.21	12.051	1.30
24.42	24.039	1.56
36.63	35.559	2.92
48.84	47.847	2.03
61.05	58.591	4.03
73.26	70.042	4.39
85.47	81.127	5.08

97.68	94.619	3.13

Note that each frequency was generated to within about 5% of the desired frequency. Screenshots of for each frequency can be found in the results section. The amplitudes match the expected amplitudes from MATLAB.

Trouble Implementing Switches

Some trouble was encountered when adding switches to this lab. In the first attempt, the switches were initialized first, then the sine wave array was generated. When done in this order, the program would hang when calling `XGpioDiscreteRead()`. If the switches were initialized after the sine wave array was generated, the program would not hang, but had the following output:

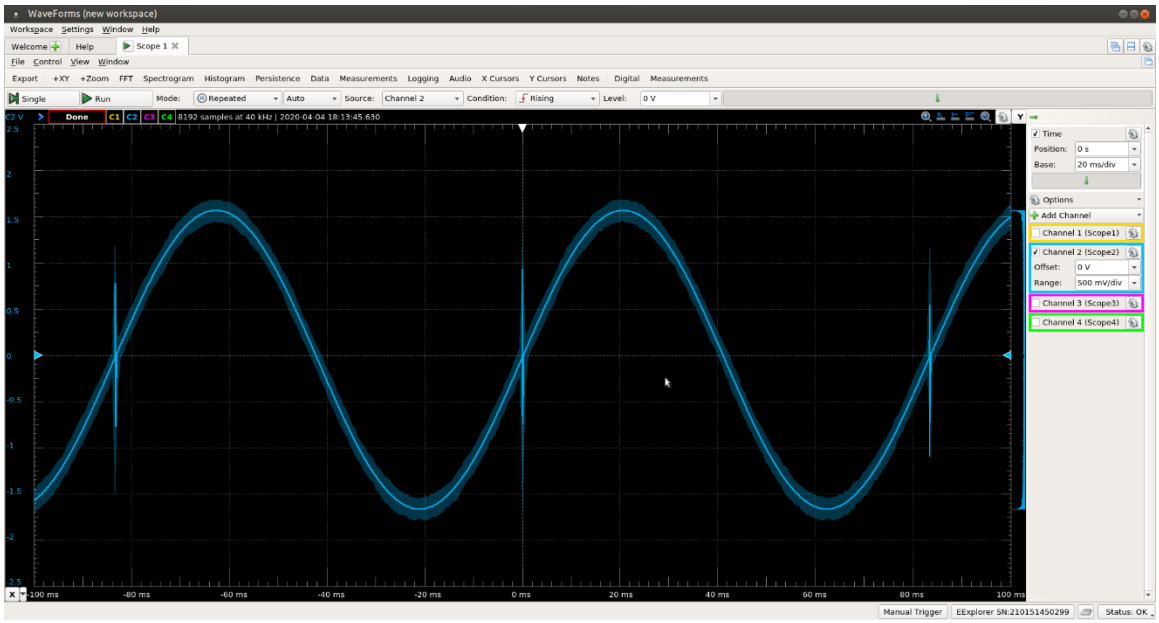

Figure 20: Noisy output

Since the noise seemed to occur at the start of each period, I assumed it was noise from the Zybo reading the switches. However, the noise remained even when the discrete read function was removed. Zooming in on the noise revealed something about its nature.

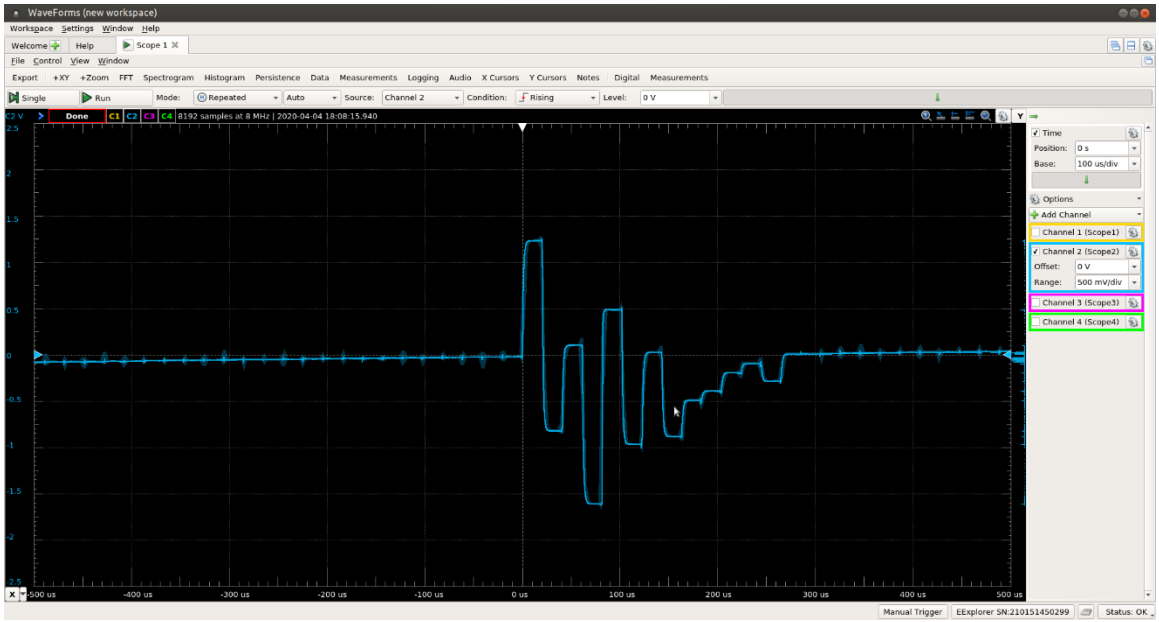

Figure 21: Close-up of noise

Instead of being random, the noise was broken discrete spikes, each lasting for 20 µs. This indicated that the noise was not random, but that something was wrong with the sine wave array. Upon printing the contents of the array before and after the switches were initialized, it was revealed that initializing the switches overwrote a few samples in the array. This in turn indicated that there was a conflict with the memory address location for the switch register. Moving the memory address of the switch register fixed the problem.

Video Link

Please see demonstration at the following link:

https://youtu.be/tkrLCJ4_Jpc

Conclusions

This lab thoroughly explored FIR filters and simulating sine waves in both MATLAB and C. Sine waves of related frequencies were successfully generated and fed through a FIR filter. The output on the DAC matched the expected results in MATLAB. Furthermore, sine waves with specific frequencies were generated in C to an acceptable error.

Appendix

Screenshots of FIR Output for each Harmonic

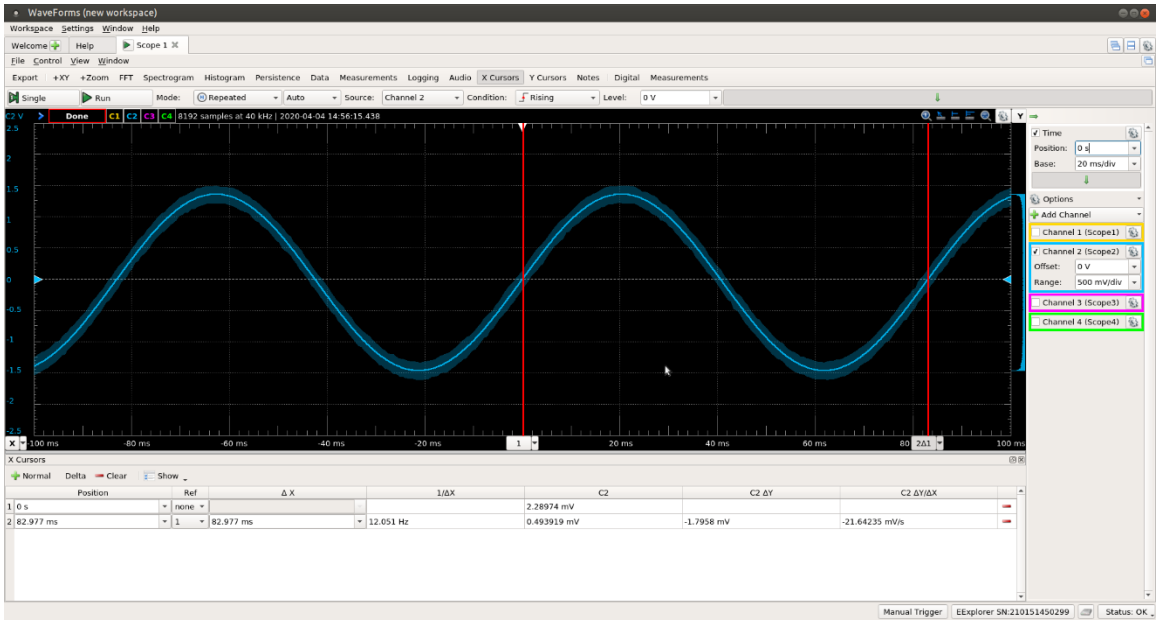

Figure 22: Output for n = 1

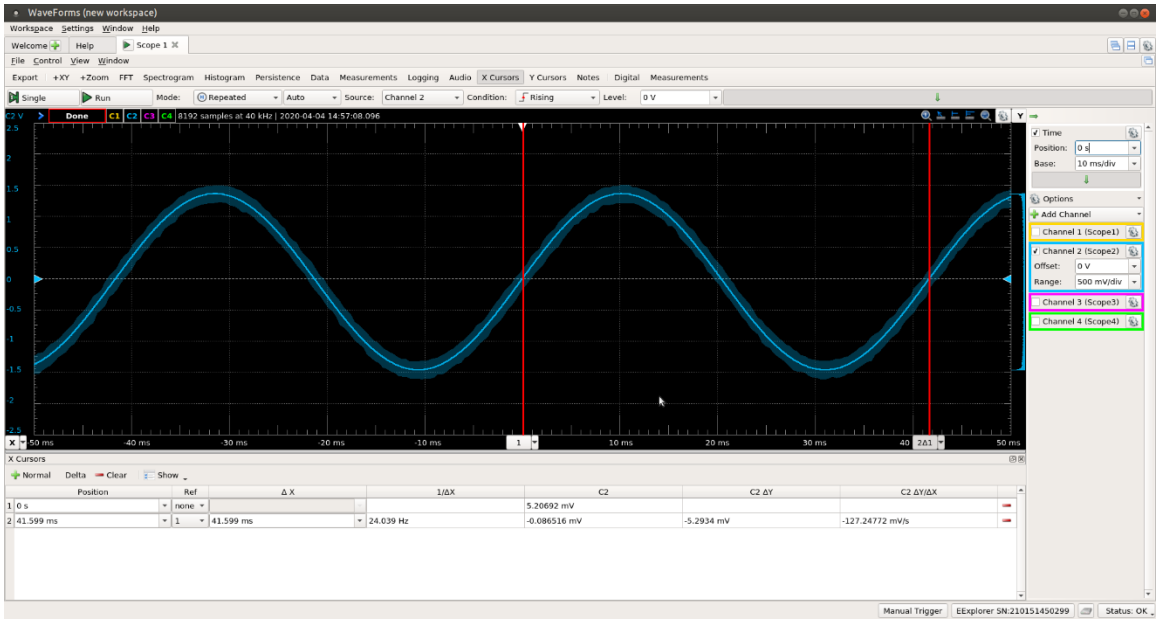

Figure 23: Output for n = 2

Figure 24: Output for n = 3

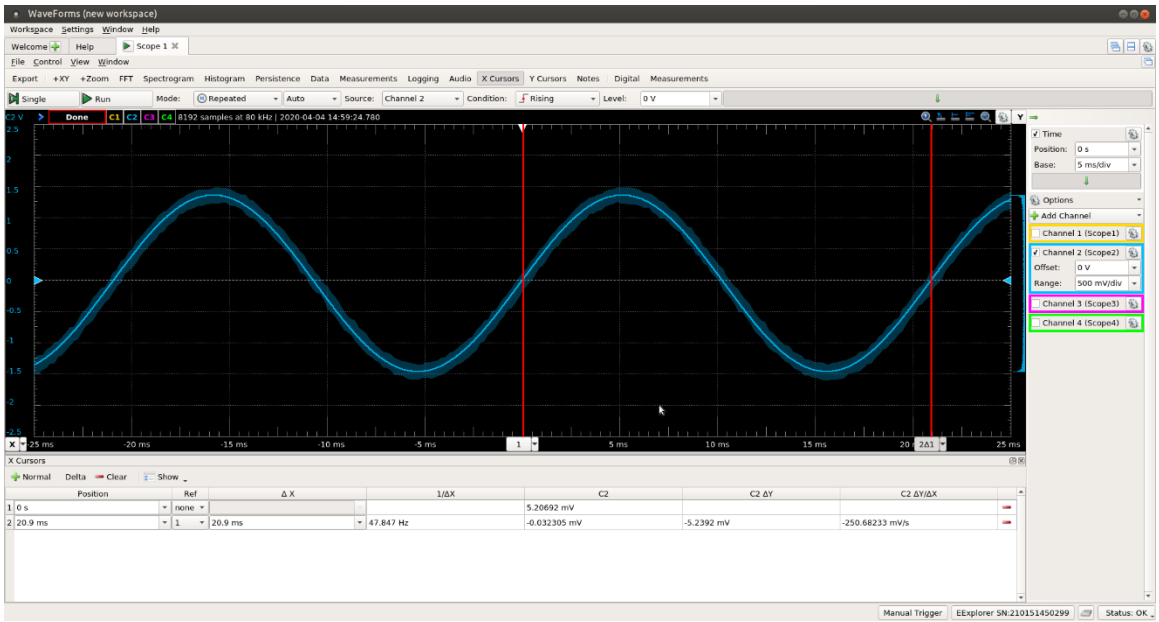

Figure 25: Output for n = 4

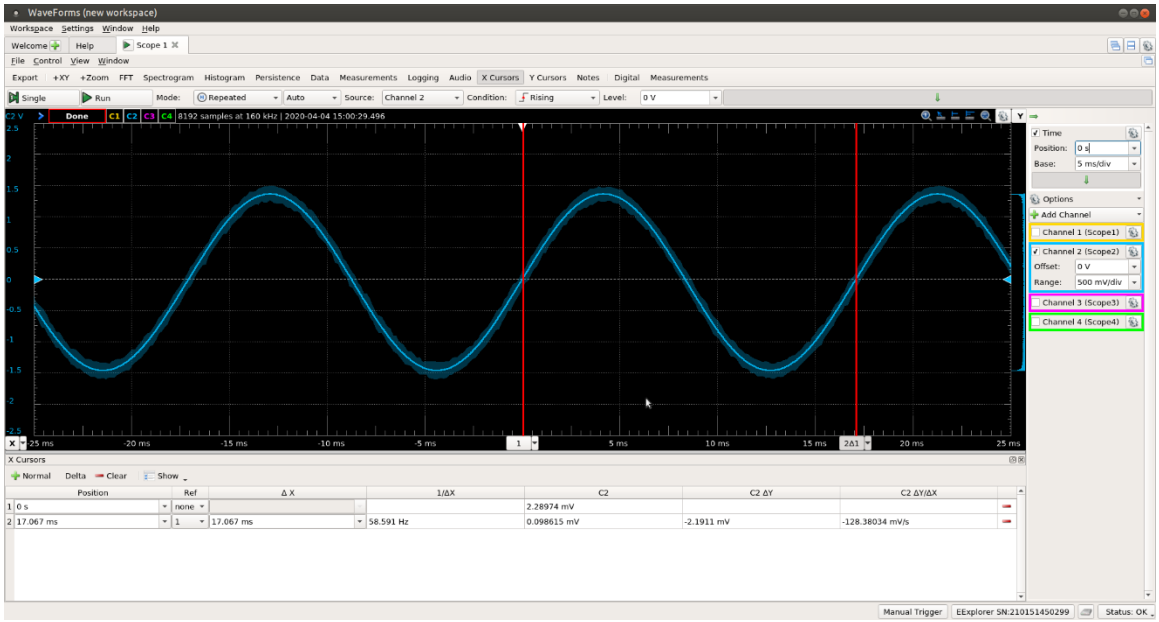

Figure 26: Output for n = 5

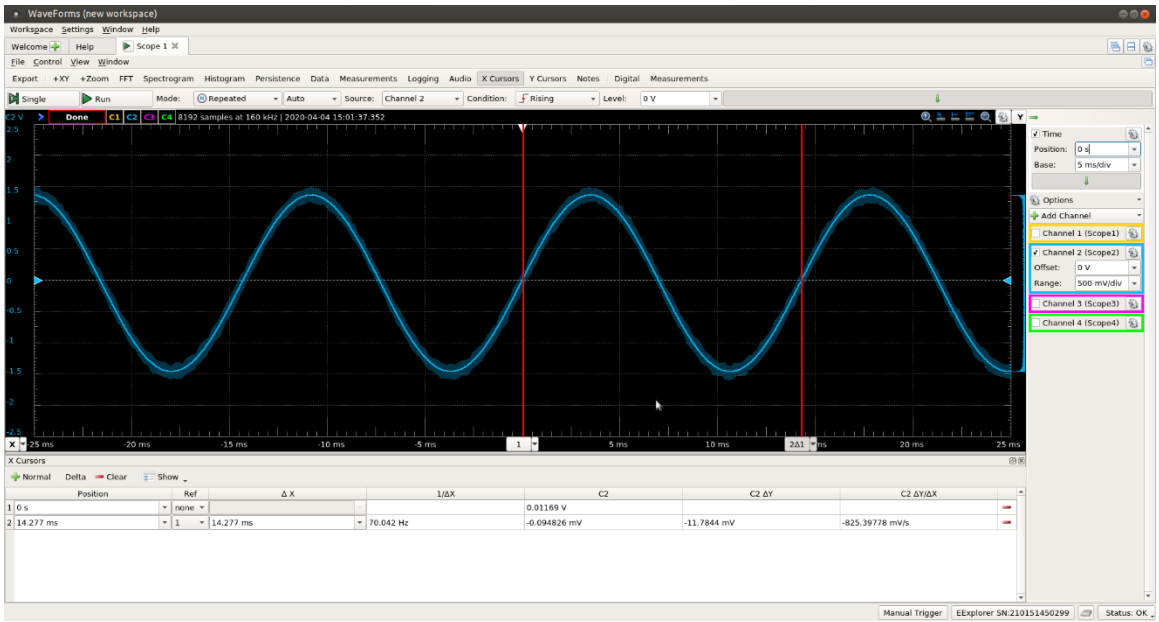

Figure 27: Output for n = 6

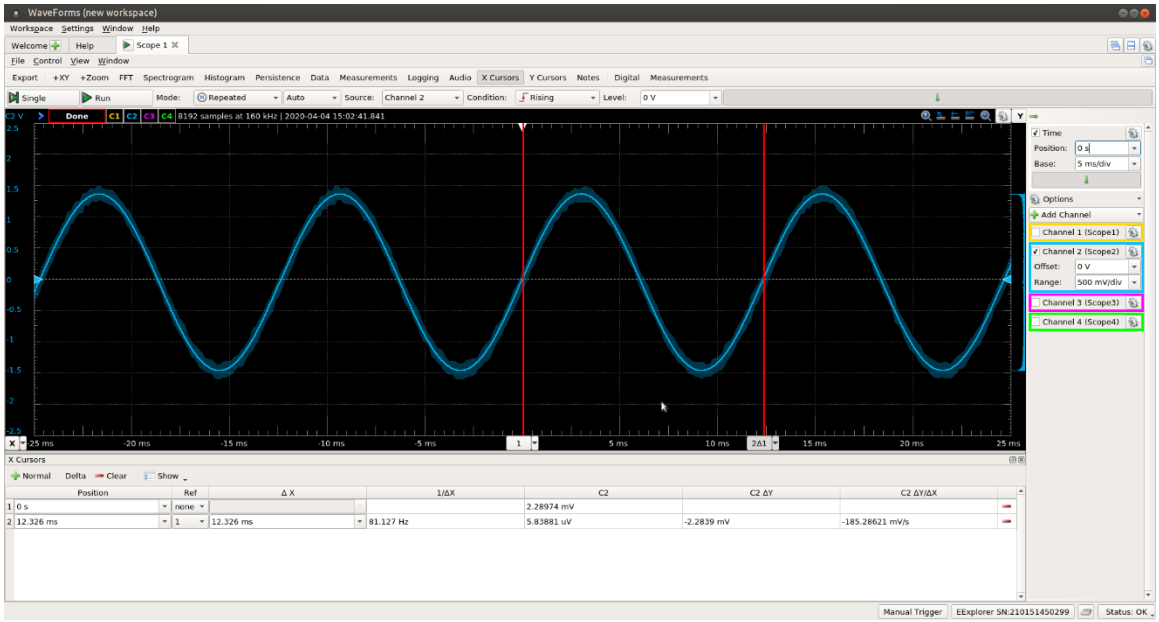

Figure 28: Output for n = 7

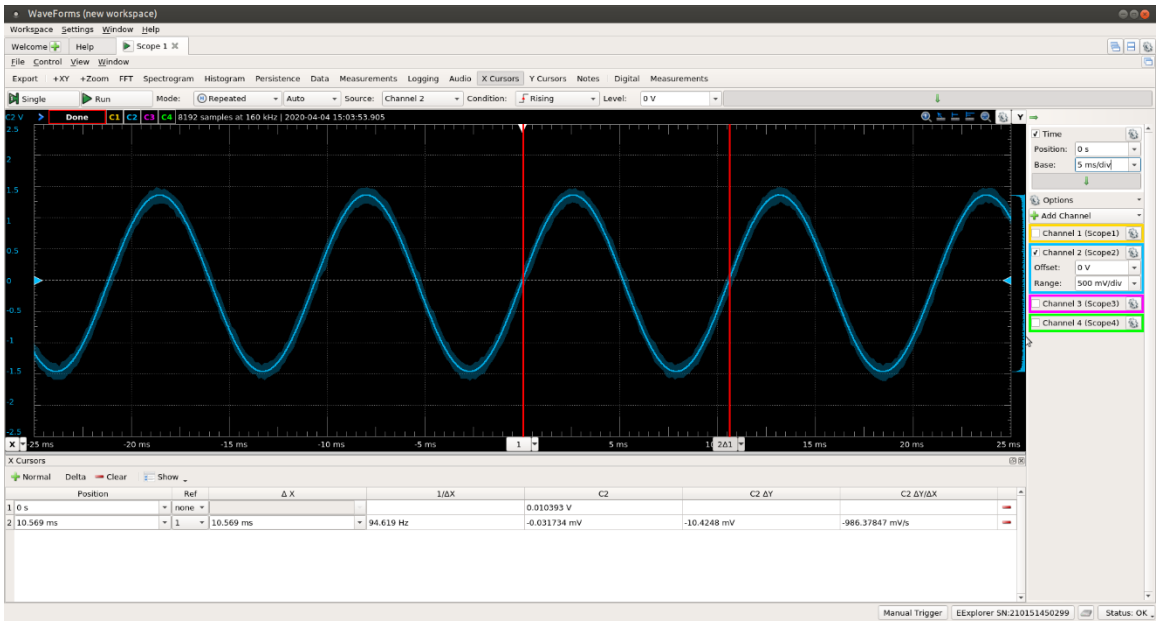

Figure 29: Output for n = 8

MATLAB Code

```
%% --------------------Task 1: Find Gain of FIR filter---------------------
clear;
clc;
figureNum = 1;

TUID = [9 1 2 2 0 1 8 1 9];
lsdPosition = length(TUID);

b = zeros(1,6); % holds coefficients based on TUID number.
fprintf("The coefficients for the TUID FIR filter are:\n");
for k = 1:length(b)
    b(k) = mod(TUID(lsdPosition - (k - 1)),5-(k-1)) + 1;
    fprintf("b%d = %d\t", k-1, b(k));
end
fprintf("\n\n");

numeratorOfFilterEquation = b; % based on z^-1 equation
denominatorOfFilterEquation = [1]; % based on z^-1 equation

% Define sampling frequency and number of points
fs = 50e3;  % Samples/Sec = Hz
ts = 1/fs;
numOfPoints = 4096;

% Create discrete transfer function based on calculated numerator and
%   denominator
figure(figureNum); figureNum = figureNum+1;
[Hz, frq] = freqz(numeratorOfFilterEquation, ...
                    denominatorOfFilterEquation, 4096, fs);
magH = abs(Hz);
plot(frq, abs(Hz));
title("Frequency Response of TUID FIR");
xlabel("Frequency (Hz)");
ylabel("Gain (magnitude)");

% Generate fundamental frequency sine wave
fundFreq = fs/numOfPoints; %Hz
discreteTime = 0:4095;
fundamentalSineWave =   floor*2000*sin(2*pi*(fundFreq/fs)*discreteTime)) …
                      + 2048;

% Plot fundamental frequency
figure(figureNum); figureNum = figureNum+1;
subplot(2,4,1);
plot(discreteTime, fundamentalSineWave);
xlim([1 4096]);
title(sprintf("Sine wave for f = %d*%0.2fHz",1,fundFreq));
xlabel("Sample Number");
ylabel("Sample Value");
hold on;

maxHarmonic = 8;  % Number of sine waves to make from fundamental frequency

% place holder for matrix holding all sines
nthHarmonicSineWaves = zeros(maxHarmonic,4096);

% put the fundamental frequency sine wave in the 1st row
nthHarmonicSineWaves(1,1:end) = fundamentalSineWave;

for k = 2:maxHarmonic
    kthHarmonicSineWave = fundamentalSineWave(k:k:end);
    len = length(kthHarmonicSineWave);
    for m = 1:k
        %Create a sine wave wave of freqency k*f by taking the kth sample
        %   of the fundamental freqency sine wave and copying it for 4096
        %   samples
        nthHarmonicSineWaves(k, 1+(m-1)*len:m*len) = kthHarmonicSineWave;
    end
    subplot(2,4,k);
```

```
    plot(discreteTime, nthHarmonicSineWaves(k, 1:end));
    title(sprintf("Sine wave for f = %d*%0.2fHz",k,fundFreq));
    xlabel("Sample Number");
    ylabel("Sample Value");
    xlim([1 4096]);
    hold on;
end

% --------------------Task 2: Create pole-zero plot-----------------------
%   H(z) = b0 + b1*z^-1 + b2*z^-2 + b3*z-3 + b4*z^-4 + b5*z^-5
%=> H(Z) = b0*z^5 + b1*z^4 + b2*z^3 + b3*z^2 + b4*z + b5
%          ---------------------------------------------
%                              z^5
%=> H(z) =    (z^5+(b1/b0)z^4+(b2/b0)z^3(b3/b0)z^2+(b4/b0)z^1 + (b5/b0))
%          b0-----------------------------------------------------------
%                                   z^5
%=> poles = {0:5} zeros = {roots from numerator}

G = b(1);
b = b./G;

% Set poles and zeros vectors. Calculations at the beginnig of theis script
poles = zeros(1,5);
zr = roots(b);

% Create complex unit circle
theta = linspace(0, 2*pi, 1e4);
unitCircle = exp(j*theta);

% Plot complex unit circle
figure(figureNum); figureNum = figureNum+1;
plot(real(unitCircle), imag(unitCircle), 'k')
axis square
hold on
% Plot zeros
plot(real(zr), imag(zr), 'bo')
hold on
% Plot Poles
plot(real(poles), imag(poles), 'bx')
title("Pole-Zero Plot of TUID FIR Filter");
xlabel("Re");
ylabel("Im");

% Find gain at fs/4
fs = 50e3;
f = fs/4;

pointOnUnitCircle = [cos(2*pi*(f/fs)), sin(2*pi*(f/fs))];

numeratorRadicand =   (pointOnUnitCircle(1) + real(zr)).^2 ...
                    + (pointOnUnitCircle(2) + imag(zr)).^2;
numerator = prod( sqrt(numeratorRadicand));

denominatorRadicand =   (pointOnUnitCircle(1) + real(poles)).^2 ...
                      + (pointOnUnitCircle(2) + imag(poles)).^2;
denominator= prod( sqrt(denominatorRadicand));

calcGainAtOneQuarterFs = G*(numerator/denominator);
fprintf("The calculated gain at fs/4 is %0.2f\n", calcGainAtOneQuarterFs);

gainAtOneQuarterFsFromFreqz = magH(find(frq == fs/4));
fprintf("The gain at fs/4 from freqz is %0.2f\n", ...
            gainAtOneQuarterFsFromFreqz);



%------------ Task 5 : Plot nth Harmonic filtered Sine Waves --------------
% Run each harmonic through the filter
filteredSines = filter(numeratorOfFilterEquation, ...
                  denominatorOfFilterEquation, nthHarmonicSineWaves,[],2);
```

```
% Plot all filtered harmonics, scaled appropriately
figure(figureNum); figureNum = figureNum+1;
for k = 1:maxHarmonic
    subplot(2,4,k);
    indexOfkthHarmonic = find(frq == k*fundFreq);
    firGain = magH(indexOfkthHarmonic);
    plot(discreteTime, (1/firGain)*filteredSines(k, 1:end));
    xlim([1 numOfPoints]);
    title(sprintf("Filtered wave for f = %d*%0.2fHz",k,fundFreq));
    xlabel("Sample Number");
    ylabel("Sample Value");
end

%------- Task 6 : Plot nth Harmonic filtered Sine Waves in Voltz ----------
% Plot all filtered harmonics in voltzs, shifted down to match oscilloscope
%   output.

VREF = 3.3;
MAX_SAMPLE_VAL = 4095;
fundamentalPeriod = numOfPoints*ts;

figure(figureNum); figureNum = figureNum+1;
for k = 1:maxHarmonic
    subplot(2,4,k);
    indexOfkthHarmonic = find(frq == k*fundFreq);
    firGain = magH(indexOfkthHarmonic);
    filteredSineShiftedDown =   (1./firGain).*filteredSines(k, 1:end) ...
                              - ceil(MAX_SAMPLE_VAL/2);
    filteredSineInVoltz =    (VREF/MAX_SAMPLE_VAL)...
                            * filteredSineShiftedDown;

    time = 1000*linspace(0, fundamentalPeriod, 4096);
    plot(time, filteredSineInVoltz);
    xlim([0 max(time)]);
    title(sprintf("Filtered wave for f = %d*%0.2fHz",k,fundFreq));
    xlabel("Time (ms)");
    ylabel("Voltage");
end
```

C Code

```
// Xilinx include statements
#include "xgpio.h"
#include "xparameters.h"
#include "xstatus.h"
#include "xil_io.h"
#include "xil_printf.h"

#include <math.h>       // used for sin
#include <unistd.h>     // used for usleep

#define printf xil_printf // smaller, optimized printf

//DAC2Pmod from Address Editor in Vivado, second IP
#define DA2acq          0x43C00000      //DA2 acquisition           - output
#define DA2dav          0x43C00004      //DA2 data available - input
#define DA2dat1         0x43C00008      //DA2 channel 1 data - output
#define DA2dat2         0x43C0000C      //DA2 channel 2 data - output

// Switch GPIO constants
#define SW_DEVICE_ID  XPAR_AXI_GPIO_0_DEVICE_ID     // GPIO device that Switches are
connected to
#define SW_CHANNEL 1                                // GPIO port for Switches

XGpio SWInst;

// Define constants
#define pi M_PI
#define F       12.21   // Hz
#define FS      50e3    // Hz
#define NUM_OF_SAMPLES 4096
#define NUM_OF_COEFFICIENTS 6
#define FIR_GAIN 16     // from MATLAB

// Function prototypes
int InitDA2();
int writeSampleToDA2(int, int);
int InitSwitches();

int main(void)
{
        // Initialize switches
        int status = InitSwitches();
        if(status != XST_SUCCESS)
        {
                printf("Failed to initialize switches!\n");
                return XST_FAILURE;
        }

        // Initialize DA2
        int dacDataAvailable = InitDA2();

        // Generate sine wave
        static int fundamentalFreqSine[NUM_OF_SAMPLES];
        printf("Generating fundamental sine wave...\n");
```

```
for (int n = 0; n < NUM_OF_SAMPLES; n++)
{
	fundamentalFreqSine[n] = (int)(2000*sin(2*pi*(F/FS)*n));
}
printf("Fundamental sine wave complete.\n");

// Print coefficients found in MATLAB
int b[NUM_OF_COEFFICIENTS] = {5, 2, 3, 2, 1, 3};

int sample, nthHarmonic, m;
unsigned int switchValue;
int lastFiveSamples[] = {0,0,0,0,0,0};

// Initialize iterator for lastFiveSamples[m]
m = 0;

printf("Entering main while loop....\n");
while (1)
{
	switchValue = XGpio_DiscreteRead(&SWInst, SW_CHANNEL) & 0xF;
	nthHarmonic = (switchValue & 7) +1;

	// Generate output based on selected harmonic
	for (int sampleNum = 0; sampleNum < NUM_OF_SAMPLES; sampleNum++)
	{	// Check to see if the sample number is divisible by the desired harmonic
		if ( sampleNum % nthHarmonic == 0 )
		{	// If it is...
			// Add the current sample to the list of last five samples
			lastFiveSamples[m] = fundamentalFreqSine[sampleNum];

			// Apply FIR filter
			sample =   b[0]*lastFiveSamples[m]
				 + b[1]*lastFiveSamples[((m-1)+6)%6]
				 + b[2]*lastFiveSamples[((m-2)+6)%6]
				 + b[3]*lastFiveSamples[((m-3)+6)%6]
				 + b[4]*lastFiveSamples[((m-4)+6)%6]
				 + b[5]*lastFiveSamples[((m-5)+6)%6];

			// Divide by FIR gain as found in the MATLAB simulation to keep
sample in -2048..+2047
			sample /= FIR_GAIN;

			// Add DC component to sample to keep it in 0..4095
			sample += 2048;

			// Increment iterator for the last five samples
			m++;

			// Keep m in 0..5
			m = m%6;

			// Write sample to DA2
			dacDataAvailable = writeSampleToDA2(sample, dacDataAvailable);

			// Wait so total time sample is on the wire is 20us
```

```
                                        usleep(18);
                                }
                        }
                }
        }
}

int InitDA2()
{
        int dacDataAvailable;
        printf("Initializing DA2...\n");
        Xil_Out32(DA2acq, 0);                   //DAC stop acquire
        dacDataAvailable = Xil_In32(DA2dav);    //DAC available?
        while (dacDataAvailable== 1)
                dacDataAvailable = Xil_In32(DA2dav);
        printf("DA2 initialized.\n");
        return dacDataAvailable;
}

int InitSwitches()
{
        printf("Initializing switches...\n");
        int status;
        status = XGpio_Initialize(&SWInst, SW_DEVICE_ID);
        if(status != XST_SUCCESS) { return XST_FAILURE; }
        printf("Switches initialized.\n");
        printf("Setting data direction for switches...\n");
        XGpio_SetDataDirection(&SWInst, SW_CHANNEL, 0xF);
        printf("Data direction set.\n");
        return XST_SUCCESS;
}

int writeSampleToDA2(int sample, int dacDataAvailable)
{
        Xil_Out32(DA2dat1, sample);             //output DAC data
        Xil_Out32(DA2acq, 1);                   //DAC acquire
        while (dacDataAvailable == 0)   //DAC data output?
                dacDataAvailable = Xil_In32(DA2dav);
        Xil_Out32(DA2acq,0);
        while (dacDataAvailable == 1)
                dacDataAvailable = Xil_In32(DA2dav);
        return dacDataAvailable;
}
```