Axi4Lite FIR Peripheral

Ben Guan, Jeremiah Brannon
CSE 465 Fall 2020

Abstract
Finite impulse response (FIR) filters are commonly used in digital signal processing to filter signals. One of the many types of signals that can be filtered using this method is sound waves.

Our project will be to implement a FIR Low Pass Filter in software and hardware using C code and Verilog HDL. We will use two sine waves as inputs, one in the passband range and one in the stop band range, to test our filter. In a later project we will use our implemented FIR Filter to filter different frequencies of sound inputs.

Introduction

We used the LPFDesign tool to generate a simulation of a filter with a passband frequency of 3.3kHz, a stopband frequency of 6kHz, a 60dB attenuation in the stopband and a 8.68dB passband ripple.

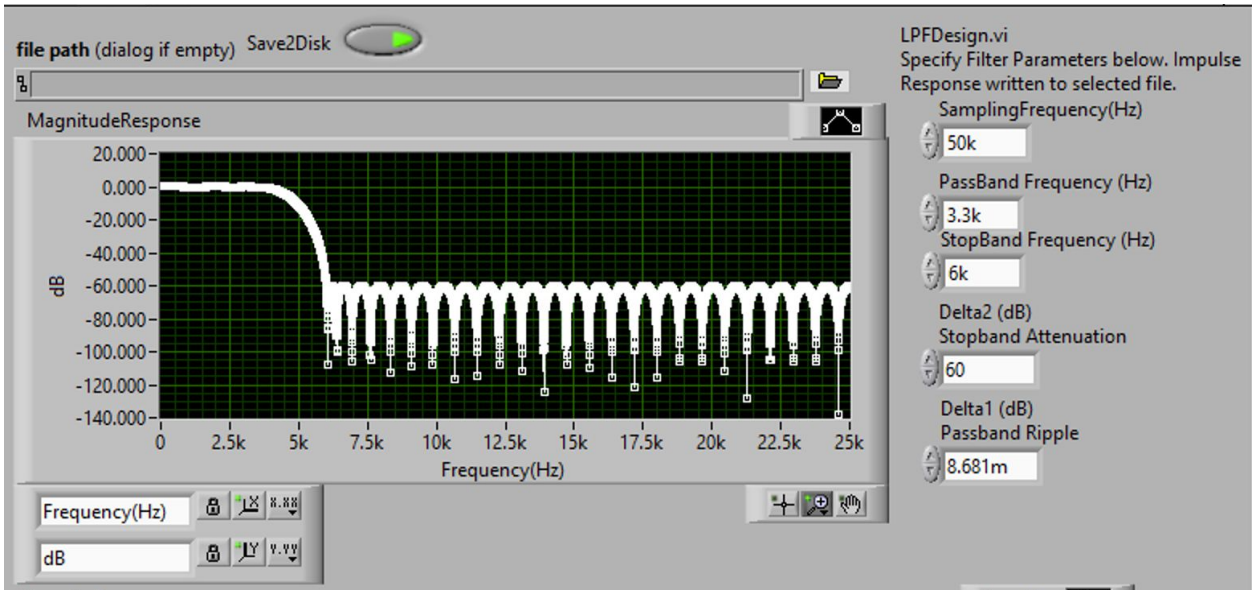

Figure 1. Magnitude Response of the FIR Filter

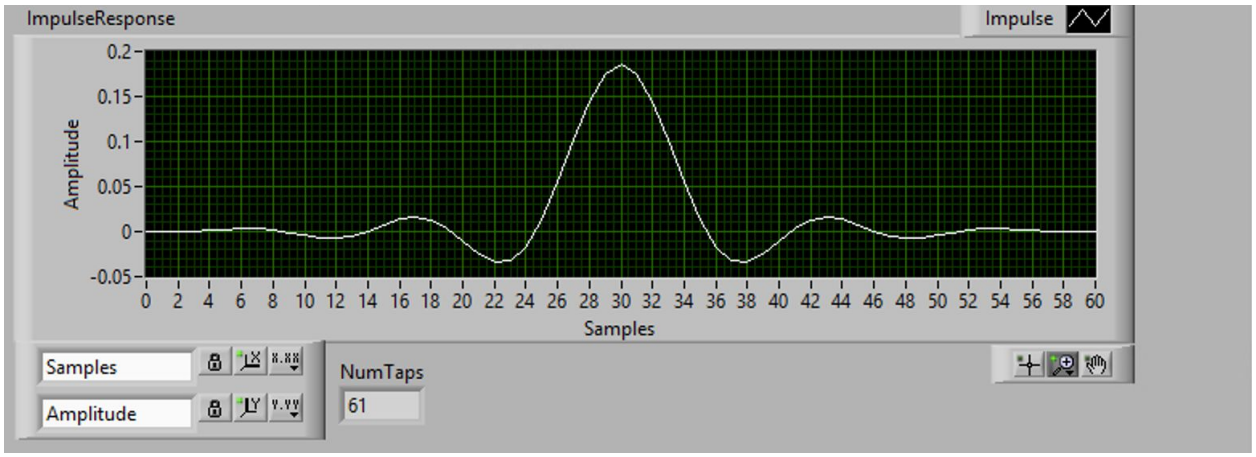

Figure 2. Frequency Response of the FIR Filter

We converted the sine wave inputs and filter coefficients into Q.15 format for easier computations. We then fed the coefficient values and a sine input into the FIR filter, which calculated the dot product of the vectors. To send data samples to our hardware implementation of the filter we wrote a SDK program in C. The SDK sends data through a microblaze across an Axi4Lite bus to our hardware implementation of the filter. In our filter we stored the sine wave values in a circular buffer, used for calculating the filtered

value. When the FIR finishes the calculation, the peripheral sends the computed result back to the microblaze. A high-level testbench is implemented to validate the design.

Design

1. Software Design:

Using the Filter we generated with the LPFDesign tool, we found two discrete points on the Magnitude response, one in the stop band state and one in the pass band state. These gave us the two frequency values and corresponding expected gain values we could use for ensuring our filter worked accurately. Our first frequency value was measured at 501Hz with an expected gain of .007595db and our second frequency value was measured at 6000Hz with an expected gain of -59.8514db. We then generated 1000 points from the functions: f1 = sin(501*2πt), f2 = sin(6000*2πt) at a sample rate of 50kHz, storing the outputs in an excel document. We then converted the values into Q.15 to make the calculations possible in binary hardware since they were decimal values.

The Q.15 values that we computed from our sine functions we stored in short integer arrays and saved them in header files that we could include in our program. Our program consisted of an initFilter() function and a Filter() function. The initFilter() function initialized the buffer for our filter coefficient values and the cyclical buffer used to store the most recent 61 sine values. The Filter() function convoluted the last 61 sine values that were currently stored in our cyclical buffer array and the filter coefficient values, returning the computed result as a Q.15 value. Our main function called our initFilter() function and entered a for loop that iterated through the 1000 Q.15 values stored in our sine array. Inside the for loop it stored the most recent value at the current index in our cyclical buffer and called our Filter() function on the current sine value.

We printed the filter values to the console from our Filter.c program and copied the values over to our excel document. Inside the excel document we converted the filter values from Q.15 back to decimal format for easier comprehension. To ensure that our Filter worked as expected, we found the maximum value returned by our Filter function and divided that by our maximum input value. For the 501Hz frequency sine function the gain was computed as 1.000875 and for the 6kHz frequency sine function the computed gain was .001017. We then converted these results into gain in db which came out to .007595db for the 501Hz frequency sine wave and -59.8514db for the 6kHz sine wave.

2. Hardware Design:

As shown in the figure below, an Axi4Lite supporter is instantiated in the Axi4Lite FIR. It is used to communicate with the Axi4Lite manager in the microblaze. The RAM used in this project is the single-port asynchronous read RAM.

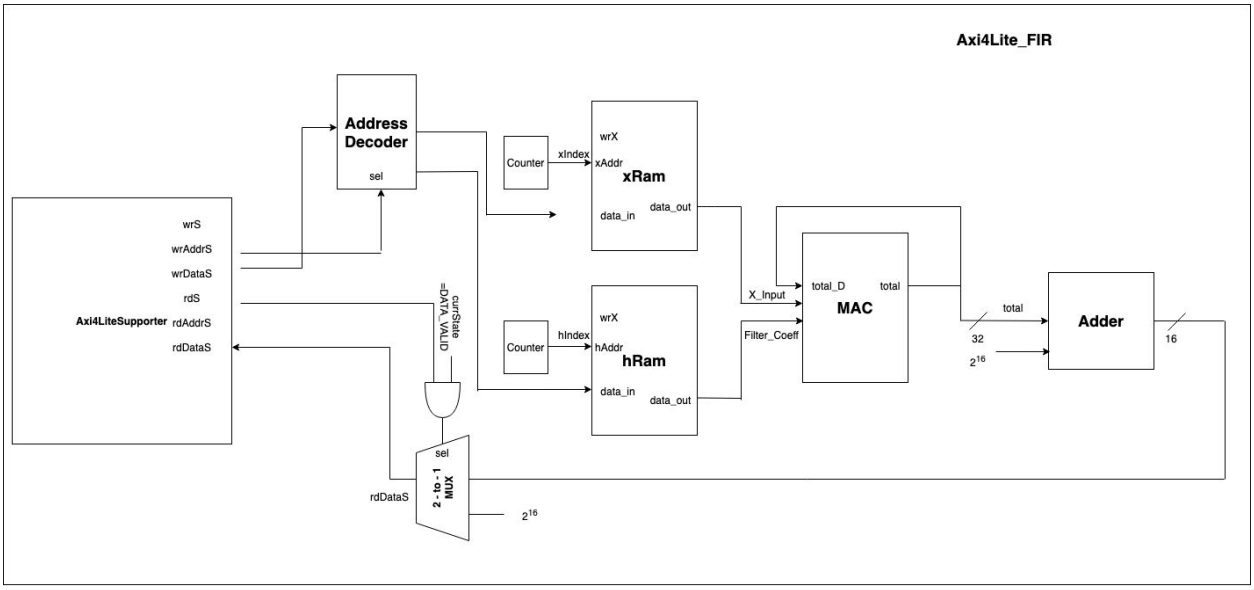

Figure 3. Block Diagram

When the SDK is launched, input and filter coefficients are written to the ram. An Axi4Lite supporter is instantiated in the Axi4LiteFIR so that it can communicate with the Axi4Lite manager in the microblaze.

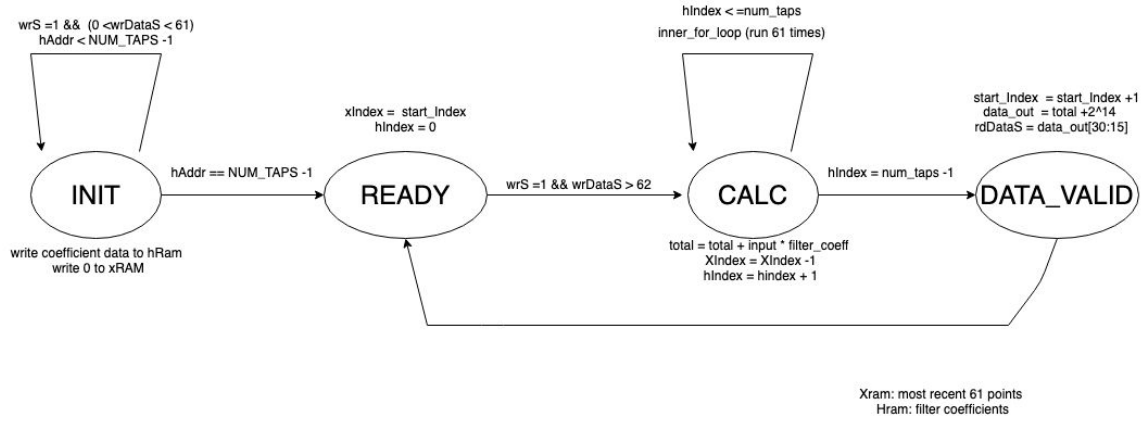

Figure 4. FSM of the FIR Peripheral

The FSM consists of 4 stages: INIT, READY, CALC, and DATA_VALID. In the INIT stage, when the Axi4Lite_FIR peripheral receives a write signal at address zero it expects to be receiving the filter size and stores that value in NUM_TAPS. When it next receives a read signal at address four, it will be ready to load filter coefficients to the hRam and initialize the xRam with zeroes.

In the READY stage, the FIR peripheral will wait to receive a sample. When the Axi4Lite_FIR peripheral receives a write signal and the wrAddr is equal to eight, then the peripheral will store the sample into xRam in a circular mannar and go to the next state CALC. xCount is used to keep track of the starting index.

In the CALC stage, the FIR peripheral will compute the dot product of the xRam and hRam. At each clock cycle, the product of the sample and filter coefficients will be added to a running sum. It takes 61 clock cycles to compute the dot product.

In the DATA_VALID stage, the FIR peripheral will send the calc_done signal to the microblaze when a rd is asserted at rdAddr twelve. Once the read at address twelve returns a nonzero value, the SDK knows the calculation has been completed and attempts to read from address eight. When the FIR peripheral receives a read signal at address eight, it adds 2^{14} to the dot product and outputs the result, 31st bit down to 15th bit, back to the microBlaze.

As shown in the figure below, the microblaze is connected to the Axi4Lite FIR via the Axi4Lite bus. An ILA is connected to the Axi4Lite bus. For debugging purposes, we constructed a long vector called “probe” which concatenates all the important information which provides more observability. There are two external ports, macs and samples, which are registers that we can write to when timing the sample rate.

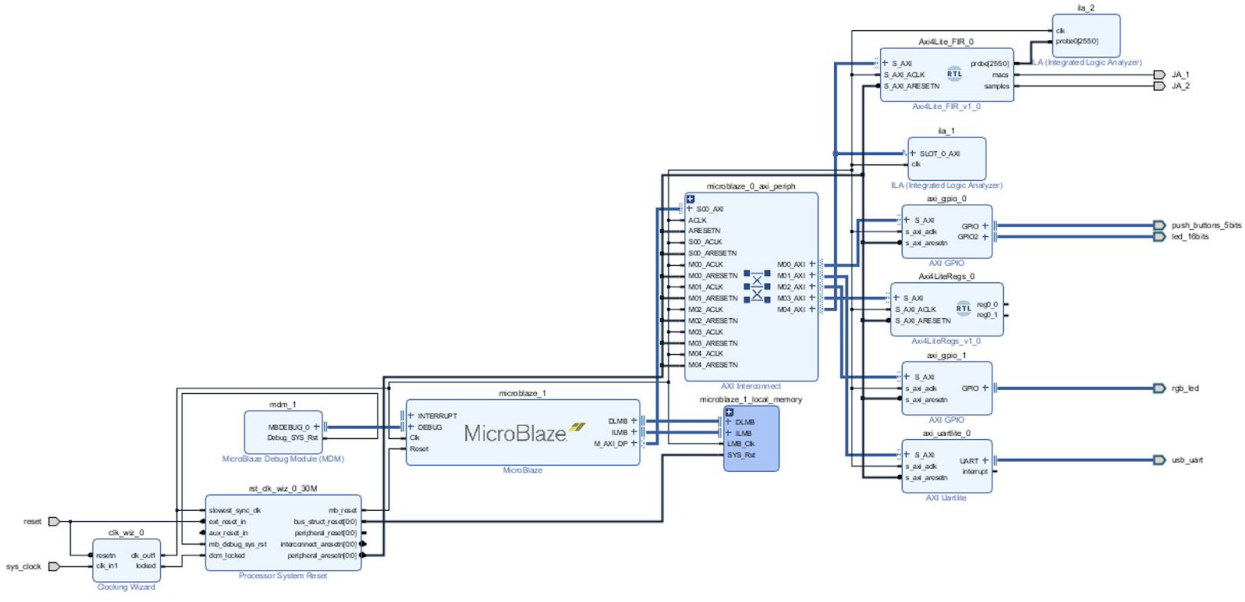

Figure 5. Block Design of System

Address Editor

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_1					
Data (32 address bits : 4G)					
Axi4LiteRegs_0	S_AXI	reg0	0x44A0_0000	64K	0x44A0_FFFF
Axi4Lite_FIR_0	S_AXI	reg0	0x44A1_0000	64K	0x44A1_FFFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_gpio_1	S_AXI	Reg	0x4001_0000	64K	0x4001_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_1_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	64K	0x0000_FFFF
Instruction (32 address bits : 4G)					
microblaze_1_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	64K	0x0000_FFFF

Figure 6. Address Editor

The base address of our FIR peripheral is located at 0x44A1_0000. To read and write to addresses in our FIR peripheral from our SDK, we had to use an offset from this base address. For example, to read from address 8 of our FIR peripheral, the SDK just needs to dereference address 0x44A10008.

Operation and Testing

Software:

The 501 Hz sine wave and the cooresponding filtered output are shown below. Since 501Hz is in the passband, the output looks almost identical except for the first 60 data points.

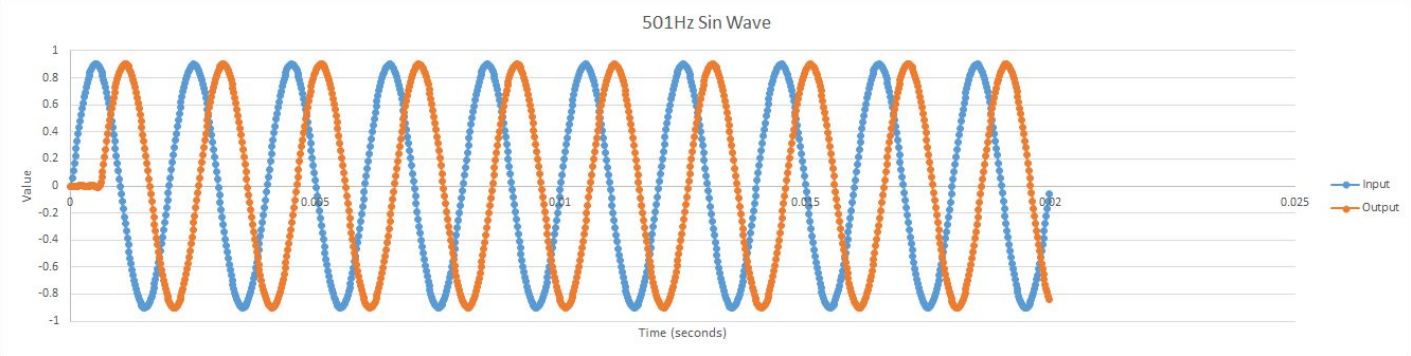

Figure 7. 501Hz Sine Wave Input and FIR Output

The 6000 Hz sine wave that is attenuated by 60.05 dB and the cooresponding filtered output are shown below. Since 6000Hz is in the stopband, the calculated gain is -59.85dB which is very close to the expected gain.

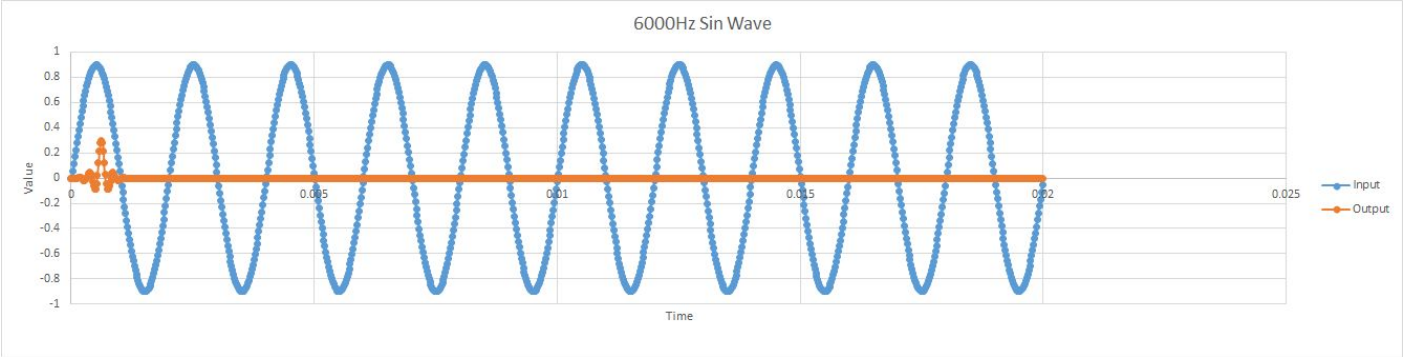

Figure 8. 6000Hz Sine Wave Input and FIR Output

The table below shows the theoretical and calculated gain of the two sine waves, 501Hz and 6000Hz. The calculated gains are consistent with the expected gain.

	Expected Gain(dB)	Calculated Gain (dB)
501Hz	0.0072	0.0075
6000Hz	-60.05	-59.85

Table 1. Expected Gain(dB) and Calculated Gain (dB) of the C implementation

Hardware:
A module-level testbench was developed to verify the functional correctness of the design. In the testbench, an Axi4Lite Manager was instantiated in the testbench. The manager will communicate with the Axi4Lite supporter in the Axi4LiteFIR module through the Axi4Lite bus. We first send a handshake signal from the testbench to the Axi4LiteFIR to set the state to INIT. Then we first write the filter coefficients to ram through the Axi4Lite bus. During the INIT stage, the XRam was zeroed out. During the Ready stage, it computes the dot product of the circular XRam and hRam. During the Data_Valid stage, the Axi4LiteFIR will wait for a read at address 12 offset from FIR base address 0x44A1_0000. That is the calc_done flag. If calc_done is 1, that means the FIR finishes the FIR calculation. Then the FIR will put the final result in Q1.15 format on the Axi4Lite bus.

As shown in the figure below, the FIR peripheral is initially at INIT (0) state. Once the microblaze asserts write at the correct address, the FIR peripheral will store that number into the hRam. In READY (1) state, the FIR will wait for a sample from the microblaze then it will go into CALC (2) state. After the dot product calculation is done, it goes into DATA_VALID (3) state. Then it will wait for a read signal from the microblaze which will read the calc_done flag located in memory 0x000c offset from the base address of the FIR peripheral. Once the calc_done flag is 1 the peripheral waits for

another read at address 0x0008 and puts the calculated result on the simple bus. Then the result will propagate through the Axi4Lite bus to the microblaze.

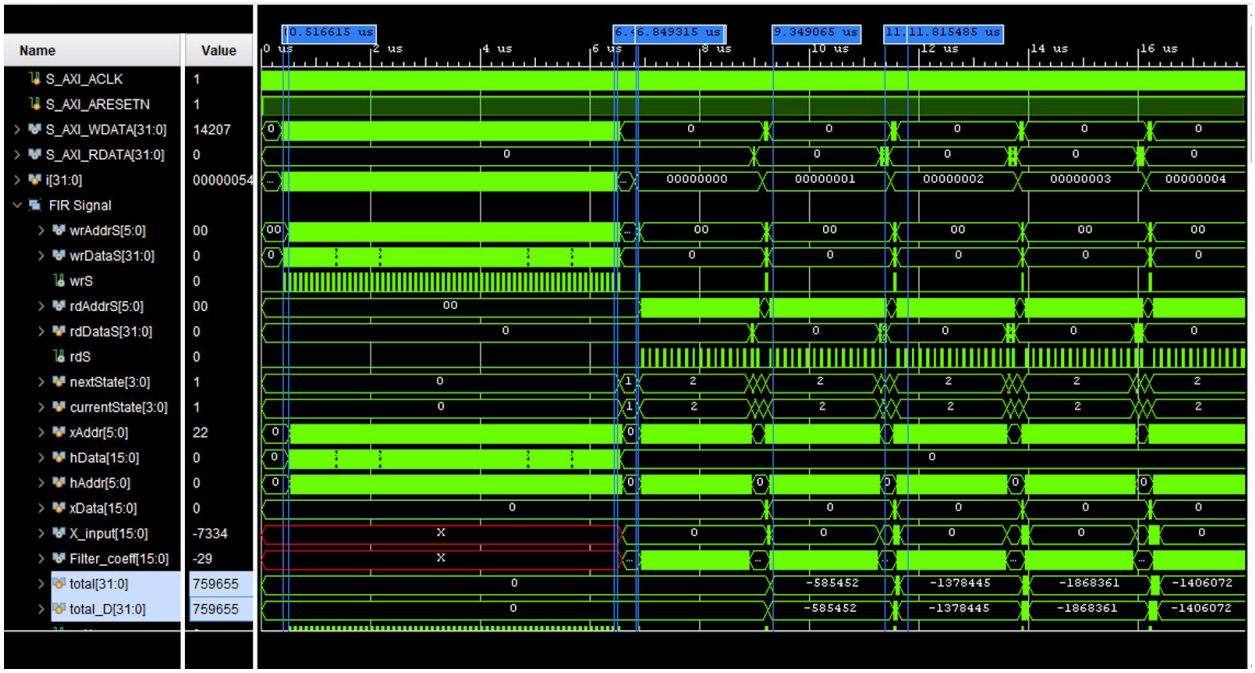

Figure 9. Simulation of FIR Peripheral

Discussion and Conclusion

Once we have demonstrated the testbench is working correctly, we started the implementation of the SDK program. However, we ran into issues where the SDK was not performing the way we expected, despite that testbench working properly. To gain more observability, we created another ILA which allows us to export a large vector out of the FIR peripheral. In the FIR, we concatenate important variables into a single vector and make it an output. From the hardware manager we parse the long vector into corresponding variables for our ILA probe to trigger on. We noticed that the FIR peripheral was not resetting to the INIT state when we ran our SDK program. To fix this issue, we created a handshake that reset the FIR at the very beginning of the SDK program and it worked as expected.

Using the oscilloscope, we found that to execute all 1000 data points with the FIR implementation in C, it took about 0.2994 s which is 3.34 sec.

During that time, there were 61,000 MAC calculations being done. Thus the MAC/s is 203,740 and the ADC rate is 3,340 sample/sec.

MAC/sec = 3.34 x 61,000 mac = 203,740 MAC/ sec

ADC Rate = 203,740 / 61 = 3,340 sample /sec

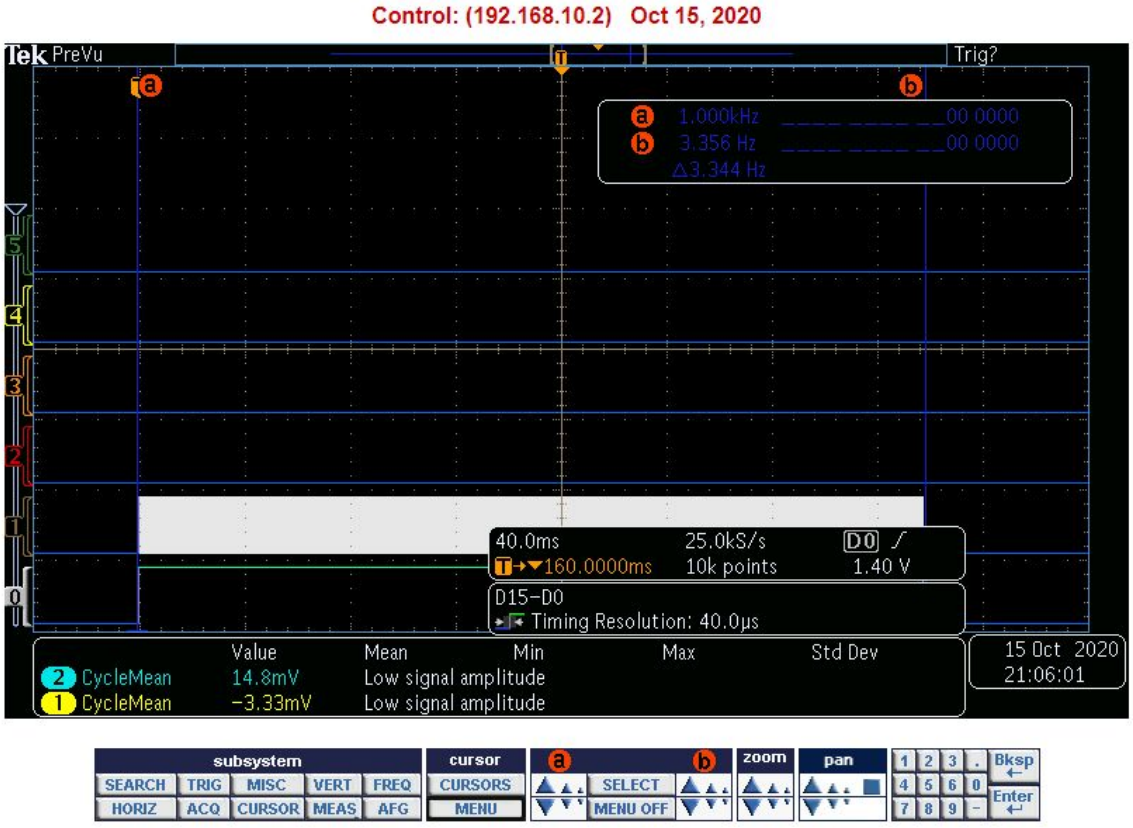

Figure 10. MAC Measurement of Oscilloscope for Software Implementation

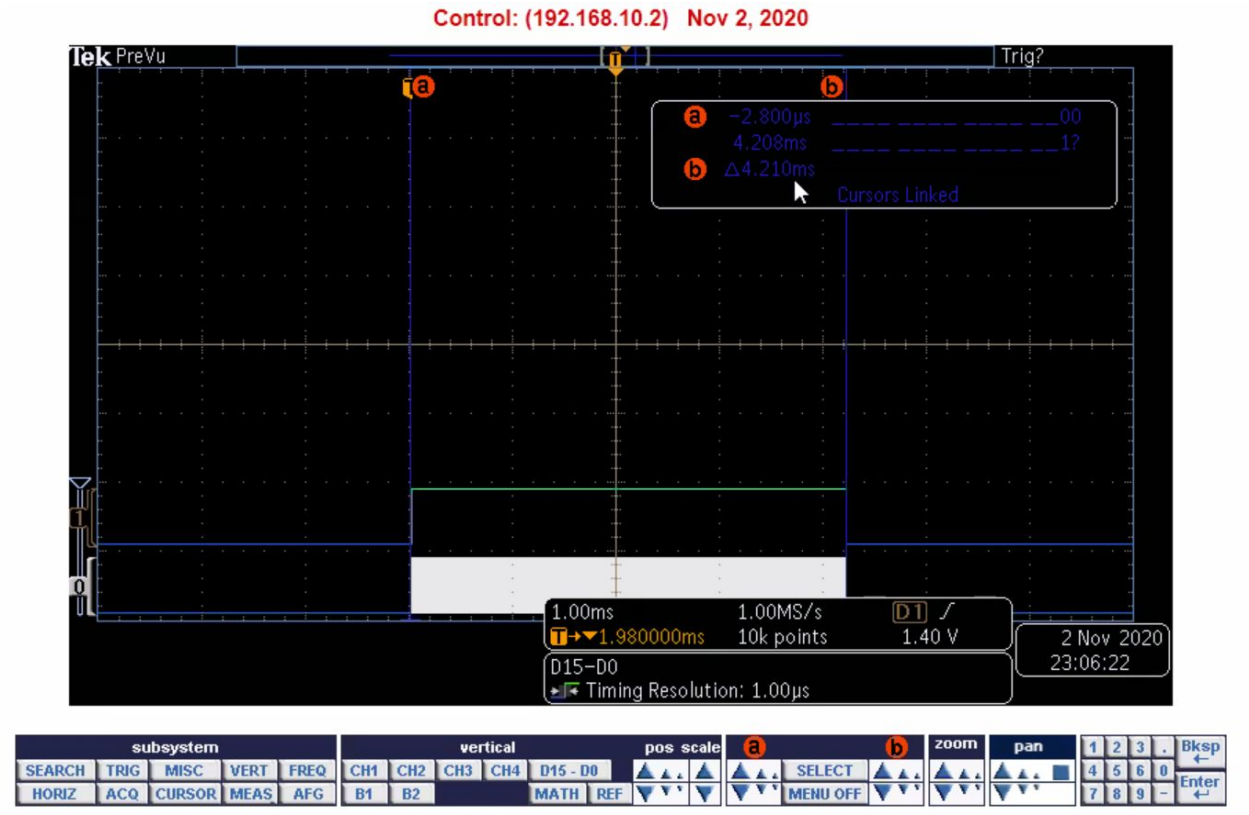

Figure 11. MAC Measurement of Oscilloscope for Hardware Implementation

For the hardware implementation, given that the duration of the execution is 4.21ms.

MAC/sec = 1/(4.21ms / 61,000 mac) = 14,489,311 MAC/ sec

ADC Rate = 14,489,311 / 61 = 237,529 sample /sec

The MAC/sec is 14,500,000. The ADC sample rate is about 238,000 samples/sec.

	MAC/sec	ADC Rate(sample /sec)
Software (C)	203,740	3,340
Hardware (Verilog)	14,489,311	237,529

Table 2.MAC/sec and ADC Rate(sample /sec) of Software and Hardware Implementation

Compared to the software implementation, the hardware implementation is significantly faster. Under ideal circumstances, the maximun ADC rate is 30 million samples/sec. Our implementation is significantly slower due to overhead in the C program and handshaking required to pass information across the Axi4Lite bus.