Indian Institute of Technology Gandhinagar

Final Project Submission, 23 April, 2024

Design Posit and Float point adders and multipliers used in Wiener Filter

---

Digital System

Project Report

Aravind Krishna 20110021
Sakshi Jain 20110181

Objective:

This project is focused on designing specialized Posit and Floating-Point adders and multipliers for integration within the Wiener Filter algorithm. The Wiener Filter plays a pivotal role in signal processing applications, particularly in tasks such as noise reduction and signal enhancement.

To ensure the Wiener Filter operates at its peak efficiency, it necessitates arithmetic units that can effectively manage the computational intricacies while upholding precision standards. Therefore, the primary aim of this project is to develop arithmetic units tailored precisely to meet the demands of the Wiener Filter and compare the performance of the two point systems.

Floating Number System:

In floating-point representation, there isn't a fixed allocation of bits for the integer or fractional parts of a number. Instead, a certain number of bits are designated for the entire number known as the **mantissa** or **significand** and another set of bits indicates the position of the decimal point within that number referred to as the **exponent**.
The value represented by a float number X, distributed as illustrated, can be expressed as follows:

$$X = (-1)^{s} \times (1 + m) \times 2^{(e-Bias)}$$

n-1 0

Sign Bit	Exponent Part	Mantissa Part

where, s is the sign bit,
m is the mantissa,
e is the exponent value, and
Bias is the bias number.

Sign Bit: The sign bit is the leftmost bit in the floating-point representation. It determines whether the number is positive or negative. In the IEEE 754 standard, the sign bit is 0 for positive numbers and 1 for negative numbers.

Exponent Bits: The exponent is a binary integer that determines the scale of the number. It represents the power of 2 by which the mantissa is multiplied. The exponent

is biased to allow both positive and negative exponents to be represented. The bias value is added to the actual exponent to obtain the stored exponent value.
For half precision(16 bits), the exponent is represented using 5 bits. The bias value is 15.
For single precision (32 bits), the exponent is represented using 8 bits. The bias value is 127, which means an exponent of 0 is stored as 127.
For double precision (64 bits), the exponent is represented using 11 bits, and the bias value is 1023.

Mantissa Bits: The mantissa (also called significand or fraction) is a binary fraction that represents the significant digits of the number. It is normalized to have a leading 1 bit followed by the fractional part. Since the leading bit is always 1, it is not explicitly stored in the representation. The remaining bits in the mantissa represent the fractional part of the number.

For half precision, the mantissa has 10 bits.
For single precision, the mantissa has 23 bits.
For double precision, the mantissa has 52 bits.

In floating-point representation, numbers are typically **normalized**, which means the leading bit of the mantissa is always 1. This allows for efficient storage and comparison of floating-point numbers. Normalization simplifies arithmetic operations and ensures that the representation is unique for each number. Normalized numbers are represented as a single nonzero digit to the left of the binary point like $1.xxxxxxxx \times 2^{yyyy}$.

Underflow happens when the result of a floating-point operation is too close to zero to be represented accurately. In other words, the magnitude of the result is smaller than the smallest positive normalized number that can be represented. When underflow occurs, the result is rounded to zero, and precision is lost.

Overflow occurs when the result of a floating-point operation exceeds the maximum representable value. In other words, the magnitude of the result is larger than the largest finite number that can be represented. When overflow occurs, the result is often represented as infinity or the maximum representable value, depending on the context and the floating-point representation used.

In normalized floating-point representation, if the result exceeds the range where the exponent is larger than 8 bit exponent field (for single precision), overflow will occur. Conversely, if the result falls below the range of representable values, specifically when it is greater than 0 but smaller than 1.0×2^{-127} underflow will occur.

Denormalized numbers are used to represent values close to zero that are smaller than the smallest normalized number. **If E = 0, but the fraction is non-zero, then the value is in denormalized form, and a leading bit of 0 is assumed.**
For single-precision, E=0, X =(-1)s 0.F 2 (-126)
In denormalized notations, the floating point has no hidden 1 i.e. no implied 1. This allows floating point numbers to define numbers very close to 0. To handle this special case, it is agreed that in this case the sign, exponent and the mantissa are set to 0s.

Under denormalized floating point representations there are three special cases: **Zero, infinity and Nan.**
Zero: Zero cannot be represented in the normalized form of floating point representations, and must be represented in denormalized form with E=0 and F=0. There are two representations for zero: +0 with S=0 and -0 with S=1.

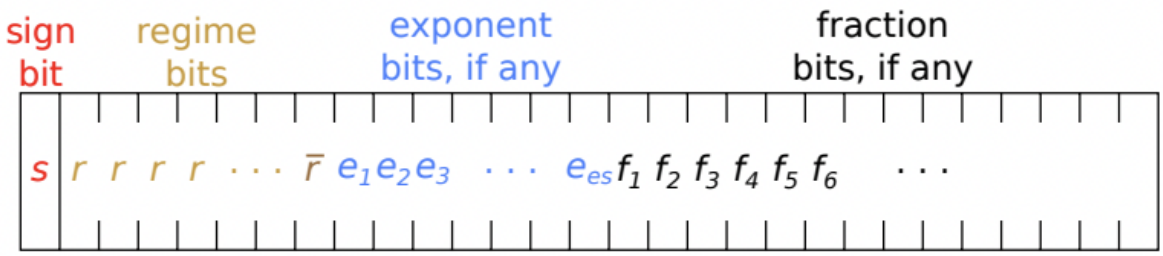
Infinity: The value of +infinity and -infinity are represented with an exponent of all 1's , F=0, and S=0 (for +INF) and S=1 (for -INF)

All the exponent bits 1 and mantissa bits non-zero represent **NaN**.

Posit Number System:

The Posit number system, introduced by John L. Gustafson in as a substitute for the widely used IEEE 754 standard for floating-point arithmetic, is gaining recognition for its potential advantages. These include improved dynamic range and accuracy within the same bit width, as well as enhanced computational consistency across different machines compared to conventional floating-point formats..
The value represented by a posit number X, distributed as illustrated, can be expressed as follows:

$$X = (-1)^{s} * useed^{k} * 2^{e} * 1.f$$



where $useed = 2^{2^{es}}$,

k is regime value,

e is unsigned value (if es > 0) ,

f is the mantissa of the number without an implicit one.

Sign Bit: The conventional sign bit operates as usual: 0 signifies positive numbers, while 1 denotes negative numbers. In the case of negative numbers, the 2's complement is applied before interpreting the regime, exponent, and fraction.

Regime Bits (**k**) : Binary sequences start with a sequence of consecutive 0s or 1s, concluding either when the next bit differs or upon reaching the end of the sequence. Let "m" denote the count of identical bits in the consecutive run; if these bits are 0, then $k = -m$; whereas, if they are 1, $k = m - 1$. The regime denotes a scale factor of $useed^k$.

Exponent Bits (**e**)**:** They signify scaling by 2^e. The number of exponent bits can be up to es bits, contingent on the remaining bits to the right of the regime.

Fraction (**f**) **:** If any bits remain after accounting for the regime and exponent bits, they signify the fraction f, analogous to the fraction 1.f in a floating-point format, where there is always a hidden bit set to 1.

Posit numbers have only two exceptional values: 0 (represented by all 0 bits) and ±∞ (indicated by 1 followed by all 0 bits). These values do not conform to positional notation in terms of their bit string meanings.

The primary encoding distinction between float and posit formats lies in the inclusion of a runtime-varying scaling component, which encompasses the regime and the available exponent bits.

Algorithm:

Floating Point Multiplication:

It involves following steps:

1. Determine the sign of the product by XOR of the sign bit of Multiplicand and Multiplier.
2. Compute the exponent of the product by adding the exponents and subtract $2^{k-1} - 1$ to maintain the bias representation, where k is the length of exponent bits.
3. Multiply the mantissas.
4. Normalize and round the product.
    a. If MSB of the product of mantissas is 1, then it is normalized, otherwise left shift the product and increase the sum of exponents by 1.
    b. Rounding: If the first bit in the Guard bits is 1 and OR of the rest of the Guard bit is 1, then add 1 to the product of mantissas. Otherwise do nothing.
5. Check for overflow and underflow conditions.

Posit Multiplication:

1. Given inputs: IN1 and IN2, each consisting of several sub-components like XIN1, S1, R1, E1, M1, Inf1, Z1, etc.
2. Z and Inf are initialized based on the logical AND and OR operations applied to Z1 and Z2, and Inf1 and Inf2 respectively.
3. Determine the actual (+ve / -ve) of regime (RG1 and RG2) using respective regime check bits (RC1 and RC2)
4. M1 and M2 (mantissas) are multiplied to get M, which is then shifted left by one position if the most significant bit (MSB) is set.
5. Exp is computed by adding the exponent parts (E1 and E2) with adjustments based on RG1, RG2, and the value of the most significant bit in M.
6. The computed values are then composed together, rounded, and the final output is generated.

Data Extraction : The module takes an input in representing the posit number and decomposes it into several output components: the regime sign bit (rc), the regime value (regime), the left shift amount (Lshift), the exponent (exp), and the mantissa (mant). The posit number's size is defined by the parameter N, and it calculates the regime's bit size (Bs) and uses a fixed exponent size (es). It utilizes Leading One Detector (LOD_N) and Leading Zero Detector (LZD_N) modules to determine the regime value and shift amount and also a dynamic shift register (DSR_left_N_S) to adjust the number's alignment for extracting the exponent and mantissa. For detecting a sequence of zeros terminated by a one (negative regime), we use LOD on XIN[N-2:0]. Similarly, for a sequence of ones terminated by a zero (positive regime), we employ LZD on X[N-3:0] (one bit less as the regime value will be one less than the count of 1s.).

Verilog:

Top Module:

1. Parameter: n - bit width of the input
   exp - bit width of exponent
2. Input : select - 2 bit
   A - n bit
   B - n bit
3. Output: out - n bit

select can take 4 values:

- 00 - posit multiplier
- 01 - posit adder
- 10 - float point multiplier
- 11 - float point adder

A 4X1 multiplexer is used to get the output based on the value of the select line.

Simulation Results:

1. For (**16,5**)

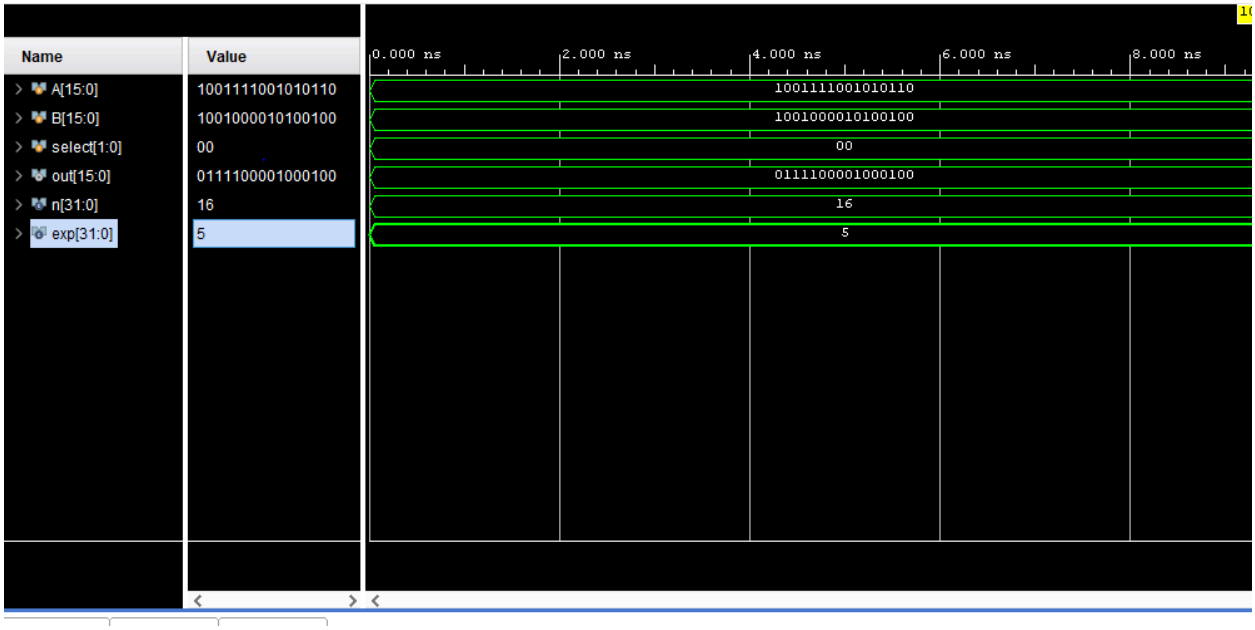
Posit Multiplier:

select = 00

A = 1001111001010110 Decimal = -1.1601e-17

B = 1001000010100100 Decimal = -8.2598e-20

out = 0111100001000100 Decimal = 1.2605e+29



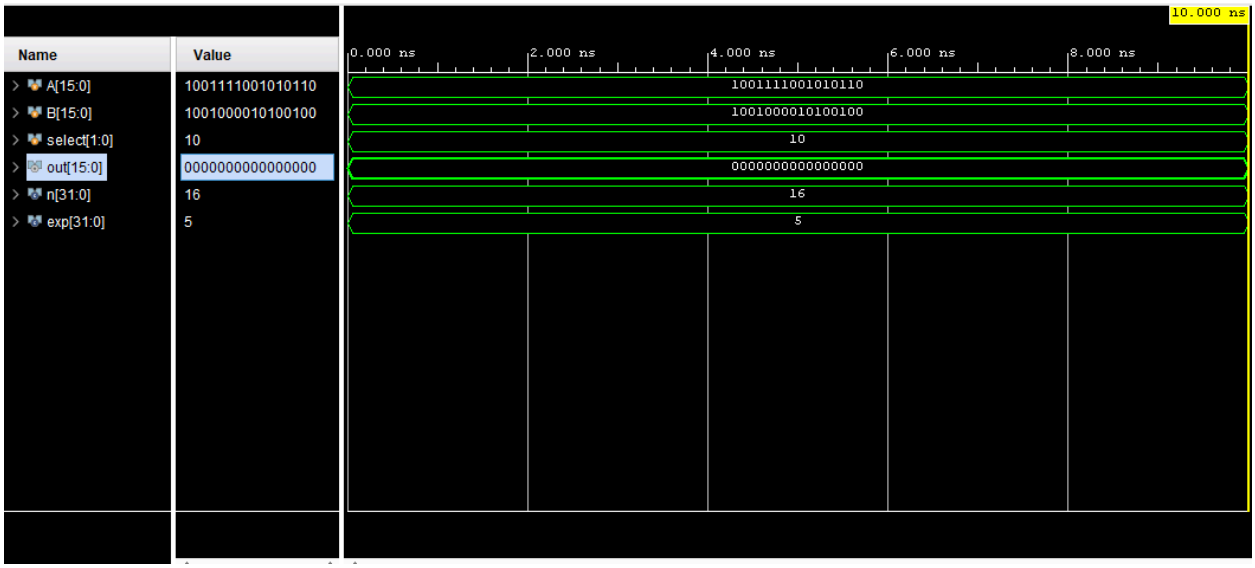
Float Point Multiplier:

select = 10

A = 1001111001010110 Decimal = -0.0061874

B = 1001000010100100 Decimal = -0.0005664825

out = 0000000000000000 Decimal = 0.00000



2. For (32,8)

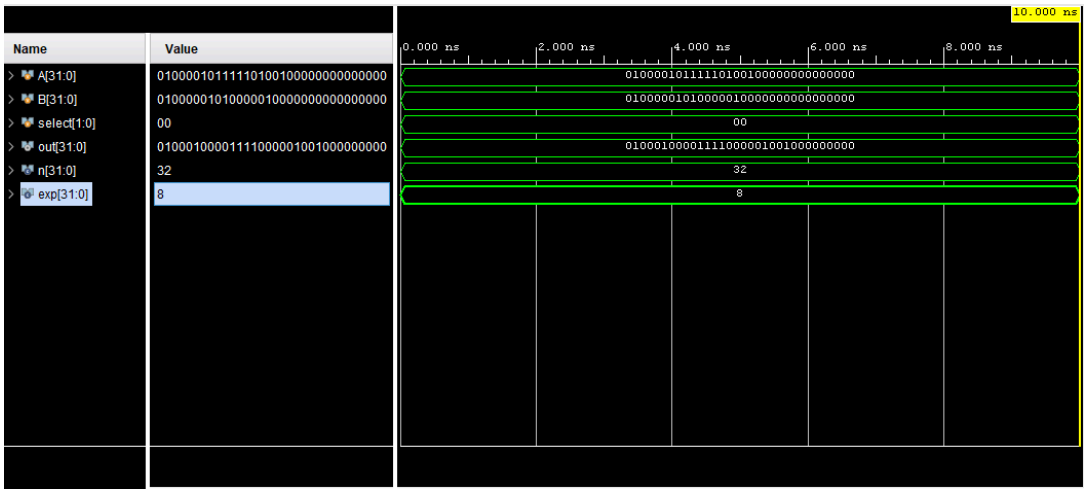
Posit Multiplier:

select = 00

A = 01000010111110100100000000000000 Decimal = 2.9958

B = 01000001010000010000000000000000 Decimal = 1.2807

out = 01000100001111000001001000000000 Decimal = 3.8366



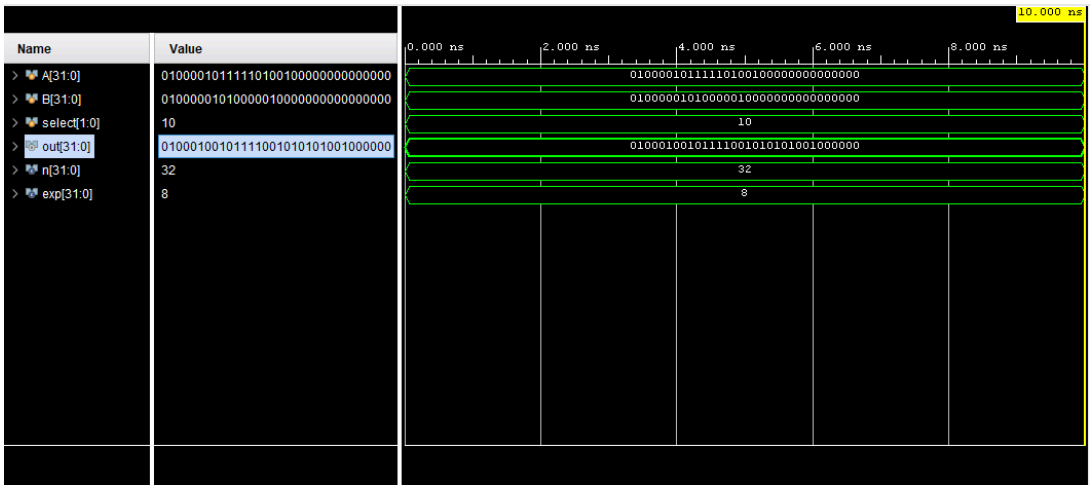
Float Point Multiplier:

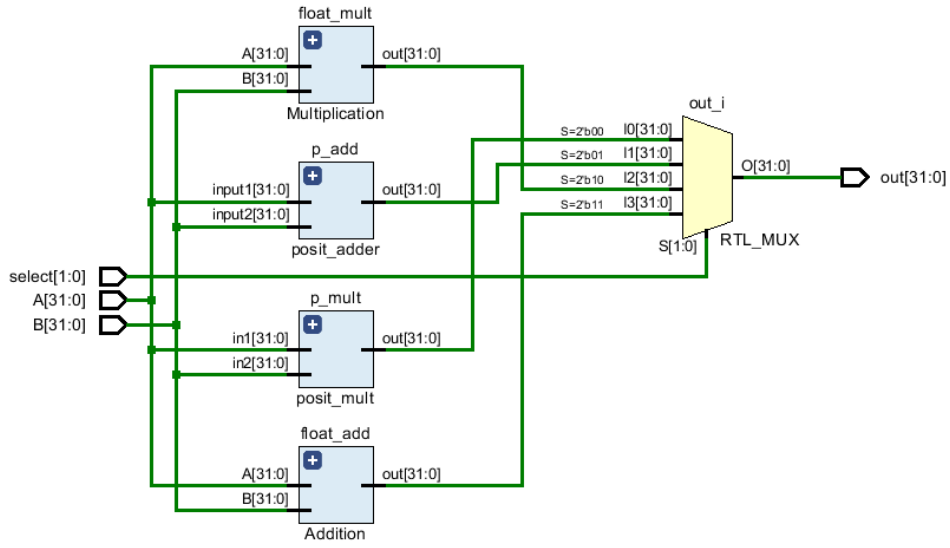
select = 10

A = 01000010111110100100000000000000; Decimal = 125.125

B = 01000001010000010000000000000000; Decimal = 12.0625

out = 01000100101111001010101001000000 Decimal = 1509.3203125





Elaborated Design

Wiener Filter:

Noise and Local Variance Conversion:

- The noise variance is predetermined and the calculated local variance within each neighborhood of the image is converted to binary.

Local Region Processing:

- For each pixel in the image, the function defines a local region based on the specified kernel size. This region is centered around the current pixel (i, j) and its size is given by kernel_size.
- The local mean and variance of this region are then computed. These statistical measures are crucial as they describe the average intensity and variability of pixel values within the local region, respectively.

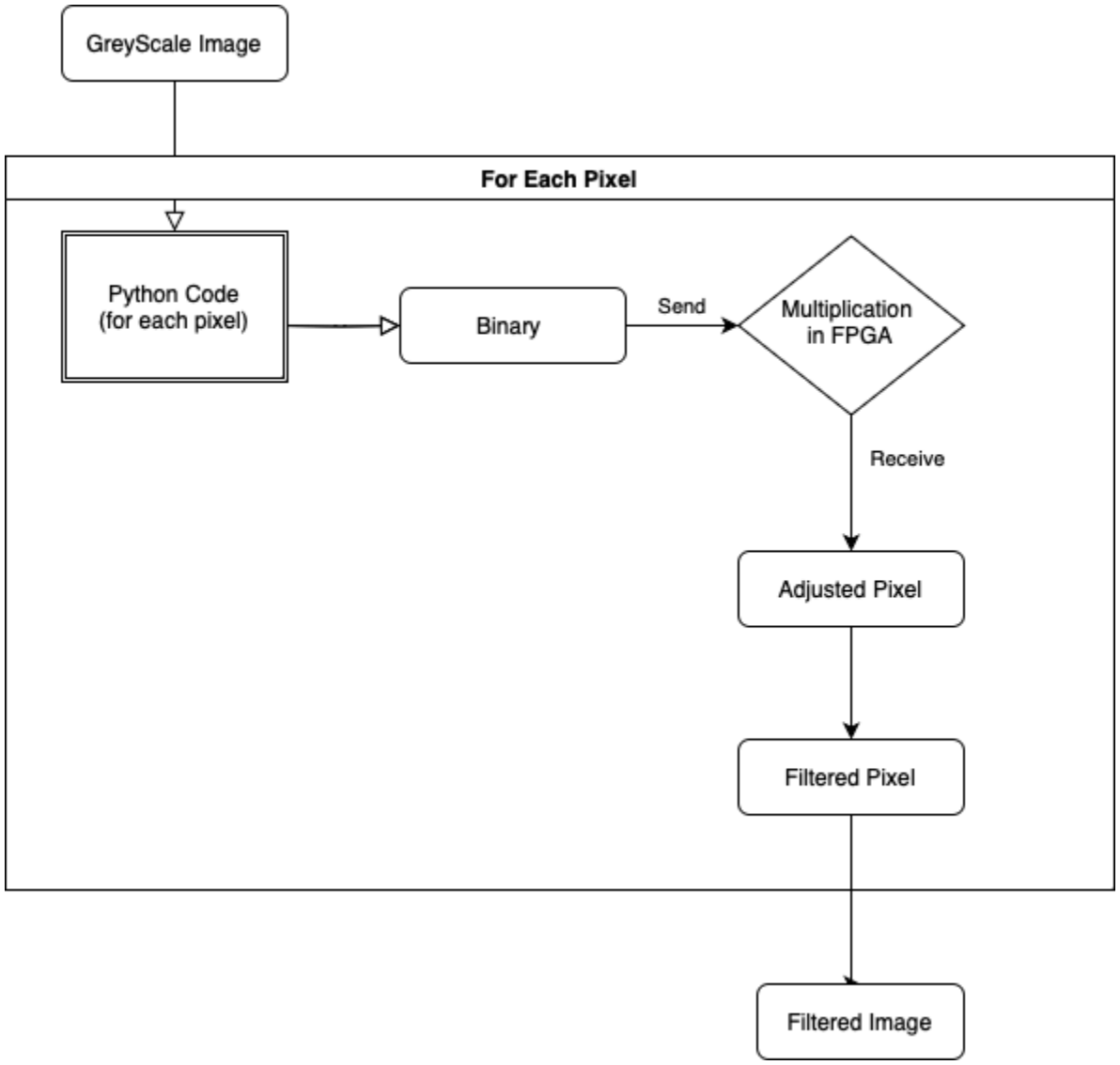
FPGA Offloading for Multiplication:

- The local variance and noise variance (both in binary form) are sent to the FPGA for multiplication. The Wiener filter requires a comparison between the local variance and the noise variance to determine how much noise reduction should be applied.
- The multiplication result (the product of local variance and noise variance) is received back from the FPGA and converted from binary to a floating-point number or a posit number.

Filter Application:

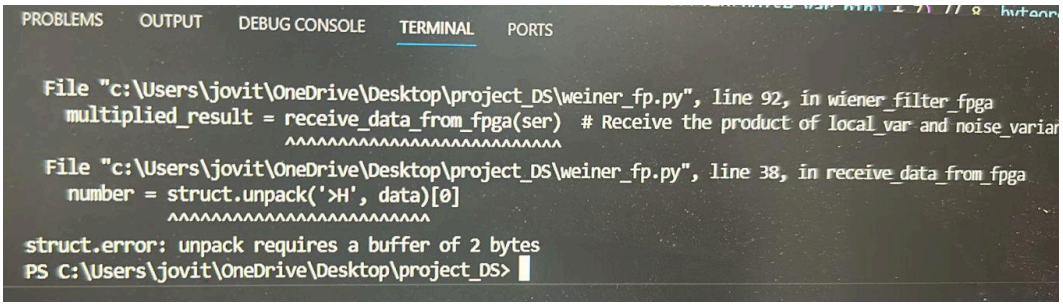
- The actual filtering operation for each pixel adjusts the original pixel value based on the local mean, the ratio of the noise variance to the local variance, and the calculated product from the FPGA.
- Specifically, the pixel value is adjusted by subtracting the local mean (to center the pixel values around zero), then scaled by the factor (multiplied result / local variance). This factor adjusts the influence of the original pixel value based on how noisy the local region is compared to the overall noise level.
- If the local variance is significantly higher than the noise variance, the original pixel value is mostly preserved, reflecting a lower level of noise.
- Finally, the local mean is added back to reposition the adjusted value within the correct intensity range.

Flow Chart:



Simulation & Observations:

We tried to simulate the filter but we faced issues with UART communication, even though the data was sent to the fpga, no data was getting received back to the python code due to which we were not able to produce results.
UART can transmit only 8 bits at a time, so we tried to use bit frames to receive and send the data but still it wasn't working..



Receiver.v

- Inputs and Outputs: It has inputs for the system clock (clk), reset (reset), and serial data (RxD). The outputs are a 16-bit data output (RxData) and a flag (isNewData) to indicate that new data has been received.
- State Machine: The code employs a state machine with several states for handling serial communication:
- State 0: Idle state, waiting for the start bit (a '0').
- State 1: Verifies the presence of the start bit.
- State 2 to DATA_WIDTH+1: Sequentially reads each bit of the data frame.
- State DATA_WIDTH+2: Processes the end of a frame. If it's the first frame, the data is stored temporarily. If it's the second frame, the data from both frames are combined into one 16-bit number and stored in RxData, toggling the isNewData flag to indicate completion.

Sender.v

- Sender is designed to transmit serial data divided into two 8-bit frames.
- Configuration: It uses parameters for system clock frequency, baud rate, and data width.
- Inputs/Outputs: Includes system clock (clk), reset (reset), serial data output (TxD), a control signal to start transmission (doTransmit), 16-bit data input (TxData), and a busy flag (isBusy).
- Transmission Process:

- Initialization: Waits for a trigger (doTransmit) to start sending data while not busy.
- First Frame: Loads the first 8 bits of TxData, transmits them bit by bit, followed by a stop bit.
- Second Frame: Automatically loads and sends the second set of 8 bits as the next frame, also followed by a stop bit.
- State Machine: Controls data transmission, ensuring each bit is sent sequentially followed by stop bits, toggling isBusy during transmission to prevent new data from starting.