Contents

Moving Average Filter

November 27, 2024

1 Overview

The moving average filter is a FIR Filter (Finite Impulse Response), the filter has following properties.

1.1 Pros:

- Easy to implement.
- Great for signals encoded in the time domain.
- Effective at removing random noise, and mean-free noise.
- Maintains a fast step response, if the filter size is reasonably short.
- Simple computational requirements, making it suitable for real-time processing (best performance when filter size is $(2^N, \quad N \in \mathbb{N})$
- Effective in smoothing out short-term fluctuations in a data set, leading to a clearer signal trend.
- Does not require extensive knowledge of the signal characteristics for effective application.

1.2 Cons:

- Causal filter: introduces phase shift in the filteroutput, which gets worse for higher N.
- Bad at filtering data in the frequency domain. It's a low-pass filter with sub-optimal attenuation.
- Reduces the amplitude of signal components, potentially leading to loss of important information in the signal.
- Poor performance in distinguishing between high-frequency noise and high-frequency signal components.
- Not ideal for applications requiring precise frequency domain analysis.
- Moving average filters have a fixed window size, which might not be optimal for all types of signals, but the filter size can be varied.

2 Digital Signal Processing

This filter has the advantage that it only needs one multiplication, which can be realised with a shift right operator, if $N+1$ is a power of 2. The filter works by saving the last N values of the input signal and the current signal. Then the sum of the $(N+1)$ values are divided by $N+1$.

$$\text{filter\_output} = \frac{1}{N+1} \sum_{n=0}^{N} a[n] \quad N = \text{Filter\_length} \tag{1}$$

2.1 Impulse response

The impulse response $h[n]$ of a moving average filter of length $N+1=8$ in response to a unit impulse $\delta[n]$ is given by:

$$h[n] = \frac{1}{N+1} \cdot \delta[n], \quad 0 \leq n \leq N \tag{2}$$

where $\delta[n]$ is the unit impulse function. The impulse response of the filter can be seen in 1 on the following page.

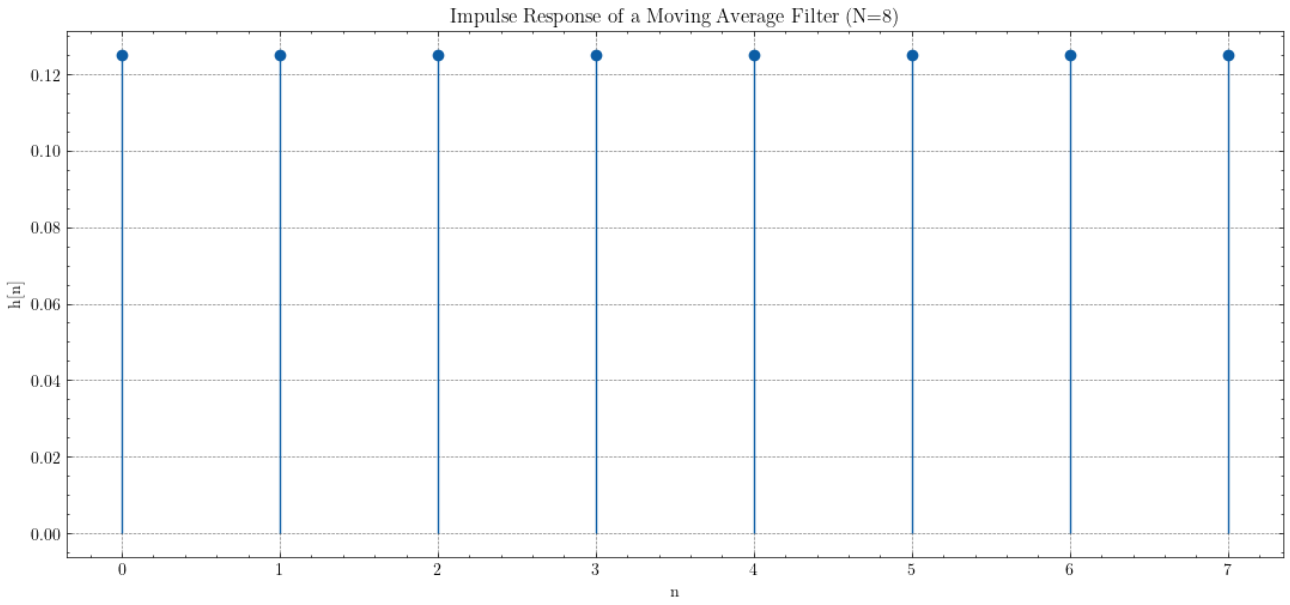

Figure 1: Impulse response moving average

Here it can be seen that the response of finite, hence it is an FIR filter.

2.2 Step response

The step response of the moving average filter can be seen in 2.

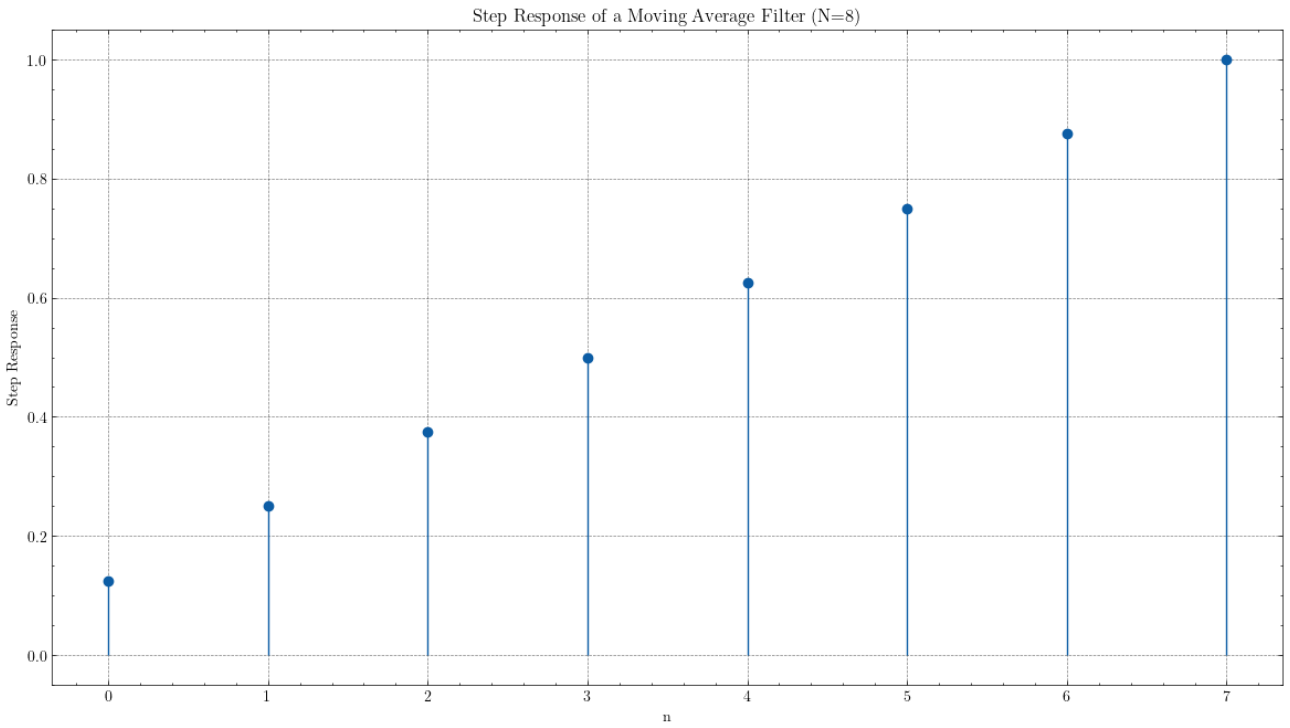

Figure 2: Step response moving average

3 Z-transform of a Moving Average Filter

Consider a moving average filter of size N. The impulse response $h(n)$ of the filter is given by:

$$h(n) = \begin{cases} \frac{1}{N} & \text{for } 0 \leq n < N, \\ 0 & \text{otherwise.} \end{cases}$$

The z-transform $H(z)$ of $h(n)$ is defined as the sum:

$$H(z) = \sum_{n=0}^{\infty} h(n) \cdot z^{-n}.$$

Substituting the values of $h(n)$ for a moving average filter:

$$H(z) = \frac{1}{N} \sum_{n=0}^{N-1} z^{-n}.$$

This sum is a finite geometric series. The sum of a geometric series a_n can be calculated as

$$a_n = \sum_{n=0}^{N-1} a \cdot q^n = a \cdot \frac{(1-q^N)}{1-q},$$

where $a = \frac{1}{N}$ and $q = z^{-1}$, for this case, additionally $|q| < 1$.

$$H(z) = \frac{1}{N} \cdot \frac{1-z^{-N}}{1-z^{-1}}.$$

This is the z-transform of a moving average filter of size N.

3.1 Discret time fourier transform

Now that we have the z-transform we can get the frequency response by substituting $z = e^{j\Omega}$.

$$H(z)|_{z=e^{j\Omega}} = \frac{1}{N} \frac{1-e^{-j\Omega N}}{1-e^{-j\Omega}} = \frac{1}{N} \frac{e^{-\frac{j\Omega N}{2}}}{e^{-\frac{j\Omega}{2}}} \cdot \frac{e^{\frac{j\Omega N}{2}} - e^{-\frac{j\Omega N}{2}}}{e^{\frac{j\Omega}{2}} - e^{-\frac{j\Omega}{2}}}$$

If we now use the exponential representation of the sin

$$\sin\left(\frac{N\Omega}{2}\right) = \frac{1}{2j}\left(e^{\frac{j\Omega N}{2}} - e^{-\frac{j\Omega N}{2}}\right)$$

The expression above simplifies to

$$H(e^{j\Omega}) = \frac{1}{N} \cdot \frac{e^{-\frac{j\Omega N}{2}}}{e^{-\frac{j\Omega}{2}}} \frac{\sin\left(\frac{N\Omega}{2}\right)}{\sin\left(\frac{\Omega}{2}\right)}$$

This can be further be split into a phase and amplitude representation.

3.2 Amplitude Response

The amplitude response, denoted as $|H(e^{j\Omega})|$, is the magnitude of the frequency response:

$$|H(e^{j\Omega})| = \left| \frac{1}{N} \cdot \frac{e^{-j\Omega N/2}}{e^{-j\Omega/2}} \cdot \frac{\sin\left(\frac{N\Omega}{2}\right)}{\sin\left(\frac{\Omega}{2}\right)} \right|$$

Simplifying, as the magnitude of a complex exponential is 1:

$$|H(e^{j\Omega})| = \frac{1}{N} \left| \frac{\sin\left(\frac{N\Omega}{2}\right)}{\sin\left(\frac{\Omega}{2}\right)} \right|$$

3.3 Phase response

The phase response, denoted as $\Phi(H(e^{j\Omega}))$, is the argument of the frequency response:

$$\Phi(H(e^{j\Omega})) = \arg\left(\frac{1}{N} \cdot \frac{e^{-j\Omega N/2}}{e^{-j\Omega/2}} \cdot \frac{\sin\left(\frac{N\Omega}{2}\right)}{\sin\left(\frac{\Omega}{2}\right)}\right)$$

This simplifies to:

$$\Phi(H(e^{j\Omega})) = -\frac{\Omega N}{2} + \frac{\Omega}{2} = -\frac{\Omega(N-1)}{2}$$

This indicates a linear phase shift characteristic of FIR filters.

3.4 Frequency and phase plots

The frequency response of the filter can be seen in 3, it can be seen that the rolloff for filtering higher frequencies is not great. This can be improved by using higher order filters. The phase response can be seen in 4 on the next page.

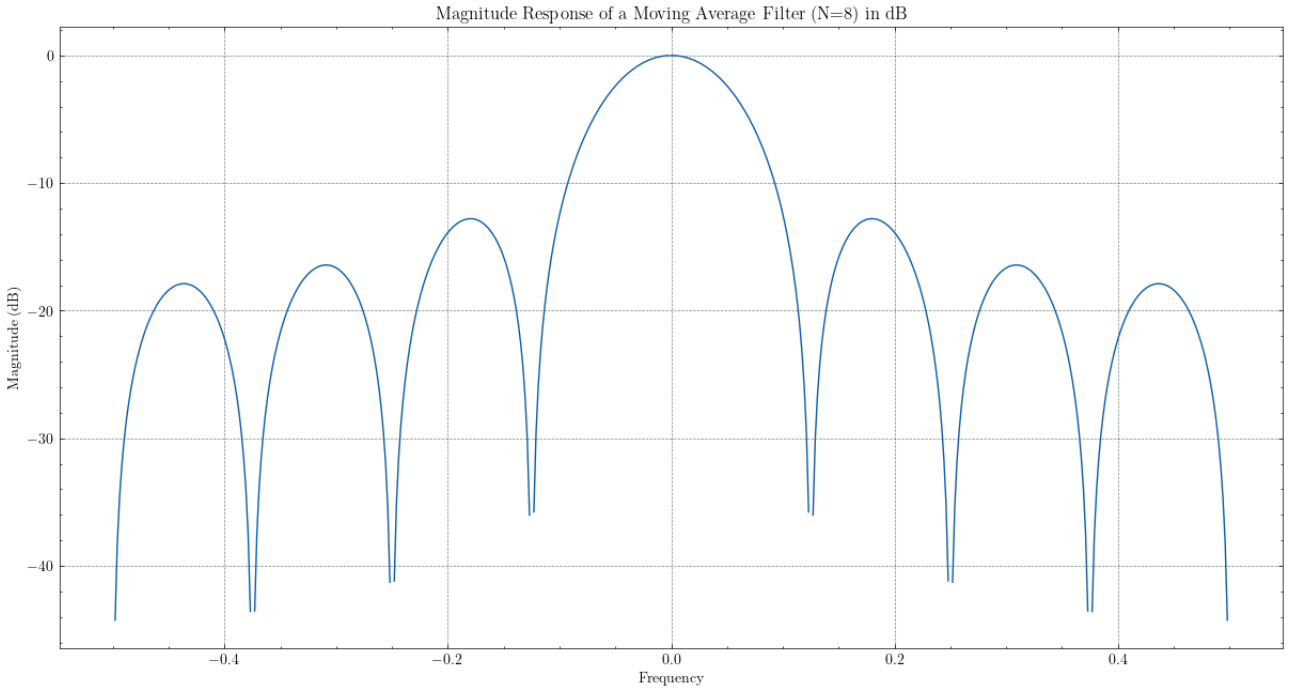

Figure 3: Frequency response moving average

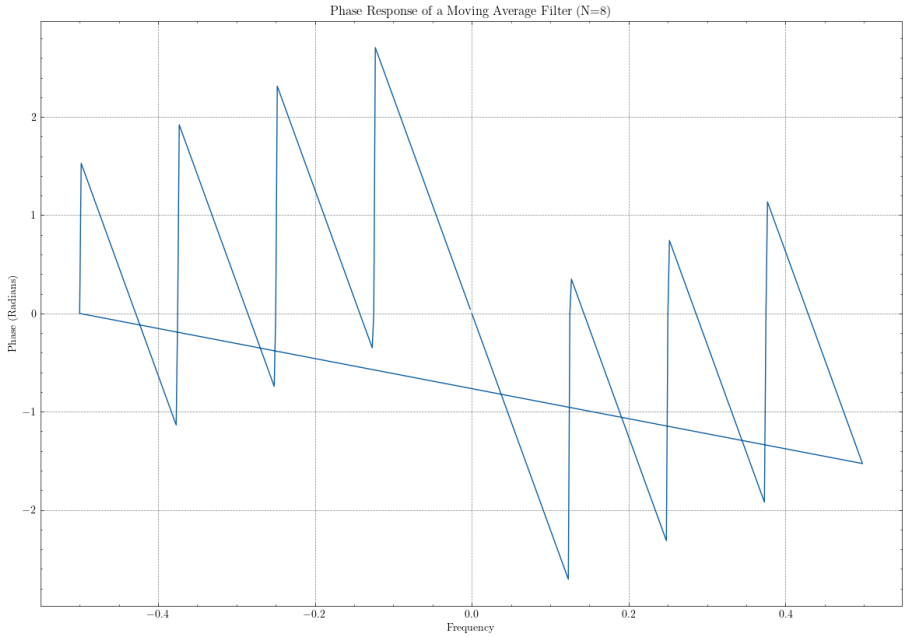

Figure 4: Phase response moving average

4 Building the Filter in Verilog

4.1 Overview

Building the filter in a hardware description language is straightforward. Figure 5 illustrates the essential functional blocks of the filter. The filter operates as follows for a filter size $\bar{N} = 8$ (i.e., $N + 1$):

Upon activation of the input strobe, the filter accepts the next input value. It maintains an array of length 8 to store the most recent data values. The filter sums these values using an internal counter that iterates over each array position, accumulating the total sum.

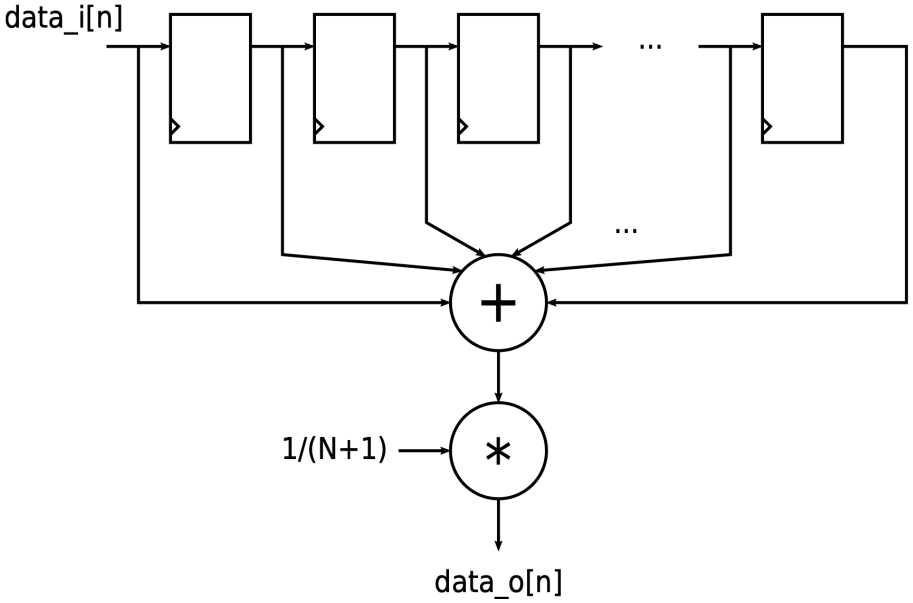

Figure 5: Block diagram filter

As new data arrives, it is placed at the beginning of the array, displacing the oldest data sample which is then discarded. Once all samples are summed, the filter computes the average. This computation is efficiently performed through a simple shift operation, given the filter's specific length. The filter then remains idle, awaiting the next data input.

This can be done by using a state machine, the 3 states are:

- Wait for Strobe
- Add
- Average

4.2 Inputs and Outputs

- **Inputs:**
  - `data_in`: Input data (Width: DATA_IN_LEN bits).
  - `strobe_in`: Strobe input signal (1 bit).
  - `clk`: Clock input (1 bit).
  - `reset`: Reset signal, active high (1 bit).
- **Outputs:**
  - `data_out`: Output data (Width: DATA_IN_LEN bits).
  - `strobe_out`: Strobe output signal (1 bit).

4.3 Parameters

- FILTER_POWER: Window length as a power of 2.
- DATA_IN_LEN: Number of bits for input data.

5 Tiny Tapeout

For the tinytapeout module there are a few things to consider. The in -and outputs have specific names and sizes. There are also only around 1000 gates available. The input clock is max $f_{max} = 50\,\text{MHz}$.

5.1 Ports

The ports of the module are used as following.

The module tt_um_moving_average_master in Verilog has several inputs, each serving a specific purpose in the filter's operation. The inputs and their bit usages are as follows:

5.1.1 Inputs

- ui_in [7:0]: This 8-bit input is used as a primary input for the moving average filter. Each bit contributes to the filter's input data.
- uio_in [7:0]: This 8-bit bidirectional input serves multiple functions:
  - uio_in[0]: Used as a strobe input. It triggers the processing of new data within the filter.
  - uio_in[1]: Unused and typically configured as an input.
  - uio_in[2] and uio_in[3]: Unused inputs, reserved for potential future use.
  - uio_in[4] and uio_in[5]: Additional bits, often used as output bits in the configuration.
  - uio_in[6] and uio_in[7]: These bits are utilized for selecting the filter width, allowing dynamic control over the filter's characteristics.
- clk: The clock input, which orchestrates the timing of the filter's operations.
- rst_n: Active low reset signal. It initializes or resets the filter's internal state when asserted.
- ena: Enable signal for the module.

5.1.2 Outputs

- uo_out [7:0]: This 8-bit output is the primary output of the moving average filter. It delivers the processed data resulting from the filter's computation.
- uio_out [7:0]: This 8-bit bidirectional output is used for multiple purposes:
  - uio_out[0]: Typically set to high impedance as it's an unused output bit.
  - uio_out[1]: Used as a strobe output. It signals when the filter has completed processing and the output data is ready.
  - uio_out[2] and uio_out[3]: Set to high impedance, indicating these are unused output bits.
  - uio_out[4] and uio_out[5]: Serve as additional output bits, often part of the 10-bit output data from the filter.
  - uio_out[6] and uio_out[7]: Also set to high impedance, indicating these are unused output bits.
- uio_oe [7:0]: This 8-bit output is used for controlling the direction (input or output) of the bidirectional pins uio_in and uio_out. It specifies whether each pin is functioning as an input or an output.

The combination of these inputs and outputs allows the tt_um_moving_average_master module to perform its designated filtering tasks with versatility and precision.

5.2 Used Filters

The `tt_um_moving_average_master` module is designed with multiple averagers, each with a distinct window length. The activation and utilization of these averagers are controlled by specific module inputs.

Number and Types of averagers

- The module contains **four distinct averagers**, each designed for different filtering characteristics:
  1. An averager with a window length of 2 (powered by 1).
  2. An averager with a window length of 4 (powered by 2).
  3. An averager with a window length of 8 (powered by 3).
  4. An additional averager with a window length of 2, chained after the 8-length averager for enhanced filtering.

5.3 Filter activation mechanism

- The selection and activation of these averagers are dynamically controlled through the `filter_select` input:
  - `filter_select [1:0]`: A 2-bit input, where each combination of bits selects a different averager. The specific mapping is as follows:
    - `00`: Activates the averager with a window length of 2.
    - `01`: Activates the averager with a window length of 4.
    - `10`: Activates the averager with a window length of 8.
    - `11`: Activates the additional averager with a window length of 2, chained after the 8-length averager.
  - This input allows for on-the-fly adjustment of the filtering characteristics, providing significant flexibility in the module's operation.

The design of `tt_um_moving_average_master` with multiple averagers and a dynamic selection mechanism enables it to be versatile and adaptable for various signal processing needs.

A RTL view of the design can be seen in 6 on the next page.

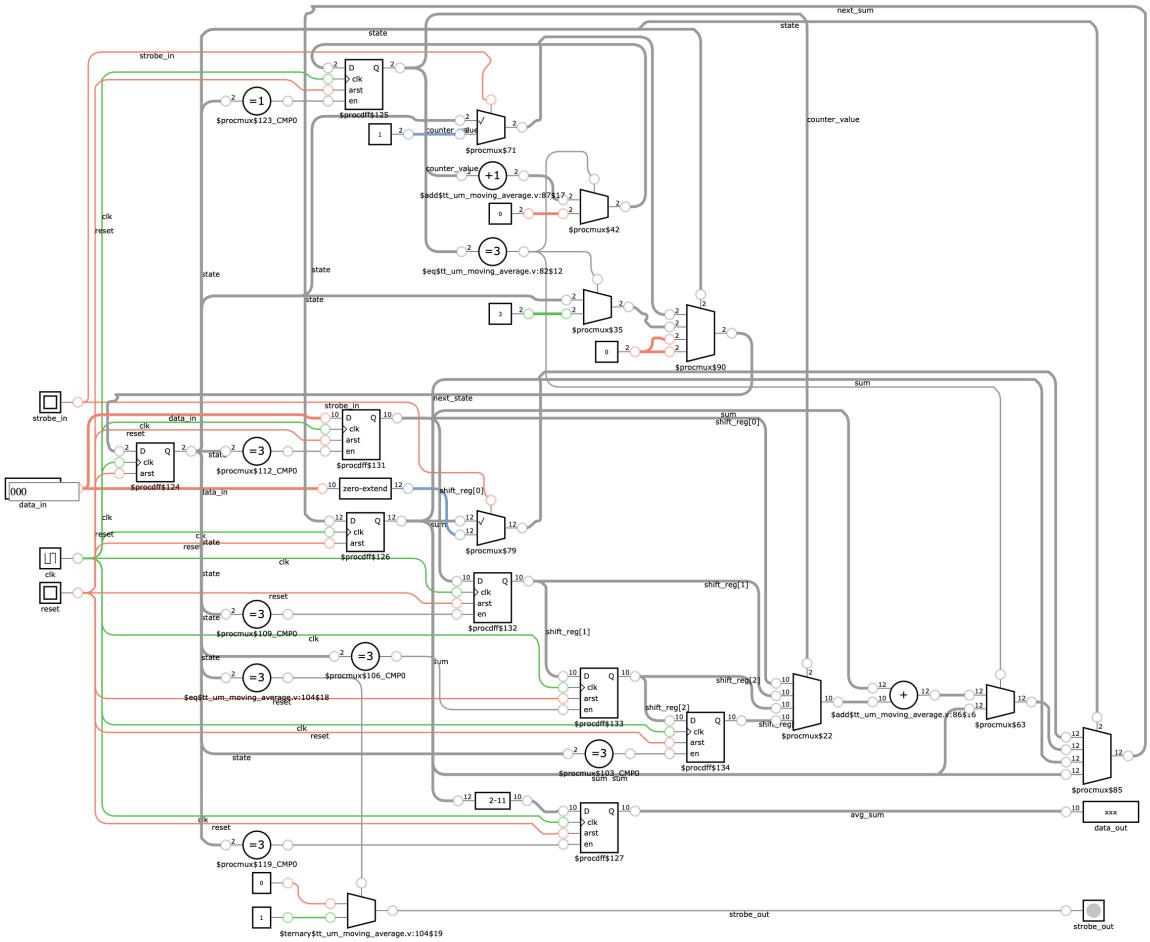

Figure 6: RTL view

6 Testing

The filters have been thoroughly tested using a Verilog test bench. The testing process focused on evaluating the impulse response of each filter configuration. The tests were conducted by sending a single 10-bit (1023) signal to the input for one input strobe and then observing the system's response.

6.1 Impulse Response

The procedure for testing the impulse response was consistent across all filter configurations.

6.1.1 Filter Configuration: N=2

The filter with a window length of two ($N = 2$) was tested first.

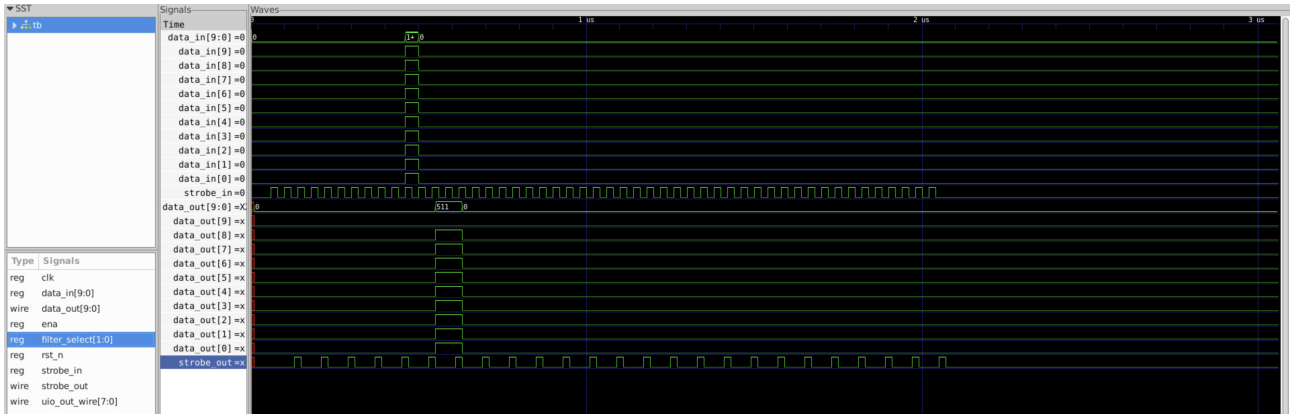

Figure 7: Impulse response for $N = 2$

The observed output confirmed the filter's expected behavior:

$$\text{data\_out}[0] = \frac{1}{2} \cdot 1023 \approx 511$$
$$\text{data\_out}[1] = \frac{1}{2} \cdot 1023 \approx 511$$
$$\text{data\_out}[n] = 0 \quad \text{for } n > 1$$

6.1.2 Filter Configuration: N=4

Next, the filter with a window of four ($N = 4$) was tested.

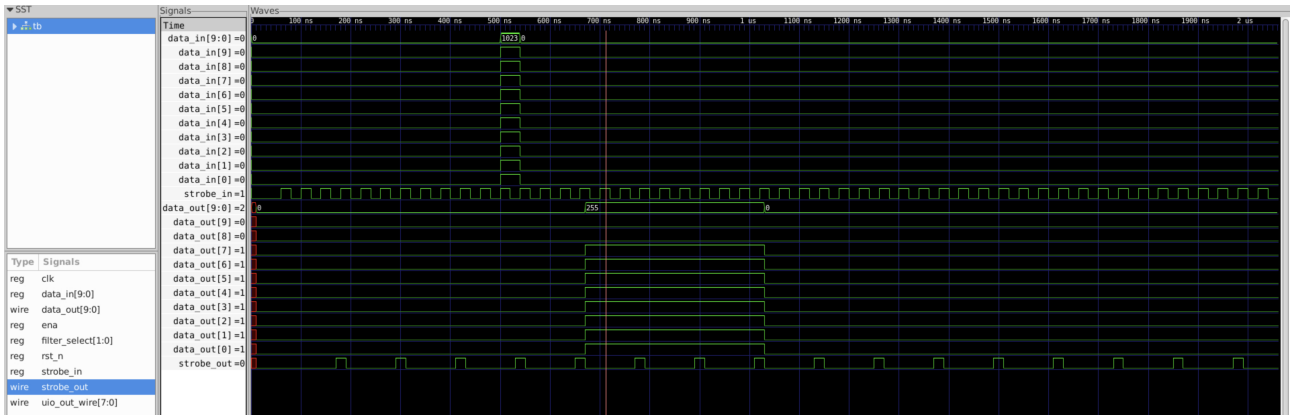

Figure 8: Impulse response for $N = 4$

The output for this filter configuration was:

$$\text{data\_out}[0:3] = \frac{1}{4} \cdot 1023 \approx 255$$
$$\text{data\_out}[n] = 0 \quad \text{for } n > 3$$

6.1.3 Filter Configuration: N=8

Finally, the filter with a window of eight ($N = 8$) was tested.

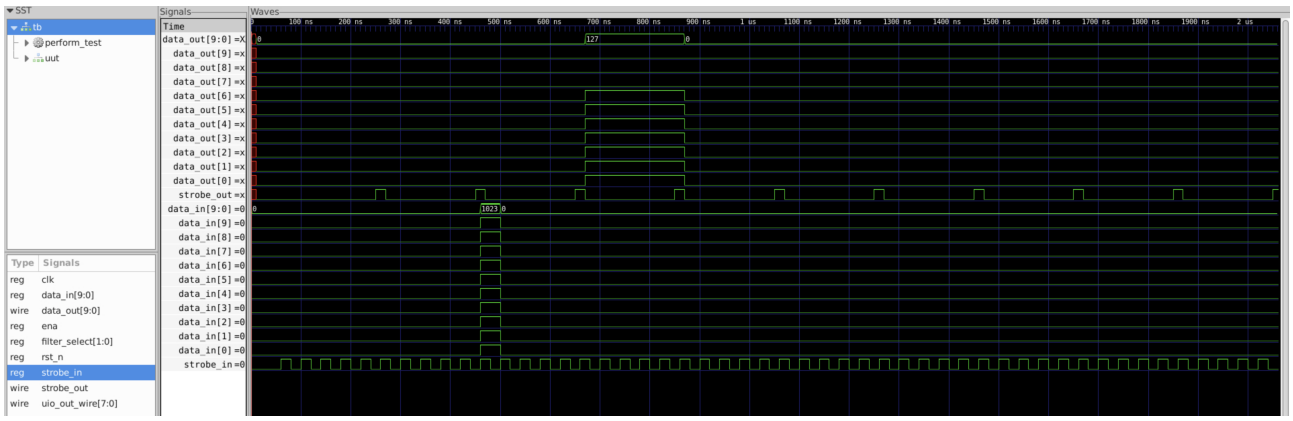

Figure 9: Impulse response for $N = 8$

The output for this configuration was:

$$\text{data\_out}[0:7] = \frac{1}{8} \cdot 1023 \approx 127$$
$$\text{data\_out}[n] = 0 \quad \text{for } n > 7$$

6.2 Step Response

The step response of the filters was also tested using a Verilog test bench. The test involved sending a constant 10-bit (1023) signal to the filter and monitoring the system's response over time.

The procedure for each filter was consistent. The closed-form expression for the step response of a moving average filter with a window size N is:

$$s(n) = \begin{cases} \frac{n+1}{N} & \text{for } 0 \leq n < N, \\ 1 & \text{for } n \geq N. \end{cases}$$

6.2.1 Filter Configuration: N=2

The filter with a window length of two ($N = 2$) was tested first.

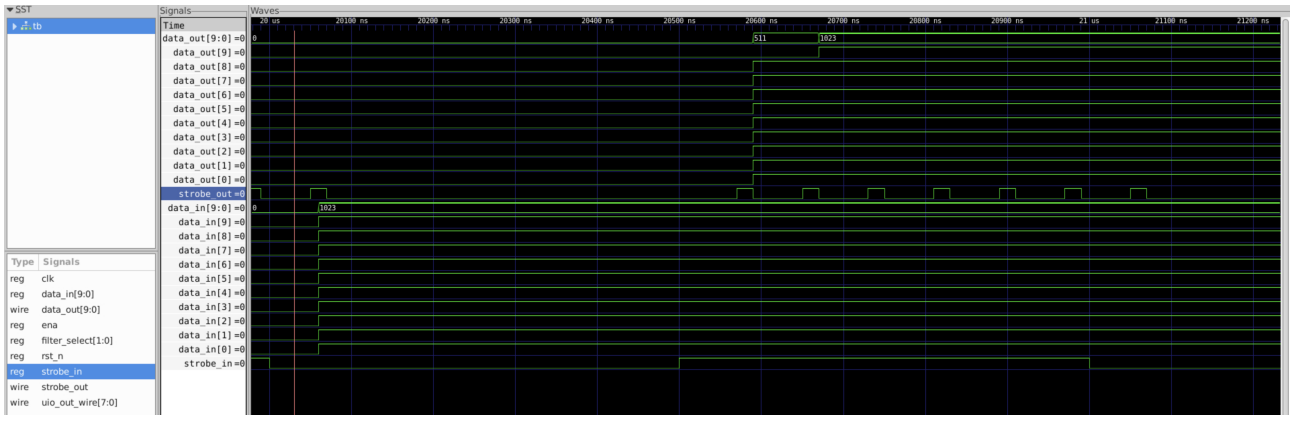

Figure 10: Step response for $N = 2$

The observed output was as expected:

$$\text{data\_out}[0] = \frac{1}{2} \cdot 1023 \approx 511$$
$$\text{data\_out}[1] = \frac{2}{2} \cdot 1023 = 1023$$
$$\text{data\_out}[n] = 1023 \quad \text{for } n \geq 2$$

6.2.2 Filter Configuration: N=4

Next, the filter with a window of four ($N = 4$) was tested.

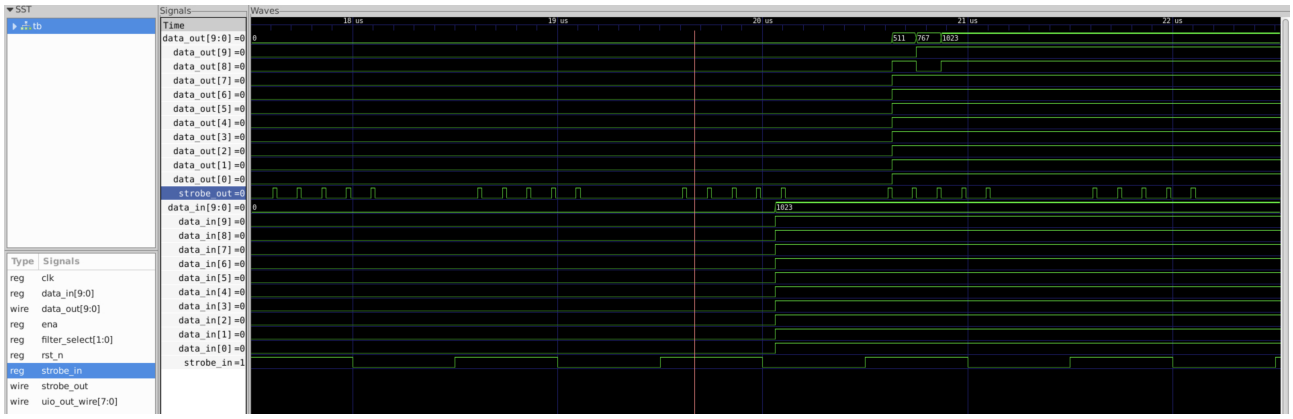

Figure 11: Step response for $N = 4$

The output for this configuration was:

$$\text{data\_out}[0] = \frac{1}{4} \cdot 1023 \approx 255$$
$$\text{data\_out}[1] = \frac{2}{4} \cdot 1023 \approx 511$$
$$\text{data\_out}[2] = \frac{3}{4} \cdot 1023 \approx 767$$
$$\text{data\_out}[3] = \frac{4}{4} \cdot 1023 = 1023$$
$$\text{data\_out}[n] = 1023 \quad \text{for } n \geq 4$$

6.2.3 Filter Configuration: N=8

Finally, the filter with a window of eight ($N = 8$) was tested.

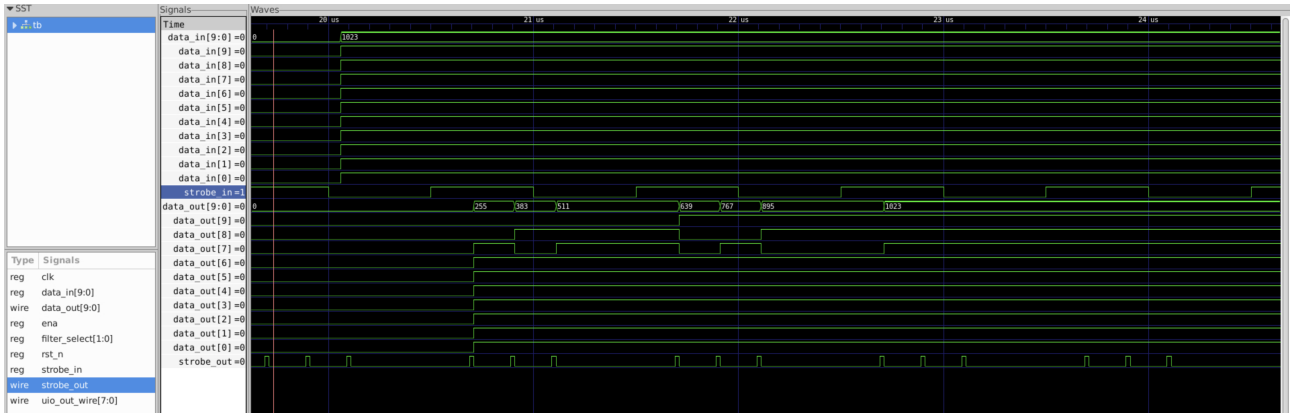

Figure 12: Step response for $N = 8$

The output for this configuration was:

$$\text{data\_out}[0] = \frac{1}{8} \cdot 1023 \approx 127$$
$$\text{data\_out}[1] = \frac{2}{8} \cdot 1023 \approx 255$$
$$\text{data\_out}[2] = \frac{3}{8} \cdot 1023 \approx 383$$
$$\text{data\_out}[3] = \frac{4}{8} \cdot 1023 \approx 511$$
$$\text{data\_out}[4] = \frac{5}{8} \cdot 1023 \approx 639$$
$$\text{data\_out}[5] = \frac{6}{8} \cdot 1023 \approx 767$$
$$\text{data\_out}[6] = \frac{7}{8} \cdot 1023 \approx 895$$
$$\text{data\_out}[7] = \frac{8}{8} \cdot 1023 = 1023$$
$$\text{data\_out}[n] = 1023 \quad \text{for } n \geq 8$$

6.3 Testing with Various Signals

In this section, we evaluate the performance of the moving average filter by testing it with different types of input signals. These tests aim to assess the filter's ability to remove high-frequency noise and its effectiveness in processing signals with varying frequency components.

6.3.1 Sinwave with Noise

A noisy sine wave was used as the first test signal. The original signal, shown in Figure 13, was generated using a Python script and contains both the sine wave and additive white Gaussian noise. The expectation is that the filter will remove the high-frequency noise components.

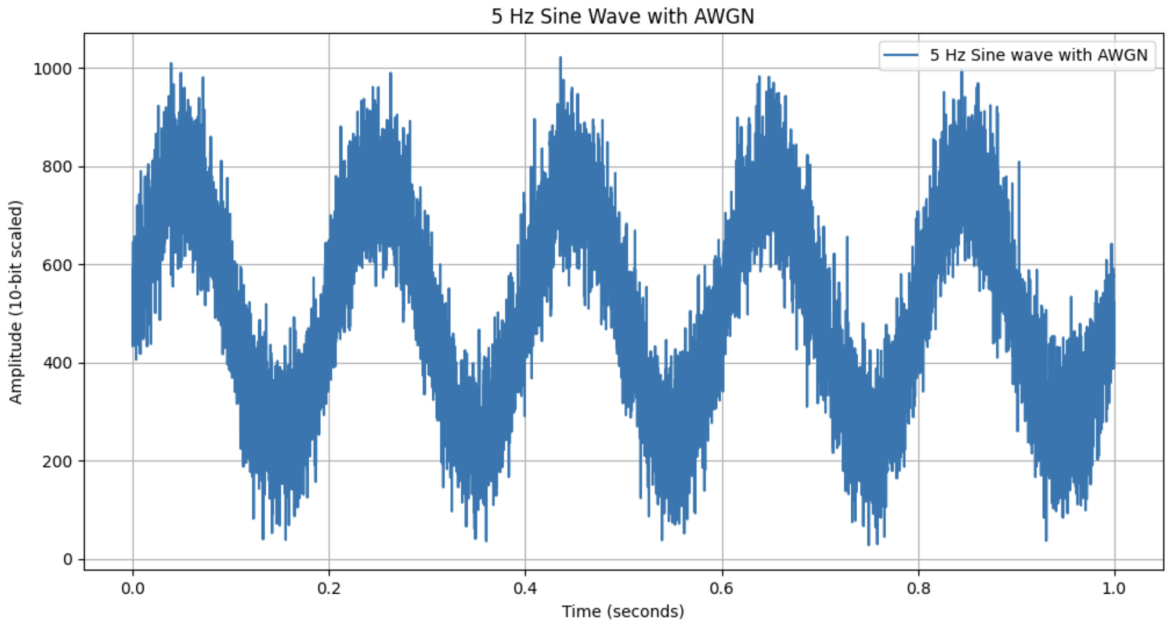

Figure 13: Noisy Sinwave Input Signal

Filtering with $N = 8$ The filtered signal using a filter size of $N = 8$ is displayed in Figure 14, with its spectrum shown in Figure 15. As expected, the filter effectively removes the high-frequency noise.

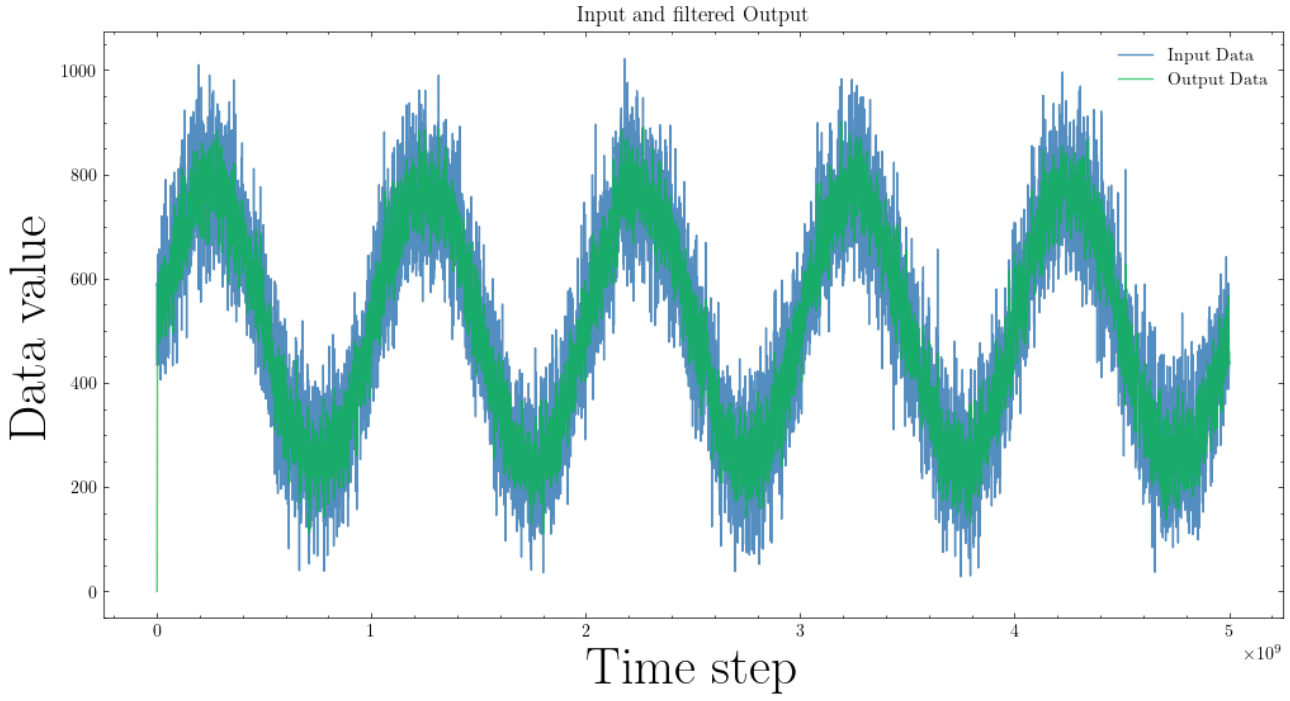

Figure 14: Filtered Sinwave with $N = 8$

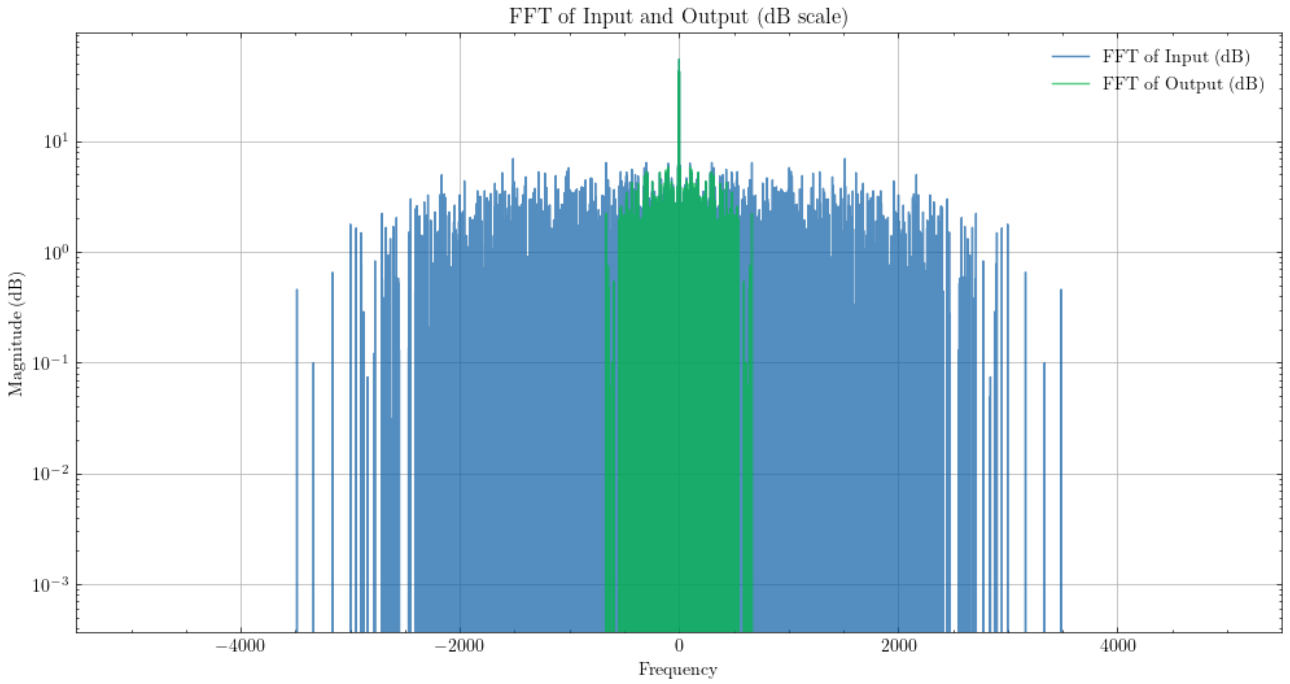

Figure 15: Spectrum of Filtered Sinwave with $N = 8$

Filtering with $N = 16$ Using a larger filter size of $N = 16$ further improves the performance, as shown in Figures 16 and 17, with even better noise reduction.

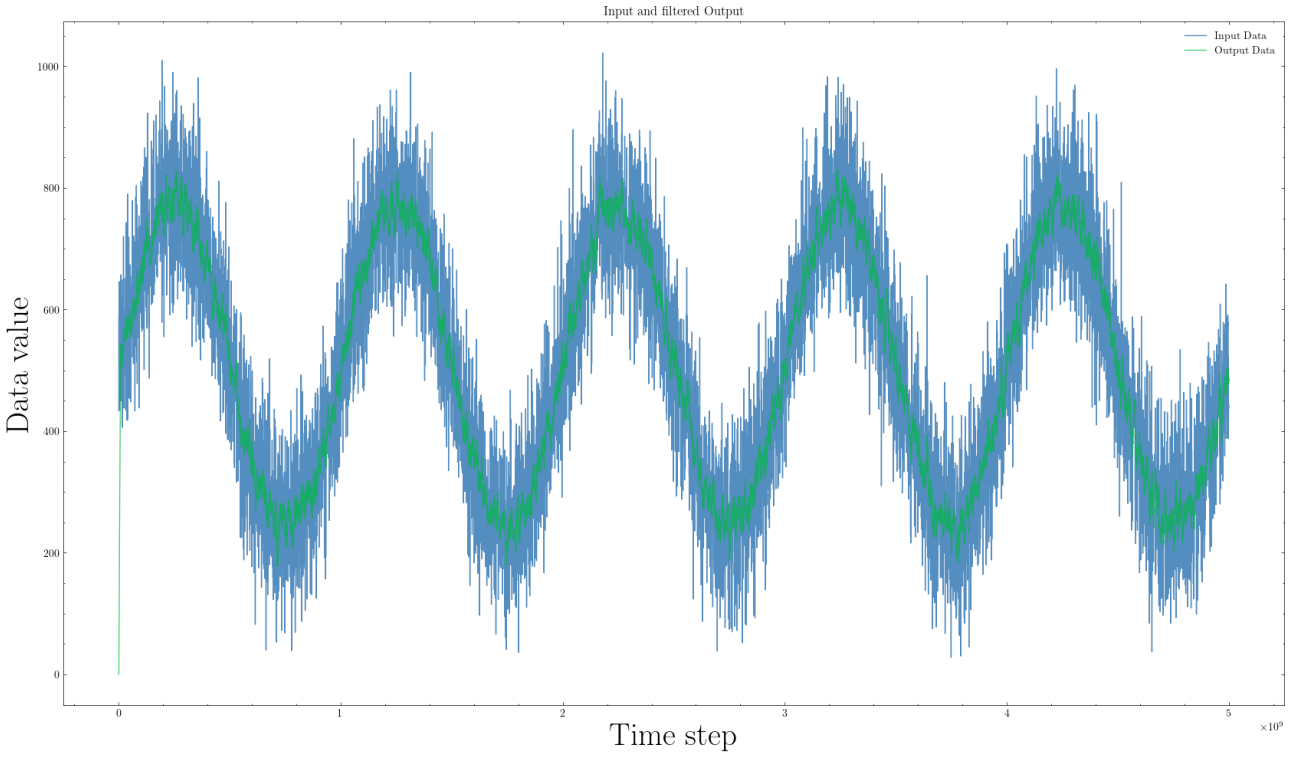

Figure 16: Filtered Sinwave with $N = 16$

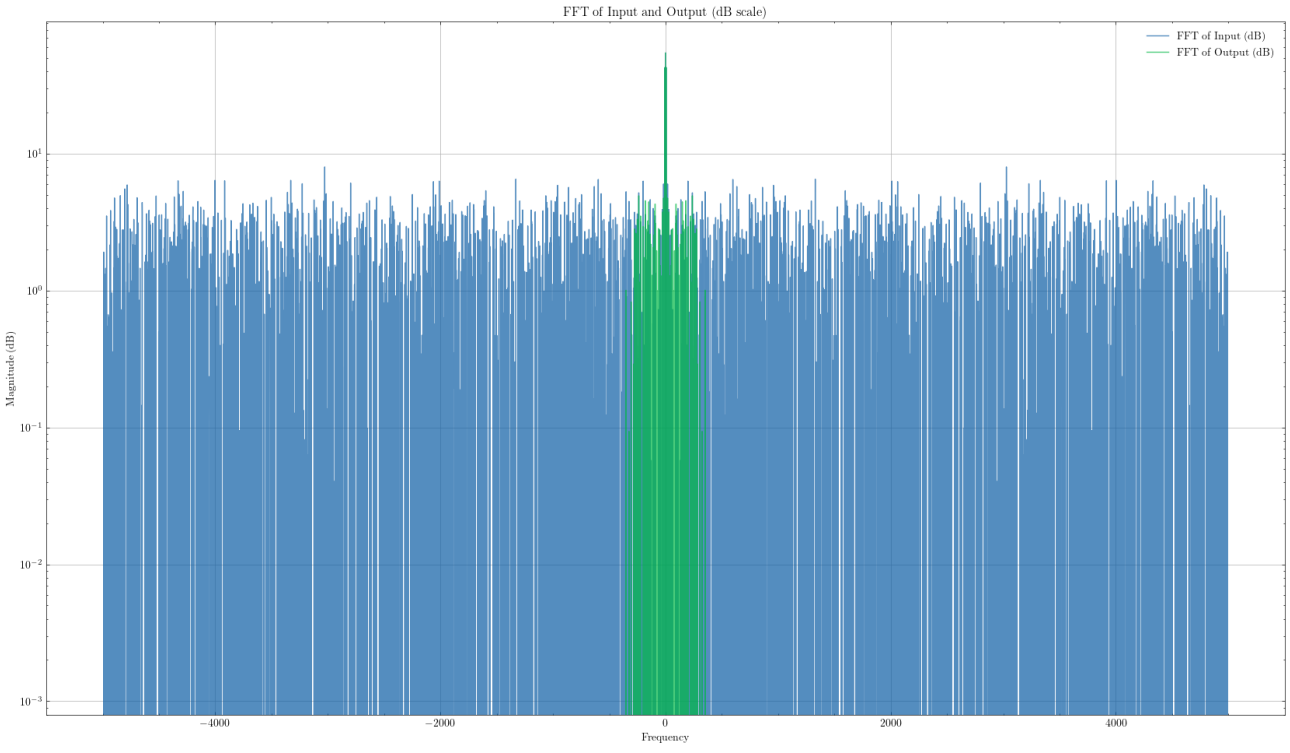

Figure 17: Spectrum of Filtered Sinwave with $N = 16$

6.3.2 Testing with a Sawtooth Signal

The filter's response to a sawtooth signal was also tested, as shown in Figures 18 and 19. The filter effectively damps rapid signal changes, as evidenced by the attenuated high-frequency components in the output.

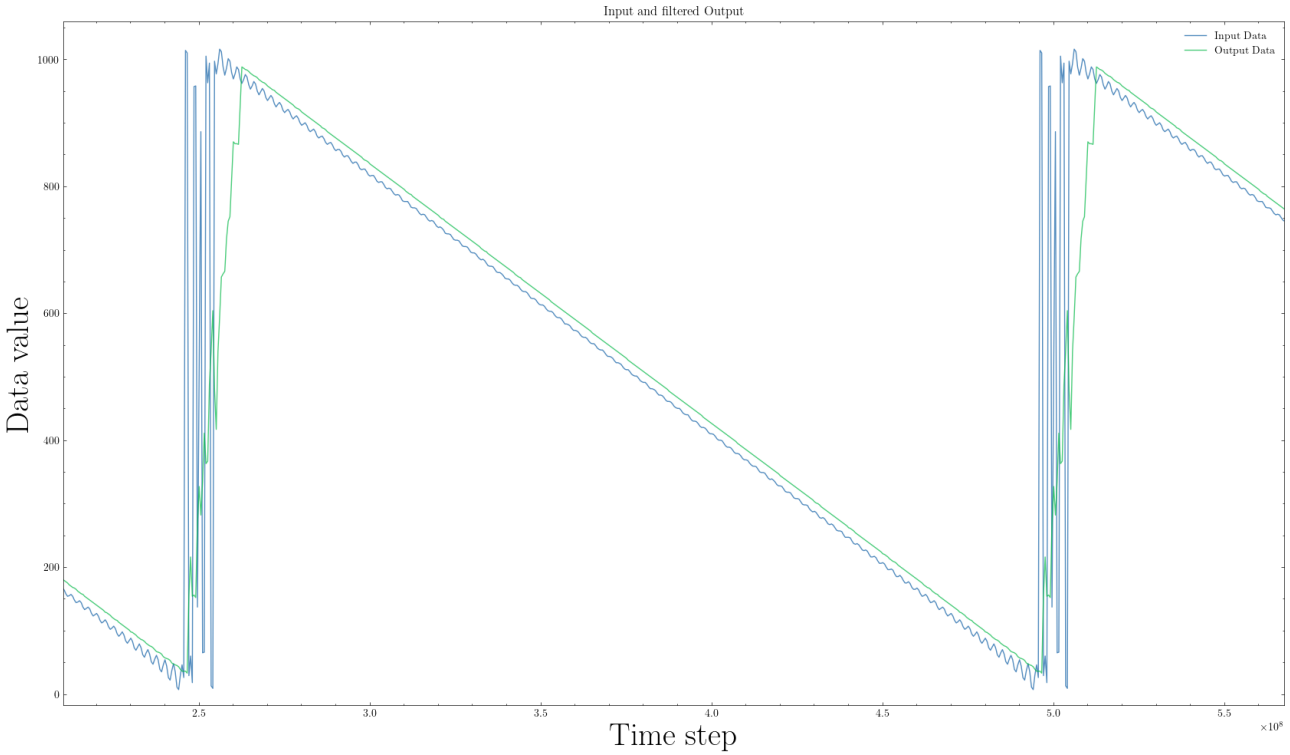

Figure 18: Filtered Sawtooth Signal

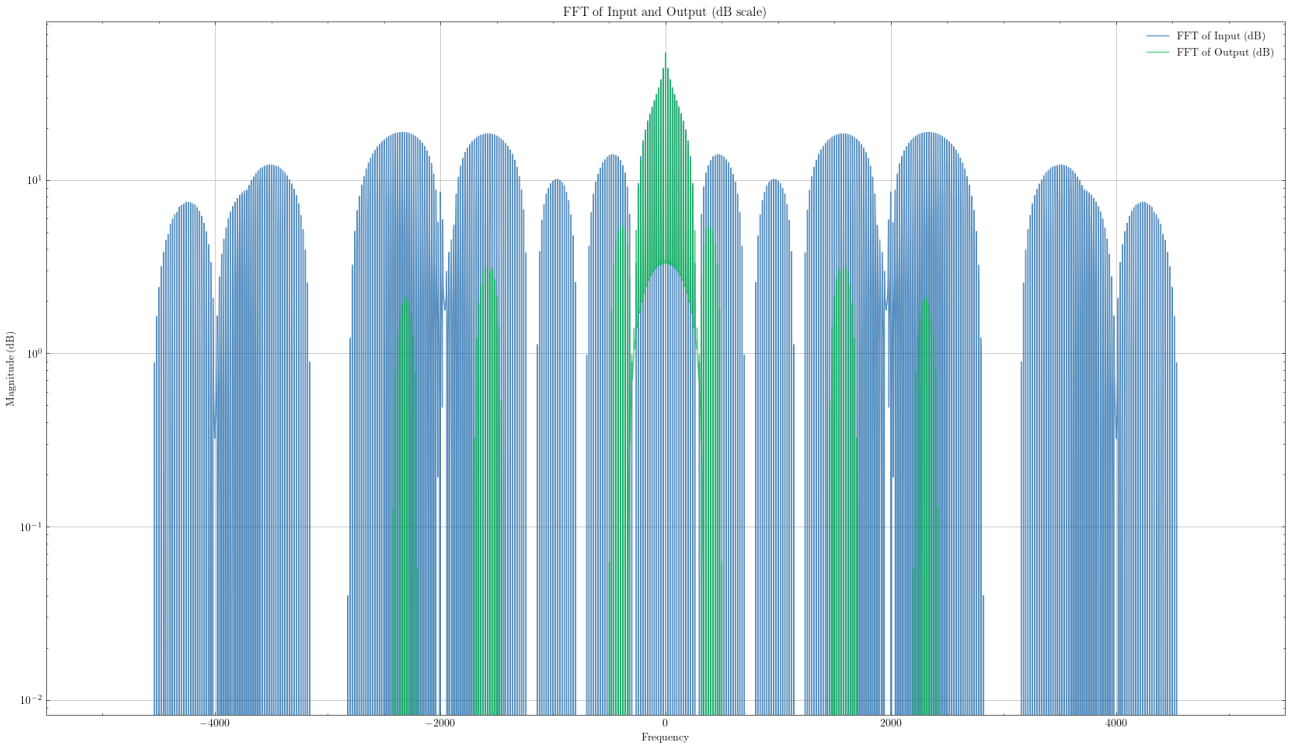

Figure 19: Spectrum of Filtered Sawtooth Signal

6.3.3 Testing with a Sinwave with Harmonics

Finally, a sine wave with harmonics was used as a test signal. The filter's response, depicted in Figure 20, demonstrates its capability to dampen fast-changing components, leading to a smoother output signal. A more detailed picure can be seen in 21 on the following page.

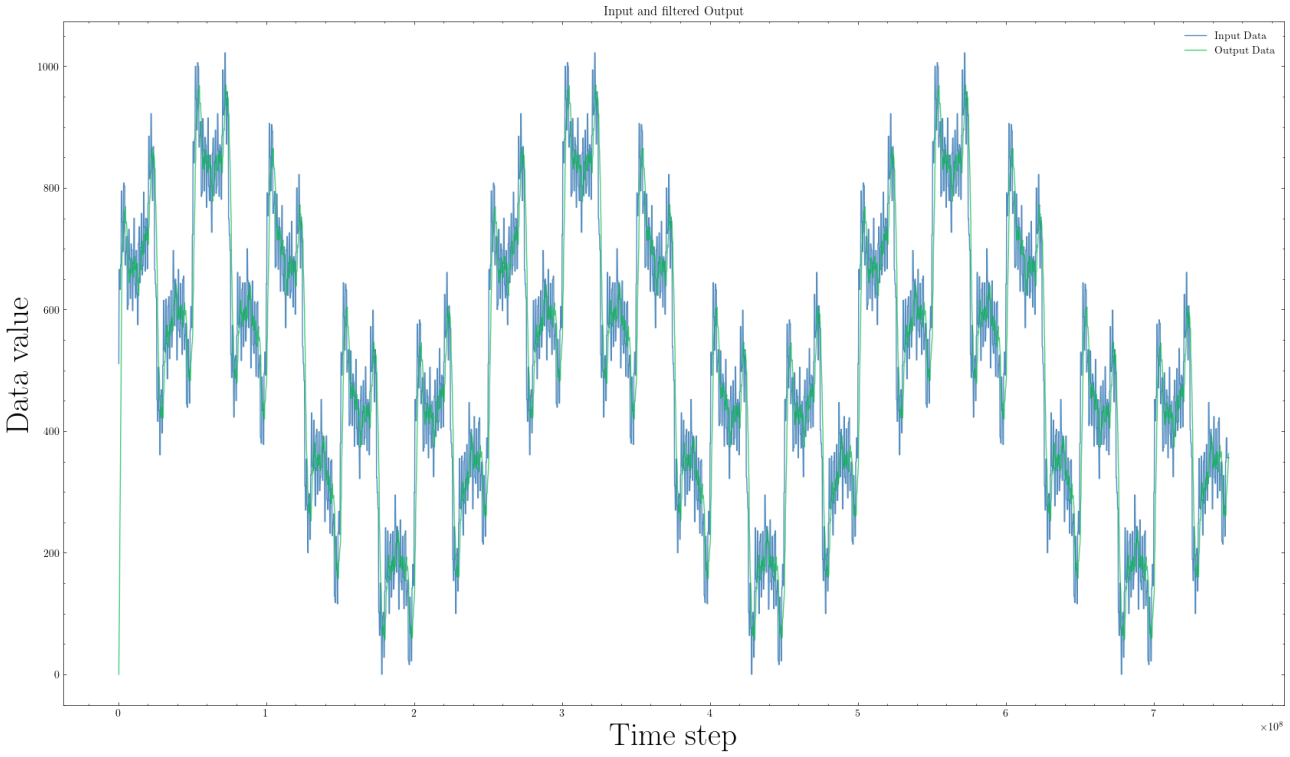

Figure 20: Filtered Sinwave with Harmonics

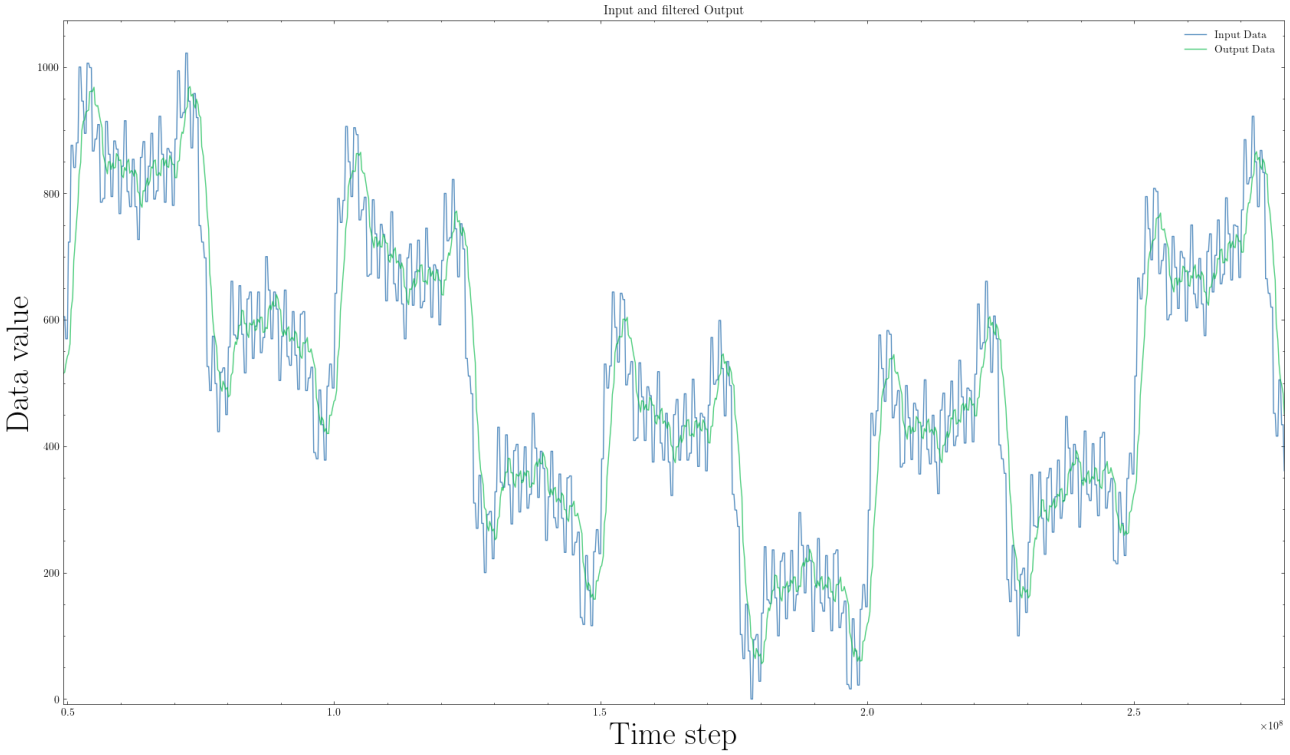

Figure 21: Detailed filtered Sinwave with Harmonics

These tests confirm that the moving average filter performs effectively in reducing high-frequency noise and smoothing out signals, making it suitable for applications requiring such signal processing.

7 GDS

The code was uploaded to GitHub where the chip was build. All actions were run successful as seen in 22.

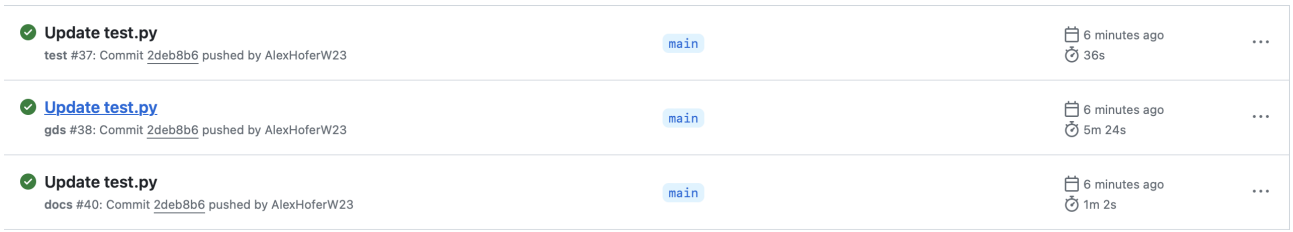

Figure 22: GitHub actions

The finished chip can was produced, and can be seen in 23.

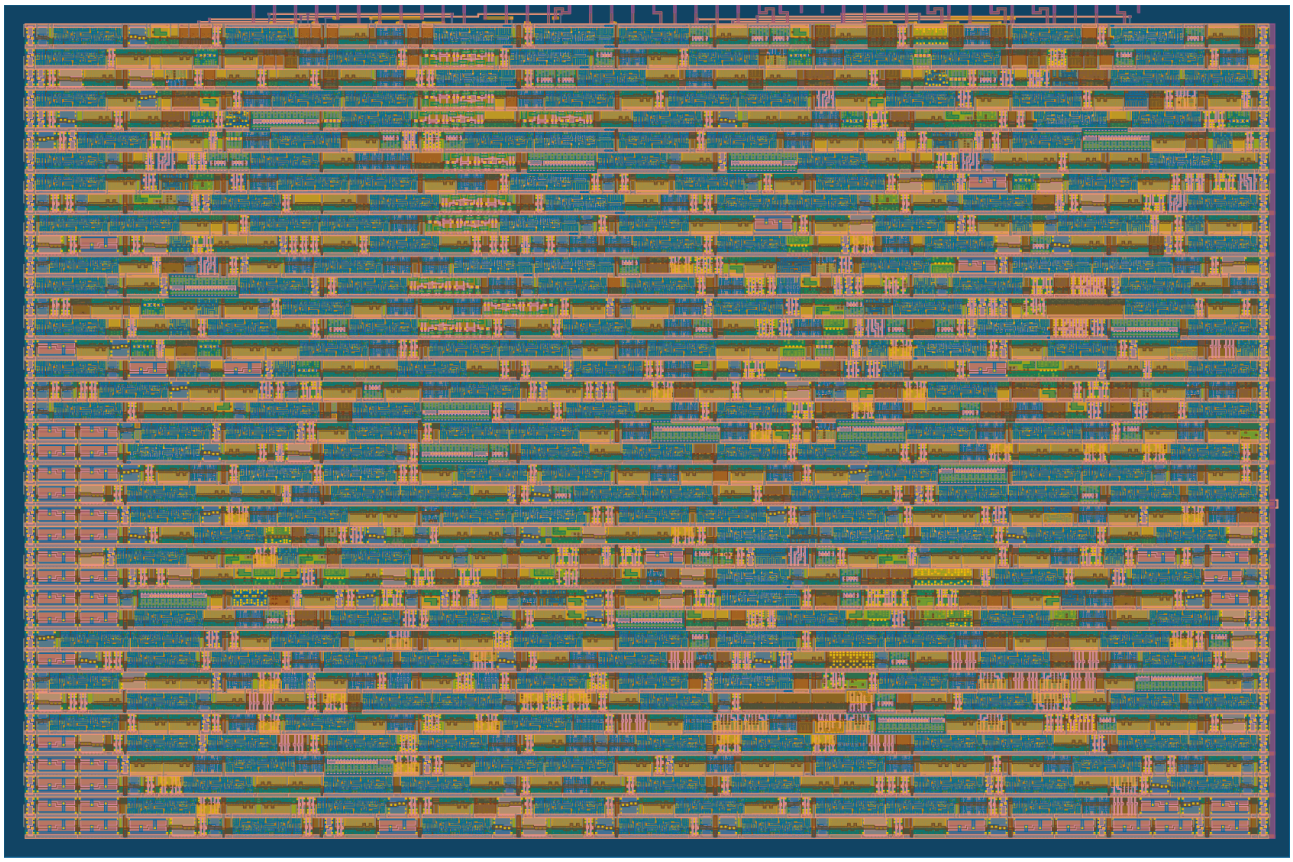

Figure 23: GDS Render

The cell usage can be seen in 24 on the next page.

Linter output

```
%Warning-UNUSEDSIGNAL: /work/src/tt_um_moving_average_master.v:14:22: Bits of signal are not used: 'uio_in'[5:4,1]
                                                                      : ... In instance tt_um_moving_average_master
   14 |     input wire [7:0] uio_in,
      |                      ^~~~~~
                       ... For warning description see https://verilator.org/warn/UNUSEDSIGNAL?v=5.009
                       ... Use "/* verilator lint_off UNUSEDSIGNAL */" and lint_on around source to disable this message.
%Warning-UNUSEDSIGNAL: /work/src/tt_um_moving_average_master.v:19:16: Signal is not used: 'ena'
                                                                      : ... In instance tt_um_moving_average_master
   19 |     input wire ena
      |                ^~~
```

Figure 25: Linter output

Routing stats

Utilisation (%)	Wire length (um)
79.71	27784

Cell usage by Category

Category	Cells	Count
Fill	decap fill	726
Buffer	buf clkbuf	309
Tap	tapvpwrvgnd	246
Multiplexer	mux4 mux2	237
Flip Flops	dfrtp	233
Combo Logic	a21oi a41o o32a o21ba and2b and3b nor3b and4b a221o a21o o31a or2b o21a o2bb2a a22o o41a a2111o a2bb2o o311a a211o o22a a31o o31ai a211oi o21ai a21bo or3b a31oi nor2b o211ai nand3b	194
Misc	dlygate4sd3 dlymetal6s2s conb	96
NOR	nor2 xnor2	56
NAND	nand2 nand3 nand2b nand4	43
OR	or3 or2 or4 xor2	35
AND	and3 and2 a21boi	33
Inverter	inv	15
Diode	diode	5
Clock	clkinv	1

1257 total cells (excluding fill and tap cells)

Figure 24: Cell usage

In 25, the warnings of the GDS can be found. There are only two left, and those can be ignored. The unused signal can be ignored, since those pins are used as outputs. The enable is also not used in the module, this is used by TinyTapeout.

8 GitHub

The link to the GitHub repository can be found here: GitHub Repository

Appendix

A Complete Code Listings

This appendix contains the complete code listings used for the implementation and testing of the moving average filter.

A.1 Moving Average Filter Implementation

```
`default_nettype none

module tt_um_moving_average #(
    parameter FILTER_POWER = 2,  // Window length as a power of 2
    parameter DATA_IN_LEN = 10   // Number of bits for input data
)(
    input wire [DATA_IN_LEN-1:0] data_in,     // Input data
    output wire [DATA_IN_LEN-1:0] data_out,    // Output data
    input wire strobe_in,                     // Strobe input signal
    output wire strobe_out,                   // Strobe output signal
    input wire clk,                           // Clock
    input wire reset                          // Reset signal (active high)
);

    localparam FILTER_SIZE = 1 << FILTER_POWER; // Filter size
    localparam SUM_WIDTH = DATA_IN_LEN + FILTER_POWER;
    localparam PAD_WIDTH = FILTER_POWER;

    // FSM states
    reg [1:0] state, next_state;
    localparam WAIT_FOR_STROBE = 2'b00;
    localparam ADD             = 2'b01;
    localparam AVERAGE         = 2'b11;

    // Data buffer
    reg [DATA_IN_LEN - 1:0] shift_reg [FILTER_SIZE - 1:0];
    reg [DATA_IN_LEN - 1:0] next_shift_reg [FILTER_SIZE - 1:0];

    // Other signals
    reg [FILTER_POWER - 1:0] counter_value, next_counter_value;
    reg [SUM_WIDTH - 1:0] sum, next_sum;
    reg [DATA_IN_LEN - 1:0] avg_sum, next_avg_sum;

    // Main FSM logic
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            counter_value <= 0;
            state <= WAIT_FOR_STROBE;
            sum <= 0;
            avg_sum <= 0;
            for (integer i = 0; i < FILTER_SIZE; i = i + 1) begin
            shift_reg[i] <= 0;
            end
        end else begin
            counter_value <= next_counter_value;
            state <= next_state;
            sum <= next_sum;
            avg_sum <= next_avg_sum;
            for (integer i = 0; i < FILTER_SIZE; i = i + 1) begin
            shift_reg[i] <= next_shift_reg[i];
            end
        end
    end
```

```
54
55      // FSM
56      always @(*) begin
57          next_state = state;
58          for (integer i = 0; i < FILTER_SIZE; i = i + 1) begin
59              next_shift_reg[i] = shift_reg[i];
60          end
61          next_sum = sum;
62          next_avg_sum = avg_sum;
63          next_counter_value = counter_value;
64
65          case(state)
66              WAIT_FOR_STROBE: begin
67                  if (strobe_in) begin
68                      next_sum = {{PAD_WIDTH{1'b0}}, data_in}; //zero padding
69                      next_state = ADD;
70                  end
71              end
72
73              ADD: begin
74                  if (counter_value == FILTER_SIZE - 1) begin
75                      next_counter_value = 0;
76                      next_state = AVERAGE;
77                  end else begin
78                      next_sum = sum + {{PAD_WIDTH{1'b0}}, shift_reg[counter_value]};
79                      next_counter_value = counter_value + 1'b1;
80                  end
81              end
82
83              AVERAGE: begin
84                  next_shift_reg[0] = data_in;
85                  for (integer i = 1; i < FILTER_SIZE; i = i + 1) begin
86                      next_shift_reg[i] = shift_reg[i - 1];
87                  end
88                  next_avg_sum = sum[SUM_WIDTH-1:FILTER_POWER];
89                  next_state = WAIT_FOR_STROBE;
90              end
91              default: next_state = WAIT_FOR_STROBE;
92          endcase
93      end
94
95      assign data_out = avg_sum;
96      assign strobe_out = (state == AVERAGE) ? 1'b1 : 1'b0;
97
98  endmodule
```

A.2 Moving Average Tiny Tapeout

```
1   `default_nettype none
2
3   module tt_um_moving_average_master(
4       input wire [7:0] ui_in,     // Dedicated inputs - Input for the moving averager
5       output wire [7:0] uo_out,   // Dedicated outputs - Output for the moving averager
6       input wire [7:0] uio_in,    // IOs: Bidirectional Input path
7       output wire [7:0] uio_out,  // IOs: Bidirectional Output path
8       output wire [7:0] uio_oe,   // IOs: Bidirectional Enable path (active high: 0=input
9       input wire clk,             // Clock
10      input wire rst_n,           // Reset (active low)
11      input wire ena
12  );
13
14      localparam DATA_IN_LEN = 10;
15      wire reset = !rst_n;
```

```
16      wire [DATA_IN_LEN - 1:0] data_i = {uio_in[3:2], ui_in};
17      wire strobe_i = uio_in[0];
18      wire [1:0] filter_select = uio_in[7:6]; // Filter width control
19
20      // Filter instantiations
21      wire [DATA_IN_LEN - 1 :0] filter_out_2, filter_out_4, filter_out_8, filter_out_8_e
22      wire strobe_out_2, strobe_out_4, strobe_out_8, strobe_out_8_extra;
23
24      // Filter with window length 2
25      tt_um_moving_average #(.FILTER_POWER(1), .DATA_IN_LEN(10)) filter_2 (
26          .data_in(data_i),
27          .data_out(filter_out_2),
28          .strobe_in(strobe_i),
29          .strobe_out(strobe_out_2),
30          .clk(clk),
31          .reset(reset)
32      );
33
34      // Filter with window length 4
35      tt_um_moving_average #(.FILTER_POWER(2), .DATA_IN_LEN(10)) filter_4 (
36          .data_in(data_i),
37          .data_out(filter_out_4),
38          .strobe_in(strobe_i),
39          .strobe_out(strobe_out_4),
40          .clk(clk),
41          .reset(reset)
42      );
43
44
45
46      tt_um_moving_average #(.FILTER_POWER(3), .DATA_IN_LEN(10)) filter_8 (
47          .data_in(data_i),
48          .data_out(filter_out_8),
49          .strobe_in(strobe_i),
50          .strobe_out(strobe_out_8),
51          .clk(clk),
52          .reset(reset)
53      );
54
55      // Additional filter (2-length) after 8-length filter
56      tt_um_moving_average #(.FILTER_POWER(1), .DATA_IN_LEN(10)) filter_8_extra (
57          .data_in(filter_out_8),
58          .data_out(filter_out_8_extra),
59          .strobe_in(strobe_out_8),
60          .strobe_out(strobe_out_8_extra),
61          .clk(clk),
62          .reset(reset)
63      );
64
65      // Multiplexer for selecting output based on filter_select
66      reg [9:0] selected_filter_out;
67      reg selected_strobe_out;
68
69      // Multiplexer for selecting output based on filter_select
70      always @(posedge clk or posedge reset) begin
71          if (reset) begin
72              selected_filter_out <= 0;
73              selected_strobe_out <= 0;
74          end else begin
75              case(filter_select)
76
77                  2'b00: begin
78                      selected_filter_out <= filter_out_2;
```

```
79                      selected_strobe_out <= strobe_out_2;
80                  end
81
82                  2'b01:
83                      begin
84                          selected_filter_out <= filter_out_4;
85                          selected_strobe_out <= strobe_out_4;
86                      end
87
88
89                  2'b10:
90                      begin
91                          selected_filter_out <= filter_out_8;
92                          selected_strobe_out <= strobe_out_8;
93                      end
94
95
96                  2'b11: begin
97                      selected_filter_out <= filter_out_8_extra;
98                      selected_strobe_out <= strobe_out_8_extra;
99                  end
100
101                 default: begin
102                     selected_filter_out <= 0;
103                     selected_strobe_out <= 0;
104                 end
105             endcase
106         end
107     end
108
109
110     // uio_in  and uio_out pin usage:
111     // uio_in[0] - Strobe input (configured as input)
112     // uio_in[1] - Unused (configured as input)
113     // uio_in[2] - Unused (configured as input)
114     // uio_in[3] - Unused (configured as input)
115     // uio_in[4] - Additional output bit (configured as output)
116     // uio_in[5] - Additional output bit (configured as output)
117     // uio_in[6] - Filter width input
118     // uio_in[7] - Filter width input
119
120     // uio_out[0] - High impedance (unused output bit)
121     // uio_out[1] - Strobe output
122     // uio_out[2] - High impedance (unused output bit)
123     // uio_out[3] - High impedance (unused output bit)
124     // uio_out[4] - Additional output bit (part of 10-bit output)
125     // uio_out[5] - Additional output bit (part of 10-bit output)
126     // uio_out[6] - High impedance (unused output bit)
127     // uio_out[7] - High impedance (unused output bit)
128
129
130     // Assigning output to selected filter output
131     assign {uio_out[5:4], uo_out} = selected_filter_out;
132     assign uio_out[1] = selected_strobe_out;
133
134     // IOs configuration
135     assign uio_oe[0] = 1'b0; // Strobe input set as input
136     assign uio_oe[1] = 1'b1; // Strobe output set as output
137     assign uio_oe[3:2] = 2'b00; // Additional input bits for data
138     assign uio_oe[5:4] = 2'b11; // Additional output bits for data
139     assign uio_oe[7:6] = 2'b00; // Filter select input
140
141     // Default configuration for unused output bits
```

```
    assign uio_out[7:6] = 2'bz;
    assign uio_out[3:2] = 2'bz;
    assign uio_out[0] = 1'bz;

endmodule
```

A.3 Impulse Response Test Bench

```
`default_nettype none
`timescale 1ns/1ps

module tb;
    // Testbench Signals
    reg [9:0] data_in = 0;        // 10-bit Input for the moving averager
    wire [9:0] data_out;          // 10-bit Output for the moving averager
    reg strobe_in = 0;            // Strobe Input
    wire strobe_out;              // Strobe Output
    reg clk = 0;                  // Clock
    reg rst_n = 1;                // Reset (active low)
    reg ena = 1;                  // Enable
    reg [1:0] filter_select = 0; // Filter selection bits

    wire [7:0] uio_out_wire; // Wire for uio_out port

    // Instantiate the Unit Under Test (UUT)
    tt_um_moving_average_master uut (
        .ui_in(data_in[7:0]),
        .uo_out(data_out[7:0]),
        .uio_in({filter_select, 2'b00, data_in[9:8], 1'b0, strobe_in}),
        .uio_out(uio_out_wire),
        .uio_oe(),
        .clk(clk),
        .rst_n(rst_n),
        .ena(ena)
    );

    assign {data_out[9:8], strobe_out} = {uio_out_wire[5:4], uio_out_wire[1]};

    // Clock Generation
    always #10 clk = ~clk;

    initial begin
        $dumpfile ("tb.vcd");
        $dumpvars (0, tb);

        // Reset the system
        rst_n = 0;
        #20;
        rst_n = 1;
        #40;

        // Test for Filter Size 2
        filter_select = 2'b00;
        perform_test();
        #1000;

        $finish;
    end

    task perform_test;
        for (integer i = 0; i < 50; i++) begin
            if (i == 10) begin
                data_in = 1023;
```

```
                strobe_in = 1; // Strobe signal active
                #20;
                strobe_in = 0; // Strobe signal inactive
                #20;
            end else begin
                data_in = 0; // Generate 10-bit test data
                strobe_in = 1; // Strobe signal active
                #20;
                strobe_in = 0; // Strobe signal inactive
                #20;
            end
        end
    endtask
endmodule
```

A.4 Step Response Test Bench

```
`default_nettype none
`timescale 1ns/1ps

module tb;
    // Testbench Signals
    reg [9:0] data_in = 0;        // 10-bit Input for the moving averager
    wire [9:0] data_out;          // 10-bit Output for the moving averager
    reg strobe_in = 0;            // Strobe Input
    wire strobe_out;              // Strobe Output
    reg clk = 0;                  // Clock
    reg rst_n = 1;                // Reset (active low)
    reg ena = 1;                  // Enable
    reg [1:0] filter_select = 0; // Filter selection bits

    wire [7:0] uio_out_wire;

    // (UUT)
    tt_um_moving_average_master uut (
        .ui_in(data_in[7:0]),
        .uo_out(data_out[7:0]),
        .uio_in({filter_select, 2'b00, data_in[9:8], 1'b0, strobe_in}),
        .uio_out(uio_out_wire),
        .uio_oe(),
        .clk(clk),
        .rst_n(rst_n),
        .ena(ena)
    );

    assign {data_out[9:8], strobe_out} = {uio_out_wire[5:4], uio_out_wire[1]};

    // Clock Generation
    always #10 clk = ~clk;

    always # 500 strobe_in =~ strobe_in;

    initial begin
        $dumpfile ("tb.vcd");
        $dumpvars (0, tb);

        // Reset the system
        rst_n = 0;
        #20;
        rst_n = 1;
        #40;

        filter_select = 2'b00;
```

```
        data_in = 0;
        #20000
        data_in = 1023;
        #20000;
        $finish;
    end

endmodule
```

A.5 Testing with Sinwave and Noise

```
`timescale 1ns / 1ps

module tb;
    // Testbench Signals
    reg [9:0] data_in = 0;
    wire [9:0] data_out;
    reg strobe_in = 0;
    wire strobe_out;
    reg clk = 0;
    reg rst_n = 1;
    reg ena = 1;
    reg [1:0] filter_select = 0;

    integer i;
    integer input_file, output_file;
    integer scan_result;
    reg [9:0] sine_wave[0:1499];

   wire [7:0] uio_out_wire; // Wire for uio_out port

    //(UUT)
    tt_um_moving_average_master uut (
        .ui_in(data_in[7:0]),
        .uo_out(data_out[7:0]),
        .uio_in({filter_select, 2'b00, data_in[9:8], 1'b0, strobe_in}),
        .uio_out(uio_out_wire),
        .uio_oe(),
        .clk(clk),
        .rst_n(rst_n),
        .ena(ena)
    );

    assign {data_out[9:8], strobe_out} = {uio_out_wire[5:4], uio_out_wire[1]};

    // Clock Generation
    always #10 clk = ~clk;

    always # 250 strobe_in =~ strobe_in;

    initial begin
        $dumpfile("tb.vcd");
        $dumpvars(0, tb);

        // Open the file with sine wave data for reading
        input_file = $fopen("sine_wave_input.txt", "r");
        // Open a file for writing the output data
        output_file = $fopen("data_out.txt", "w");

        if (input_file) begin
            for (i = 0; i < 1500; i = i + 1) begin
                scan_result = $fscanf(input_file, "%d\n", sine_wave[i]);
                // Check if the reading is successful
```

```
53                  if (scan_result != 1) begin
54                      $display("Error reading sine_wave_input.txt at line %d", i+1);
55                      $fclose(input_file);
56                      $finish;
57                  end
58              end
59              $fclose(input_file);
60          end else begin
61              $display("Error: sine_wave_input.txt file not found!");
62              $finish;
63          end
64
65          // Reset the system
66          rst_n = 0;
67          #20;
68          rst_n = 1;
69          #40;
70
71          filter_select = 2'b01;
72          for (i = 0; i < 1500; i = i + 1) begin
73              data_in = sine_wave[i];
74              #500;
75          end
76
77          #1000;
78          $fclose(output_file);
79          $finish;
80      end
81
82          // Capture data_out when strobe_out changes
83      always @(posedge clk) begin
84          if (strobe_out) begin
85              $fwrite(output_file, "%t, %d, %d\n", $time, data_in, data_out);
86          end
87      end
88
89  endmodule
```

List of Figures