ECE 498NSU Fall 2020

MiniProject: 3-tap Linear Predictor in Systems Verilog

Abhi Kamboj

Table of Contents:

Introduction:

In this project, Python and Systems Verilog are used to investigate and prototype a 3-tap linear predictor for the following data generation model:

$$x_n = 0.1g_n + 0.5g_{n-1} - 0.5g_{n-2} + 0.1g_{n-3}$$

Where g_n are i.i.d $\mathcal{N}(0,1)$ random variables. The 3-tap linear predictor implemented, predicts x_n from it's 3 previous samples x_{n-1}, x_{n-2}, x_{n-3}. The prediction function takes the following form:

$$\hat{x}_n = w_1 x_{n-1} + w_2 x_{n-2} + w_3 x_{n-3}$$

Where the Mean Squared Error (MSE) cost function shown here is minimized:

$$\mathbb{E}\left[e_n^2\right] = \mathbb{E}\left[(\hat{x}_n - x_n)^2\right]$$

The software prototyping was used to first determine the optimal weights, and then determine the fixed-point minimum bit precision needed to remain within .5 dB of the true output value using floating-point. The hardware prototyping was used to design a hardware system composed of adders, multipliers and registers that implement the quantized inference of this 3-tap system. This hardware design was then simulated to determine if it functions as desired using ModelSim. Finally it was synthesized and the resource utilization, power consumption, and timing were analzed.

Software:

All the software code was implemented in python on the jupyter notebook (see appendix). First the data generation model was used to generate 1 million data samples. The Weiner-Hopf method was then used to determine the optimal weights and the optimal Mean Squared Error (MSE) to be the following:

Optimal Weights: [-0.67938811 -0.41286511 -0.17928317]

MSE: 0.35208181039207065

Finally, the software was used to determine the optimal bit precisions such that the floating-point output Signal to Noise Ratio (SNR), what is labeled optimal_snr in the code, is within .5db of the fixed-point SNR which would include the quantization error, which is referred to as sqnr in the code. The following equations were used:

$$SNR = \frac{\sigma_y^2}{MSE}$$

$$SQNR = \frac{\sigma_y^2}{\sigma^2(x - Q[x]) + MSE}$$

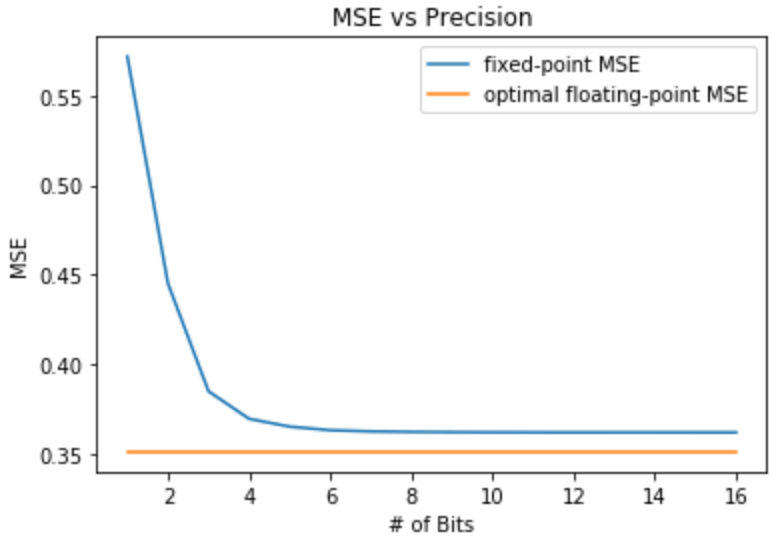
Where y is the output, x is the input and Q is that quantization function.

A “predict” function was created that takes in weights and data, and runs the weights through the 1 million data points predicting the outputs and returning the MSE. A for loop was used to quantize the data and weights to bit precisions 1 through 16, call this prediction function and determine each MSE and SQNR. Here is the table created from this code:

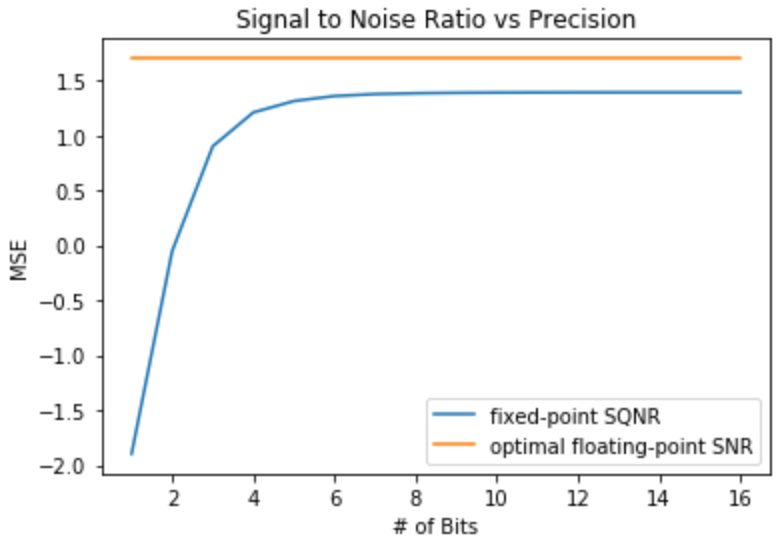
```
Optimal MSE: 0.3513533155728955
Optimal SNR (dB): 1.698789959975315
Bits     | MSE    | SQNR   |SQNR-SNR|       Within .5dB of SNR?
1.0000   |0.5718 |-1.896  |3.5949|         NO
2.0000   |0.4453 |-0.053  |1.7515|         NO
3.0000   |0.3851 |0.900   |0.7987|         NO
4.0000   |0.3698 |1.208   |0.4906|         YES
5.0000   |0.3654 |1.312   |0.3870|         YES
6.0000   |0.3634 |1.357   |0.3417|         YES
7.0000   |0.3627 |1.375   |0.3239|         YES
8.0000   |0.3625 |1.383   |0.3162|         YES
9.0000   |0.3623 |1.387   |0.3121|         YES
10.0000  |0.3623 |1.389   |0.3101|         YES
11.0000  |0.3622 |1.390   |0.3091|         YES
12.0000  |0.3622 |1.390   |0.3085|         YES
13.0000  |0.3622 |1.391   |0.3082|         YES
14.0000  |0.3622 |1.391   |0.3081|         YES
15.0000  |0.3622 |1.391   |0.3081|         YES
16.0000  |0.3622 |1.391   |0.3080|         YES
```

As we can see from the results, at bit precision of 4 bits, the fixed-point SQNR is .49 dB away from the true value, so that is the smallest precision when the output is within .5 dB away from the true value when considering the SNR due to MSE. At this point we can see that the MSE is (.37-.35)/.35 * 100 = 5.7% away from the true MSE value!

To further visualize these results two graphs were plotted. The first shows the MSE values as a function of bit precision:



The second shows the SQNR values as a function of bit precision:

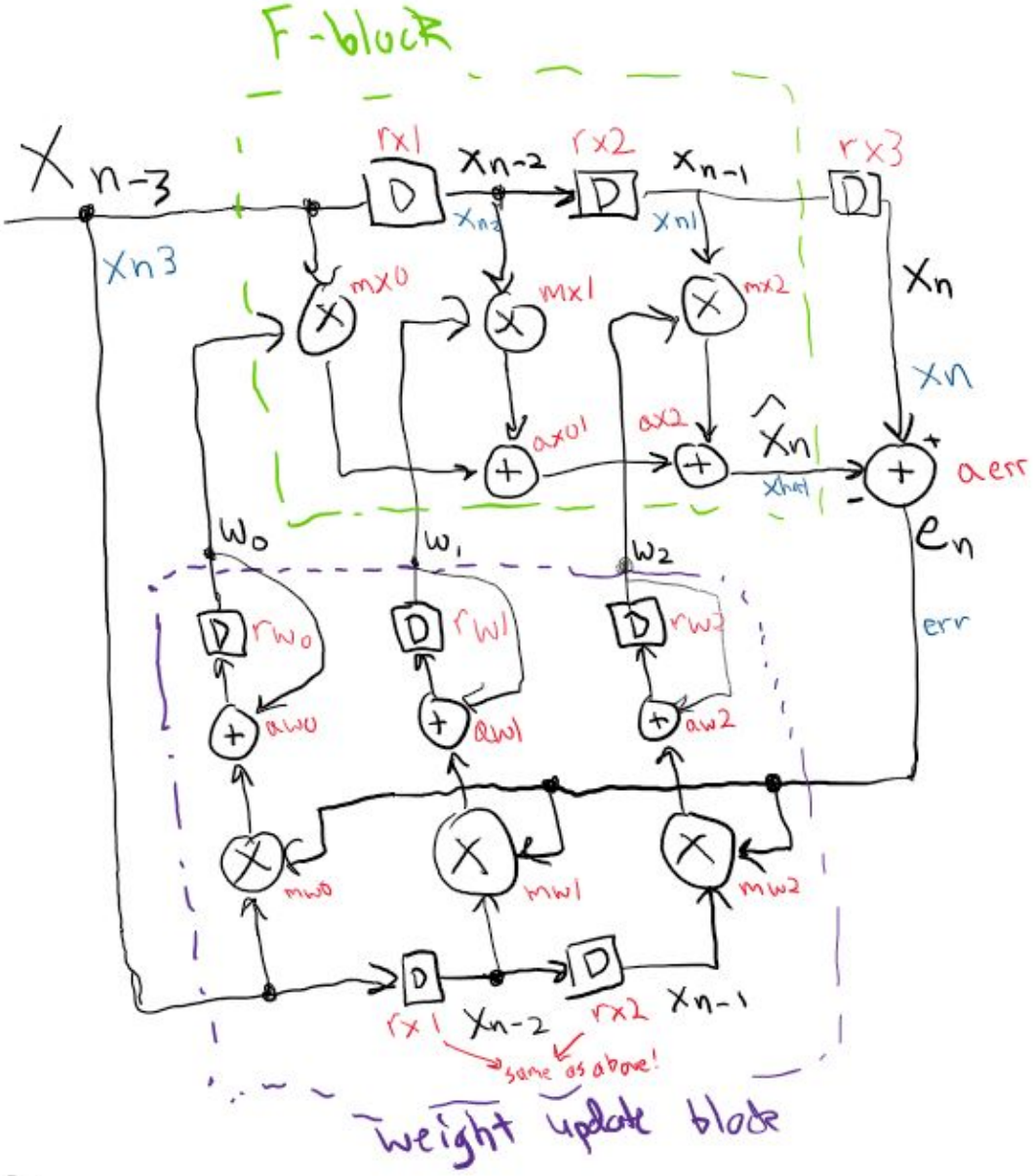


Hardware

The hardware section was done in Systems Verilog code. A 3-Tap predictor system and a corresponding testbench was created. ModelSim was used to write and simulate the testbench, and Vivado was used to synthesize and implement the code.

Architecture Design:

The design was created by analyzing the 3-Tap Linear Predictor architecture drawn for Homework 2 part 6. This design provided a baseline for the Systems Verilog implementation and the diagram was to include the names of signals and modules that were implemented. The red color indicates an instantiated component and the blue color indicates the name for an internal signal or wire. For example the rx1 stands for the first data register, mx0 stands for the first data multiplier, etc. The sketch of the architecture is shown below:



D is a one-delay (register)

module instantiations

wires/internal signals

I ended up implementing the entire architecture, however, for this Mini Project I only tested and used the filter-block (F-block) portion boxed off in green.

Precision:

As determined in the software prototyping, the precision which still allows for the output to be within .5dB of the optimal SNR, was 4 bits. Therefore, all my input data and weights are 4 bits. However, to deal with bit growth in the multipliers I implemented the architecture of the F-block using 8 bits. While I was unit testing, I further noticed some errors due to overflow with the addition since one bit was being used for sign. To accommodate this, I increased the architecture by one bit, and these errors went away. All the Adders, Multipliers and Registers used by the predictor are thus 9 bits.

Modules:

In order to build this architecture I created separate modules, and then created a top level predictor module that combines them.
Predictor: This module instantiates 3 multipliers, 3 registers and 3 adders and connects them according to the architecture above.
Arithmetic Modules: The add module outputs the sum of two inputs, and the multiply module outputs their product.
Register: This is a positive edge triggered synchronous register. I implemented it with a load and clear signal as well.

Simulation:

I wrote a test bench performing an inference on the first 3 values generated by my data generation model. These values were: {0.19778589, 0.55186864, 0.73500254, -0.69549231}. My testbench uses my predictor module to predict the last data point from the first 3 data points.

Here is the testbench I wrote:

```
module testbench();
timeunit 10ns;
timeprecision 1ns;

logic reset, Clk;
logic [8:0] xn3, w0, w1, w2;;
logic [8:0] x_hat, err;

always begin : CLOCK_GENERATION

 #1 Clk = ~Clk;

end

initial begin : CLOCK_INITIALIZATION
        Clk = 0;
end

predictor p(.*);

parameter [3:0] data [4] = {0.19778589 * 16, 0.55186864 * 16, 0.73500254 * 16, -0.69549231 * 16};

//x_hat should be = -0.69549231
real result;
assign result = x_hat/16/16;

initial begin : TEST_VECTORS
#2 reset = 0;
#2 reset = 1;
#2 reset = 0;

//set weights: [-0.67776111 -0.41394865 -0.17928979]
#2 w0 = -0.67776111*16; w1=-0.41394865*16; w2=-0.17928979*16;

#2 xn3 = data[3]; //xn is the one at the end and xn-1 is the one before that, etc
#2 xn3 = data[2];
#2 xn3 = data[1];
#2 xn3 = data[0];

end

endmodule
```

Recall that I determined a minimal precision of 4 bits, however Verilog doesn't automatically store floating point decimals, so I multiplied each decimal by 16 before storing it in the bits. As a result, the simulation waveform shows integers. The values that are being passed are the rounded results:

0.19778589 * 16 = 3

0.55186864 * 16 = 9

0.73500254 * 16 = 12

-0.69549231 * 16 = 5.

These integers are the 4-bit quantized representation of the data points. Similarly for the weights we have:

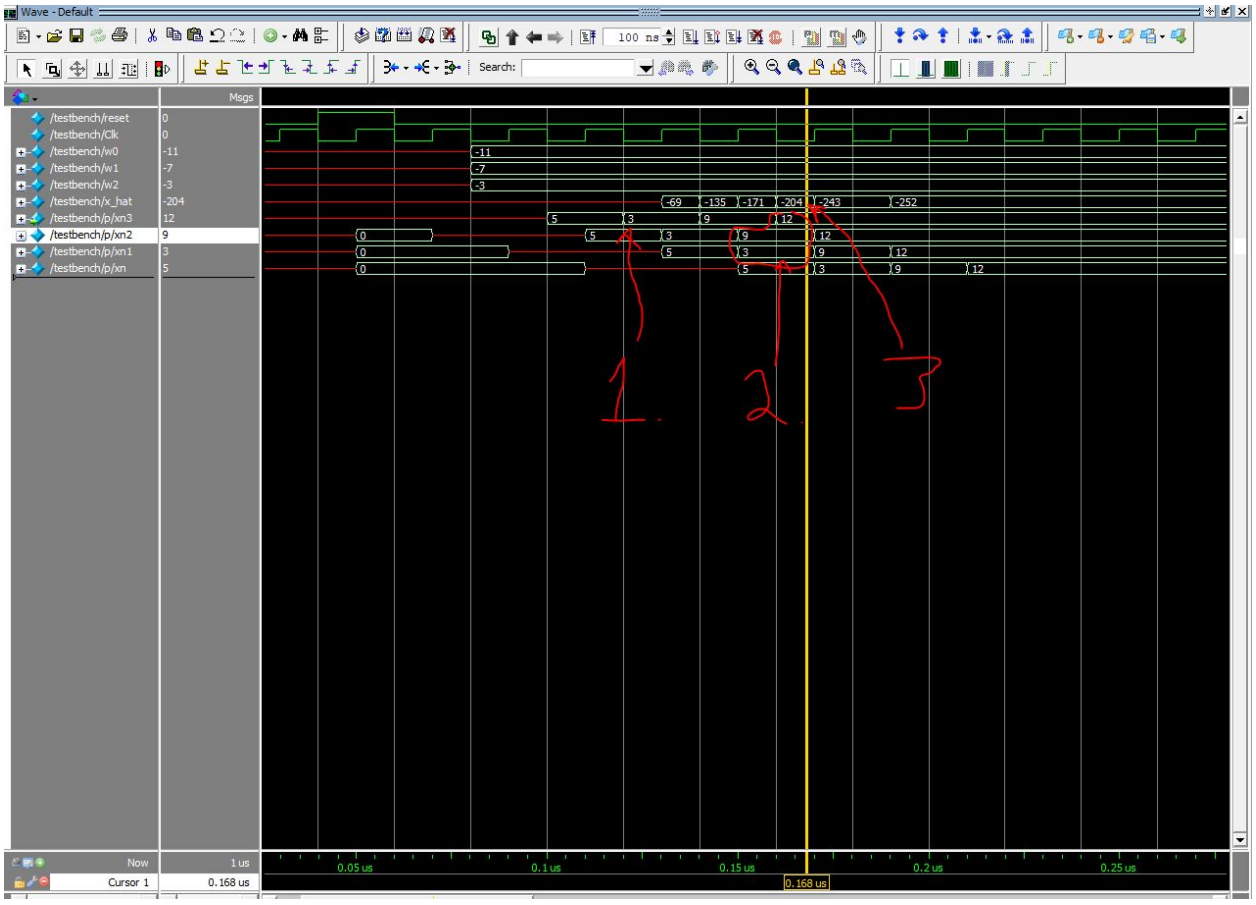
w0 = -0.67776111*16 = -11

w1=-0.41394865*16 = -7

w2=-0.17928979*16 = -3

Below is the waveform of the simulation:

1. As we can see in the simulation below, the values 3, 9, 12 and 5 are shifted into the registers in the couple of clock cycles.
2. At the .168 us when the xn3=12, xn2=9, xn1=12 and the system can perform the dot product.
3. Then when the clock is low, the combinational logic (the adders and multipliers) will instantaneously put the next prediction on x_hat. So the system computes:
    a. $\hat{x} = (x_{n-3}, x_{n-2}, x_{n-1}) \cdot (w_0, w_1, w_2)$
    $= (12 \times -11) + (9 \times -7) + (3 \times -3) = -204$
    b. This is the result shown in x_hat



Simulation Results:

In base 2 notation, when multiplying by 16, I am simply shifting the first four bits after the decimal point to be before the decimal point. To transfer this result back to decimal notation we can divide by 16^2 = 256, since we multiplied both the weights and inputs by 16, before running the inference. Thus the inference result is -204/256 = -0.797. The actual next data point is -0.695, so the MSE error is (-0.797 - -0.695)^2 = 0.01. This is a unique case in which the estimate is very close to the true result, however, this is not a good measure since only one inference is being measured. Instead, I decided to compare the inference result to the inference result of a floating point inference. A small python code was hardcoded and written to confirm this result:

```
x_dptest = np.array([0.73500254,  0.55186864,  0.19778589])
w_dptest = np.array([-0.68014248, -0.41424331, -0.17878953])
print("Floating point dot-product:", x_dptest.dot(w_dptest))
```

```
Floating point dot-product: -0.7638763887944293
```

,

With floating point weights and inputs, the optimal inference is actually -0.764. Thus the percent difference between my hardware implementation and the optimal floating point inference is only:

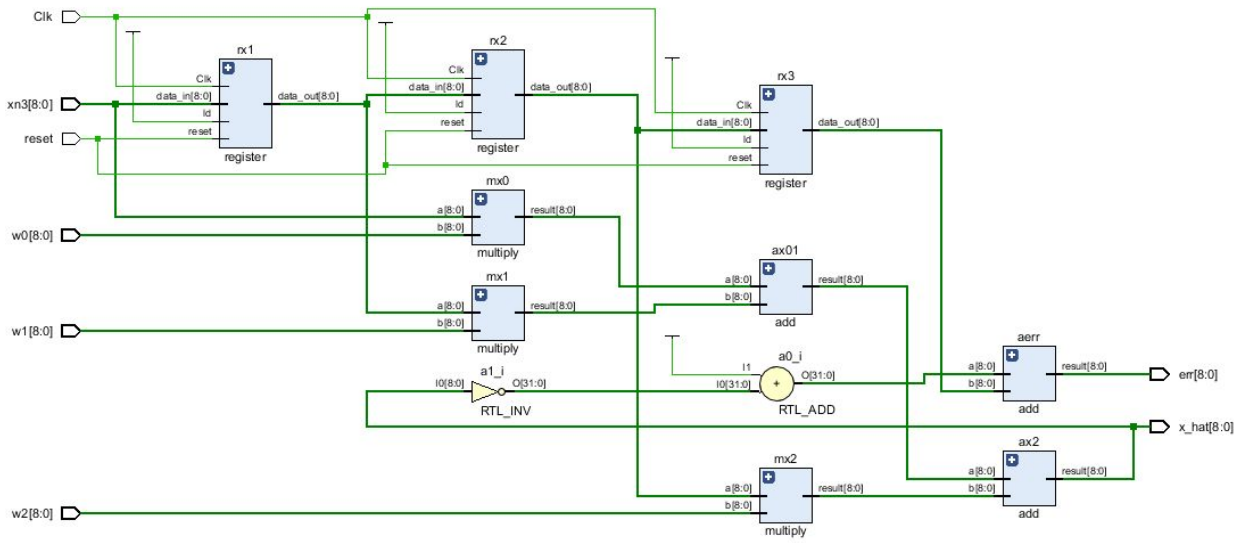
$$Percentdifference = \frac{|(-0.764 - -0.797)|}{|-0.764|} \times 100 = 4.32\%$$

This is surprisingly small, but again could be because only one data point is being measured. Ideally, to get a better picture of the architecture I could perform 1 million of these inferences and average them, as I did very easily in the python code, but that is much more difficult to implement given the resources on the FPGA. Nonetheless, this inference demonstrates proof of concept of the architecture. It clearly shows that the system can successfully perform a dot product and can therefore perform an inference when quantized to 4 bits.

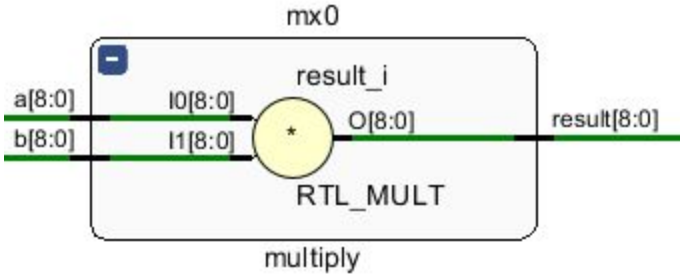
Schematic:

After testing in ModelSim, the design was synthesized and implemented in Vivado. The following schematic diagrams show the result. The schematic diagrams are very similar to my hand drawn architecture, but they are more thorough and detailed. Vivado synthesized a few different schematics.

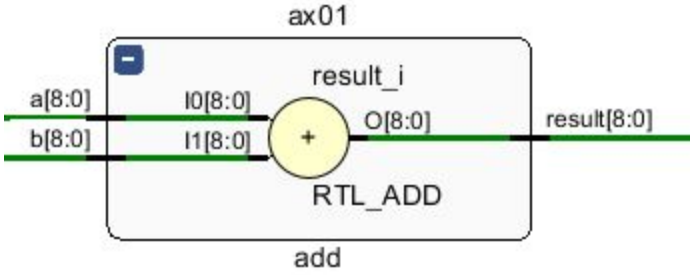
RTL Schematic



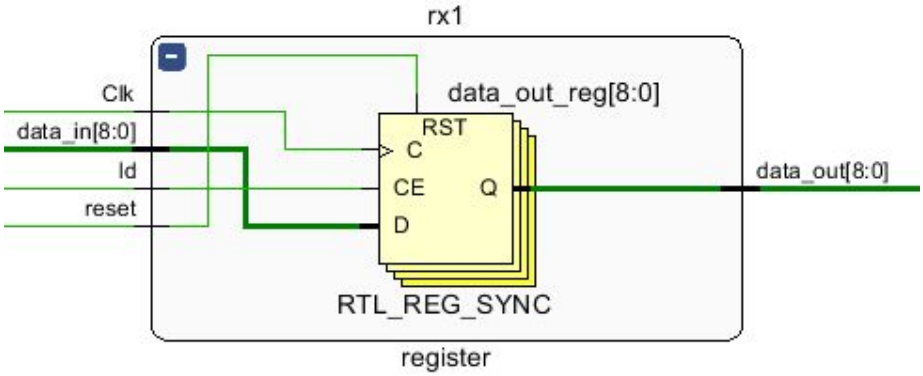
Inside the multiply module:



Inside the add module:

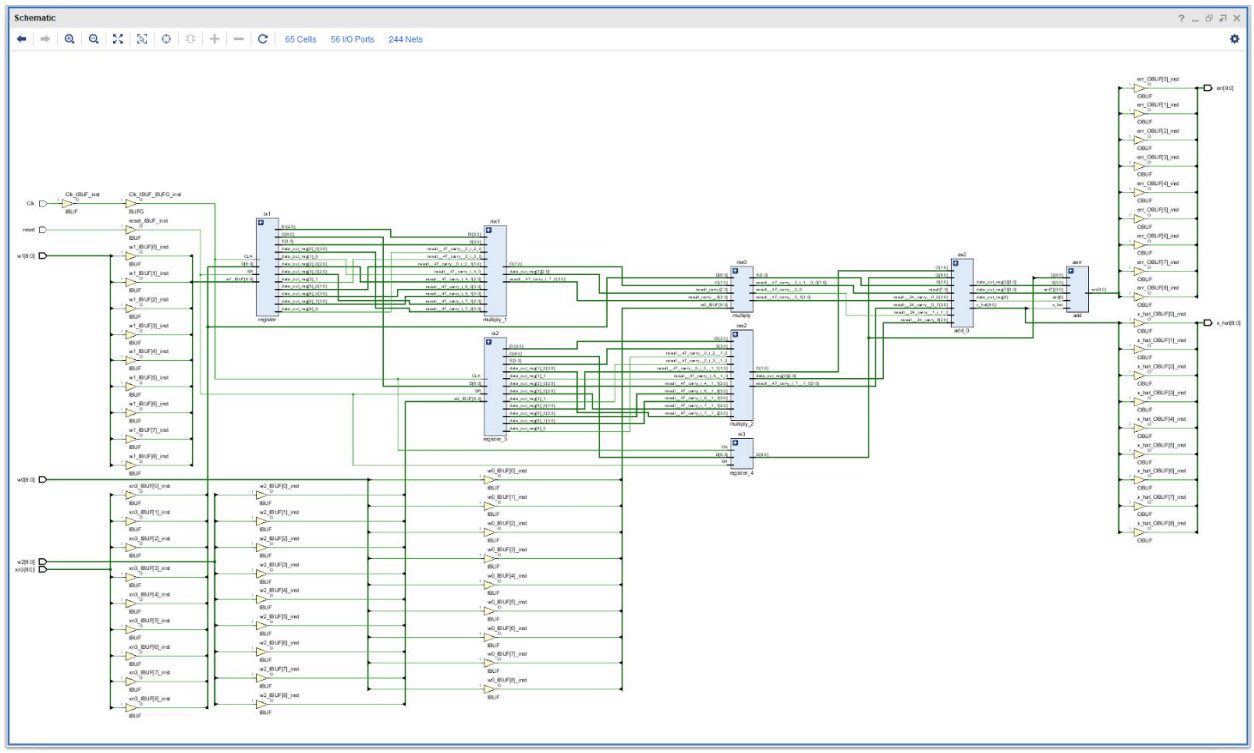


Inside the register module:



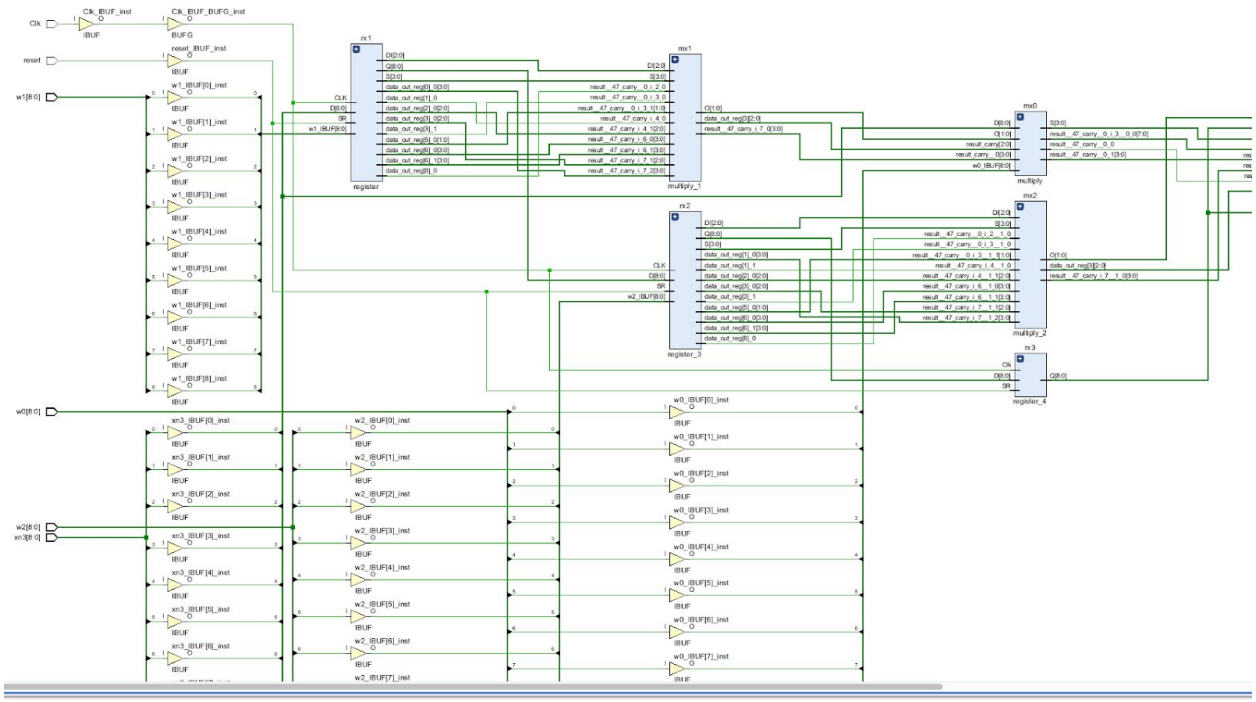
Synthesized Schematics:

Overall:



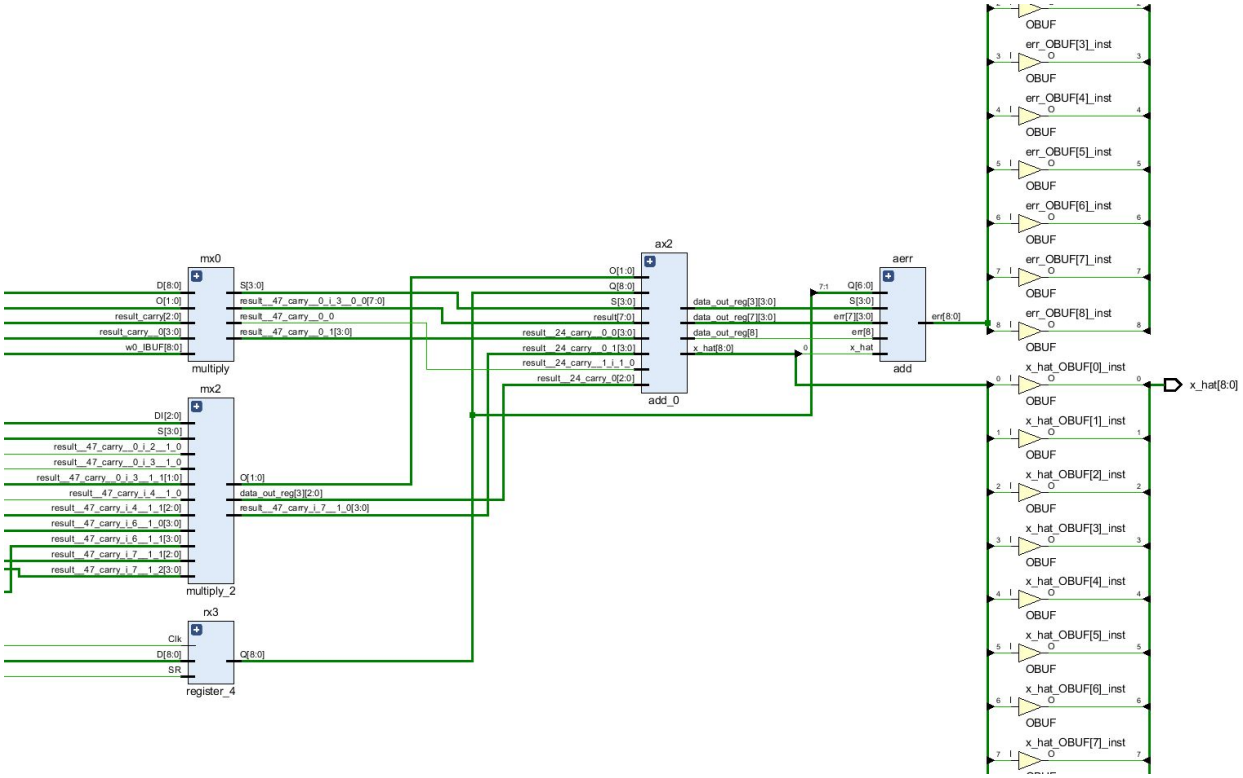
Left Side:

On the left side, the inputs xn3, w0, w1, and w2, can be seen as a buffer of 9 signals (9 bits). The input xn3, feeds into rx1. Rx1 connects to rx2, and rx2 to rx3. The weight buffers are fed through the registers into the multipliers.



Right side:

On the right side, there are the output buffers for error and x_hat. Right before that are the adders that use the outputs of the multipliers to finish the dot product and compute the error.

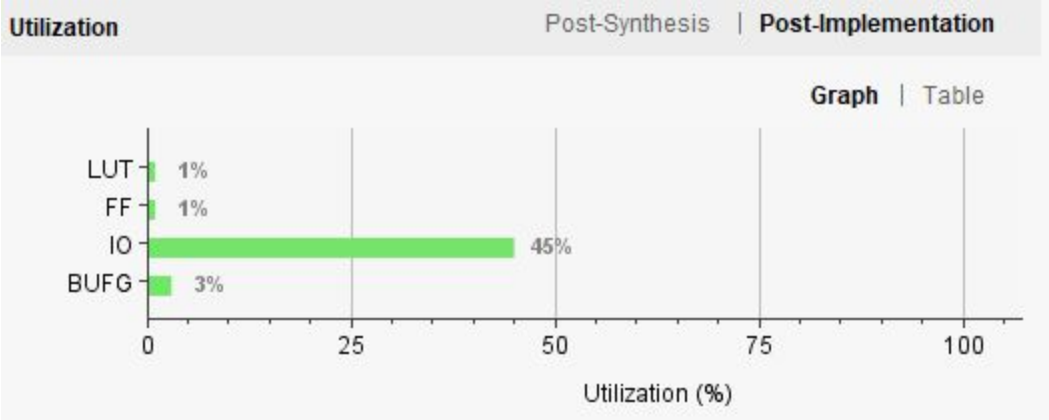


Design Resource Usage:

In general, we see a very small utilization and given the size of the large number of resources on the board and the small task at hand of performing inference with a 3-tap linear predictor.

Utilization:

In this design, the input-output elements (IO) had the largest utilization. Relative to the capacity of the FPGA, the lookup tables (LUT), flip-flops (FF) and the global buffer (BUFG) were not heavily utilized. I looked a little further into the IO usage, and I believe the reason it is large is because the inputs and outputs of the predictor module, which is currently the top level, is being mapped to pin headers on the pynq-z2 board. In reality, this system would most likely be connected to another system on the board, which feeds its inputs and outputs, not connected directly to the user through hardware pins.



Utilization — Post-Synthesis | **Post-Implementation**

Graph | **Table**

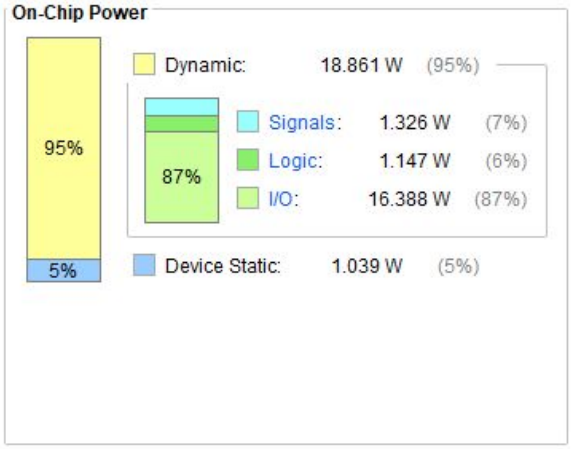
Resource	Utilization	Available	Utilization %
LUT	159	53200	0.30
FF	27	106400	0.03
IO	56	125	44.80
BUFG	1	32	3.13

Power:

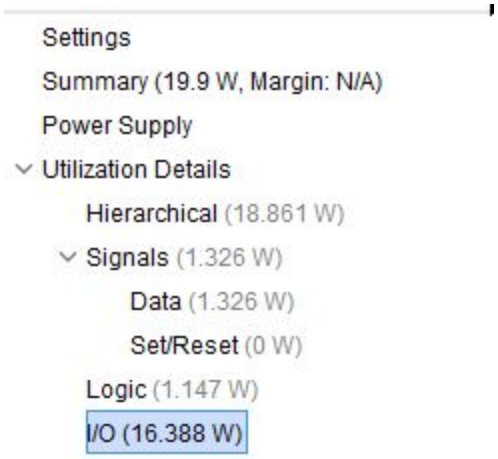
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**19.9 W (Junction temp exceeded!)**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**125,0°C**
Thermal Margin:	-169,5°C (-13,9 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



From the initial power report, it looks like the power consumption is too high, so I investigated the power consumption further.



Utilization	Name	I/O Type	I/O Standard	Drive Strength	Input Pins	Output Pins	Bidir Pins	IO LOGIC SERDES	IO DELAY	IBUF LOW PWR	Input Term	Output Imped
16.388 W (82% of total)	predictor											
8.213 W (41% of total)	err	HR	LVCMOS18	12.000	0	9	0	No	Off	No	NONE	RDRV_NONE
8.027 W (40% of total)	x_hat	HR	LVCMOS18	12.000	0	9	0	No	Off	No	NONE	RDRV_NONE
0.036 W (<1% of total)	w0	HR	LVCMOS18	N/A	9	0	0	No	Off	No	RTT_NONE	NONE
0.036 W (<1% of total)	w1	HR	LVCMOS18	N/A	9	0	0	No	Off	No	RTT_NONE	NONE
0.036 W (<1% of total)	w2	HR	LVCMOS18	N/A	9	0	0	No	Off	No	RTT_NONE	NONE
0.036 W (<1% of total)	xn3	HR	LVCMOS18	N/A	9	0	0	No	Off	No	RTT_NONE	NONE
0.004 W (<1% of total)	Clk	HR	LVCMOS18	N/A	1	0	0	No	Off	No	RTT_NONE	NONE
0 W	reset	HR	LVCMOS18	N/A	1	0	0	No	Off	No	RTT_NONE	NONE

The second largest power consumption source is the IO. Similar to the utilization problem, mapping x_hat and err outputs to pin headers seems to be taking up most of the power consumption. In addition, since the system is continuous the system is constantly supplying power to these output pins. The system can't perform multiple inferences at once, and then output the 1 result, like the MSE, onto the pins, but instead for every inference at every clock cycle rising edge a new output is mapped to pins and the pin needs power.

Utilization	Name	Signals (W)	Data (W)	Logic (W)	I/O (W)
18.861 W (95% of total)	predictor				
16.771 W (84% of total)	Leaf Cells (57)				
0.507 W (3% of total)	mx0 (multiply)	0.158	0.158	0.35	<0.001
0.482 W (2% of total)	ax2 (add_0)	0.253	0.253	0.228	<0.001
0.26 W (1% of total)	rx1 (register)	0.102	0.102	0.158	<0.001
0.246 W (1% of total)	rx2 (register_3)	0.089	0.089	0.157	<0.001
0.22 W (1% of total)	mx2 (multiply_2)	0.105	0.105	0.115	<0.001
0.21 W (1% of total)	mx1 (multiply_1)	0.096	0.096	0.114	<0.001
0.15 W (1% of total)	aerr (add)	0.133	0.133	0.017	<0.001
0.014 W (<1% of total)	rx3 (register_4)	0.012	0.012	0.003	<0.001

The first largest power consumption is the hierarchical elements. From the diagram above, it can be seen that "Leaf Cells" take up most of the power. These "Leaf Cells" are referring to all the input and output buffers / wires shown in the synthesized schematic. Again, since this is a standalone design, the inputs are being mapped to some sort of input pin headers on the board, and at every clock cycle a new 9 bits need to be read in for xn3. In addition, the weights are not being held in a register (I used the weight registers in my Weight Update Block implementation, but not for inference using the F-block). Instead, for inference the weights are constantly being held as inputs into the system which is probably causing a high power consumption.

Timing Check:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-15,298 ns	Worst Hold Slack (WHS):	0,156 ns	Worst Pulse Width Slack (WPWS):	9,500 ns
Total Negative Slack (TNS):	-238,694 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	18	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	72	Total Number of Endpoints:	72	Total Number of Endpoints:	28

Timing constraints are not met.

I created a .sdc timing constraints file to create a clock signal and implement input and output constraints for those ports. Shown above is the initial timing report. It shows that I should have

given more time for the IO signals to read and write, because currently 18 endpoints are failing. I went back and edited the timing constraints to fix these errors, and was able to pass the initial timing check with no errors, but still have a red highlighted Total Negative Slack.

Appendix:

Python Notebook:

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:

```
#here i genereate 1,000,000 data points using the generation model
n_sample = 1000000
gn = [np.random.normal(),np.random.normal(),np.random.normal()]
data = []
for idx in range(3,n_sample+3):
    g = np.random.normal()
    gn.append(g)
    data.append(.1*g+.5*gn[idx-1]-.5*gn[idx-2]+.1*gn[idx-3]) # x_[n] = u_[n] + 0.8*x_[
```

In [5]:

```
# the following code obtains the optimal weights and MSE

X = []
y = []
COV_X = []
for three_sample_before, two_sample_before, one_sample_before, curr in zip(data[0:-3:]
    vector1 = np.array([one_sample_before, two_sample_before, three_sample_before])
    vector2 = np.array([curr, one_sample_before, two_sample_before]) # for R
    X.append(vector1)
    y.append(curr)
    COV_X.append(vector2)
X = np.vstack(X)
y = np.array(y)
COV_X = np.vstack(COV_X)


R = np.cov(COV_X.T)
p = y.dot(X)/len(y)
w = np.linalg.inv(R).dot(p)
prediction = X.dot(w)
print("Optimal Weights:",w)
error = y.dot(y)/n_sample - p.dot(w)
print("Optimal MSE:",error)
```

```
Optimal Weights: [-0.68014248 -0.41424331 -0.17878953]
Optimal MSE: 0.3513533155728955
```

In [44]:

```
# here are some helper functions
def quantize(values,BW):
    quant = np.minimum(np.round(np.array(values)*np.power(2.0,BW-1.0))*np.power(2.0,1.(
    return quant

def calc_MSE(x,x_hat):
    #here x is true values, x_hat is prediction
        return np.sum(np.square(np.array(x) - np.array(x_hat)))/n_sample

def evaluate_sqnr(data, dq,mserr):
    SNR = 10*np.log10(np.true_divide(np.sum(np.var(data)),np.sum(np.var(np.array(data)
    return SNR

def predict(wq,xq):
    x_hats = np.matmul(xq,np.transpose(wq))
    mse = calc_MSE(y,x_hats)
    return mse


#first without quantizing:
print("Optimal MSE:",error)
optimal_SNR = 10*np.log10(np.var(y)/error)
print("Optimal SNR (dB):", optimal_SNR)

#now quantize for various bit precisions
precisions = np.arange(1,17,1)

print("Bits \t| MSE \t| SQNR\t|SQNR-SNR|\tWithin .5dB of SNR?")
MSEs = []
sqnrs = []
for BW in precisions:
    wq = quantize(w,BW)
    dq = quantize(X,BW)
    MSE = predict(wq,dq)
    MSEs.append(MSE)
    sqnr = evaluate_sqnr(X,dq,MSE)
    sqnrs.append(sqnr)
    print("{:.4f}\t|{:.4f}\t|{:.3f}\t|{:.4f}|\t{}".format(BW,MSE,sqnr,abs(sqnr-optimal_
```

```
Optimal MSE: 0.3513533155728955
Optimal SNR (dB): 1.698789959975315
Bits     | MSE    | SQNR   |SQNR-SNR|      Within .5dB of SNR?
1.0000   |0.5718 |-1.896 |3.5949|        NO
2.0000   |0.4453 |-0.053 |1.7515|        NO
3.0000   |0.3851 |0.900   |0.7987|        NO
4.0000   |0.3698 |1.208   |0.4906|        YES
5.0000   |0.3654 |1.312   |0.3870|        YES
6.0000   |0.3634 |1.357   |0.3417|        YES
7.0000   |0.3627 |1.375   |0.3239|        YES
8.0000   |0.3625 |1.383   |0.3162|        YES
9.0000   |0.3623 |1.387   |0.3121|        YES
10.0000 |0.3623 |1.389   |0.3101|        YES
11.0000 |0.3622 |1.390   |0.3091|        YES
12.0000 |0.3622 |1.390   |0.3085|        YES
```

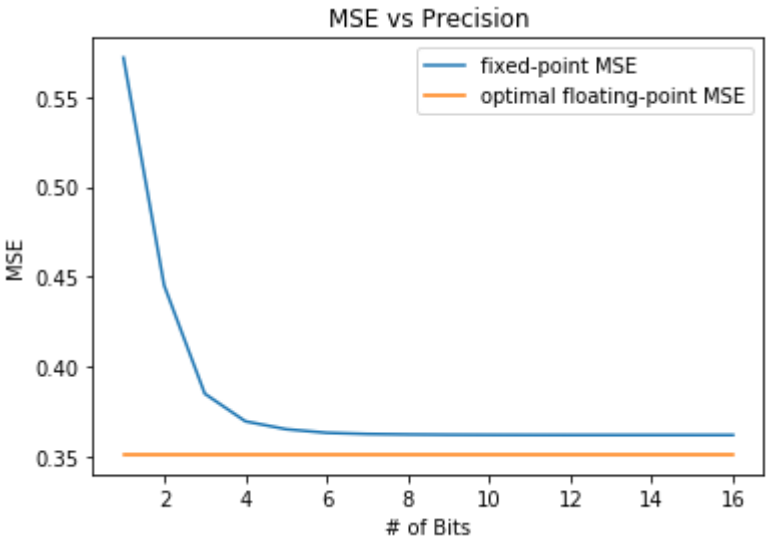
```
13.0000 |0.3622 |1.391  |0.3082|        YES
14.0000 |0.3622 |1.391  |0.3081|        YES
15.0000 |0.3622 |1.391  |0.3081|        YES
16.0000 |0.3622 |1.391  |0.3080|        YES
```

In [28]:

```
# we can also plot the MSE and SQNRs across different bit widths
plt.figure()
plt.plot(precisions, MSEs, label = "fixed-point MSE")
plt.title("MSE vs Precision")
plt.xlabel("# of Bits")
plt.ylabel("MSE")
plt.plot(precisions,[error]*len(precisions),label = "optimal floating-point MSE")
plt.legend()
```

Out[28]:

```
<matplotlib.legend.Legend at 0x17a0a5b5198>
```

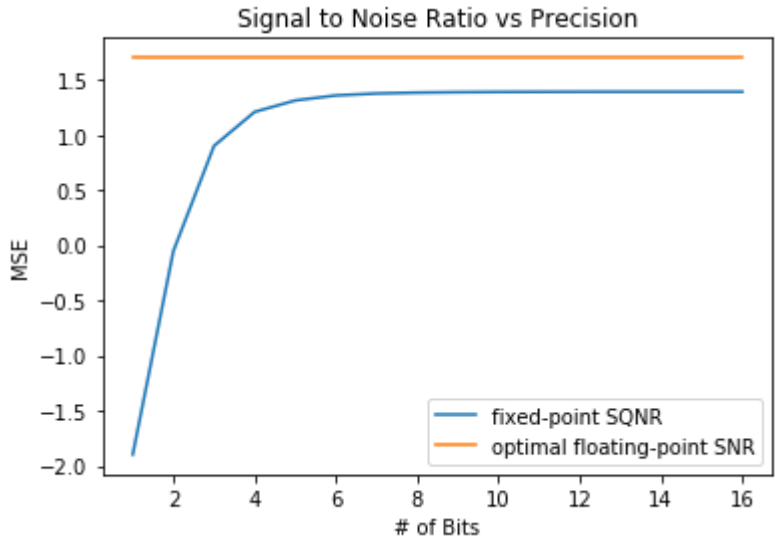


In [47]:

```
plt.figure()
plt.plot(precisions, sqnrs, label = "fixed-point SQNR")
plt.title("Signal to Noise Ratio vs Precision")
plt.xlabel("# of Bits")
plt.ylabel("MSE")
plt.plot(precisions,[optimal_SNR]*len(precisions),label = "optimal floating-point SNR"
plt.legend()
```

Out[47]:

```
<matplotlib.legend.Legend at 0x17a0a48c9e8>
```



In [81]:

```
#verilog confirmation:
xs = np.array([np.array([0.73500254,  0.55186864,  0.19778589]),[-0.69549231, 0.735002
y_dec= [-0.69549231, -0.66702645]
ws_quant = quantize(w,4)*16
for x in xs[:-1]:
    xs_quant = quantize(x,4)
    print(xs_quant*16,ws_quant)
    xs_quant = xs_quant*16
    print(xs_quant.dot(ws_quant)/256,x.dot(w))
```

```
[12.  8.  4.] [-10.  -6.  -2.]
-0.6875 -0.7638763927922381
```

In [86]:

```
x_dptest = np.array([0.73500254,  0.55186864,  0.19778589])
w_dptest = np.array([-0.68014248, -0.41424331, -0.17878953])
print("Floating point dot-product:", x_dptest.dot(w_dptest))
```

```
Floating point dot-product: -0.7638763887944293
```

VHDL Code:

Predictor.sv

```
//3 tap linear predictor code

module predictor(
	input logic reset, Clk,
	input logic [8:0] xn3, w0, w1, w2,
	output logic [8:0] x_hat, err);

//internal signals across registers
logic [8:0] xn2, xn1, xn;
logic ld;
assign ld = 1; //we are loading the registers every clock cycle
//assign xn3 = x;

//internal signals, outputs between adders and multipliers
logic [8:0] mx0_out, mx1_out, mx2_out, ax01_out, ax2_out, aerr_out, aw0_out, aw1_out,
aw2_out, mw0_out, mw1_out, mw2_out;

//internal registers:
register rx1(.data_in(xn3), .data_out(xn2), .*); //.* should connect ld, Clk, reset
register rx2(.data_in(xn2), .data_out(xn1), .*);
register rx3(.data_in(xn1), .data_out(xn), .*);


//filter block:
multiply mx0(.a(xn3), .b(w0), .result(mx0_out));
multiply mx1(.a(xn2), .b(w1), .result(mx1_out));
add ax01(.a(mx0_out), .b(mx1_out), .result(ax01_out));
multiply mx2(.a(xn1), .b(w2), .result(mx2_out));
add ax2(.a(ax01_out), .b(mx2_out), .result(x_hat));

//error
add aerr(.a(~x_hat + 1), .b(xn), .result(err));


//Weight update block:
/*
register rw0(.data_in(aw0_out), .data_out(w0), .*);
register rw1(.data_in(aw1_out), .data_out(w1), .*);
register rw2(.data_in(aw2_out), .data_out(w2), .*);
*/
```

```
/*
multiply mw0(.a(err), .b(xn3), .result(mw0_out));
multiply mw1(.a(err), .b(xn2), .result(mw1_out));
multiply mw2(.a(err), .b(xn1), .result(mw2_out));

add aw0(.a(mw0_out), .b(w0), .result(aw0_out));
add aw1(.a(mw1_out), .b(w1), .result(aw1_out));
add aw2(.a(mw2_out), .b(w2), .result(aw2_out));
*/
endmodule

module add(
	input logic [8:0] a, b,
	output logic[8:0] result);
assign result = a + b;//(real'(a) + real'(b));
endmodule

module multiply(
	input logic [8:0] a,b,
	output logic [8:0] result);

assign result = a*b;//(real'(a) * real'(b));

endmodule

module register(
	input logic[8:0] data_in,
	input logic ld, reset, Clk,
	output logic[8:0] data_out);

always_ff @(posedge Clk)
begin
	if (reset)
		data_out <= 9'h00;
	else if(ld==1'b1)
		data_out <= data_in;
	else
		data_out <= data_out;

end
endmodule
```

Testbench.sv

```
module testbench();
timeunit 10ns;
timeprecision 1ns;

logic reset, Clk;
logic [8:0] xn3, w0, w1, w2;;
logic [8:0] x_hat, err;

always begin : CLOCK_GENERATION

 #1 Clk = ~Clk;

end

initial begin : CLOCK_INITIALIZATION
          Clk = 0;
end

predictor p(.*);

parameter [3:0] data [4] = {0.19778589 * 16, 0.55186864 * 16, 0.73500254 * 16, -0.69549231 *
16};
//{4, 8, 12, -0.69549231 * 16};

//x_hat should be = -0.69549231
real result;
assign result = x_hat/16/16;

initial begin : TEST_VECTORS

#2 reset = 0;
#2 reset = 1;
#2 reset = 0;

//set weights: [-0.67776111 -0.41394865 -0.17928979]
#2 w0 = -0.67776111*16; w1=-0.41394865*16; w2=-0.17928979*16;

#2 xn3 = data[3]; //xn is the one at the end and xn-1 is the one before that, etc
#2 xn3 = data[2];
#2 xn3 = data[1];
#2 xn3 = data[0];

end
```

```
endmodule
```