Digital Systems Design (CS-431)

Assignment

NAME	S.M. MUBASHIR RIZVI	SAFIA FAIZ
ROLL NO.	CS-20071	CS-20058
SECTION	B	B

Submitted To: **Dr. Majida Kazmi**

Computer & Information Systems Engineering

NED University of Engineering & Technology

Task: Audio FIR Filtering

Objective: To implement a low-pass FIR filter for audio signal processing on Xilinx FPGA, showcasing the efficacy of DSP slices over a CLB-based approach.

Filter Specifications:

- Filter Type: Low-pass FIR Filter
- Filter Order (N): 8
- Cut-off Frequency (f_c): 1 kHz
- Sampling Frequency (f_s): 10 kHz

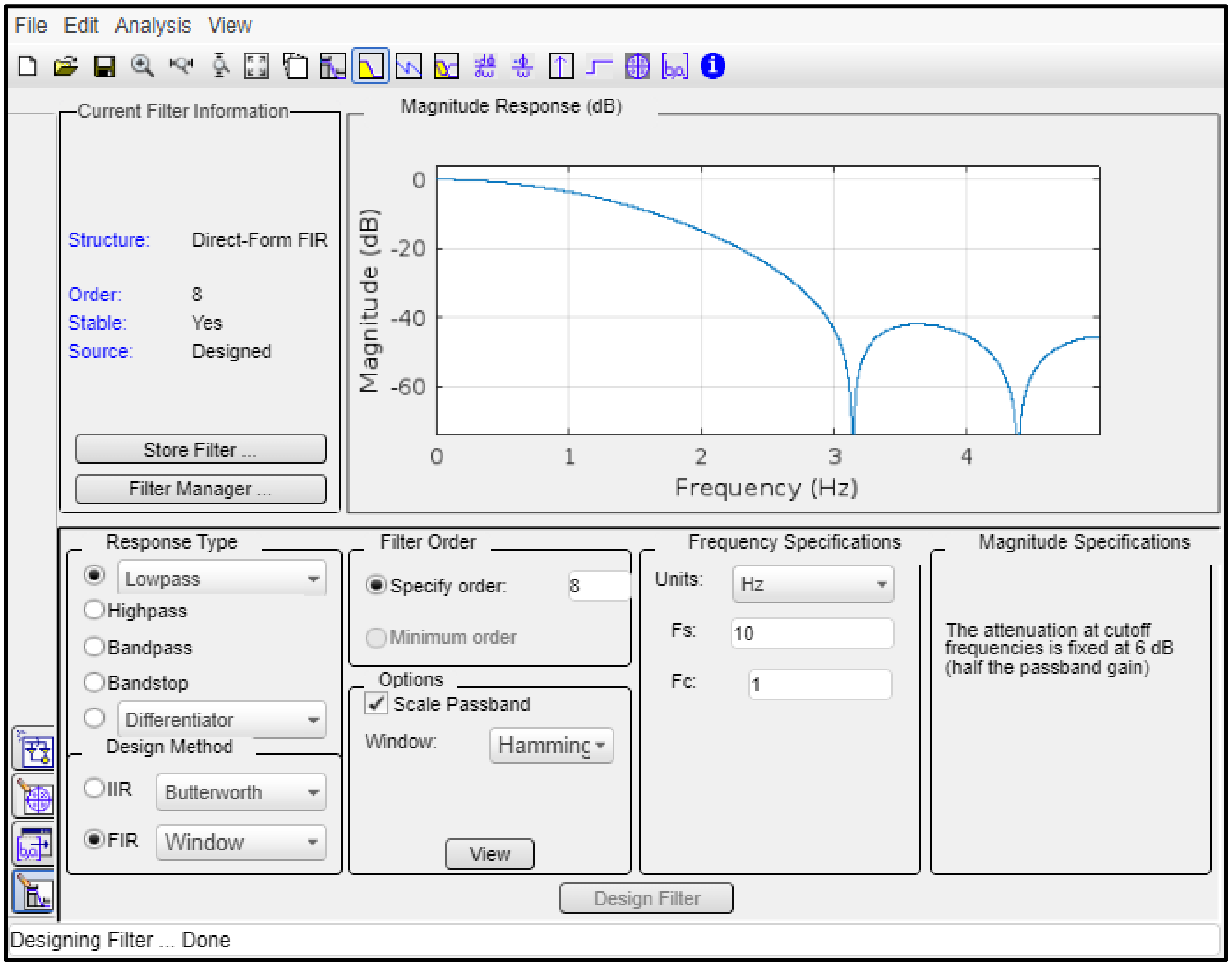
Task Steps:

1. **Design the Filter Coefficients:**
    - Design the coefficients of the low-pass FIR filter using a hamming window.
2. **Implement the FIR Filter Using CLBs:**
    - Use general-purpose logic available in CLBs to implement the low-pass FIR filter. Design the filter structure, and handle coefficients, input, and output processing.
3. **Implement the FIR Filter Using DSP Slices:**
    - Utilize DSP slices to implement the same low-pass FIR filter. Take advantage of the dedicated multiply-accumulate units in DSP slices for efficient computation.
4. **Performance Metrics:**
    - Measure and compare the throughput for both implementations (operations per second).
    - Record the resource utilization, including CLBs, LUTs, and FFs, and its power consumption.
    - Evaluate the clock frequency.
5. **Simulation and Verification:**
    - Simulate both designs to verify correctness. Ensure that both CLB and DSP slice implementations produce similar filtered outputs.
6. **Documentation and Analysis:**
    - Prepare a report documenting the design and implementation details for both CLB and DSP slice approaches.
    - Include tables or graphs showing performance metrics and resource utilization.
    - Analyse and discuss the strengths and limitations of each approach.

1. Design the Filter Coefficients:

- Design the coefficients of the low-pass FIR filter using a hamming window.

MATLAB – Filter Design and Analysis Tool:

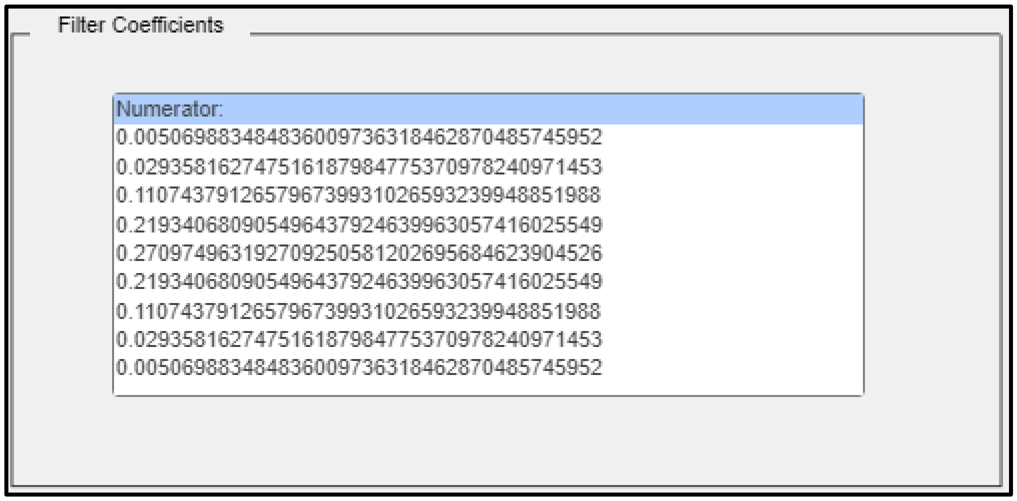


Structure: Direct-Form FIR
Order: 8
Stable: Yes
Source: Designed

Response Type: Lowpass
Design Method: FIR Window
Filter Order: Specify order: 8
Options: Scale Passband ✓, Window: Hamming
Units: Hz
Fs: 10
Fc: 1

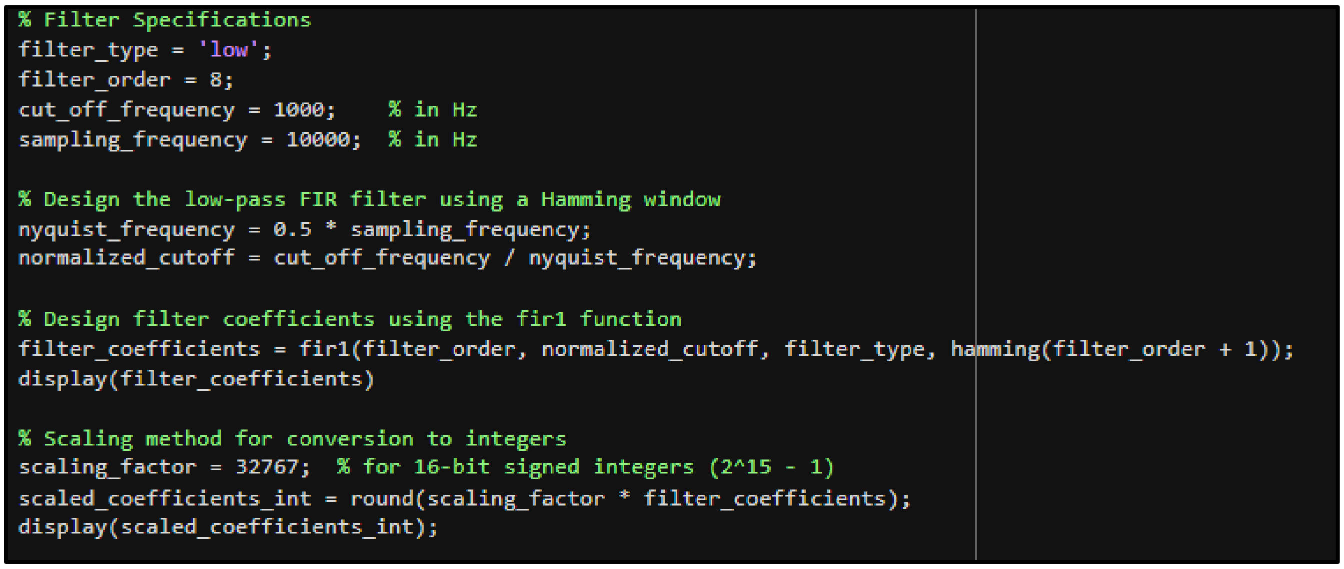
The attenuation at cutoff frequencies is fixed at 6 dB (half the passband gain)

Designing Filter ... Done

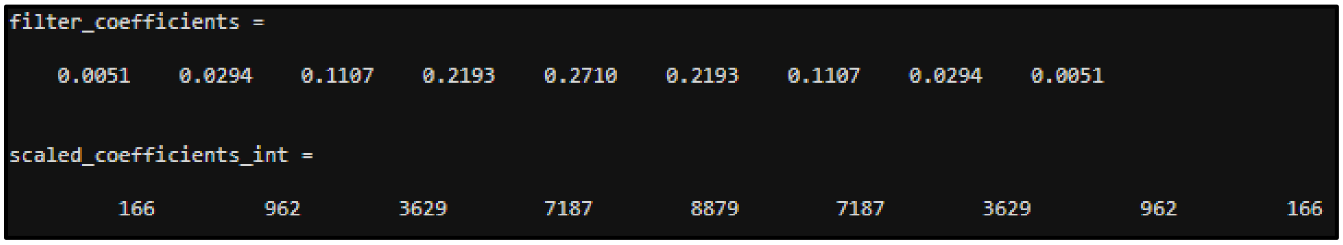


```
Numerator:
0.005069883484836009736318462870485745952
0.029358162747516187984775370978240971453
0.110743791265796739931026593239948851988
0.219340680905496437924639963057416025549
0.270974963192709250581202695684623904526
0.219340680905496437924639963057416025549
0.110743791265796739931026593239948851988
0.029358162747516187984775370978240971453
0.005069883484836009736318462870485745952
```

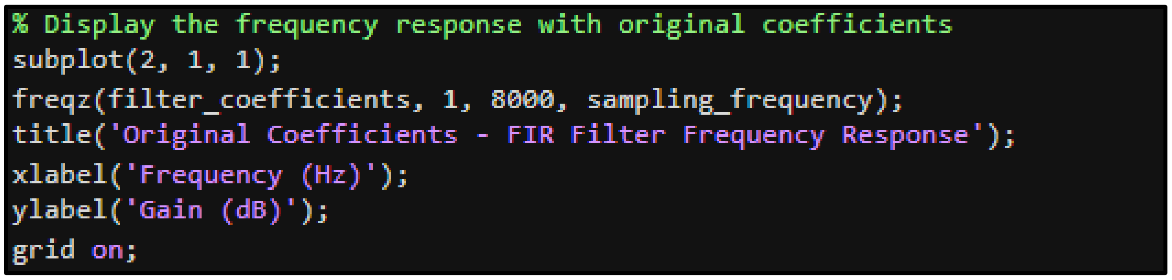
MATLAB – Code:



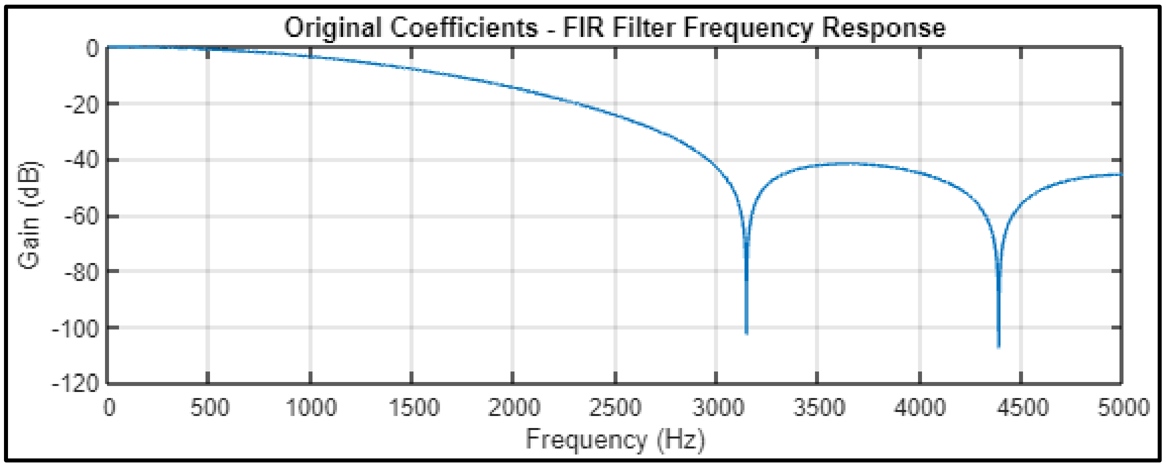
```
% Filter Specifications
filter_type = 'low';
filter_order = 8;
cut_off_frequency = 1000;    % in Hz
sampling_frequency = 10000;  % in Hz

% Design the low-pass FIR filter using a Hamming window
nyquist_frequency = 0.5 * sampling_frequency;
normalized_cutoff = cut_off_frequency / nyquist_frequency;

% Design filter coefficients using the fir1 function
filter_coefficients = fir1(filter_order, normalized_cutoff, filter_type, hamming(filter_order + 1));
display(filter_coefficients)

% Scaling method for conversion to integers
scaling_factor = 32767;  % for 16-bit signed integers (2^15 - 1)
scaled_coefficients_int = round(scaling_factor * filter_coefficients);
display(scaled_coefficients_int);
```

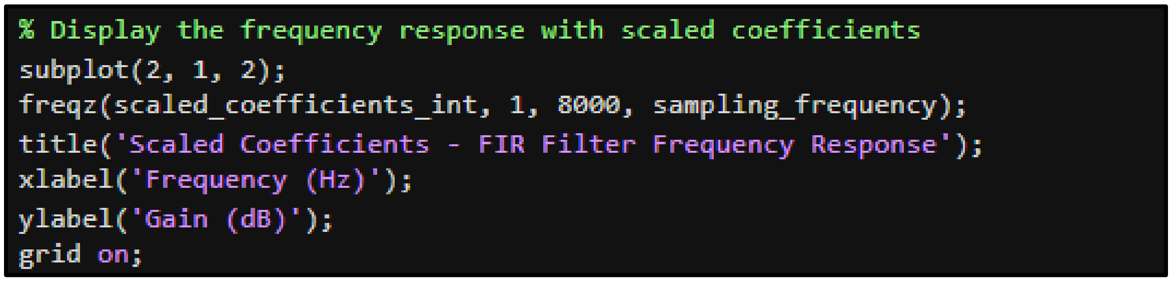


```
filter_coefficients =

    0.0051    0.0294    0.1107    0.2193    0.2710    0.2193    0.1107    0.0294    0.0051


scaled_coefficients_int =

         166          962         3629         7187         8879         7187         3629          962          166
```

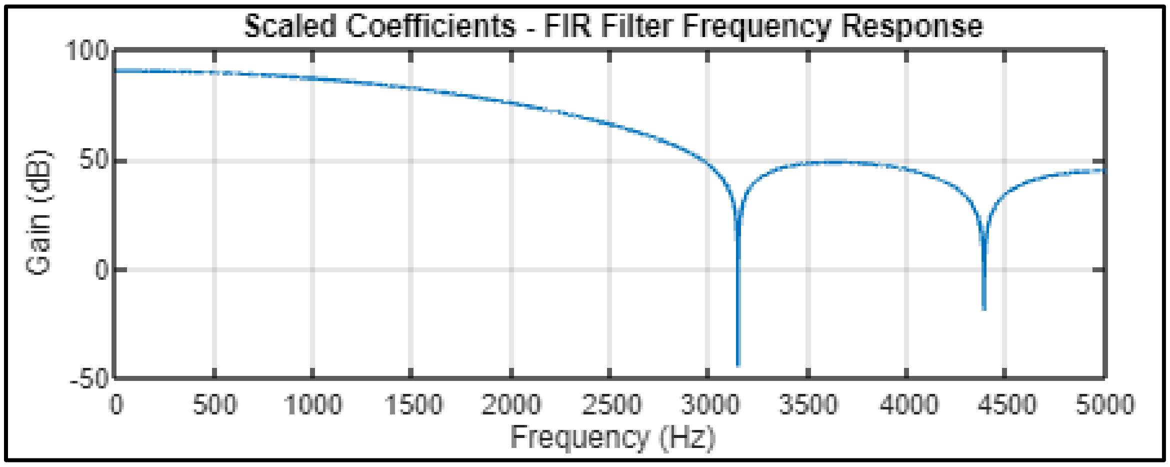


```
% Display the frequency response with original coefficients
subplot(2, 1, 1);
freqz(filter_coefficients, 1, 8000, sampling_frequency);
title('Original Coefficients - FIR Filter Frequency Response');
xlabel('Frequency (Hz)');
ylabel('Gain (dB)');
grid on;
```



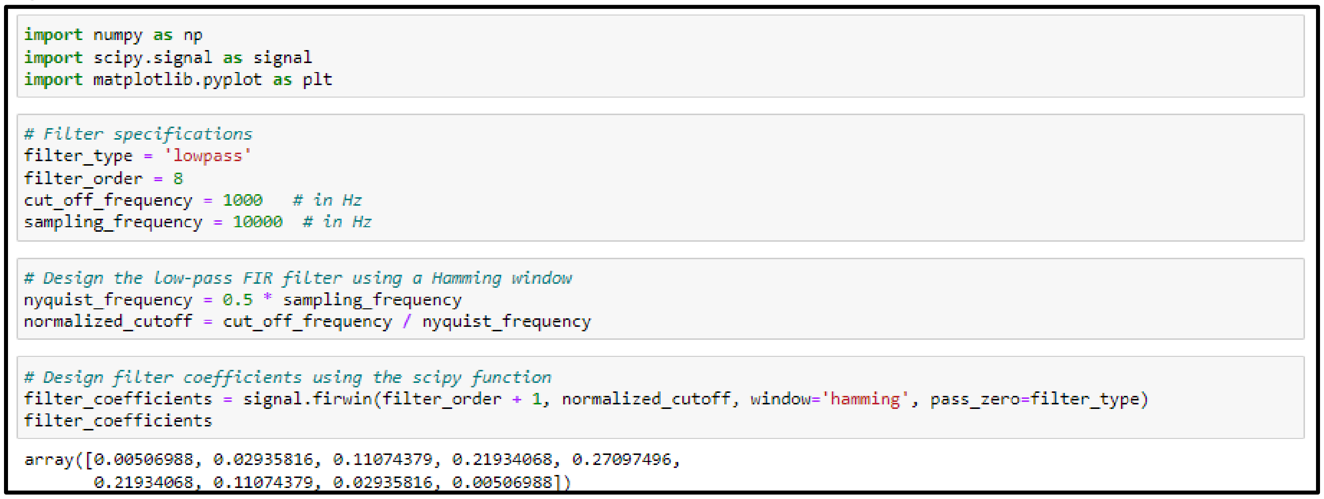


```matlab
% Display the frequency response with scaled coefficients
subplot(2, 1, 2);
freqz(scaled_coefficients_int, 1, 8000, sampling_frequency);
title('Scaled Coefficients - FIR Filter Frequency Response');
xlabel('Frequency (Hz)');
ylabel('Gain (dB)');
grid on;
```



- The purpose of the above two plots was to verify if the same filter was being generated by using the scaled coefficients.
- It is evident that the filter made using the scaled coefficients is almost exactly the same as the one made with using the original filter coefficients.
- **Note:** There would be some information lost due to the scaling technique as we have converted real number coefficients into integers.

Python – Code:



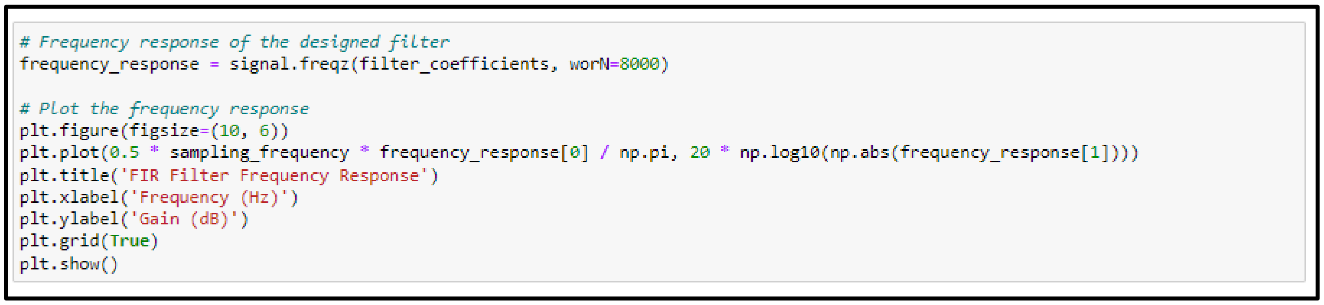
```python
import numpy as np
import scipy.signal as signal
import matplotlib.pyplot as plt
```

```python
# Filter specifications
filter_type = 'lowpass'
filter_order = 8
cut_off_frequency = 1000   # in Hz
sampling_frequency = 10000  # in Hz
```

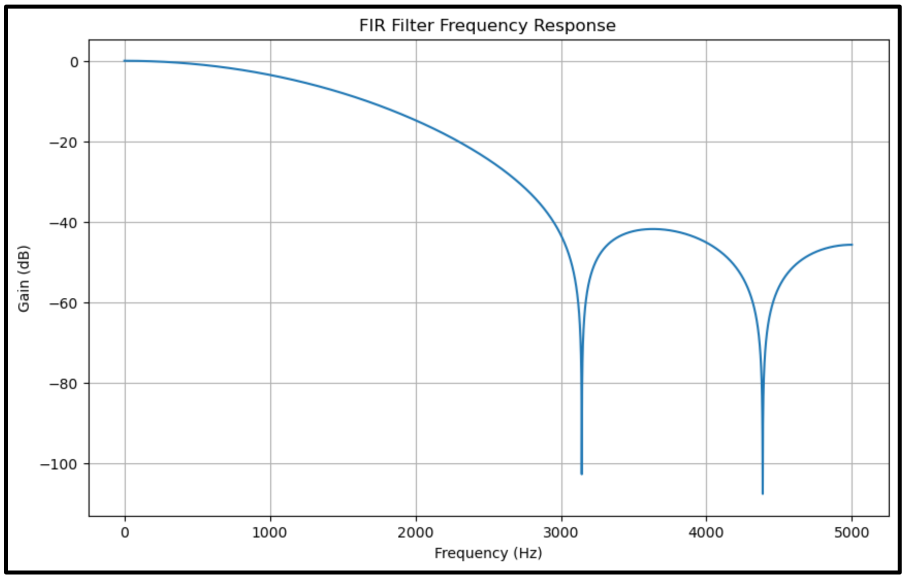
```python
# Design the low-pass FIR filter using a Hamming window
nyquist_frequency = 0.5 * sampling_frequency
normalized_cutoff = cut_off_frequency / nyquist_frequency
```

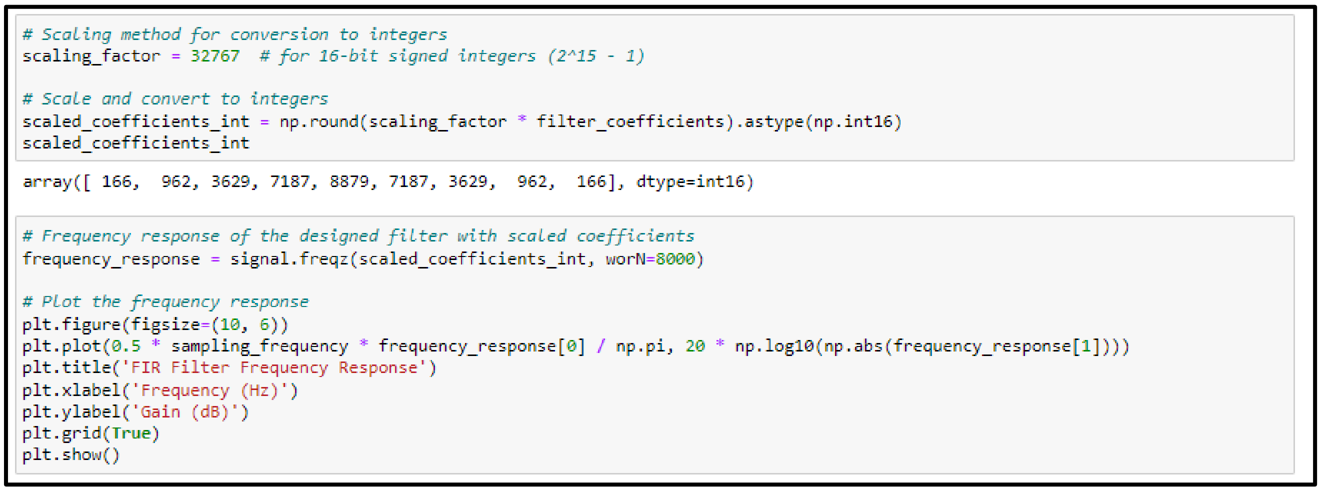
```python
# Design filter coefficients using the scipy function
filter_coefficients = signal.firwin(filter_order + 1, normalized_cutoff, window='hamming', pass_zero=filter_type)
filter_coefficients
```

```
array([0.00506988, 0.02935816, 0.11074379, 0.21934068, 0.27097496,
       0.21934068, 0.11074379, 0.02935816, 0.00506988])
```



```python
# Frequency response of the designed filter
frequency_response = signal.freqz(filter_coefficients, worN=8000)

# Plot the frequency response
plt.figure(figsize=(10, 6))
plt.plot(0.5 * sampling_frequency * frequency_response[0] / np.pi, 20 * np.log10(np.abs(frequency_response[1])))
plt.title('FIR Filter Frequency Response')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Gain (dB)')
plt.grid(True)
plt.show()
```

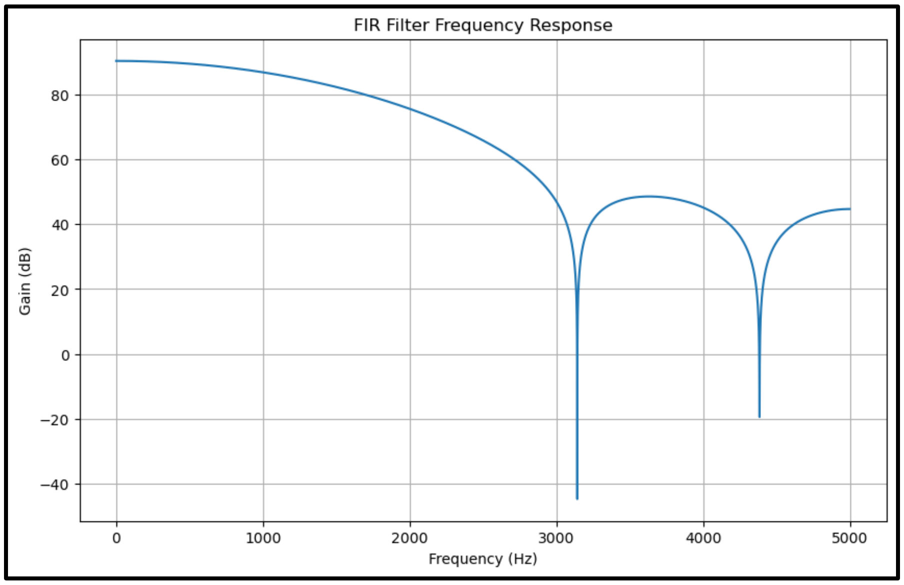




```
# Scaling method for conversion to integers
scaling_factor = 32767  # for 16-bit signed integers (2^15 - 1)

# Scale and convert to integers
scaled_coefficients_int = np.round(scaling_factor * filter_coefficients).astype(np.int16)
scaled_coefficients_int
```

```
array([ 166,  962, 3629, 7187, 8879, 7187, 3629,  962,  166], dtype=int16)
```

```
# Frequency response of the designed filter with scaled coefficients
frequency_response = signal.freqz(scaled_coefficients_int, worN=8000)

# Plot the frequency response
plt.figure(figsize=(10, 6))
plt.plot(0.5 * sampling_frequency * frequency_response[0] / np.pi, 20 * np.log10(np.abs(frequency_response[1])))
plt.title('FIR Filter Frequency Response')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Gain (dB)')
plt.grid(True)
plt.show()
```

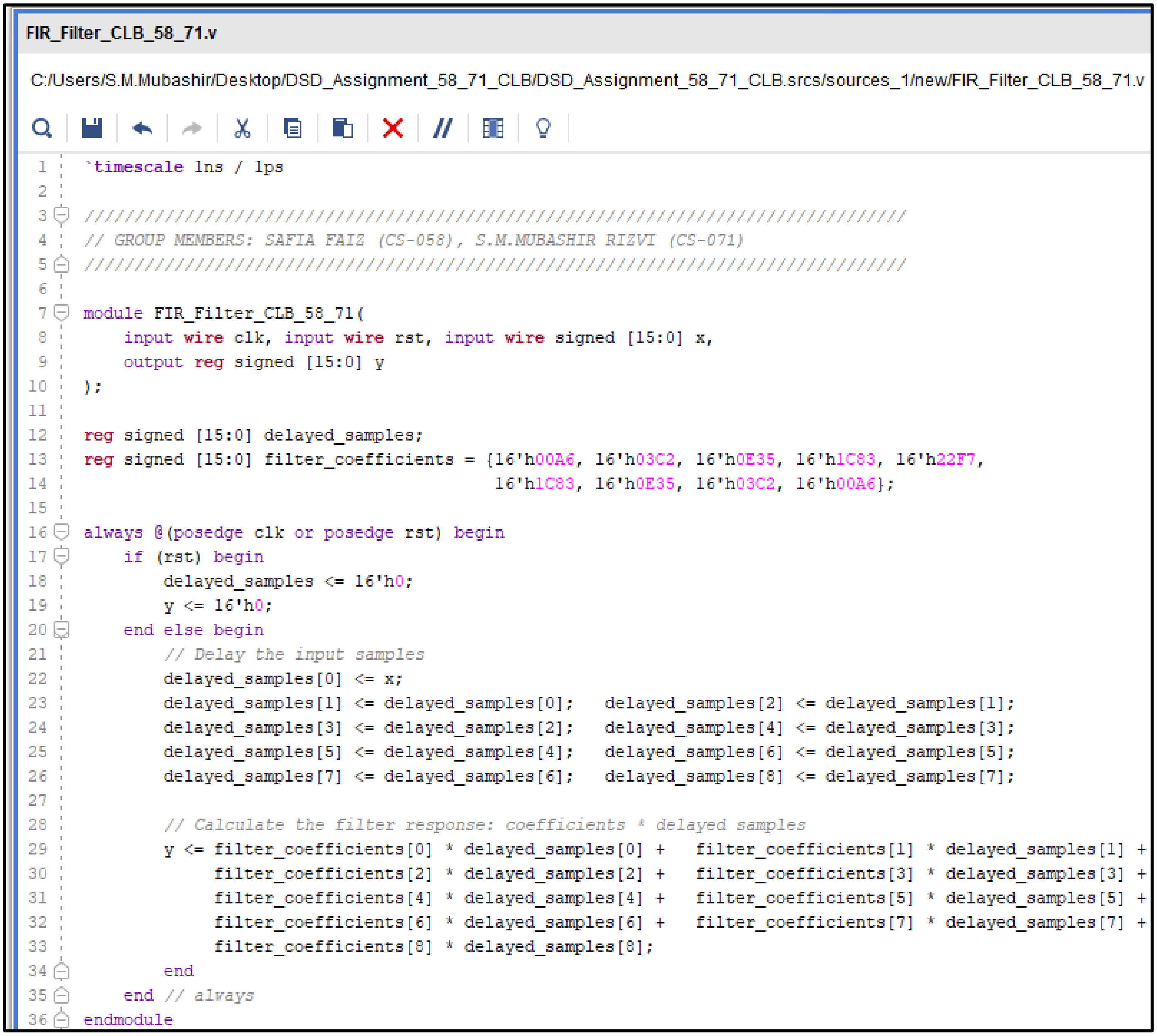


- The purpose of the above two plots is the same as mentioned in the MATLAB – Code heading.
- It is clearly seen that the scaled coefficients yield the same filter but with some information loss.

2. Implement the FIR Filter Using CLBs:

- Use general-purpose logic available in CLBs to implement the low-pass FIR filter.
- Design the filter structure, and handle coefficients, input, and output processing.

Verilog Code:

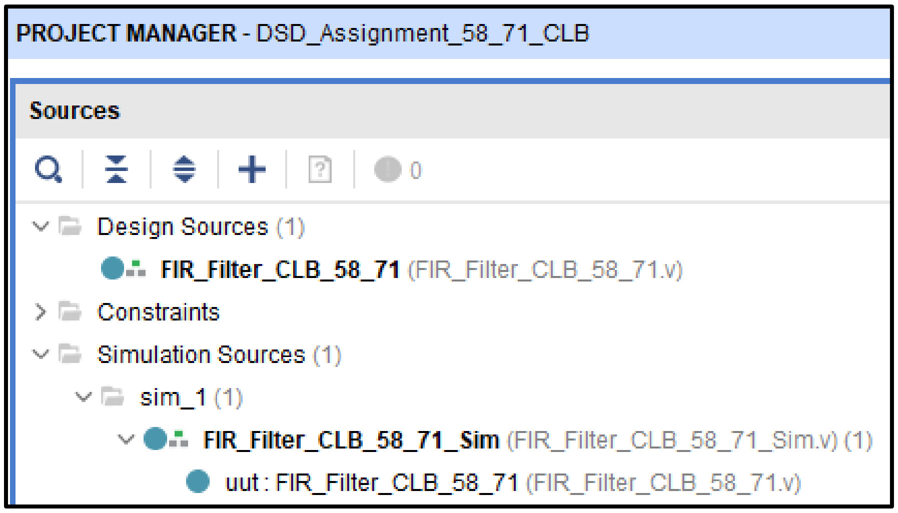


FIR_Filter_CLB_58_71.v

C:/Users/S.M.Mubashir/Desktop/DSD_Assignment_58_71_CLB/DSD_Assignment_58_71_CLB.srcs/sources_1/new/FIR_Filter_CLB_58_71.v

```verilog
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
// GROUP MEMBERS: SAFIA FAIZ (CS-058), S.M.MUBASHIR RIZVI (CS-071)
//////////////////////////////////////////////////////////////////////////////////

module FIR_Filter_CLB_58_71(
    input wire clk, input wire rst, input wire signed [15:0] x,
    output reg signed [15:0] y
);

reg signed [15:0] delayed_samples;
reg signed [15:0] filter_coefficients = {16'h00A6, 16'h03C2, 16'h0E35, 16'h1C83, 16'h22F7,
                                         16'h1C83, 16'h0E35, 16'h03C2, 16'h00A6};

always @(posedge clk or posedge rst) begin
    if (rst) begin
        delayed_samples <= 16'h0;
        y <= 16'h0;
    end else begin
        // Delay the input samples
        delayed_samples[0] <= x;
        delayed_samples[1] <= delayed_samples[0];   delayed_samples[2] <= delayed_samples[1];
        delayed_samples[3] <= delayed_samples[2];   delayed_samples[4] <= delayed_samples[3];
        delayed_samples[5] <= delayed_samples[4];   delayed_samples[6] <= delayed_samples[5];
        delayed_samples[7] <= delayed_samples[6];   delayed_samples[8] <= delayed_samples[7];

        // Calculate the filter response: coefficients * delayed samples
        y <= filter_coefficients[0] * delayed_samples[0] +   filter_coefficients[1] * delayed_samples[1] +
             filter_coefficients[2] * delayed_samples[2] +   filter_coefficients[3] * delayed_samples[3] +
             filter_coefficients[4] * delayed_samples[4] +   filter_coefficients[5] * delayed_samples[5] +
             filter_coefficients[6] * delayed_samples[6] +   filter_coefficients[7] * delayed_samples[7] +
             filter_coefficients[8] * delayed_samples[8];
        end
    end // always
endmodule
```

File Structure of This Project:



PROJECT MANAGER - DSD_Assignment_58_71_CLB

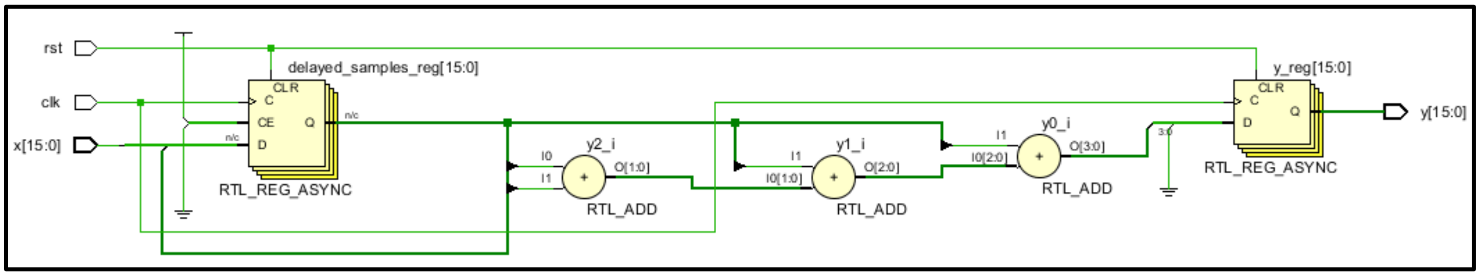
Sources

Design Sources (1)
- FIR_Filter_CLB_58_71 (FIR_Filter_CLB_58_71.v)

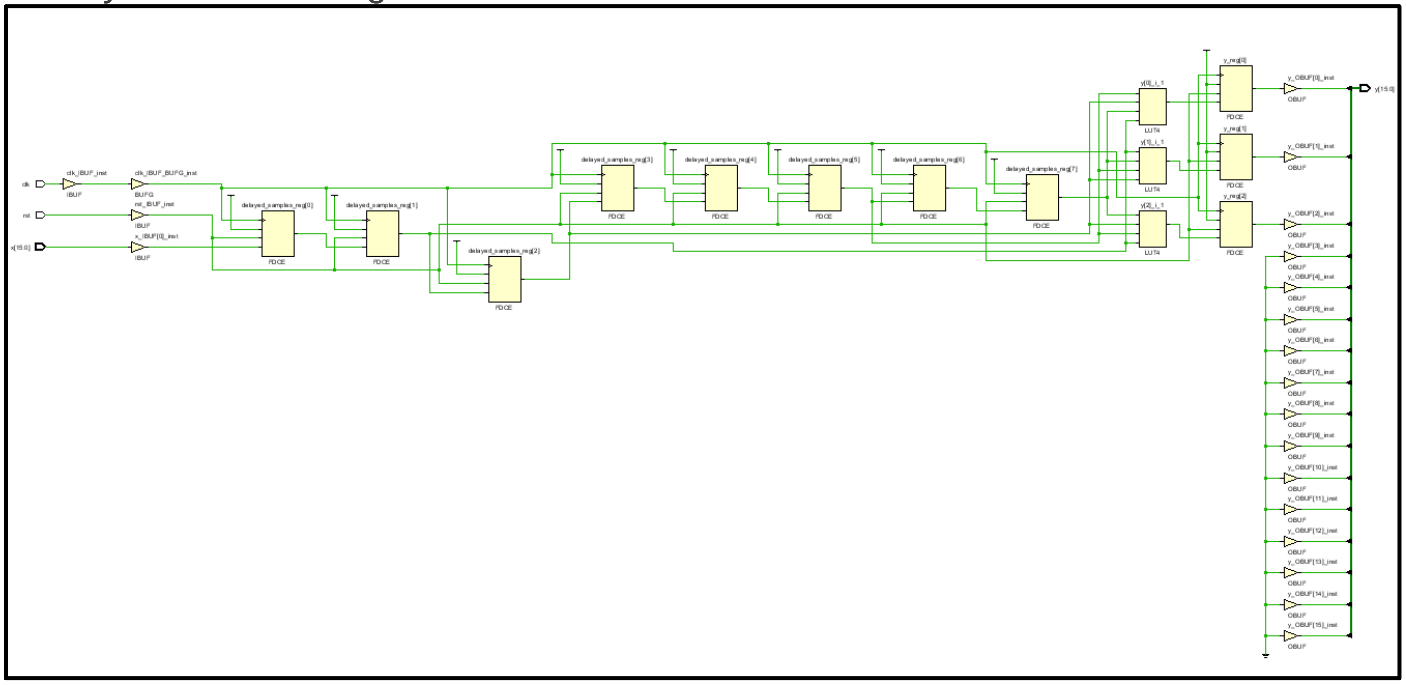
Constraints

Simulation Sources (1)
- sim_1 (1)
  - FIR_Filter_CLB_58_71_Sim (FIR_Filter_CLB_58_71_Sim.v) (1)
    - uut : FIR_Filter_CLB_58_71 (FIR_Filter_CLB_58_71.v)

RTL Schematic:



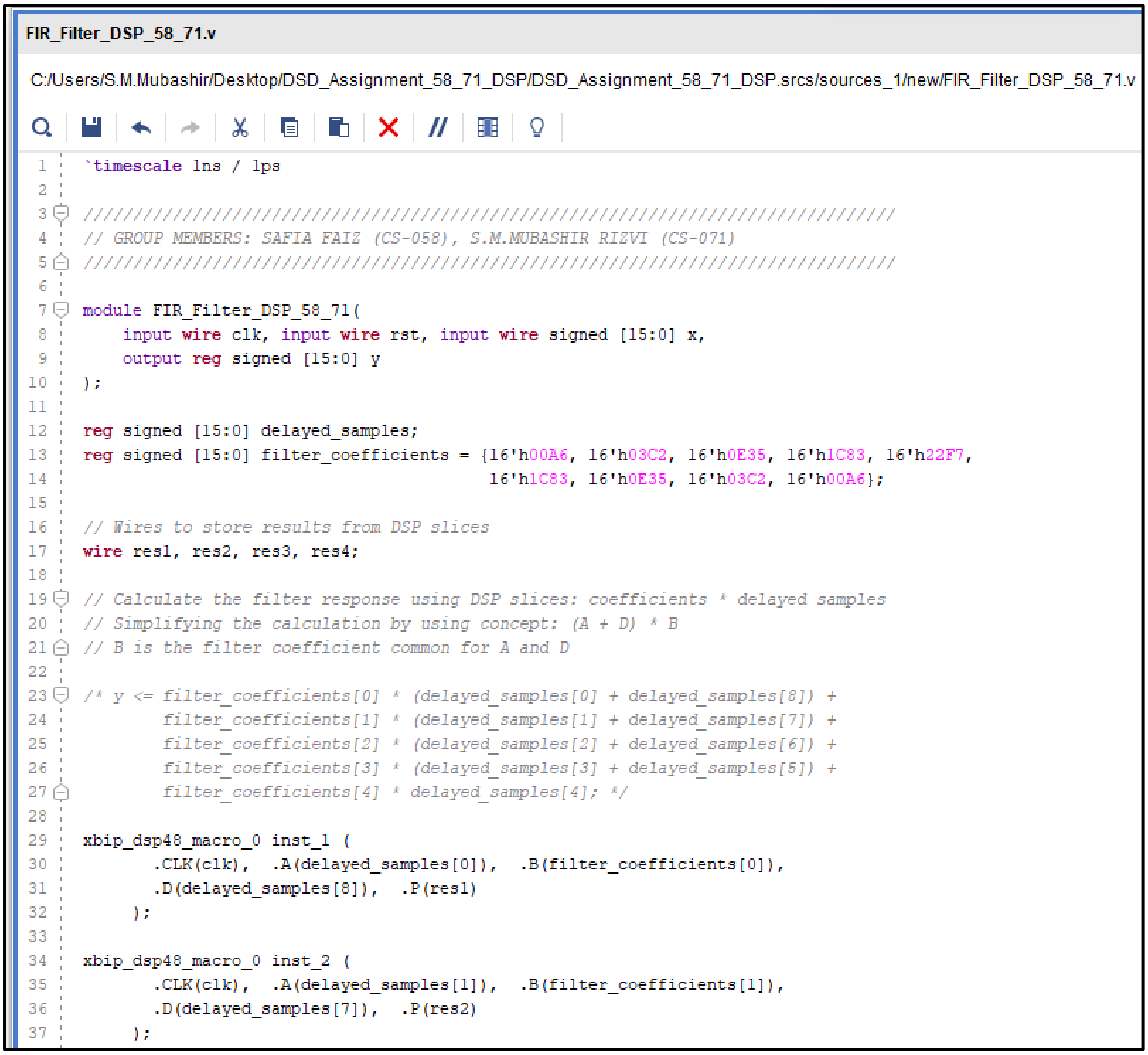
Synthesized Design:



3. Implement the FIR Filter Using DSP Slices:

- Utilize DSP slices to implement the same low-pass FIR filter.
- Take advantage of the dedicated multiply-accumulate units in DSP slices for efficient computation.

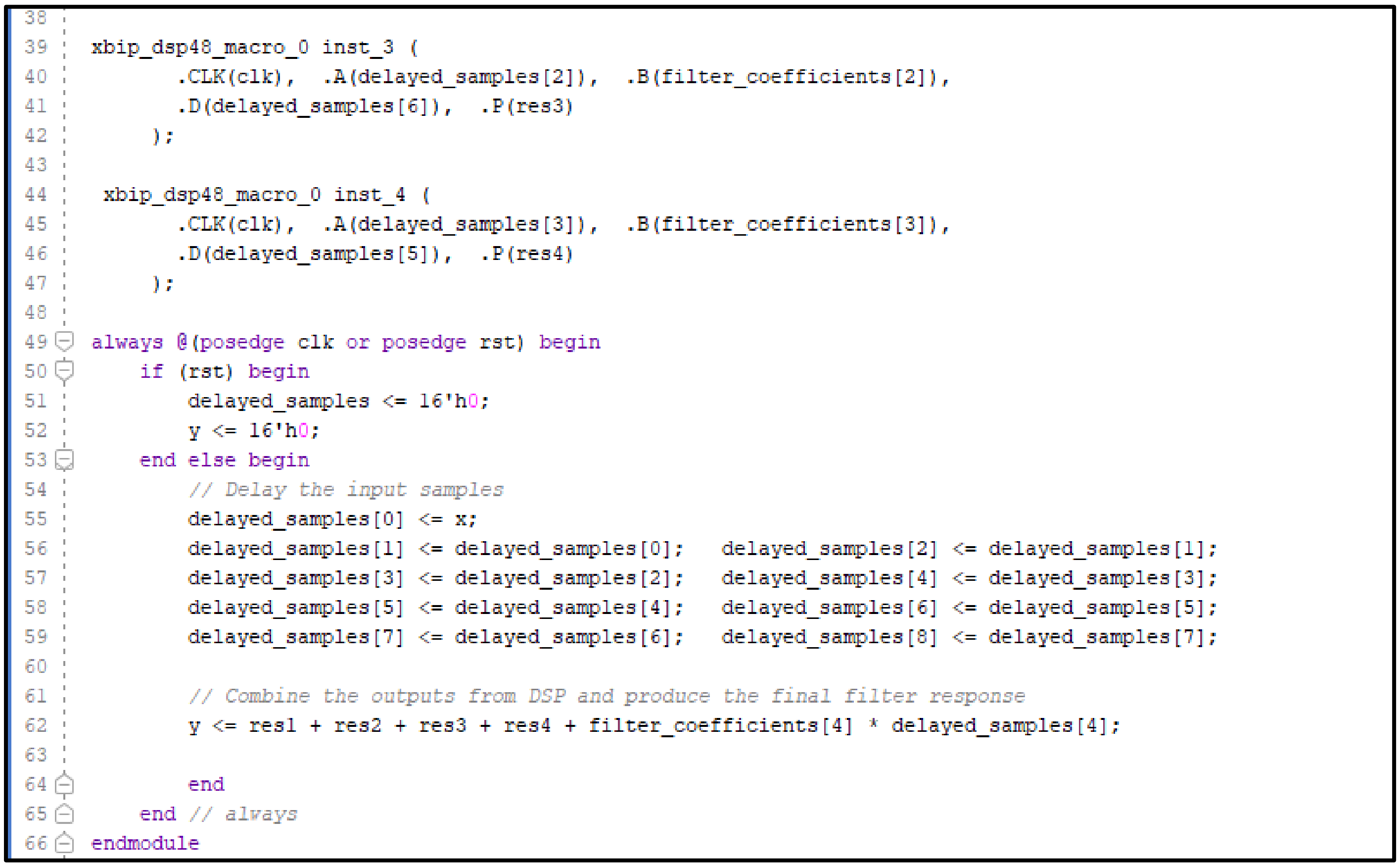
Verilog Code:



FIR_Filter_DSP_58_71.v

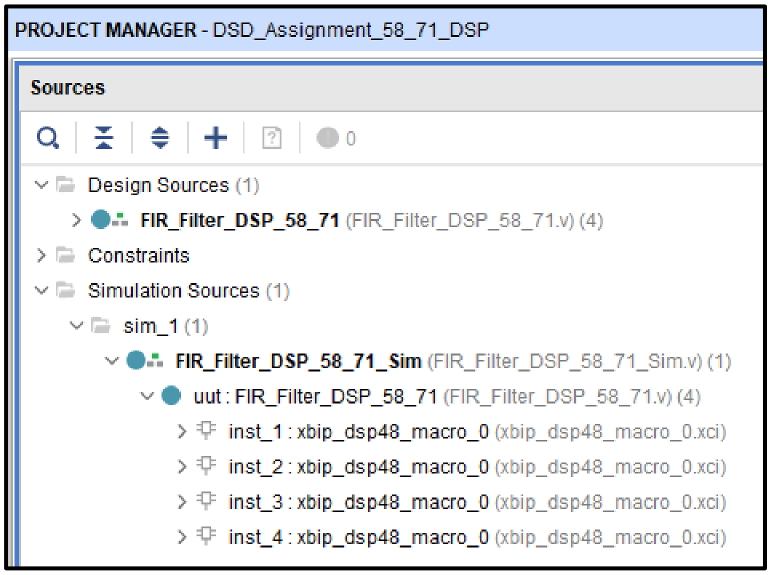
C:/Users/S.M.Mubashir/Desktop/DSD_Assignment_58_71_DSP/DSD_Assignment_58_71_DSP.srcs/sources_1/new/FIR_Filter_DSP_58_71.v

```verilog
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
// GROUP MEMBERS: SAFIA FAIZ (CS-058), S.M.MUBASHIR RIZVI (CS-071)
//////////////////////////////////////////////////////////////////////////////////

module FIR_Filter_DSP_58_71(
    input wire clk, input wire rst, input wire signed [15:0] x,
    output reg signed [15:0] y
);

reg signed [15:0] delayed_samples;
reg signed [15:0] filter_coefficients = {16'h00A6, 16'h03C2, 16'h0E35, 16'h1C83, 16'h22F7,
                                         16'h1C83, 16'h0E35, 16'h03C2, 16'h00A6};

// Wires to store results from DSP slices
wire res1, res2, res3, res4;

// Calculate the filter response using DSP slices: coefficients * delayed samples
// Simplifying the calculation by using concept: (A + D) * B
// B is the filter coefficient common for A and D

/* y <= filter_coefficients[0] * (delayed_samples[0] + delayed_samples[8]) +
        filter_coefficients[1] * (delayed_samples[1] + delayed_samples[7]) +
        filter_coefficients[2] * (delayed_samples[2] + delayed_samples[6]) +
        filter_coefficients[3] * (delayed_samples[3] + delayed_samples[5]) +
        filter_coefficients[4] * delayed_samples[4]; */

xbip_dsp48_macro_0 inst_1 (
        .CLK(clk),  .A(delayed_samples[0]),  .B(filter_coefficients[0]),
        .D(delayed_samples[8]),  .P(res1)
     );

xbip_dsp48_macro_0 inst_2 (
        .CLK(clk),  .A(delayed_samples[1]),  .B(filter_coefficients[1]),
        .D(delayed_samples[7]),  .P(res2)
     );
```



```
 xbip_dsp48_macro_0 inst_3 (
        .CLK(clk),  .A(delayed_samples[2]),  .B(filter_coefficients[2]),
        .D(delayed_samples[6]),  .P(res3)
      );

  xbip_dsp48_macro_0 inst_4 (
        .CLK(clk),  .A(delayed_samples[3]),  .B(filter_coefficients[3]),
        .D(delayed_samples[5]),  .P(res4)
      );

always @(posedge clk or posedge rst) begin
    if (rst) begin
        delayed_samples <= 16'h0;
        y <= 16'h0;
    end else begin
        // Delay the input samples
        delayed_samples[0] <= x;
        delayed_samples[1] <= delayed_samples[0];   delayed_samples[2] <= delayed_samples[1];
        delayed_samples[3] <= delayed_samples[2];   delayed_samples[4] <= delayed_samples[3];
        delayed_samples[5] <= delayed_samples[4];   delayed_samples[6] <= delayed_samples[5];
        delayed_samples[7] <= delayed_samples[6];   delayed_samples[8] <= delayed_samples[7];

        // Combine the outputs from DSP and produce the final filter response
        y <= res1 + res2 + res3 + res4 + filter_coefficients[4] * delayed_samples[4];

        end
    end // always
endmodule
```

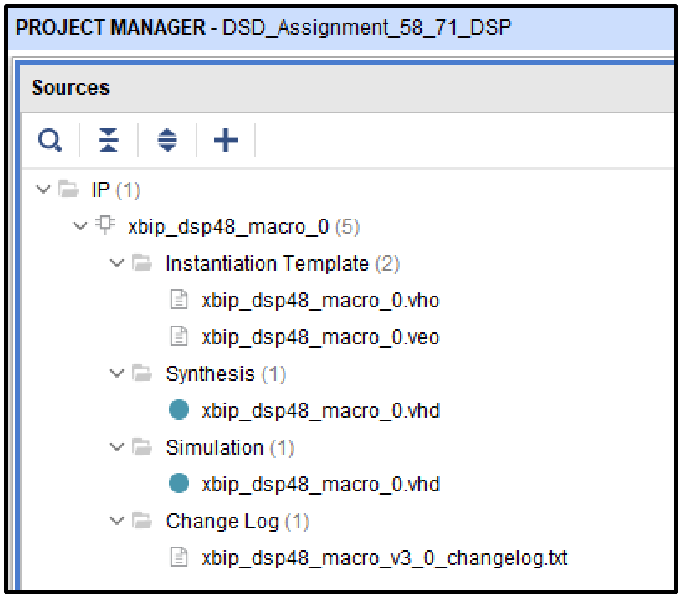
File Structure of This Project:



PROJECT MANAGER - DSD_Assignment_58_71_DSP

Sources

```
Design Sources (1)
  FIR_Filter_DSP_58_71 (FIR_Filter_DSP_58_71.v) (4)
Constraints
Simulation Sources (1)
  sim_1 (1)
    FIR_Filter_DSP_58_71_Sim (FIR_Filter_DSP_58_71_Sim.v) (1)
      uut : FIR_Filter_DSP_58_71 (FIR_Filter_DSP_58_71.v) (4)
        inst_1 : xbip_dsp48_macro_0 (xbip_dsp48_macro_0.xci)
        inst_2 : xbip_dsp48_macro_0 (xbip_dsp48_macro_0.xci)
        inst_3 : xbip_dsp48_macro_0 (xbip_dsp48_macro_0.xci)
        inst_4 : xbip_dsp48_macro_0 (xbip_dsp48_macro_0.xci)
```

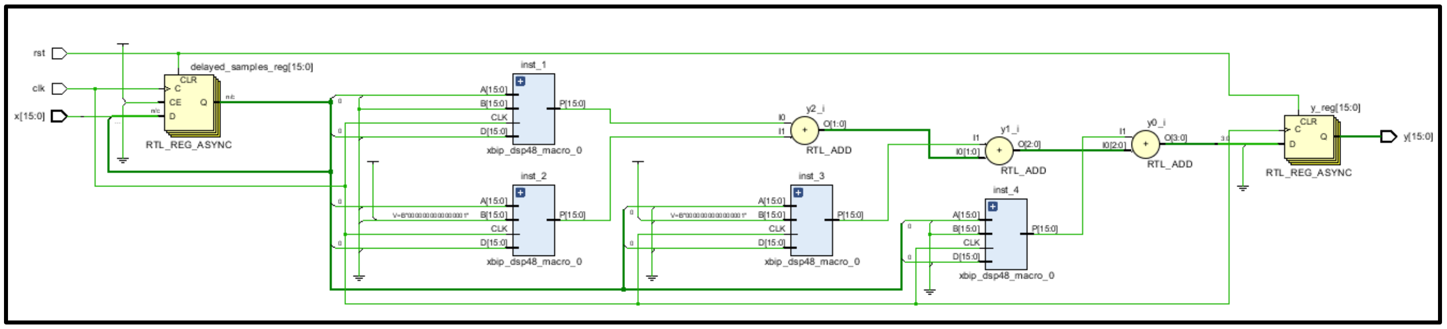


PROJECT MANAGER - DSD_Assignment_58_71_DSP

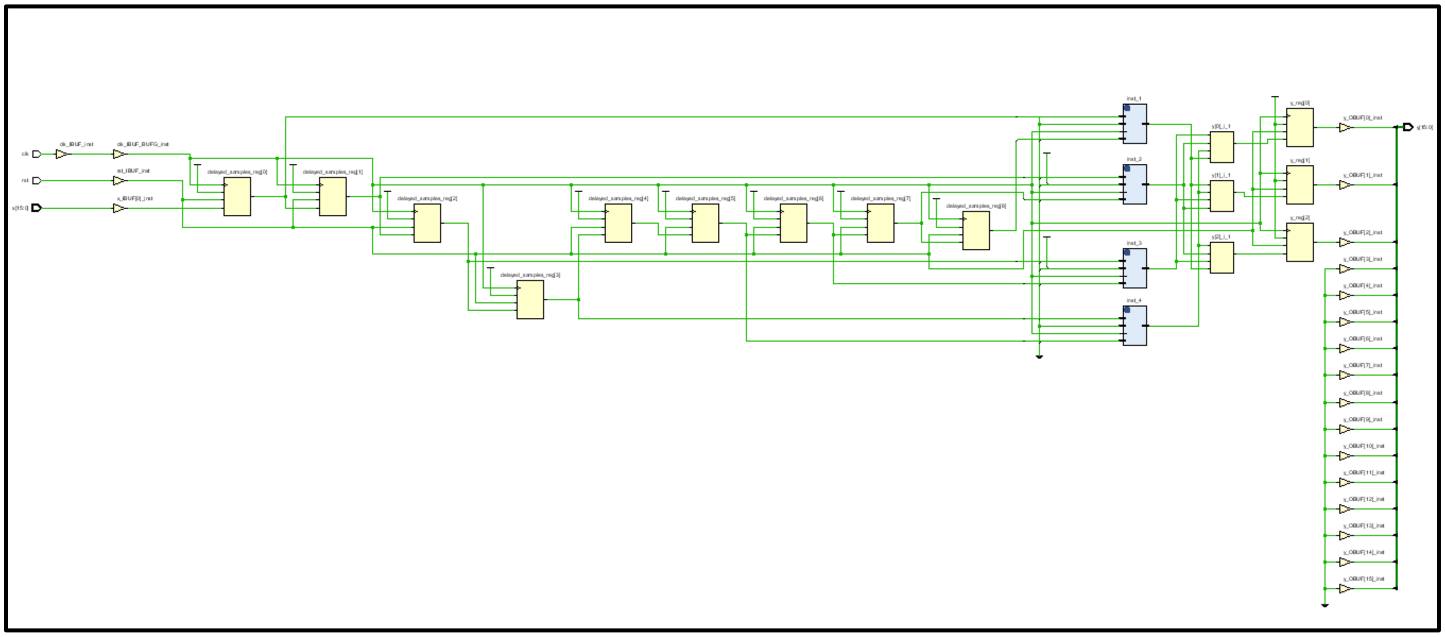
Sources

```
IP (1)
  xbip_dsp48_macro_0 (5)
    Instantiation Template (2)
      xbip_dsp48_macro_0.vho
      xbip_dsp48_macro_0.veo
    Synthesis (1)
      xbip_dsp48_macro_0.vhd
    Simulation (1)
      xbip_dsp48_macro_0.vhd
    Change Log (1)
      xbip_dsp48_macro_v3_0_changelog.txt
```

RTL Schematic:



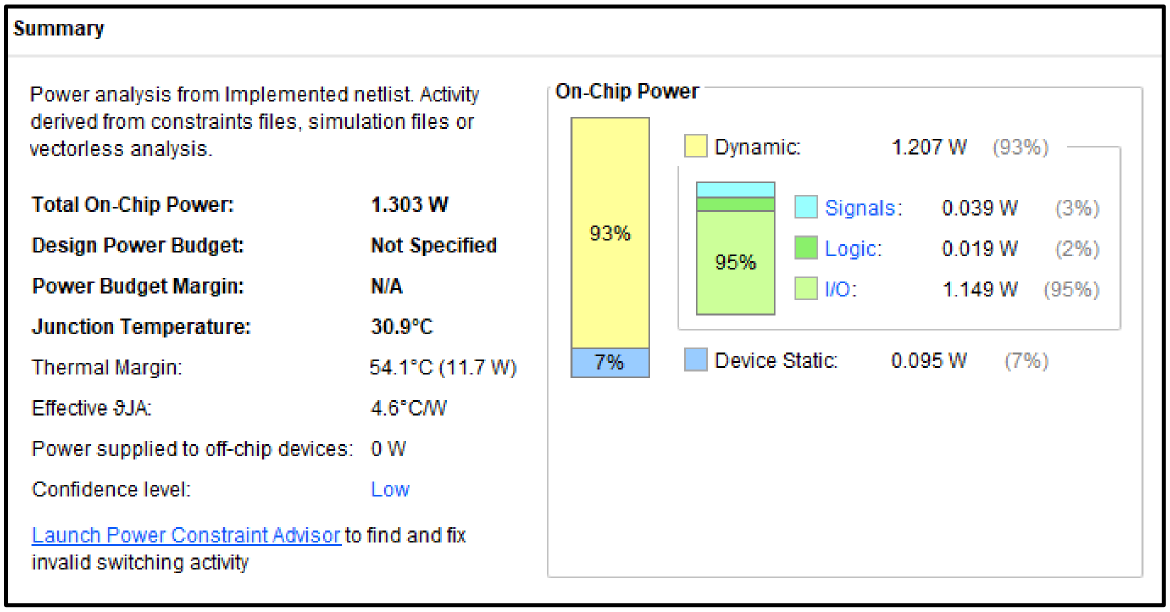
Synthesized Design:



4. Performance Metrics:

- Measure and compare the throughput for both implementations (operations per second).
- Record the resource utilization, including CLBs, LUTs, and FFs, and its power consumption.
- Evaluate the clock frequency.

Power Consumption CLB:



Summary

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

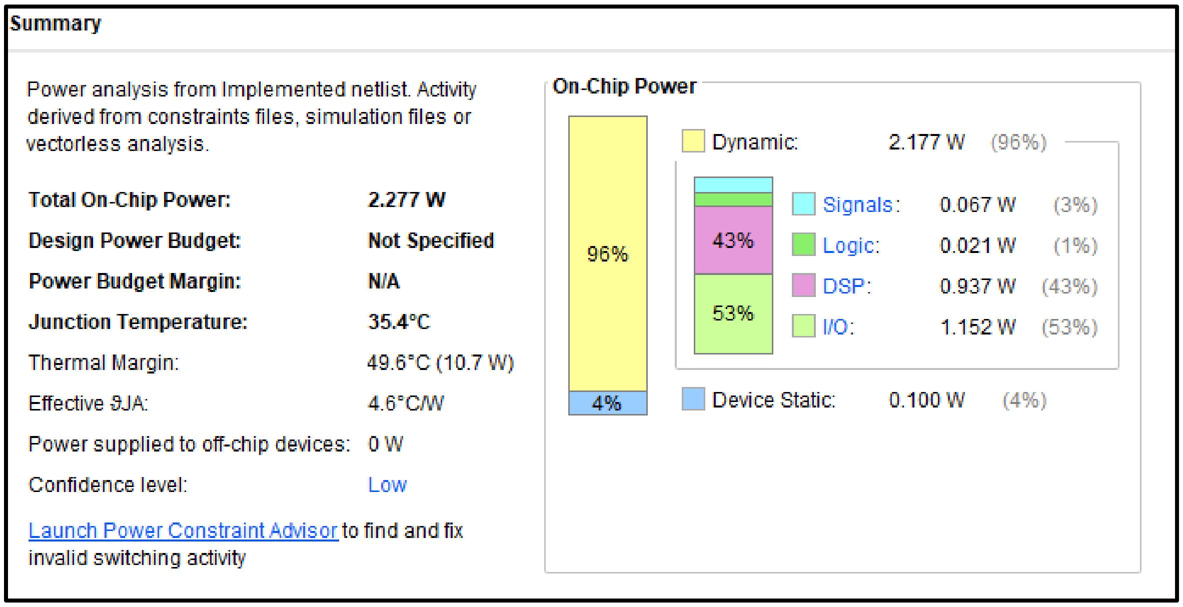
Total On-Chip Power:	**1.303 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**30.9°C**
Thermal Margin:	54.1°C (11.7 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	1.207 W	(93%)
Signals:	0.039 W	(3%)
Logic:	0.019 W	(2%)
I/O:	1.149 W	(95%)
Device Static:	0.095 W	(7%)

Power Consumption DSP:



Summary

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

Total On-Chip Power:	**2.277 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**35.4°C**
Thermal Margin:	49.6°C (10.7 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

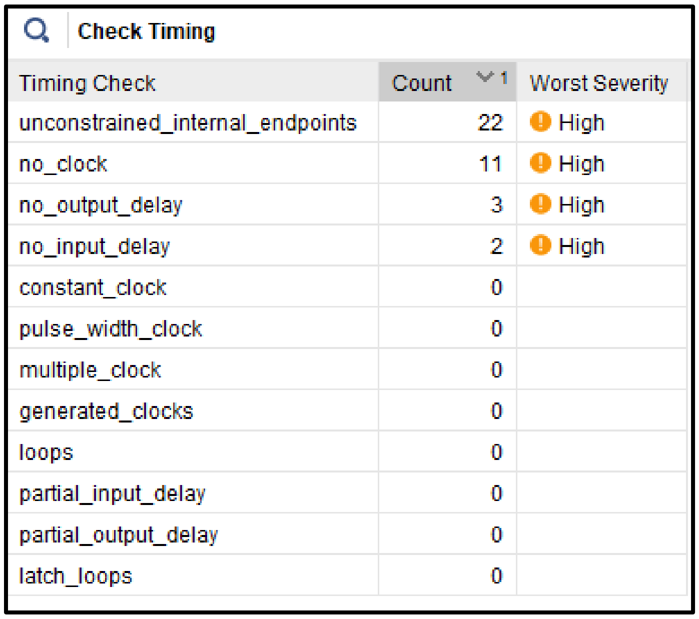
Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	2.177 W	(96%)
Signals:	0.067 W	(3%)
Logic:	0.021 W	(1%)
DSP:	0.937 W	(43%)
I/O:	1.152 W	(53%)
Device Static:	0.100 W	(4%)

- From the above power consumption data of CLB and DSP implementation of the filter, we see that DSP implementation uses more power than the CLB.
- However, there is a flaw that we have used 4 DSP slices, but we believe this task can be accomplished with a smaller number of DSP slices.
- Almost 43% of the power consumption depends on DSP slices and if the DSP slices are to be reduced, the total power consumption would reduce substantially.

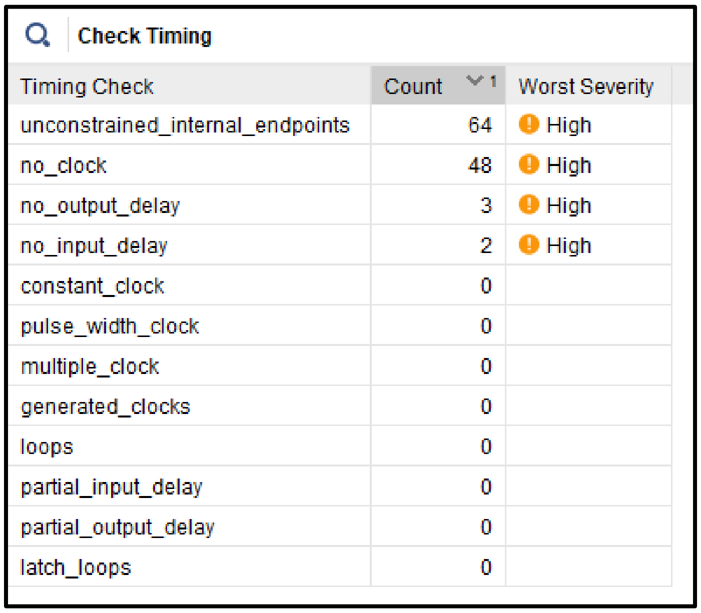
Timing Count and Severity of CLB:



Check Timing

Timing Check	Count	Worst Severity
unconstrained_internal_endpoints	22	High
no_clock	11	High
no_output_delay	3	High
no_input_delay	2	High
constant_clock	0	
pulse_width_clock	0	
multiple_clock	0	
generated_clocks	0	
loops	0	
partial_input_delay	0	
partial_output_delay	0	
latch_loops	0	

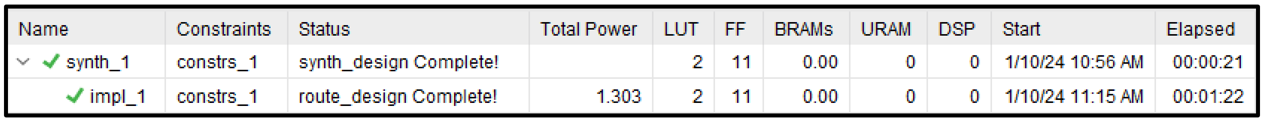
Timing Count and Severity of DSP:



Check Timing

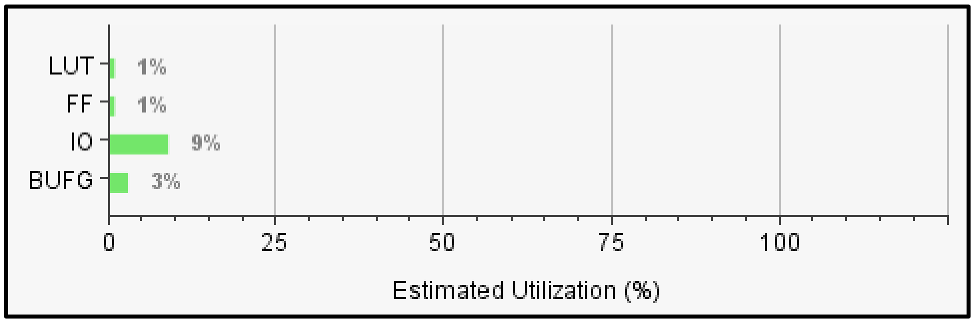
Timing Check	Count	Worst Severity
unconstrained_internal_endpoints	64	High
no_clock	48	High
no_output_delay	3	High
no_input_delay	2	High
constant_clock	0	
pulse_width_clock	0	
multiple_clock	0	
generated_clocks	0	
loops	0	
partial_input_delay	0	
partial_output_delay	0	
latch_loops	0	

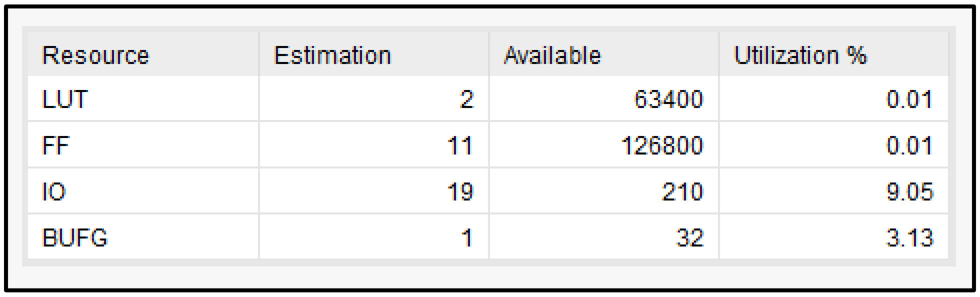
Resource Utilization of CLB:



Name	Constraints	Status	Total Power	LUT	FF	BRAMs	URAM	DSP	Start	Elapsed
✓ synth_1	constrs_1	synth_design Complete!		2	11	0.00	0	0	1/10/24 10:56 AM	00:00:21
✓ impl_1	constrs_1	route_design Complete!	1.303	2	11	0.00	0	0	1/10/24 11:15 AM	00:01:22

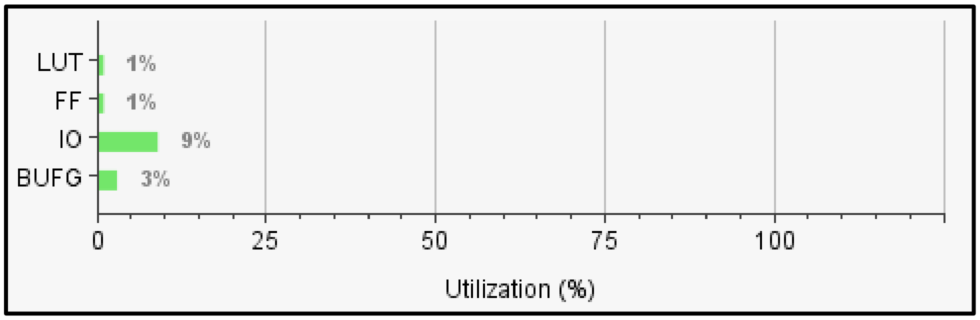
- **Post-Synthesis Utilization:**

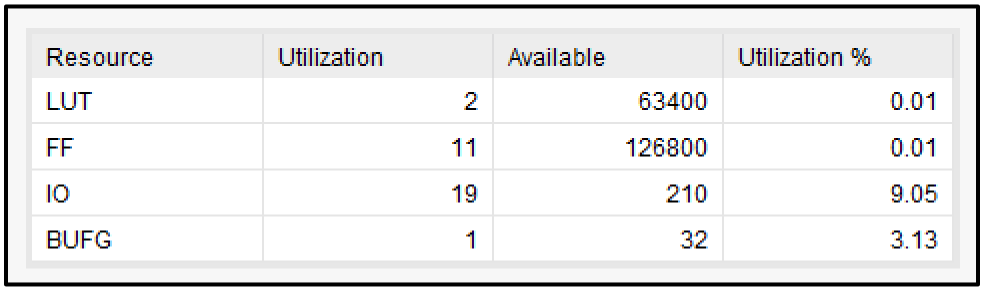




Resource	Estimation	Available	Utilization %
LUT	2	63400	0.01
FF	11	126800	0.01
IO	19	210	9.05
BUFG	1	32	3.13

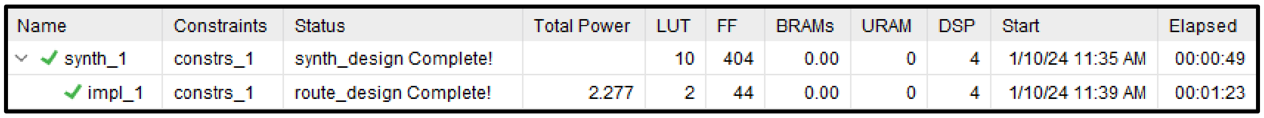
- **Post-Implementation Utilization:**





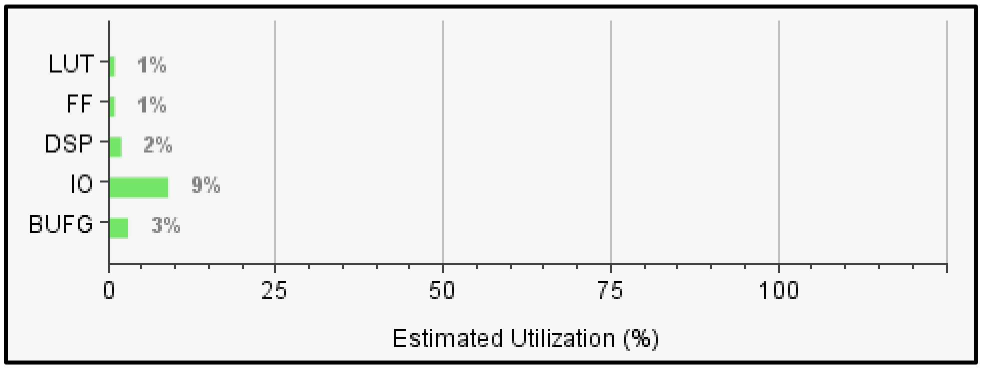
Resource	Utilization	Available	Utilization %
LUT	2	63400	0.01
FF	11	126800	0.01
IO	19	210	9.05
BUFG	1	32	3.13

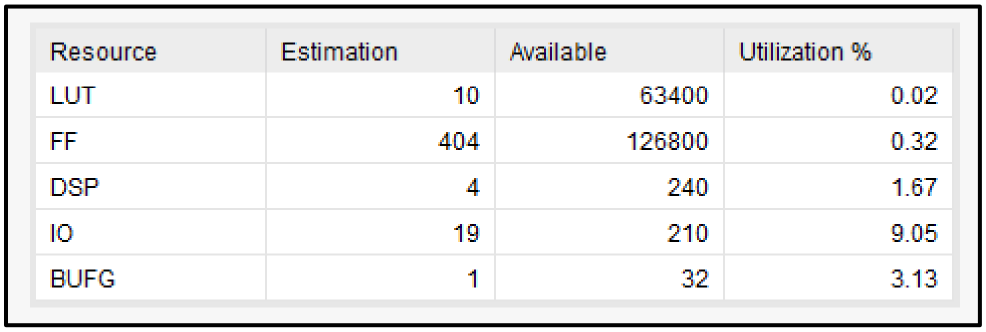
Resource Utilization of DSP:



Name	Constraints	Status	Total Power	LUT	FF	BRAMs	URAM	DSP	Start	Elapsed
✓ synth_1	constrs_1	synth_design Complete!		10	404	0.00	0	4	1/10/24 11:35 AM	00:00:49
✓ impl_1	constrs_1	route_design Complete!	2.277	2	44	0.00	0	4	1/10/24 11:39 AM	00:01:23

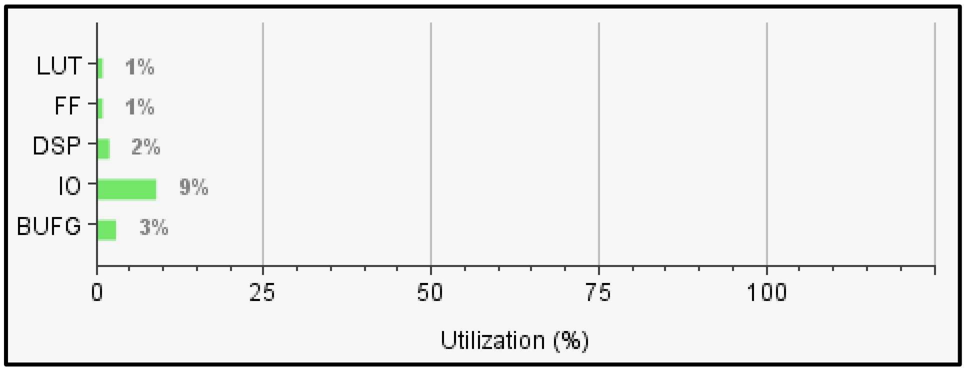
- **Post-Synthesis Utilization:**

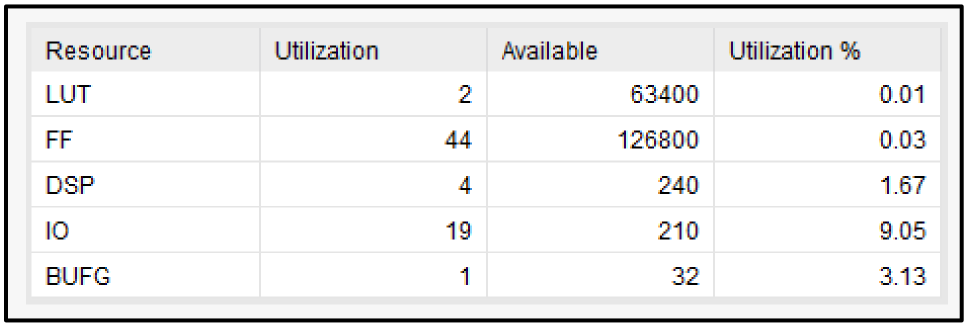




Resource	Estimation	Available	Utilization %
LUT	10	63400	0.02
FF	404	126800	0.32
DSP	4	240	1.67
IO	19	210	9.05
BUFG	1	32	3.13

- **Post-Implementation Utilization:**



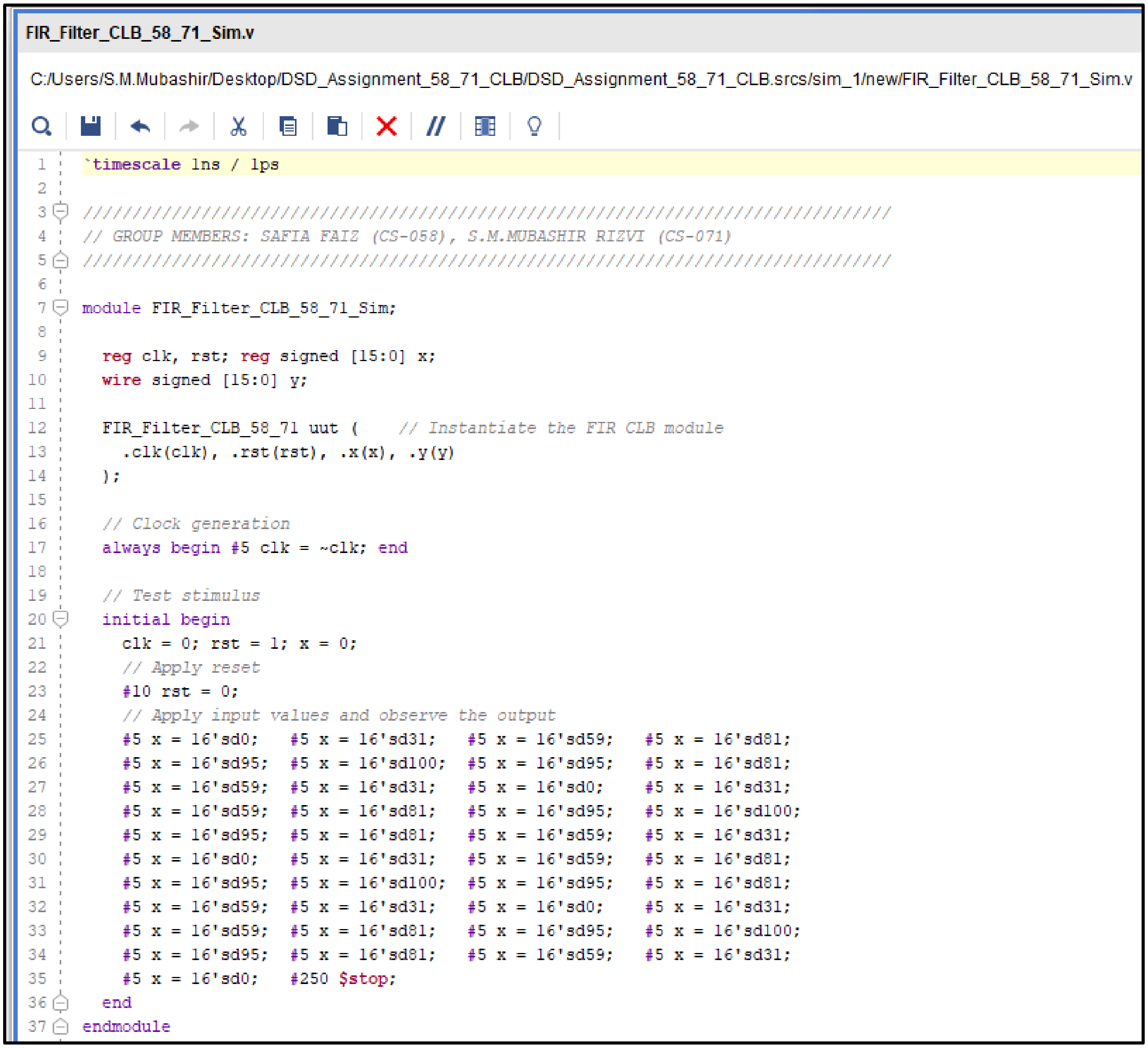


Resource	Utilization	Available	Utilization %
LUT	2	63400	0.01
FF	44	126800	0.03
DSP	4	240	1.67
IO	19	210	9.05
BUFG	1	32	3.13

5. Simulation and Verification:

- Simulate both designs to verify correctness.
- Ensure that both CLB and DSP slice implementations produce similar filtered outputs.

Test Bench for CLB Implementation:

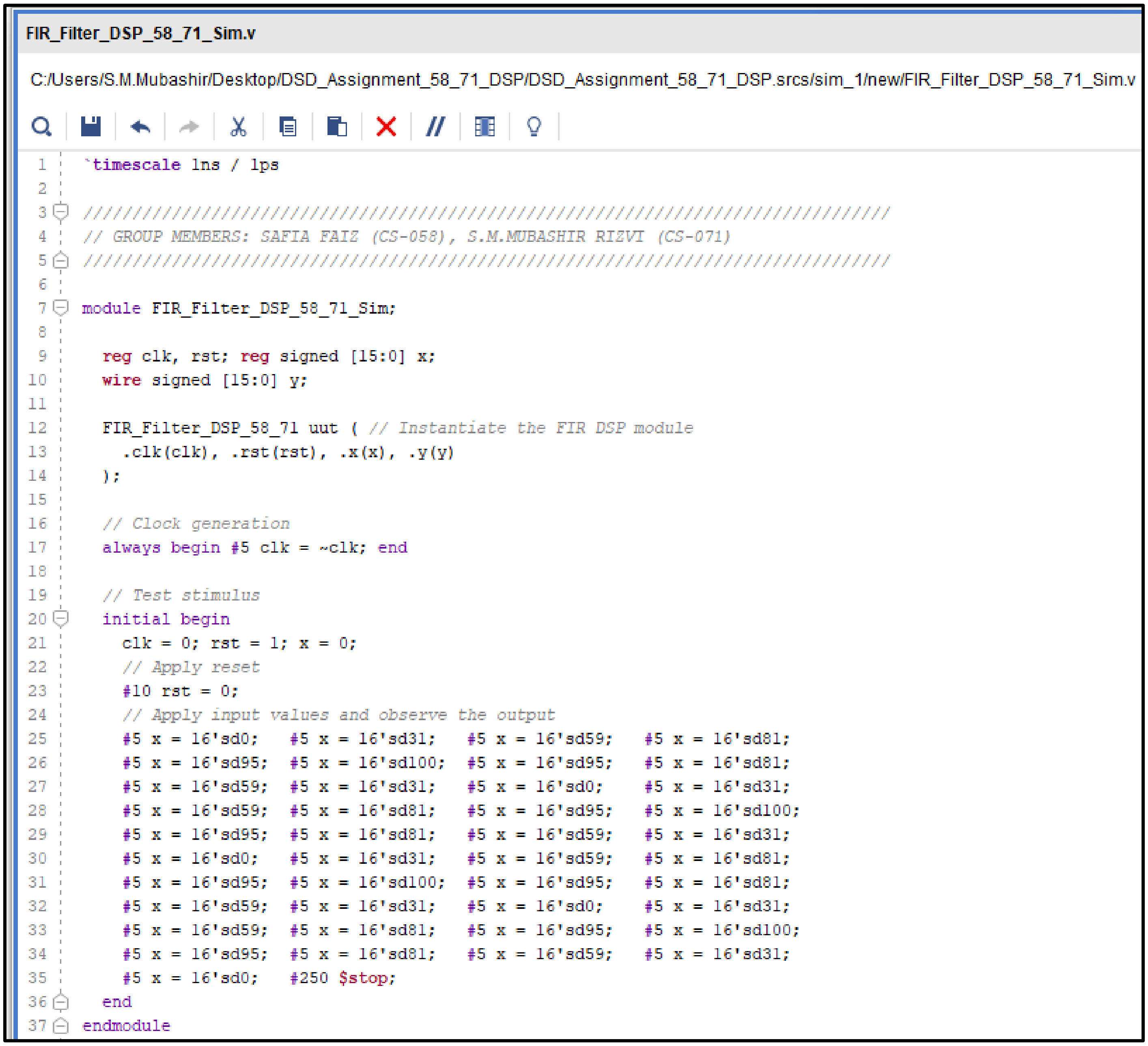


FIR_Filter_CLB_58_71_Sim.v

C:/Users/S.M.Mubashir/Desktop/DSD_Assignment_58_71_CLB/DSD_Assignment_58_71_CLB.srcs/sim_1/new/FIR_Filter_CLB_58_71_Sim.v

```verilog
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
// GROUP MEMBERS: SAFIA FAIZ (CS-058), S.M.MUBASHIR RIZVI (CS-071)
//////////////////////////////////////////////////////////////////////////////////

module FIR_Filter_CLB_58_71_Sim;

  reg clk, rst; reg signed [15:0] x;
  wire signed [15:0] y;

  FIR_Filter_CLB_58_71 uut (    // Instantiate the FIR CLB module
    .clk(clk), .rst(rst), .x(x), .y(y)
  );

  // Clock generation
  always begin #5 clk = ~clk; end

  // Test stimulus
  initial begin
    clk = 0; rst = 1; x = 0;
    // Apply reset
    #10 rst = 0;
    // Apply input values and observe the output
    #5 x = 16'sd0;   #5 x = 16'sd31;   #5 x = 16'sd59;   #5 x = 16'sd81;
    #5 x = 16'sd95;  #5 x = 16'sd100;  #5 x = 16'sd95;   #5 x = 16'sd81;
    #5 x = 16'sd59;  #5 x = 16'sd31;   #5 x = 16'sd0;    #5 x = 16'sd31;
    #5 x = 16'sd59;  #5 x = 16'sd81;   #5 x = 16'sd95;   #5 x = 16'sd100;
    #5 x = 16'sd95;  #5 x = 16'sd81;   #5 x = 16'sd59;   #5 x = 16'sd31;
    #5 x = 16'sd0;   #5 x = 16'sd31;   #5 x = 16'sd59;   #5 x = 16'sd81;
    #5 x = 16'sd95;  #5 x = 16'sd100;  #5 x = 16'sd95;   #5 x = 16'sd81;
    #5 x = 16'sd59;  #5 x = 16'sd31;   #5 x = 16'sd0;    #5 x = 16'sd31;
    #5 x = 16'sd59;  #5 x = 16'sd81;   #5 x = 16'sd95;   #5 x = 16'sd100;
    #5 x = 16'sd95;  #5 x = 16'sd81;   #5 x = 16'sd59;   #5 x = 16'sd31;
    #5 x = 16'sd0;   #250 $stop;
  end
endmodule
```

Test Bench for DSP Implementation:



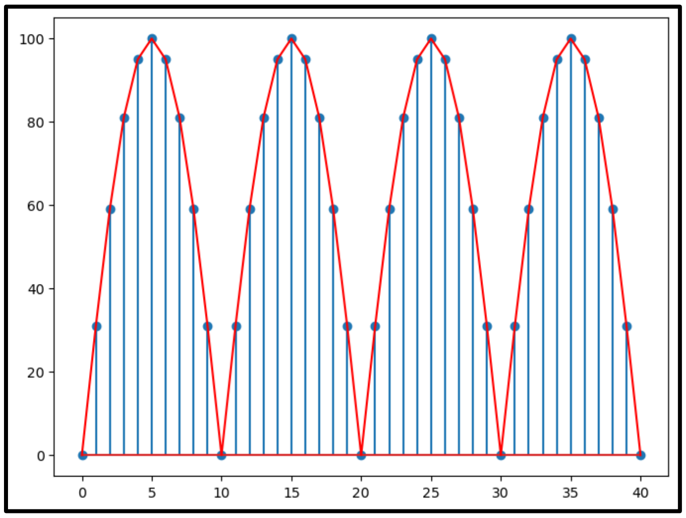
FIR_Filter_DSP_58_71_Sim.v

C:/Users/S.M.Mubashir/Desktop/DSD_Assignment_58_71_DSP/DSD_Assignment_58_71_DSP.srcs/sim_1/new/FIR_Filter_DSP_58_71_Sim.v

```verilog
`timescale 1ns / 1ps

/////////////////////////////////////////////////////////////////////////////////
// GROUP MEMBERS: SAFIA FAIZ (CS-058), S.M.MUBASHIR RIZVI (CS-071)
/////////////////////////////////////////////////////////////////////////////////

module FIR_Filter_DSP_58_71_Sim;

  reg clk, rst; reg signed [15:0] x;
  wire signed [15:0] y;

  FIR_Filter_DSP_58_71 uut ( // Instantiate the FIR DSP module
    .clk(clk), .rst(rst), .x(x), .y(y)
  );

  // Clock generation
  always begin #5 clk = ~clk; end

  // Test stimulus
  initial begin
    clk = 0; rst = 1; x = 0;
    // Apply reset
    #10 rst = 0;
    // Apply input values and observe the output
    #5 x = 16'sd0;   #5 x = 16'sd31;  #5 x = 16'sd59;  #5 x = 16'sd81;
    #5 x = 16'sd95;  #5 x = 16'sd100; #5 x = 16'sd95;  #5 x = 16'sd81;
    #5 x = 16'sd59;  #5 x = 16'sd31;  #5 x = 16'sd0;   #5 x = 16'sd31;
    #5 x = 16'sd59;  #5 x = 16'sd81;  #5 x = 16'sd95;  #5 x = 16'sd100;
    #5 x = 16'sd95;  #5 x = 16'sd81;  #5 x = 16'sd59;  #5 x = 16'sd31;
    #5 x = 16'sd0;   #5 x = 16'sd31;  #5 x = 16'sd59;  #5 x = 16'sd81;
    #5 x = 16'sd95;  #5 x = 16'sd100; #5 x = 16'sd95;  #5 x = 16'sd81;
    #5 x = 16'sd59;  #5 x = 16'sd31;  #5 x = 16'sd0;   #5 x = 16'sd31;
    #5 x = 16'sd59;  #5 x = 16'sd81;  #5 x = 16'sd95;  #5 x = 16'sd100;
    #5 x = 16'sd95;  #5 x = 16'sd81;  #5 x = 16'sd59;  #5 x = 16'sd31;
    #5 x = 16'sd0;   #250 $stop;
  end
endmodule
```

Behavioral Simulation:

- In both of test bench codes, we have given inputs of the signal using signed decimal digits of 16 bits each.
- Here is the signal of which values were converted into signed decimal digits:

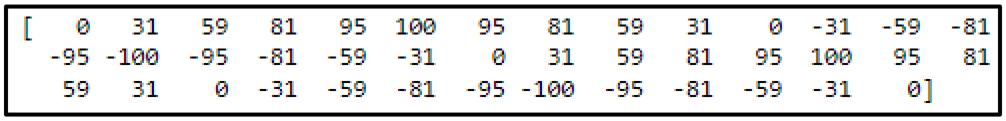


- It is a sine signal.
- $100 * \sin(2 * \pi * \frac{1000}{20000} * n)$
- Frequency: 1000Hz
- Sampling Frequency: 20000 Hz

- The signal values are rounded off to the nearest integer to input them easily into the test bench as signed decimal integers.

- First 41 samples of the signal are used in the test bench and are shown below:

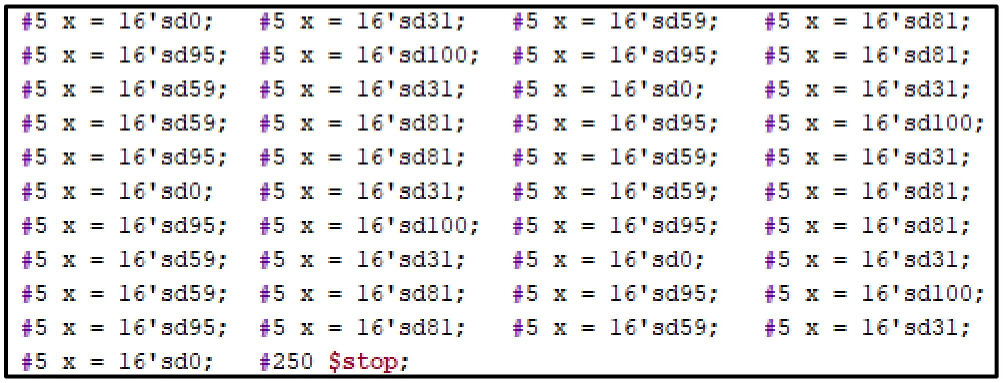
Python:



```
[   0   31   59   81   95  100   95   81   59   31    0  -31  -59  -81
  -95 -100  -95  -81  -59  -31    0   31   59   81   95  100   95   81
   59   31    0  -31  -59  -81  -95 -100  -95  -81  -59  -31    0]
```

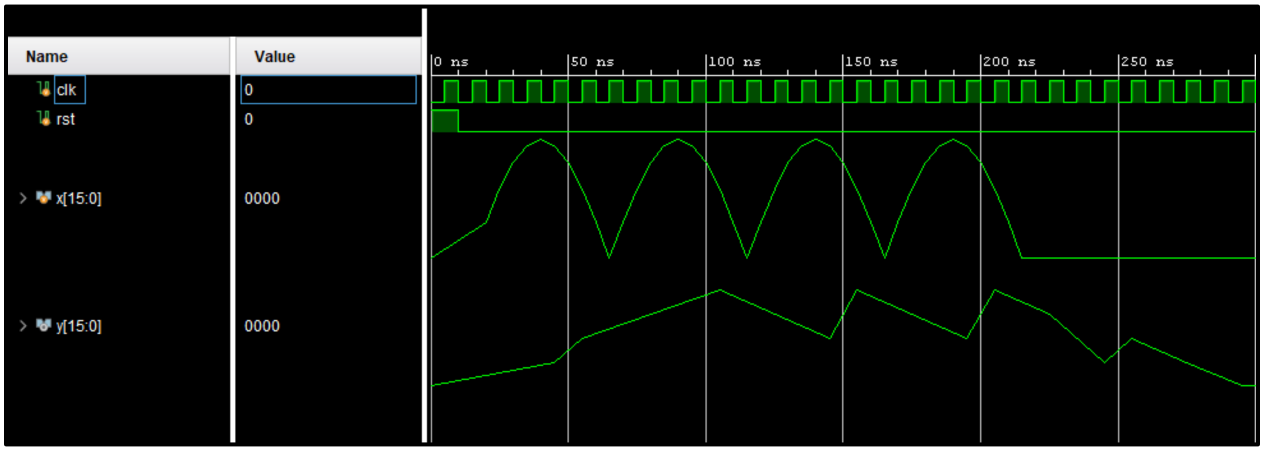
Test

Bench:

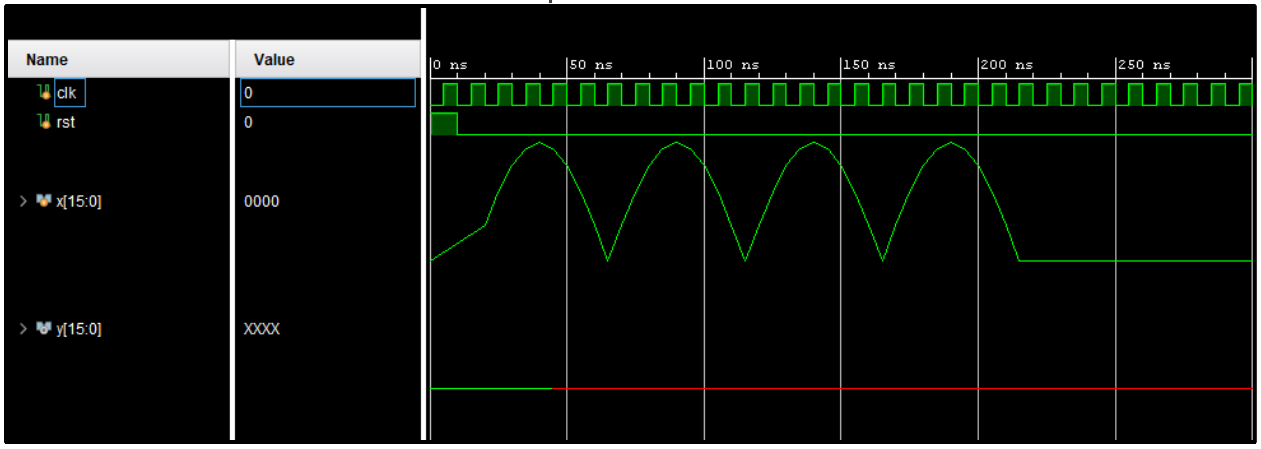


```
#5 x = 16'sd0;    #5 x = 16'sd31;    #5 x = 16'sd59;    #5 x = 16'sd81;
#5 x = 16'sd95;   #5 x = 16'sd100;   #5 x = 16'sd95;    #5 x = 16'sd81;
#5 x = 16'sd59;   #5 x = 16'sd31;    #5 x = 16'sd0;     #5 x = 16'sd31;
#5 x = 16'sd59;   #5 x = 16'sd81;    #5 x = 16'sd95;    #5 x = 16'sd100;
#5 x = 16'sd95;   #5 x = 16'sd81;    #5 x = 16'sd59;    #5 x = 16'sd31;
#5 x = 16'sd0;    #5 x = 16'sd31;    #5 x = 16'sd59;    #5 x = 16'sd81;
#5 x = 16'sd95;   #5 x = 16'sd100;   #5 x = 16'sd95;    #5 x = 16'sd81;
#5 x = 16'sd59;   #5 x = 16'sd31;    #5 x = 16'sd0;     #5 x = 16'sd31;
#5 x = 16'sd59;   #5 x = 16'sd81;    #5 x = 16'sd95;    #5 x = 16'sd100;
#5 x = 16'sd95;   #5 x = 16'sd81;    #5 x = 16'sd59;    #5 x = 16'sd31;
#5 x = 16'sd0;    #250 $stop;
```

Behavioral Simulation of CLB Implementation:



Behavioral Simulation of DSP Implementation:



- The above simulation for DSP implementation of the filter does not show the correct output.
- We deduced that this may be due to some logical error in the Verilog code as the test bench for both CLB and DSP implementations is the same.

- However, we were successful in giving the desired signal as input which we wanted to test, and it is being displayed in the same manner as it is shown above in the Behavioral Simulation heading.

6. Documentation and Analysis:

- The graphs and reports of both implementations are shown in above.
- The Verilog code for both implementations, their test bench codes, the testing signal, the outputs, everything is shown above with side-by-side comparison of both implementations.

CLB Implementation:

- **Strengths:**
    - **Flexibility:** CLBs offer a high degree of flexibility, allowing us to implement a variety of functions, including FIR filters.
    - **Parallelism:** CLBs can be configured in parallel to perform multiple operations simultaneously, which is beneficial for efficient FIR filter implementations.
    - **Customization:** We have the ability to customize the CLB configuration to meet specific filter requirements.

- **Limitations:**
    - **Limited Resources:** CLBs may have limited resources, which can be a constraint when implementing complex FIR filters.
    - **Power Consumption:** The flexibility and customization of CLBs come at the cost of potentially higher power consumption and throughput compared to more specialized approaches.
    - **Complexity:** Configuring CLBs for FIR filters may require additional effort and expertise, making the design process more complex.

DSP Slices Implementation:

- **Strengths:**
    - **Dedicated Hardware:** DSP slices are specifically designed for digital signal processing tasks, providing dedicated resources for filter computations.
    - **Optimized Arithmetic:** DSP slices often include specialized arithmetic units that are optimized for efficient multiplication and accumulation, critical for FIR filters.
    - **Efficient Resource Usage:** DSP slices can be more resource-efficient for certain applications, especially those requiring extensive signal processing.

- **Limitations:**
    - **Limited Flexibility:** DSP slices are typically optimized for specific tasks and may not be as flexible as CLBs for general-purpose logic functions.
    - **Cost:** Designs heavily reliant on DSP slices may be costlier due to the specialized nature of the hardware.

Conclusion:

- We were successful in completing and producing the output for the CLB-based implementation of the low-pass FIR filter.

- However, we failed to successfully produce an output for the DSP-based implementation. We searched for many documents and guides, but it seemed difficult to even initialize DSP slices (i.e., converting the CLB-based implementation code into a DSP-based implementation), but we managed to do that.

- We understand that we may have used more DSP slices than needed for the accumulation operation of a filter and we believe that this can be achieved using a smaller number of DSP slices.

- DSP slices are useful and efficient for arithmetic operations, especially for accumulation operations or multiplication of complex numbers.
- By using them, we can reduce the time it takes to calculate the final result and also reduce the number of resources being utilized in the design.