# IIR IP

Kevin Bloom

May 4, 2017

# Contents

# 1 Introduction

When selecting this project, I was under the assumption that it would be fairly easy to complete; seeing as completing this task in software isn't a huge deal. To my surprise, this was far from true. The calculation for IIR is quite difficult to complete in hardware, as I will discuss in this report. I will be discussing the theory behind the IIR in hardware, the major issues that can occur, and my work on the subject.

## 1.1 Important Files

Inside of this project, there are a few different files that are important. Firstly, my notebook. This file is entitled `notebook.org` and contains information on my process throughout the semester. You can open it with a text editor of choice, or you can view the exported `notebook.html` file. Please note that the HTML version doesn't contain all of the clock stamps. Inside of `presentation/` is the presentation and its source. The source for this report can be found in `report/`. Inside of `projects/` will be all of the different projects that I worked on. The important ones to note are: `single-section/`, `complex-iir/`, `multi-stage-hw-iir-final/`, and `multi-stage-gen-iir/`.

Just so that it's easier to find your way, I will do a quick description of each project. This will prevent you from having to search around. `single-section/` is a design that only uses a single section BiQuad. This was used to prove that there was something wrong with the BiQuad implementation of the IP. `complex-iir/` contains a design that doesn't use the Zynq. It does everything in hardware. The current setup uses internally selected coefficients, opposed to the normally externally fed coefficients. `multi-stage-hw-iir-final/` is the project that I use in the presentation because it meets the project requires better than the others. This project uses individual GPIOs to feed coefficients to each IIR stage. Lastly, `multi-stage-gen-iir/` is the largest project. It contains extra hardware designs called `mux` and `dmux`. These are used to allow the user to select which output they want to view and/or create a smaller ordered filter with the existing hardware.

## 1.2 Licensing

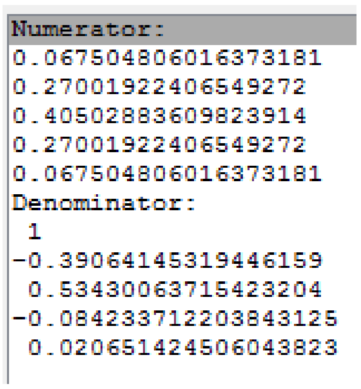
The last thing that should be noted is that any code that was written by me, Kevin Bloom, is licensed under the GNU Lesser General Public License (LGPL) version 3 or above. A copy of this license is found in the `LICENSE` file in the root directory of the project. This includes any IIR, FIR, mux, and demux VHDL files and any IIR communication C files. It is important that whomever is given the projects is also given the *same freedoms as me, Kevin Bloom.* Please make sure that this happens. Also, if someone is given this project *without those freedoms*, please see my GitHub page to receive copies of my code *with those freedoms*.

## 2 Single-Section

In this section, I will discuss the problem related to using a single-section or standard form IIR in hardware. The term *single-section* is referring to the number of sections, or stages, needed to complete the calculation. In a single-section filter, there is only 1 section needed. This means that there is no need for cascading multiple stages together. You can convert your IIR filter to a single-section in MATLAB by selecting the `Convert to Single Section` under the `Edit` tab.

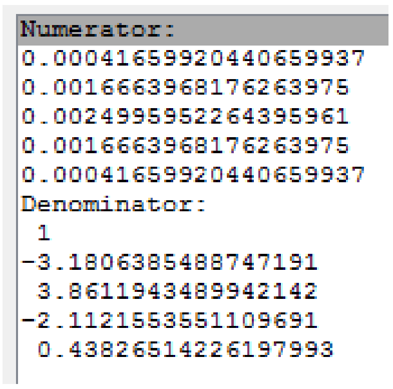
### 2.1 Coefficients

As soon as you convert your filter to a single-section, you will immediately notice that there isn't a gain listed. This is most likely because the gain is incorporated into the coefficients already and is not something you should be concerned about. Another thing you may notice is that the number of coefficients in each pair (whether numerator or denominator) is probably larger than in the multi-section or *BiQuad* form. In a BiQuad form both coefficient arrays have 3 elements yet in the single-section they may have more. Take a look at the example in **Figure 1** to see this. Here, you will see that we have 5 coefficients in each array. This may not seem problematic and it really isn't. It would be fairly easy to design hardware that could take a varying number of coefficients. All you would need to do have a maximum number of coefficients set, then depending on the number passed in, you would clear the not-in-use registers to zero. In theory, this would work great. However, there is still a major issue with using single-section that doesn't have to do with number of coefficients. It has to do with the *size* of the coefficients. Take a look at **Figure 2** and you will notice 2 interesting things: the numerator coefficients are quite small and the denominator coefficients are big.[1]



```
Numerator:
0.067504806016373181
0.27001922406549272
0.40502883609823914
0.27001922406549272
0.067504806016373181
Denominator:
 1
-0.39064145319446159
 0.53430063715423204
-0.084233712203843125
 0.020651424506043823
```

Figure 1: $F_s = 48$kHz, $F_c = 10.8$kHz



```
Numerator:
0.00041659920440659937
0.0016663968176263975
0.0024995952264395961
0.0016663968176263975
0.00041659920440659937
Denominator:
 1
-3.1806385488747191
 3.8611943489942142
-2.1121553551109691
 0.43826514226197993
```

Figure 2: $F_s = 100$kHz, $F_c = 5$kHz

So, what's the big deal? Well, if you attempt to convert those numbers to fixed point uses our standard quantisation 1.15[2] you get overloaded values in the denominator and zeros in the numerator. That being said, we can generalize this problem by saying that the the numerator coefficients approach zero and the denominator coefficients approach infinity. Due to this problem, it is nearly impossible to create a generic IIR filter using the single-section method.[3]

[1]In perspective to the numerator coefficients

[2]This is fancy for, 1 bit integer and 15 bits fractional. We could just as easily have said quantisation 2.14; 2 bits integer and 14 bits fractional.

[3]Well, one that's worth a damn that is.

### 2.2 Benefits

Just because single-section isn't the best way to do things, it still has some benefits over the BiQuad method. Firstly, it is far simpler. You don't need to mess around with extra signals and cascading. It makes the design, as a whole, much simpler. Secondly, it requires less hardware. This is a good thing, especially if you are on limited hardware such as the Lattice ICE family of FGPAs. If you have a specific application in which the cutoff isn't going to change and you can pick the sample frequency, this may be the better route. For this project, it is *not* sufficient, thus, we need to investigate another method.

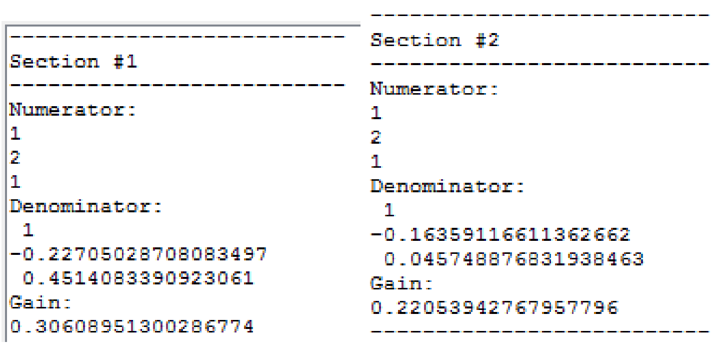
## 3 BiQuad

BiQuads are simple 2nd order filters that can be cascaded together to complete larger order applications. The BiQuad solves the biggest problem found with the single-section design, namely coefficient size. Don't get too excited because it opens up a new can of worms. Granted, these problems are far easier to handle, thus, making the BiQuad a better way to execute IIRs in hardware. Let's break down the BiQuad just as we did the single-section.

### 3.1 Coefficients

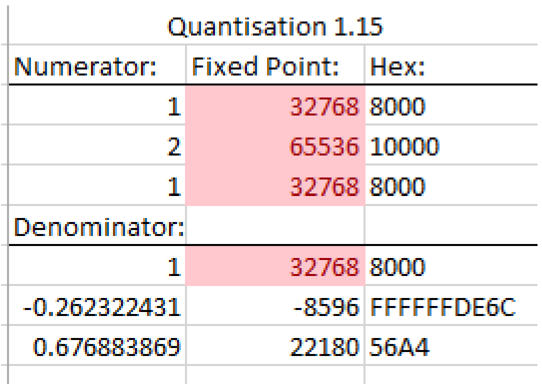
As mentioned, the BiQuad is a 2nd order filter when we look at it atomically. There 2nd order filters get cascaded together, whether in serial or parallel, to complete filters of higher order. That being said, the number of coefficients found in each stage remains the same. This is extremely beneficial in the generic case because we no longer have to modify the IP or internal design of the IIR to change the number of coefficients.[4] If we wish to increase the order of the system, we can simply add in the necessary stages. **Figure 3** is an example of a 4th order filter set of coefficients in BiQuad form.

[4]Only if we went outside of out preset limits, obviously.



```
--------------------------        --------------------------
Section #1                        Section #2
--------------------------        --------------------------
Numerator:                        Numerator:
1                                 1
2                                 2
1                                 1
Denominator:                      Denominator:
 1                                 1
-0.22705028708083497              -0.16359116611362662
 0.4514083390923061                0.045748876831938463
Gain:                             Gain:
0.30608951300286774               0.22053942767957796
                                  --------------------------
```

Figure 3: $F_s = 48\text{kHz}$, $F_c = 10.8\text{kHz}$



| Quantisation 1.15 | | |
|---|---|---|
| Numerator: | Fixed Point: | Hex: |
| 1 | 32768 | 8000 |
| 2 | 65536 | 10000 |
| 1 | 32768 | 8000 |
| Denominator: | | |
| 1 | 32768 | 8000 |
| -0.262322431 | -8596 | FFFFFFDE6C |
| 0.676883869 | 22180 | 56A4 |

Figure 4: Overflowed Coefficients

So, there we have it. The solution to everything. Coefficients look fine, we are done, right? Wrong. These coefficients have 1 major flaw still: they overflow. The numerator coefficients overflow when using our standard quantisation 1.15, therefore, we need to *scale* these coefficients. Scaling will give us coefficients that we can easily convert to fixed point with no issues. Then all we much do is multiply by some gain at the end to get the proper outputs. If you don't believe me about them overflowing, look at **Figure 4**. In that figure we the coefficients, their fixed point value, and that value in hex. You will notice that 1 and 2 *both* overflow.[5]

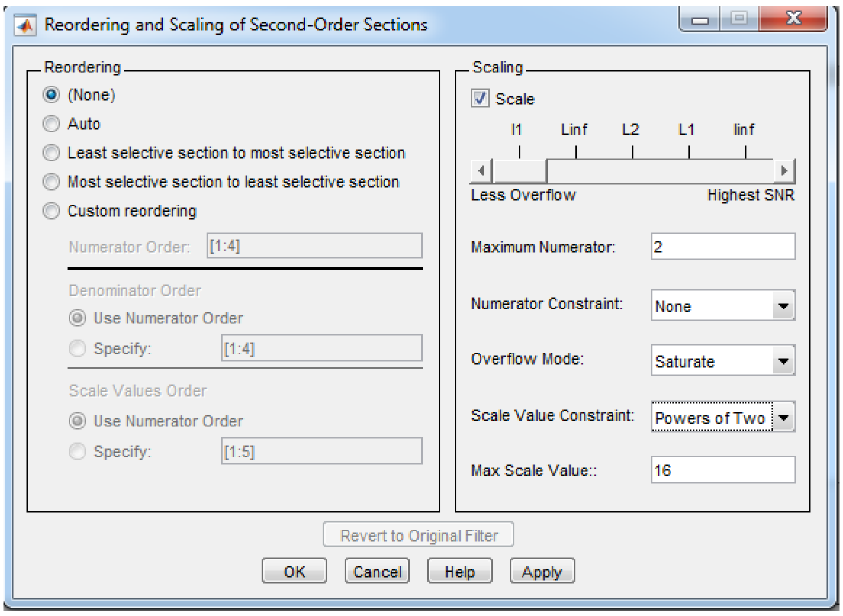
## 3.2 Scaling

Thanks to MATLAB, scaling is quite easy to do. All you must do is go into the `Edit` drop down menu, and select `Reorder and Scale Second-Order Sections ...` and you will see the dialog shown in **Figure 5**.

[5] 1 overflows because the sign bit is high, and 1 is not negative.

When you first open the window, the `scale` check box will be unchecked. Check it. You now have access to a few different scaling options. I recommend that you use the *L2* method. This is because it seems to diminish the coefficients very well yet doesn't have a huge gain at the end. The gain seems to be close to 1 for the most part. This is nice because you don't have to worry about adding in gain multipliers, since they are a waste of hardware and time. I also recommend that you keep `Maximum Numerator` and `Numerator Constraint` at the default. `Overflow Mode` should be set to *Saturate*, `Scale Value Constraint` to *Powers of Two*, and `Max Scale Value` to *16*. Then *apply* the changes.

Once the changes have been applied, you should see something similar to **Figure 6**. Your numerator coefficients should smaller and your gains should have changed to some fraction of denominator that is a power of two.[6] In the figure, I show that the values do not overflow anymore. I also show the big gains at the end. As noted earlier, the L2 scaled coefficients are slightly larger and but have a smaller gain at the end. It's close enough to 1, so you could get away with ignoring it altogether.



Reordering and Scaling of Second-Order Sections

Reordering
- [✓ (None)]
- Auto
- Least selective section to most selective section
- Most selective section to least selective section
- Custom reordering

Numerator Order: [1:4]

Denominator Order
- [✓ Use Numerator Order]
- Specify: [1:4]

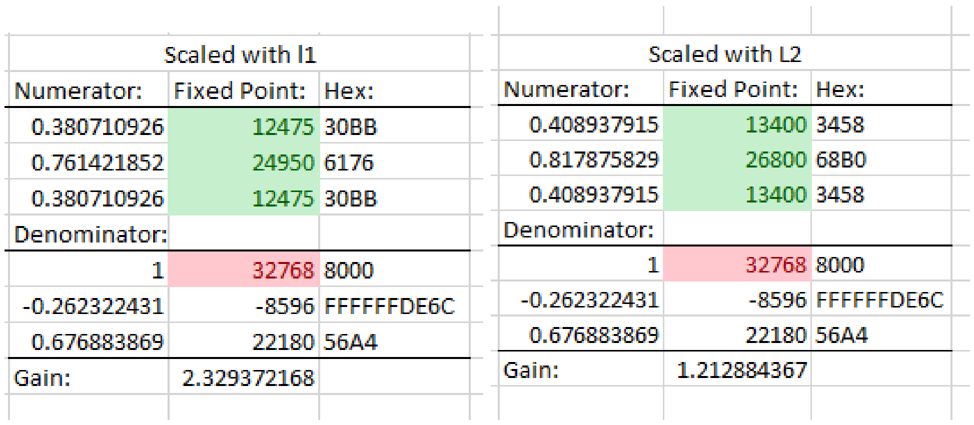
Scale Values Order
- [✓ Use Numerator Order]
- Specify: [1:5]

Scaling
- [✓ Scale]

| | |
|---|---|
| Maximum Numerator: | 2 |
| Numerator Constraint: | None |
| Overflow Mode: | Saturate |
| Scale Value Constraint: | Powers of Two |
| Max Scale Value:: | 16 |

Figure 5: Scaling Window

[6]Sometimes they won't be exact.



| Scaled with l1 | | |
|---|---|---|
| Numerator: | Fixed Point: | Hex: |
| 0.380710926 | 12475 | 30BB |
| 0.761421852 | 24950 | 6176 |
| 0.380710926 | 12475 | 30BB |
| Denominator: | | |
| 1 | 32768 | 8000 |
| -0.262322431 | -8596 | FFFFFFDE6C |
| 0.676883869 | 22180 | 56A4 |
| Gain: | 2.329372168 | |

| Scaled with L2 | | |
|---|---|---|
| Numerator: | Fixed Point: | Hex: |
| 0.408937915 | 13400 | 3458 |
| 0.817875829 | 26800 | 68B0 |
| 0.408937915 | 13400 | 3458 |
| Denominator: | | |
| 1 | 32768 | 8000 |
| -0.262322431 | -8596 | FFFFFFDE6C |
| 0.676883869 | 22180 | 56A4 |
| Gain: | 1.212884367 | |

Figure 6: Scaled Coefficients

## 3.3 Benefits

As we already discussed, the BiQuad is much better than the single-section; but what are some of the major benefits to using it. Firstly, the size of the coefficients are controllable. This allows use to have the opportunity to run any filter we want. Secondly, BiQuads can be made generic. With a little added work, you can create a BiQuad design that is generic enough that you can change the order of the filter, without having to modify the number of stages. All you need is a demux at the end. This demux would select which output you wish to take from. For example, you have a design that has 4 stages (8th order) but you wish to implement a 6th order filter. You don't need to remove that last stage if you have a output selecting demux. You can give coefficients to the first 3 stages and then select the output to come from stage 3, and there you have it! This is extremely handy because it allows us to very easily change the order of the filter without having to mess around with much. I believe that these are the 2 biggest benefits to using the BiQuad method.

# 4 IIR IP

Now that we have most of the major theory out of the way, we can start to discuss the actual IP. I'm just going to go right down the line on this one, skipping over the coefficients input process (see Zynq Communication for that). First and foremost, the IP's exterior layout is down in **Figure 7** below.

Let me explain what each input is and its function. Starting at the top, `i_clk` is the clock

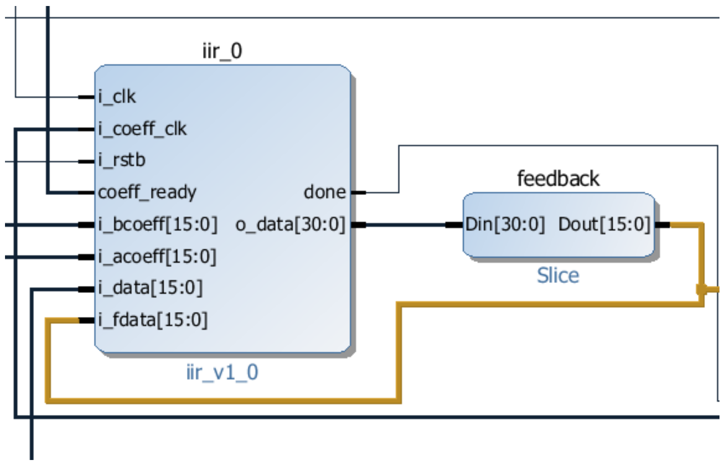


Figure 7: IIR IP in Block Format

for the system. This is hooked directly to the either the sample clock or the end-of-conversion signal from the XADC. `i_coeff_clk` is the clock for shipping in the coefficients from the Zynq. `i_rstb` is the rest for the IP; it is hooked to the system reset. `coeff_ready` is the signal that tells the IP that the Zynq is ready to ship it new coefficients. Think of it as a start signal for the Zynq communication. `i_bcoeff` and `i_acoeff` are the coefficients input signals that are 16 bits width. These are hooked directly to the GPIO that controls that stage. `i_data` is the data from the XADC. I use DRP, so this is taken from the `do_out` signal of the XADC in DRP mode. `i_fdata` is the feedback data. `done` is the done signal. This is used to start the DA2. Lastly, we have `o_data`, which is the output data. Note that this is of width 31 bits. Due to Vivado limitations on IP creation, it wouldn't let me go past 100 in/out bits. I have exactly 100 with `o_data` at 31 bits.

Let us now look inside the IP and see how it works. Due to the number of images, they were put into an appendix. Please see Appendix A: Inside the IP for the screenshots I refer to here. The first thing the IP does (besides getting coefficients) is inputting data. As seen in **Figure 8**, the feedforward and feedback data is pulled directly from the input signals and is anded with there respective internal array. This fancy operation found on lines 78 and 79 are the hardware engineer's quick and dirty way of shifting elements in an array. This pushes the data from indexes 0-2 to 1-3 then shoves the anded value into index 0. Keep in mind that both arrays are the *same size*. This is for symmetry in the design. Normally the feedback array would only need 3 values, however, this throws off the design's symmetry and makes it look bad.

The next 4 processes I will discuss in a single paragraph because they are all related very closely. They are found in **Figures 9 to 12**. These are the arithmetic processes; they do the actual IIR calculation. However, in hardware it is a good idea to follow the rules of *pipelining* when doing calculations. Pipelining allows for higher clock frequencies, reduces synthesis times, and increases throughput for the system. For those reasons, I pipelined the entire design. In **Figure 9**, you will see the multiplication. Notice that it's in a for loop and stores each multiplication into a new register in the respective array. Also note that on the final loop, when $k = 2$, the multiplication for the feedback will always be zero. In the next process, **Figure 10**, we start the additions. Notice that I must resize the registers before the addition. Both of the first stage addition arrays are 1 bit wider than the multiplication arrays. This will protect us against the overflow that may occur. The same thing is completed in the final addition stage, **Figure 11**. Finally, we subtract the feedforward data by the feedback data in **Figure 12**.

The final process in the design is the data output process found in **Figure 13**. This one simply sets the `done` signal high and outputs the data. Notice that I use cast the `r_final_sum` register into an `std_logic_vector` because all the arithmetic is done in *signed* and you cannot output a signed value.

## 5 Zynq Communication

The last piece of the puzzle is how the Zynq gets incorporated into the system. This is fairly simple because all it does is use GPIOs and a clock! Once again, due to the large number of images, they are found in Appendix B: Communication Protocol. In **Figure 14**, you will see how the GPIOs hook up in the block design. There isn't really anything special about this, so there isn't much to say. Next in the queue is **Figure 15**, which is the process within the IIR IP that handles the coefficient accepting. Notice that it uses the `coeff_ready` signal to reset the coefficient registers and the looping variable. It then uses the rising edge of `i_coeff_clk` to say when to grab the next value. Lastly, **Figure 16** shows the code from the Zynq which is very simple. The arrays, `b_coeff` and `a_coeff`, contain the coefficient that you wish to upload to the IP. Earlier in the program, the `coeff_ready` signal was set high. Then, within this loop the `i_coeff_clk` signal is toggled and the GPIO values are changed. This is how simple the communication is.

## 6 Example Outputs

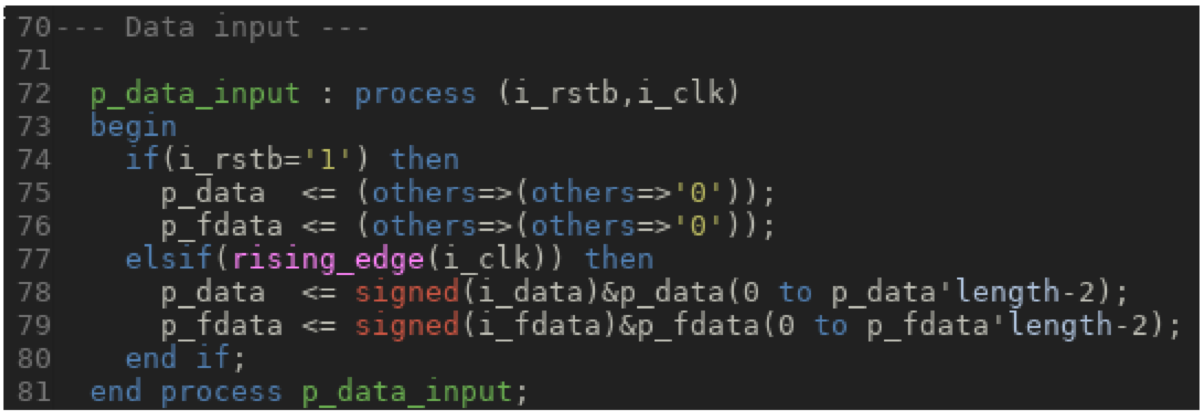
Found in Appendix C: Examples will be some example outputs of the system discussed in this report. In **Figure 17**, you will see the attempt at a lowpass filter. First thing you'll probably notice is that the amplitude is off. We are about 30dB too low. Other than that, this filter actually preforms as intended. It is dead at around 10kHz, which is what it should be. The next example isn't as good, as shown in **Figure 18**. This time the first thing that you will notice is that the passing section is very unstable. However, it does "pass" the correct frequencies. Although it isn't pretty, it still works to an extent. The final example, shown in **Figure 19**, is a complete mess. It is an attempt at a bandpass and, as you will see, isn't a very good one. It has a passing region but it's very unstable and isn't the correct frequency range. It's off by about 3kHz. In the end, these examples are not great but do so that the system *works* just very poorly.

## 7 Conclusion

In the end, the project fell flat and didn't work as intended. This isn't, however, a bad thing. Along with the struggles of the project, I learn a lot about IIRs and hardware design in general. This information may be useful later in my career. There are many reasons as to why the project didn't work. Some of the problems may have been the following: Not multiplying by the gain, no FSM in the IP, and not accurate data/coefficient. Not multiplying by the gain could be the reason the amplitude was so low. However, I don't believe this is entirely what caused the problem due to the fact all the gains where close to 1 (somewhere between .8 and 1.3). Even if we multiplied by these gains, it would only vary the amplitude slightly. Not having a FSM in the IP could be one of the things that caused it's inconsistency. Without an FSM, we don't have a way to flow control the process. Instead, I rely on sensitivity lists to move to the next process. It would have probably been a better idea to use that, but with an FSM that would only allow certain things to happen once a blocker register was set or reset. Lastly, there is a surprising amount of inaccuracy in using 16 bit coefficients. Some of the more high quality designs I saw online used 32 bit coefficients. This may or may not have had something to do with the inconsistency of the output.

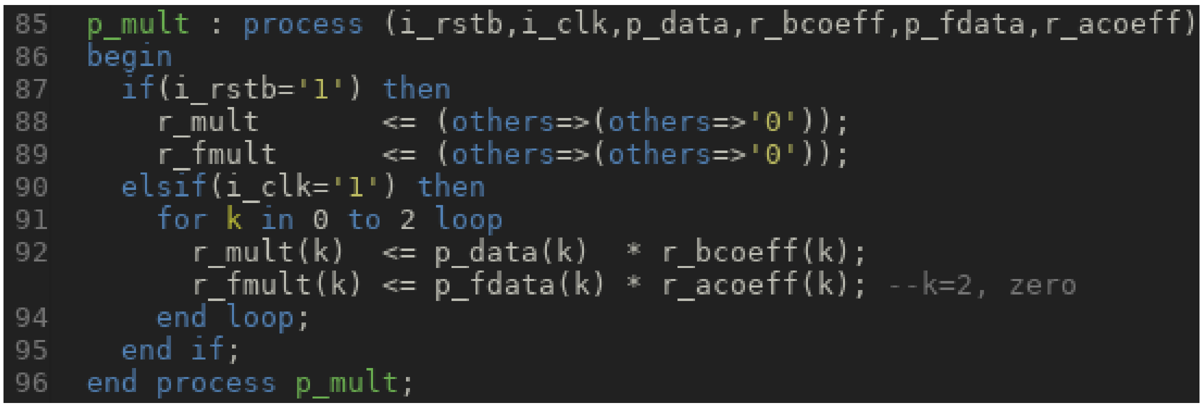
When it comes down to it, the IIR is not something that is favorable to do hardware. There are a lot of areas that can cause issues due to the complex nature of the calculation. Just the arithmetic itself is difficult in hardware because of overflowing and scaling. On top of this, attempting to make it generic is something that seems reasonable but may not be easily completed. Because of all the issues and complexities of the IIR, I believe that this is an application for a microcontroller or processor. On those systems, we don't have *any* of these issues. We can use floats or doubles for all the data, eliminating all the overflow, scaling, and coefficients related problems. In conclusion, I believe that the IIR is *not* a good application for a FPGA but an application for a microcontroller instead.

## A Inside the IP



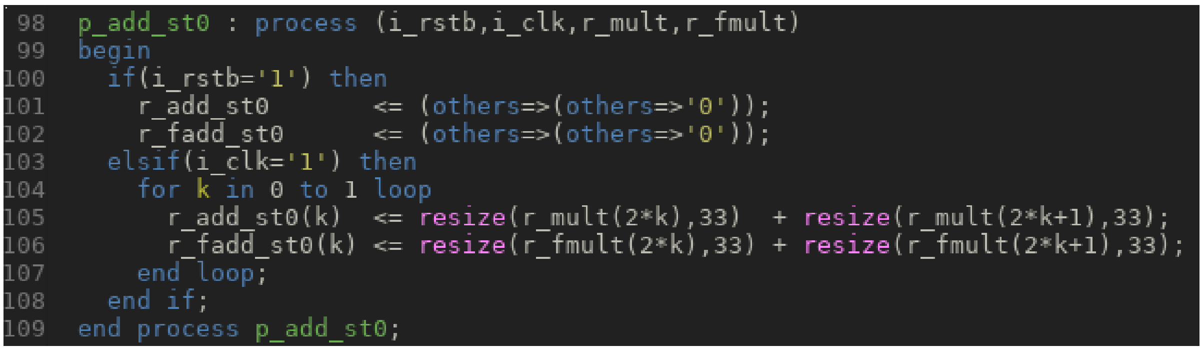
```
--- Data input ---

 p_data_input : process (i_rstb,i_clk)
 begin
   if(i_rstb='1') then
     p_data  <= (others=>(others=>'0'));
     p_fdata <= (others=>(others=>'0'));
   elsif(rising_edge(i_clk)) then
     p_data  <= signed(i_data)&p_data(0 to p_data'length-2);
     p_fdata <= signed(i_fdata)&p_fdata(0 to p_fdata'length-2);
   end if;
 end process p_data_input;
```

Figure 8: Data Input Process



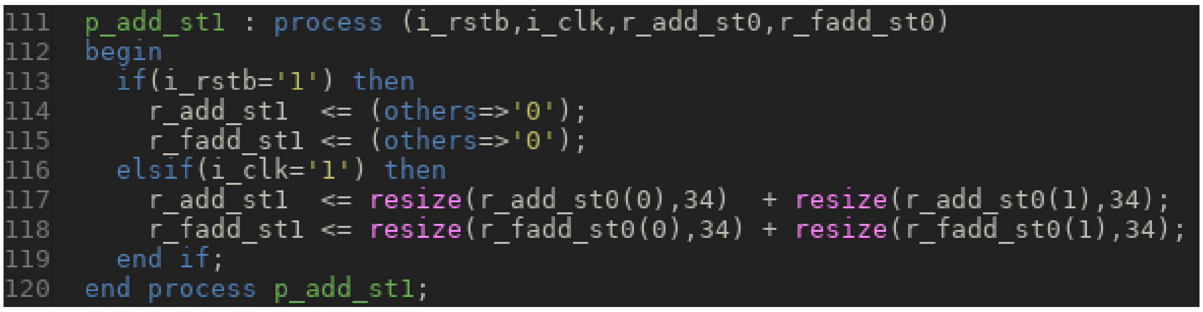
```
 p_mult : process (i_rstb,i_clk,p_data,r_bcoeff,p_fdata,r_acoeff)
 begin
   if(i_rstb='1') then
     r_mult       <= (others=>(others=>'0'));
     r_fmult      <= (others=>(others=>'0'));
   elsif(i_clk='1') then
     for k in 0 to 2 loop
       r_mult(k)  <= p_data(k)  * r_bcoeff(k);
       r_fmult(k) <= p_fdata(k) * r_acoeff(k); --k=2, zero
     end loop;
   end if;
 end process p_mult;
```

Figure 9: Multiplication Process



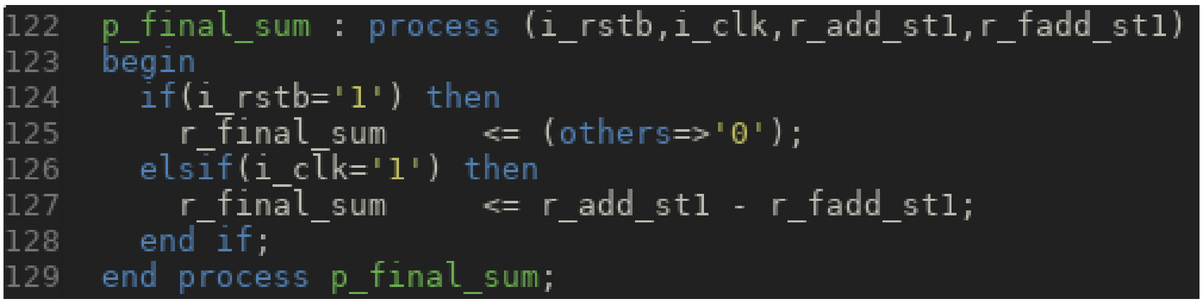
```
 p_add_st0 : process (i_rstb,i_clk,r_mult,r_fmult)
 begin
   if(i_rstb='1') then
     r_add_st0      <= (others=>(others=>'0'));
     r_fadd_st0     <= (others=>(others=>'0'));
   elsif(i_clk='1') then
     for k in 0 to 1 loop
       r_add_st0(k)  <= resize(r_mult(2*k),33)  + resize(r_mult(2*k+1),33);
       r_fadd_st0(k) <= resize(r_fmult(2*k),33) + resize(r_fmult(2*k+1),33);
     end loop;
   end if;
 end process p_add_st0;
```

Figure 10: First Addition Process



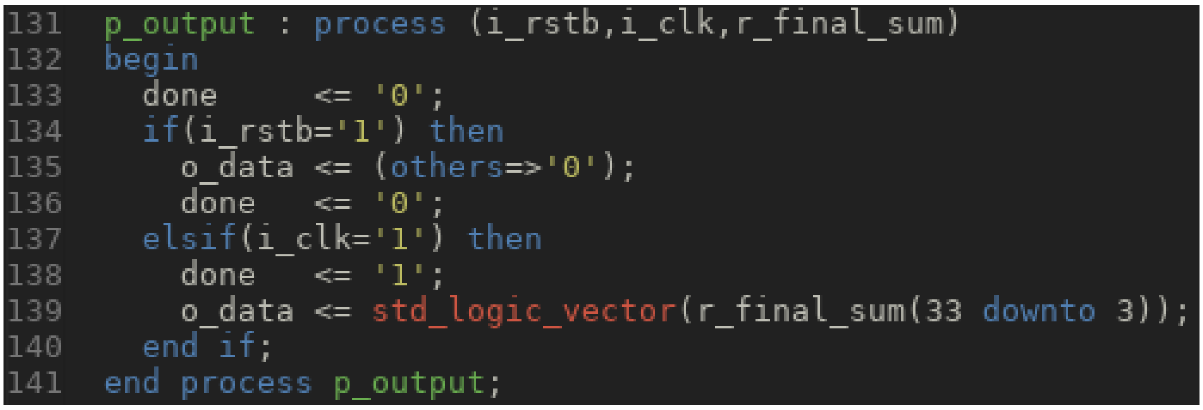
```
p_add_st1 : process (i_rstb,i_clk,r_add_st0,r_fadd_st0)
begin
  if(i_rstb='1') then
    r_add_st1  <= (others=>'0');
    r_fadd_st1 <= (others=>'0');
  elsif(i_clk='1') then
    r_add_st1  <= resize(r_add_st0(0),34)  + resize(r_add_st0(1),34);
    r_fadd_st1 <= resize(r_fadd_st0(0),34) + resize(r_fadd_st0(1),34);
  end if;
end process p_add_st1;
```

Figure 11: Second Addition Process



```
p_final_sum : process (i_rstb,i_clk,r_add_st1,r_fadd_st1)
begin
  if(i_rstb='1') then
    r_final_sum     <= (others=>'0');
  elsif(i_clk='1') then
    r_final_sum     <= r_add_st1 - r_fadd_st1;
  end if;
end process p_final_sum;
```

Figure 12: Feedfoward Feedback Subtraction Process



```
p_output : process (i_rstb,i_clk,r_final_sum)
begin
  done     <= '0';
  if(i_rstb='1') then
    o_data <= (others=>'0');
    done   <= '0';
  elsif(i_clk='1') then
    done   <= '1';
    o_data <= std_logic_vector(r_final_sum(33 downto 3));
  end if;
end process p_output;
```

Figure 13: Data Output Process

## B Communication Protocol

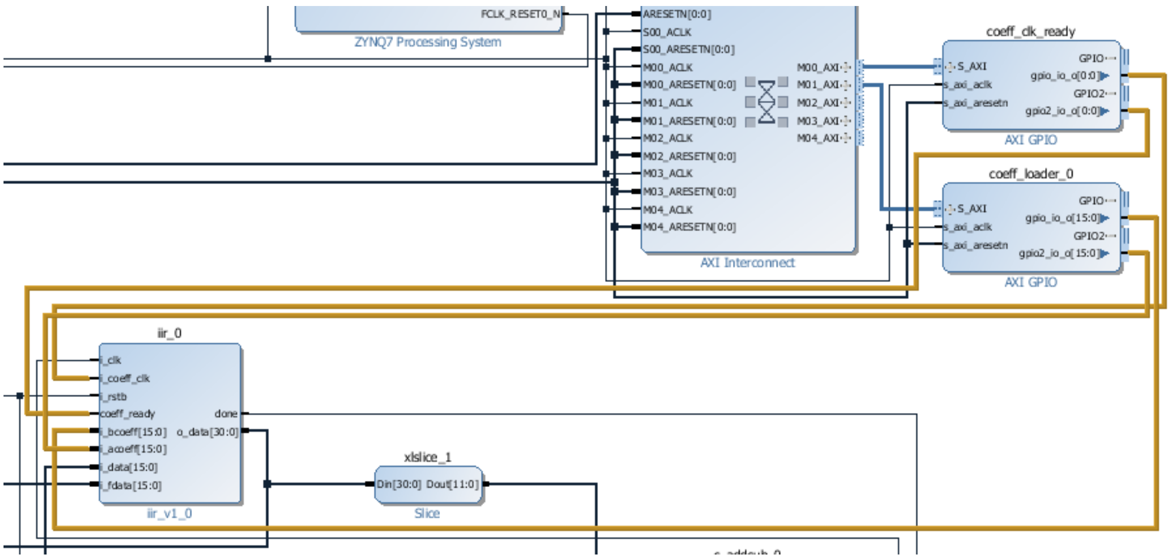


Figure 14: GPIO Hook Ups

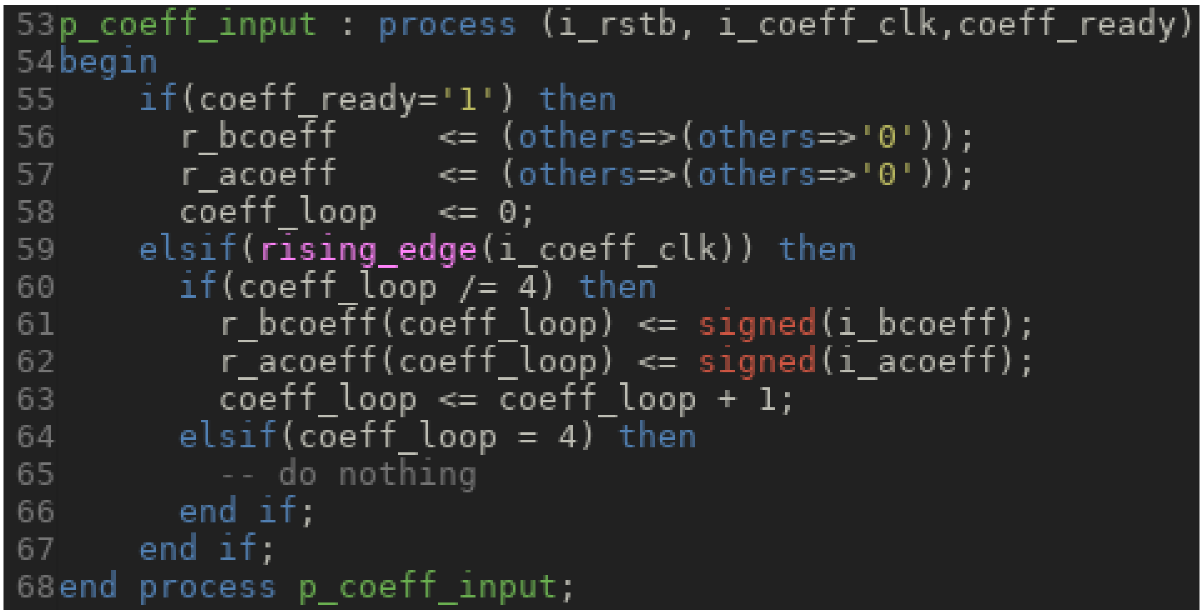

```vhdl
p_coeff_input : process (i_rstb, i_coeff_clk,coeff_ready)
begin
    if(coeff_ready='1') then
      r_bcoeff      <= (others=>(others=>'0'));
      r_acoeff      <= (others=>(others=>'0'));
      coeff_loop    <= 0;
    elsif(rising_edge(i_coeff_clk)) then
      if(coeff_loop /= 4) then
        r_bcoeff(coeff_loop) <= signed(i_bcoeff);
        r_acoeff(coeff_loop) <= signed(i_acoeff);
        coeff_loop <= coeff_loop + 1;
      elsif(coeff_loop = 4) then
        -- do nothing
      end if;
    end if;
end process p_coeff_input;
```

Figure 15: Coefficient Input Process

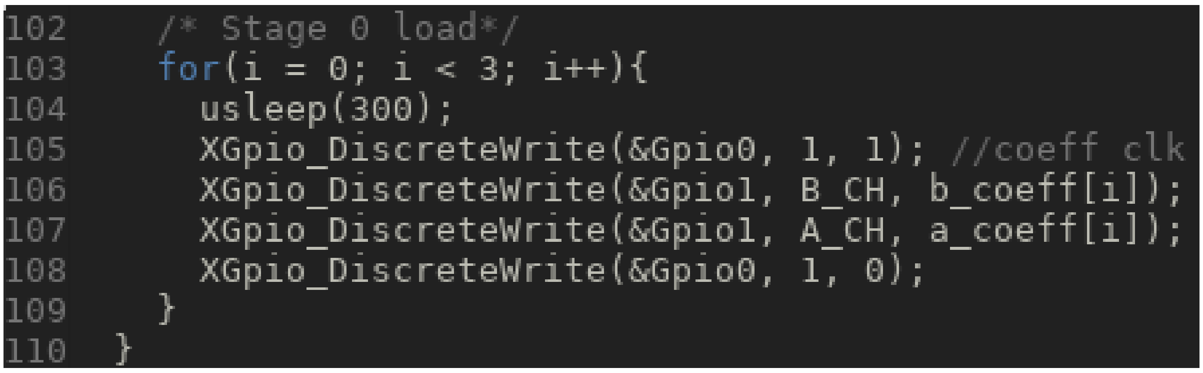

```c
    /* Stage 0 load*/
    for(i = 0; i < 3; i++){
      usleep(300);
      XGpio_DiscreteWrite(&Gpio0, 1, 1); //coeff clk
      XGpio_DiscreteWrite(&Gpio1, B_CH, b_coeff[i]);
      XGpio_DiscreteWrite(&Gpio1, A_CH, a_coeff[i]);
      XGpio_DiscreteWrite(&Gpio0, 1, 0);
    }
  }
```

Figure 16: Coefficient Output Loop

## C Examples

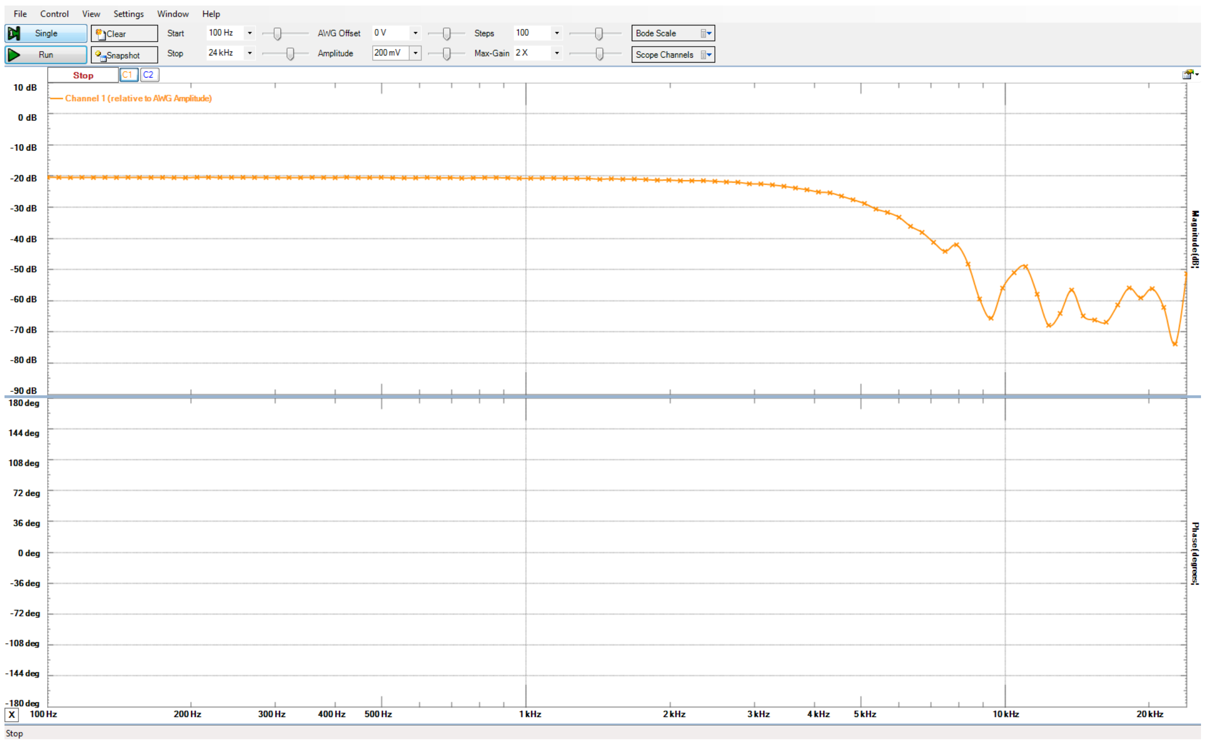

Start 100 Hz
Stop 24 kHz
AWG Offset 0 V
Amplitude 200 mV
Steps 100
Max-Gain 2 X
Bode Scale
Scope Channels
Channel 1 (relative to AWG Amplitude)

Figure 17: $F_s = 48\text{kHz}$, $F_c = 10.8\text{kHz}$

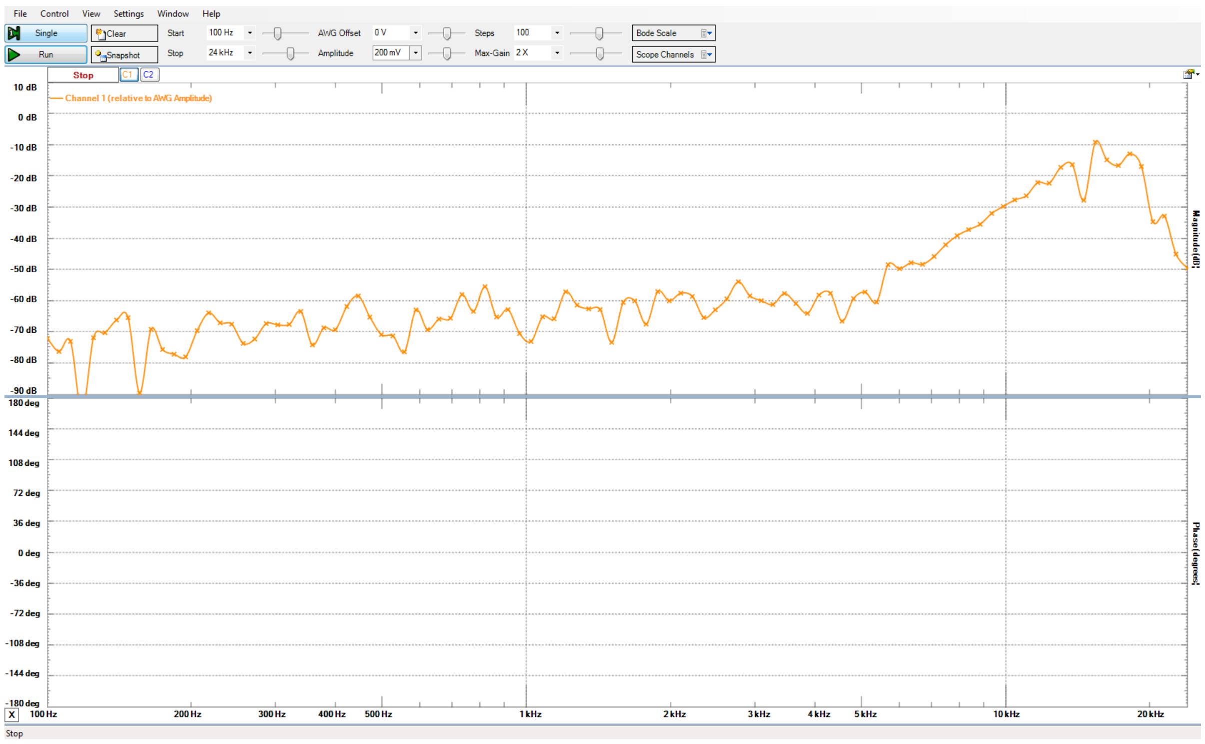

Start 100 Hz
Stop 24 kHz
AWG Offset 0 V
Amplitude 200 mV
Steps 100
Max-Gain 2 X
Bode Scale
Scope Channels
Channel 1 (relative to AWG Amplitude)

Figure 18: $F_s = 48\text{kHz}$, $F_c = 10.8\text{kHz}$

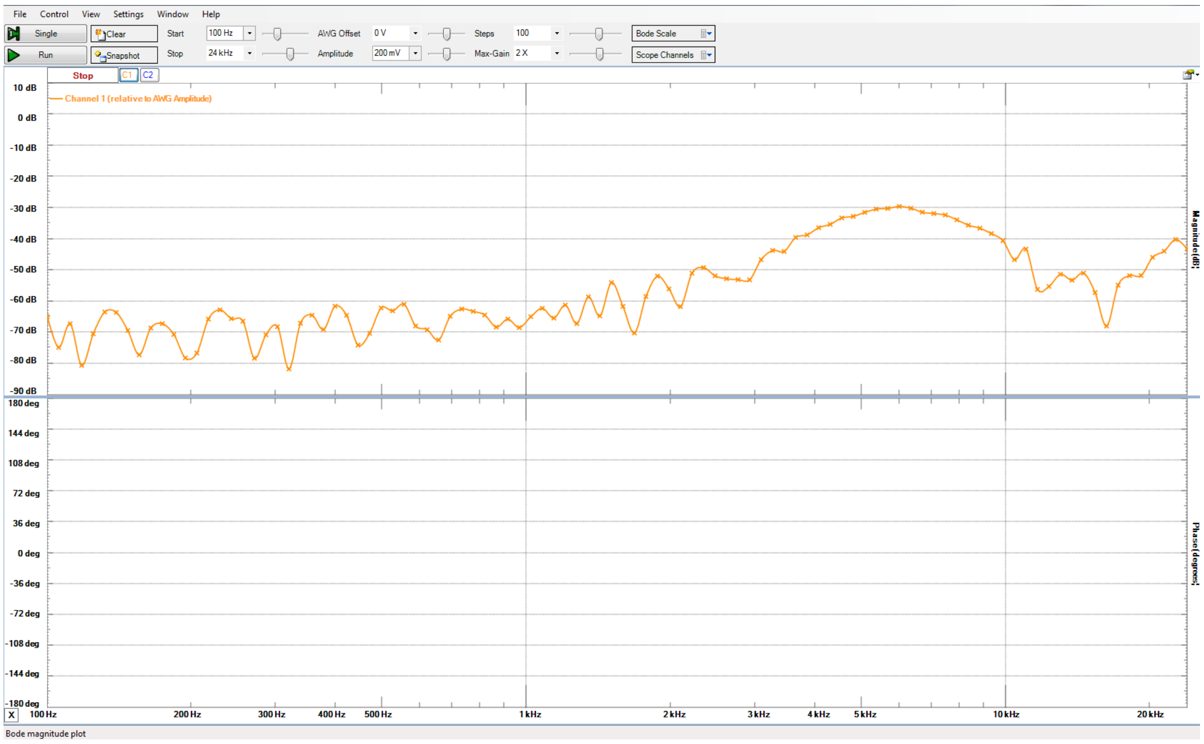

Start 100 Hz
Stop 24 kHz
AWG Offset 0 V
Amplitude 200 mV
Steps 100
Max-Gain 2 X

Figure 19: $F_s = 48\text{kHz}$, Pass band = 8.4kHz to 13.2kHz