FPGA BASED FIR & IIR FILTERS USING PMOD MIC AND PMOD DA2

A Project Report by

Prathamesh Pise (2022A8PS1085G)

Ansh Garg (2022A8PS1360G)

Under the Guidance of

Prof. Amalin Prince A

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING
BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
K.K. BIRLA GOA CAMPUS
APRIL 2025

Problem Statement :

FPGA-Based Real-Time FIR and IIR Filter Implementation.

Basic Idea :

Digital filters are essential in signal processing for noise reduction and feature extraction. Implementing FIR and IIR filters on an FPGA provides real-time processing with minimal latency.

Basic Information :

Software : Vivado 2021.2
Sensors : Pmod MIC, Pmod DA2
FPGA Board : Zedboard (xc7z020clg484-1)
Code : Verilog
Interface : SPI

Objective :

This project aims to design and implement a real-time Infinite Impulse Response (IIR) digital filter on a ZedBoard FPGA using Vivado, with the ability to configure filter coefficients dynamically. The system captures audio input through a PMOD microphone and outputs the filtered signal via a PMOD DA2 DAC module. Additionally, the project aims to compare the performance and accuracy of FPGA-based filtering with equivalent software-based filtering methods to evaluate the efficiency and real-time capabilities of the hardware implementation.

Components Used :

1. FPGA - Zynq 700 Zedboard (xc7z020clg484-1)
2. Mic - Pmod Mic
3. DAC - Pmod DA2
4. Oscilloscope
5. Monitor and Connecting Cables
6. Software - Xilinx Vivado (Verilog), Matlab

Design Approach :

Approaching the problem, we interfaced the microphone and DAC with the FPGA board using SPI communication. The FPGA operates at a clock frequency of 100 MHz. To make the Pmod modules work, we divided the clock down to 1 MHz for the microphone and 25 MHz for the DA2. The input from the microphone is 12-bit data (000–FFF), and the output from the DAC is also 12-bit data, corresponding to a voltage range of 0 V to 3.3 V.

Working Principle :

1. Input from Microphone :

   The microphone is connected to the FPGA using SPI (Serial Peripheral Interface). Input data is acquired from the microphone and validated using an onboard LED.

2. Filter Implementation :

   The coefficients for the FIR and IIR filters were generated using MATLAB. These coefficients were then used to implement the filters in Verilog. The input signal is passed through the filters to reduce noise and eliminate unwanted components.

3. Output to DA2 :

   The filtered output is sent to the DA2, which converts the digital signal into an analog signal ready for display.

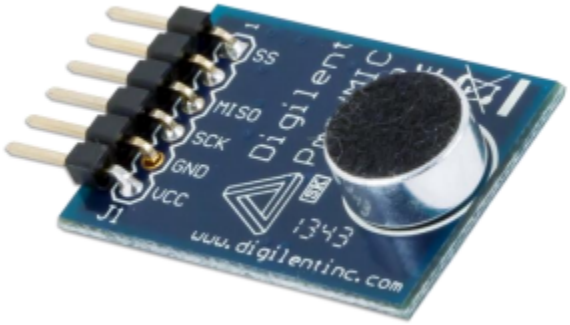
4. Output to oscilloscope :

   The DA2 output is connected to an oscilloscope using wires and probes. The final filtered waveform can be observed on the oscilloscope.

Device Information :

1. Pmod MIC :

Sensor Image :



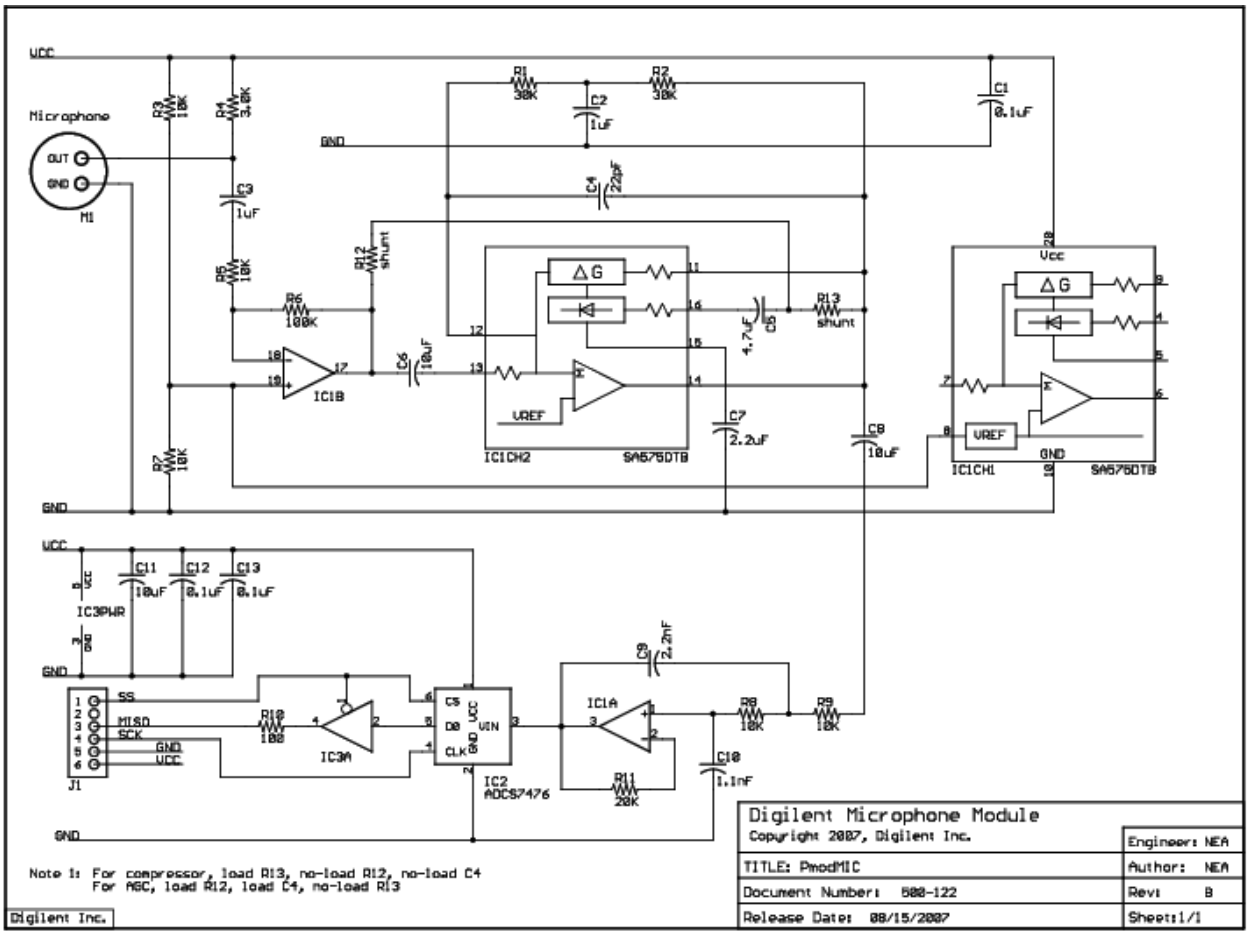
Pinout Table :

Pin	Signal	Description
1	SS	Chip Select
2	NC	Not Connected
3	MISO	Master-In-Slave-Out
4	SCK	Serial Clock
5	GND	Power Supply Ground
6	VCC	Power Supply (3.3V/5V)

Functional Description :

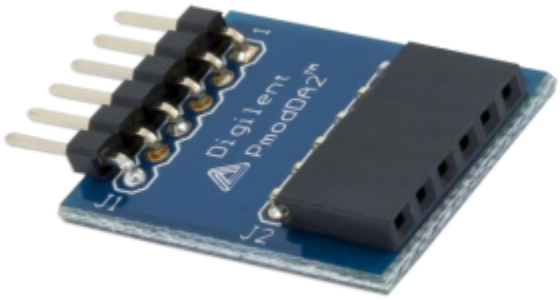
The PmodMIC is designed to digitally report to the host board whenever it detects any external noise. By sending a 12-bit digital value representative of the frequency and volume of the noise, this number can be processed by the system board.

Schematic :


VCC
Microphone
OUT
GND
M1
R3 10K
R4 3.0K
C3 1uF
R5 10K
R7 10K
R1 30K
R2 30K
C2 1uF
C1 0.1uF
C4 22pF
R12 shunt
R6 100K
IC1B
C6 10uF
IC1CH2
SA575DTB
VREF
C7 2.2uF
C8 10uF
R13 shunt
4.7uF C5
IC1CH1
SA575DTB
Vcc
GND
C11 10uF
C12 0.1uF
C13 0.1uF
IC3PWR
J1
SS
MISO
SCK
GND
VCC
R10 100
IC3A
IC2
ADCS7476
C9 2.2nF
IC1A
R8 10K
R9 10K
C10 1.1nF
R11 20K
Digilent Microphone Module
Copyright 2007, Digilent Inc.
TITLE: PmodMIC
Document Number: 500-122
Release Date: 08/15/2007
Engineer: NEA
Author: NEA
Rev: B
Sheet:1/1
Note 1: For compressor, load R13, no-load R12, no-load C4
For AGC, load R12, load C4, no-load R13
Digilent Inc.

2. Pmod DA2 :

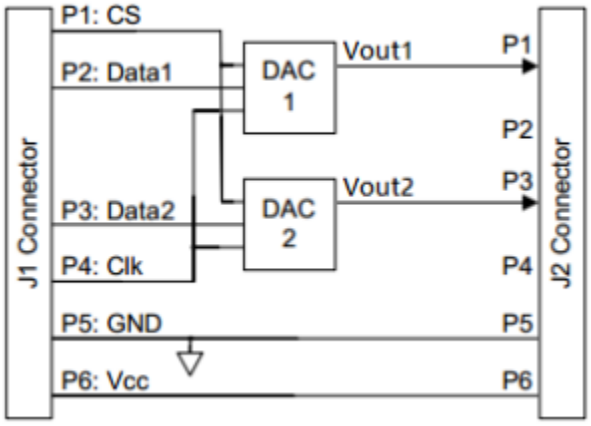
Device Image :



Pinout Table :

Pin	Signal	Description
1	~SYNC	Chip Select
2	DINA	Data In for Channel A
3	DINB	Data In for Channel B
4	SCLK	Serial Clock
5	GND	Power Supply Ground
6	VCC	Power Supply (3.3V/5V)

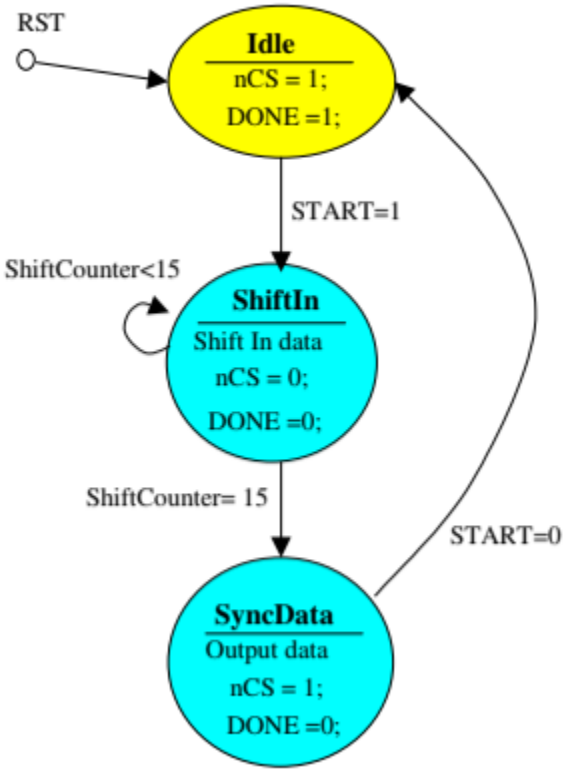
Schematic :



Functional Description :

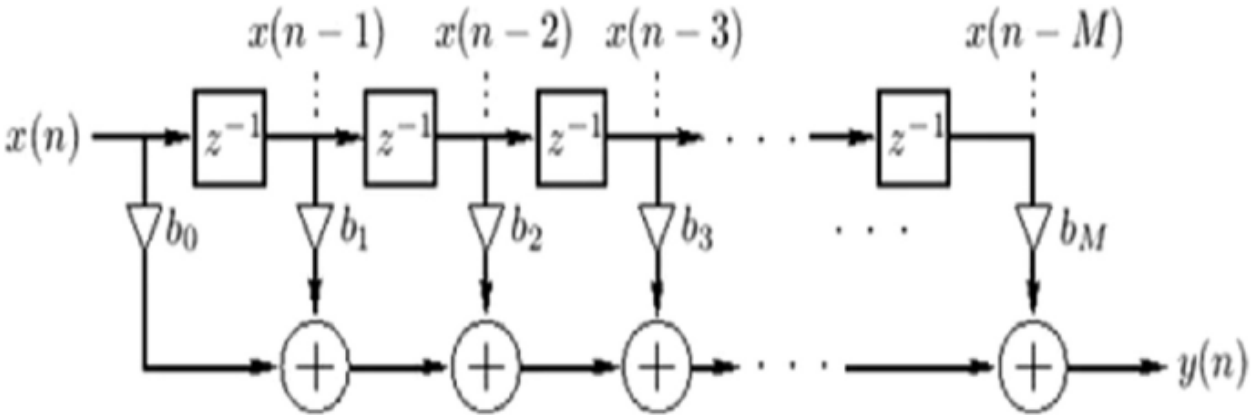
The PmodDA2 provides two channels of 12-bit Digital-to-Analog conversion, allowing users to achieve a resolution up to about 1 mV.

State Diagram :



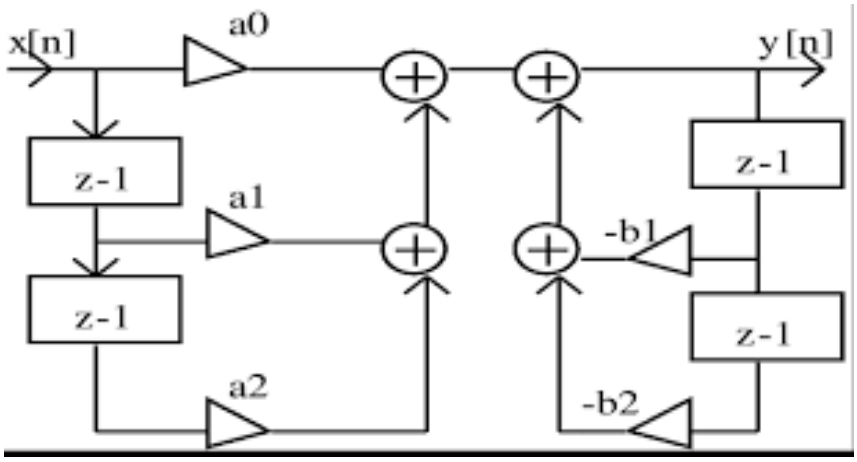
Block Diagram :

1. FIR Filter



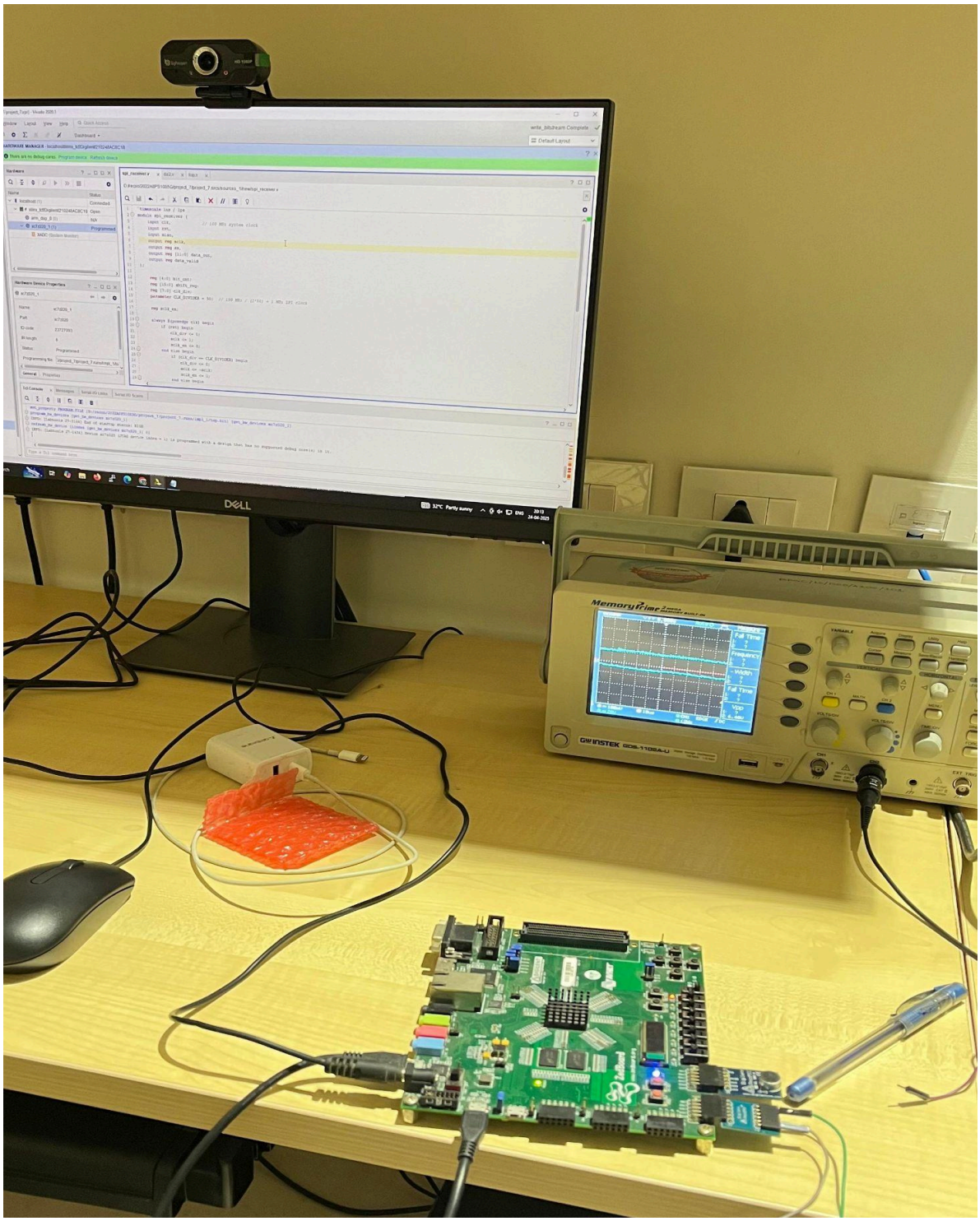
We have implemented an 8-tap FIR Filter.

2. IIR Filter

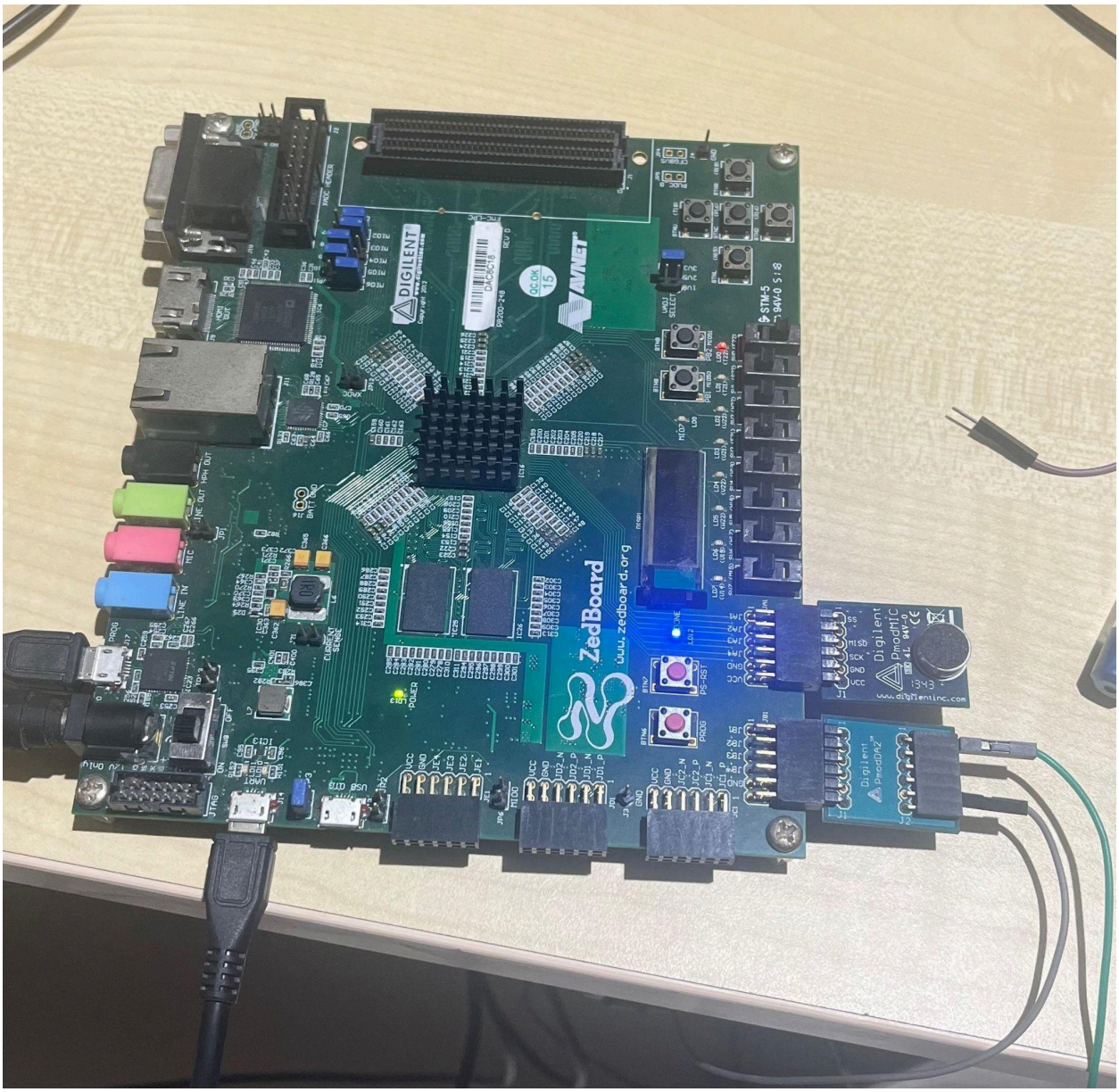


Working Setup :

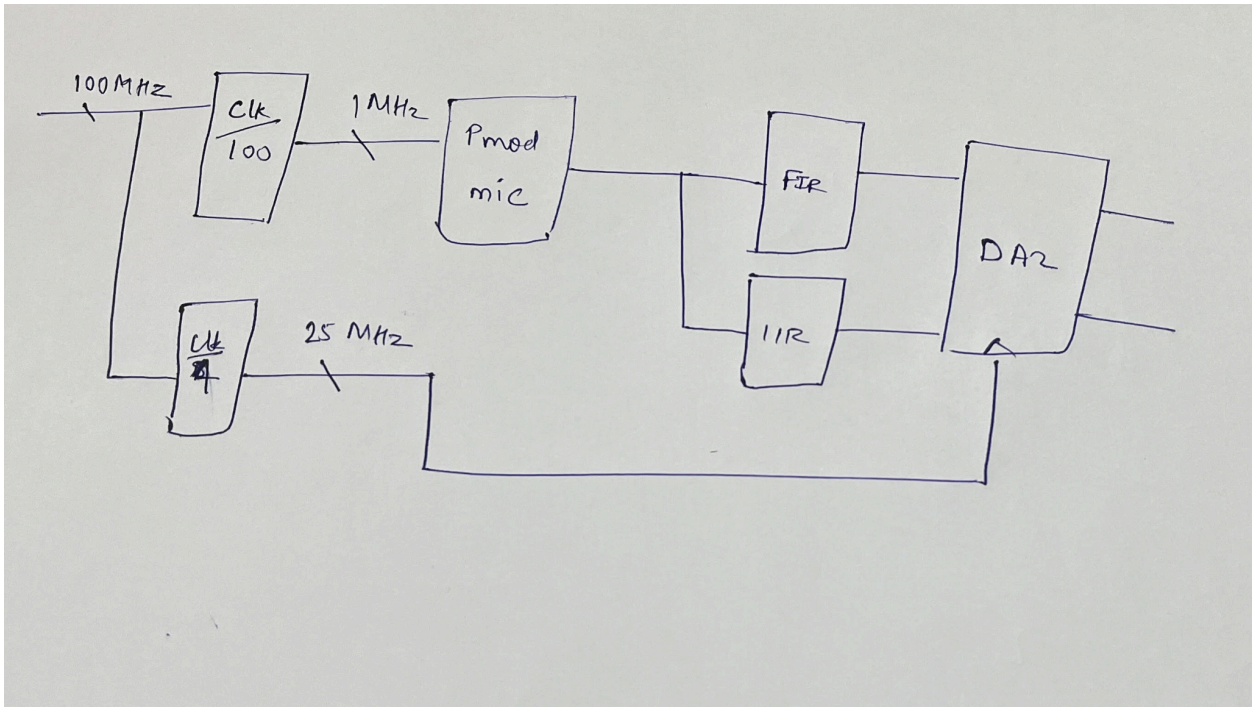
1.



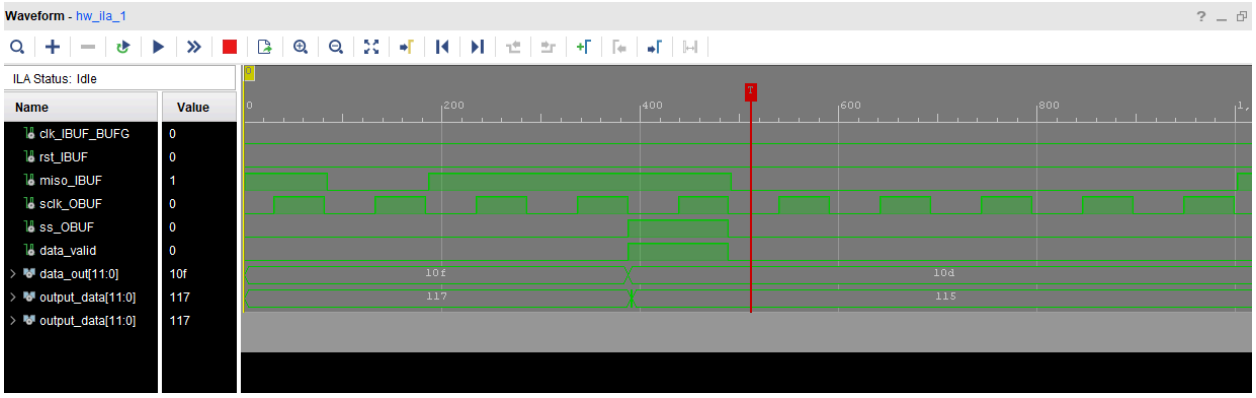
2.


DIGILENT
AVNET
ZedBoard
www.zedboard.org
Digilent
PmodMIC
Digilent
PmodDA2

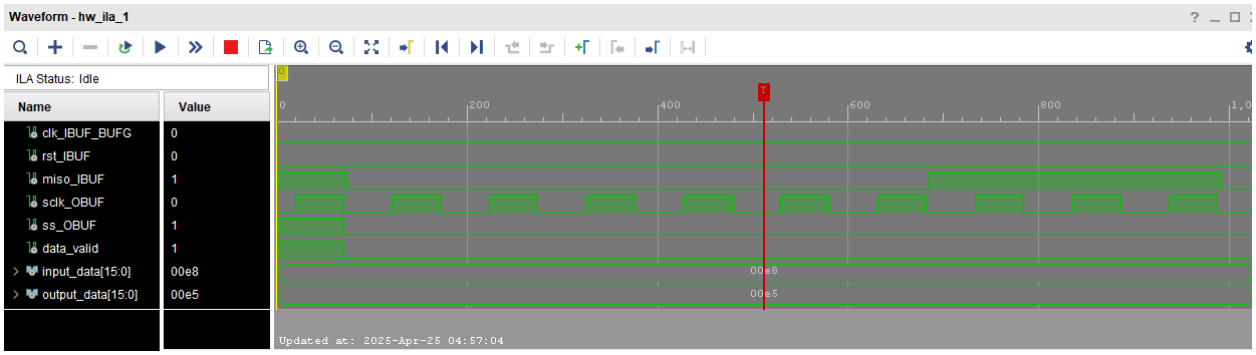
Block Diagram :



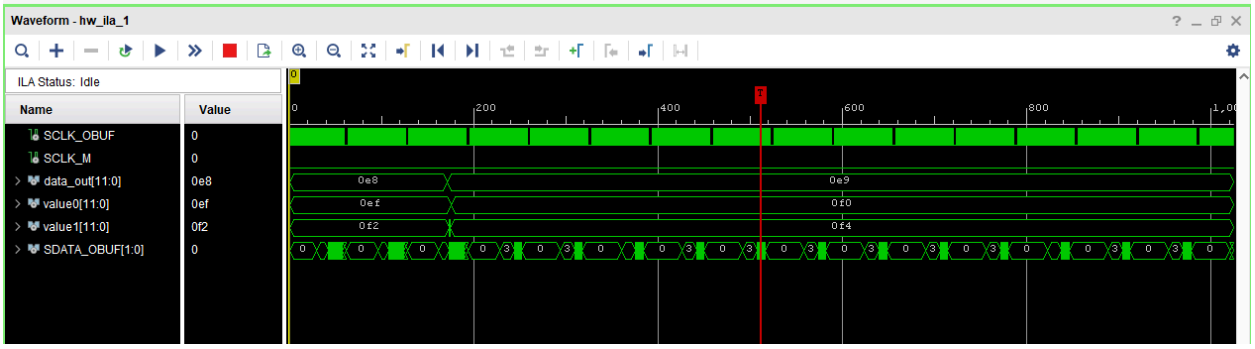
ILA FIR Simulation :



ILA IIR Simulation :

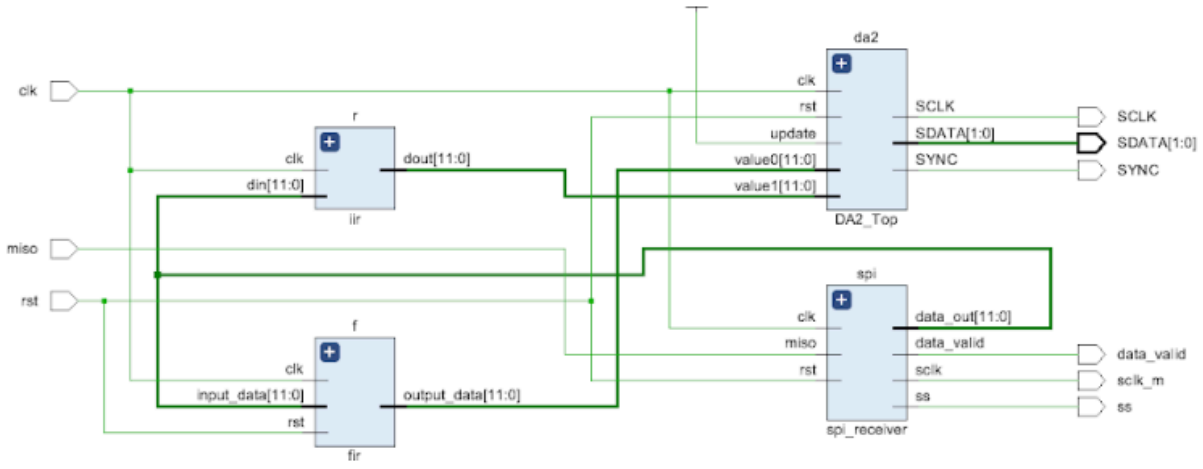


Complete ILA simulation :

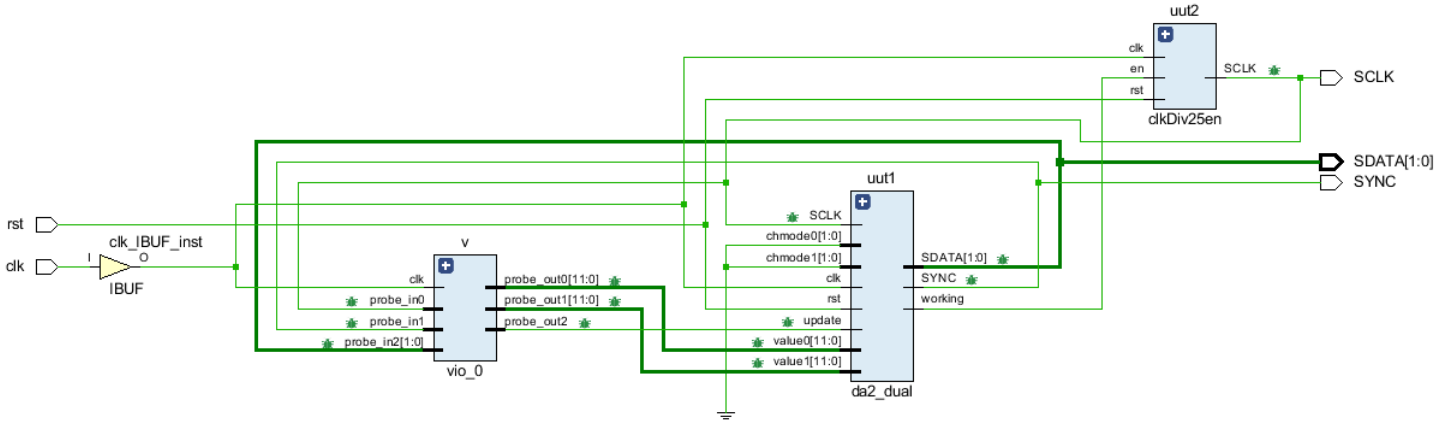


RTL Schematic :

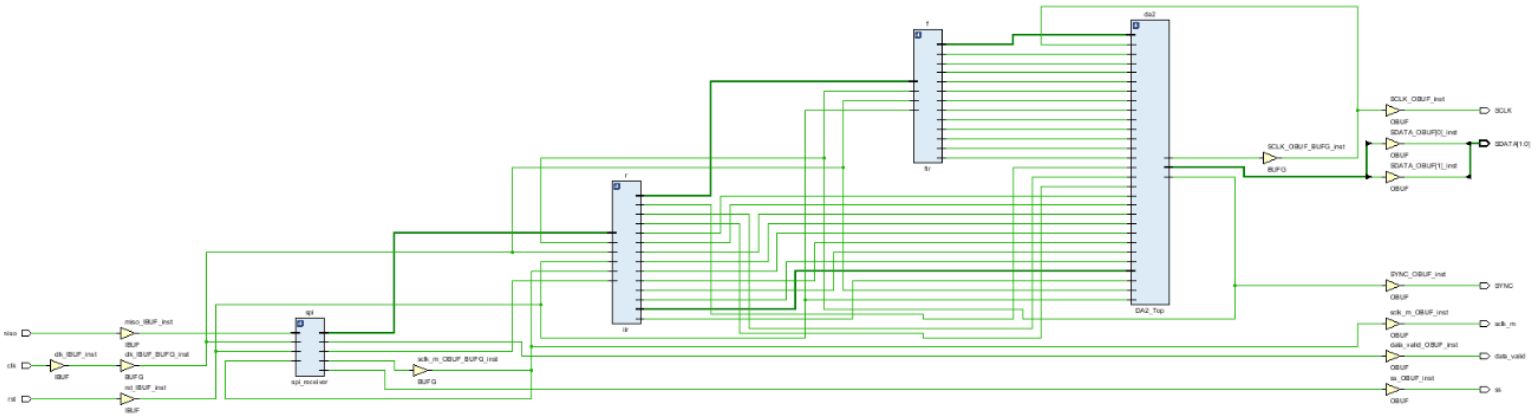
Complete RTL :



DA2 RTL :



Synthesis Schematic :



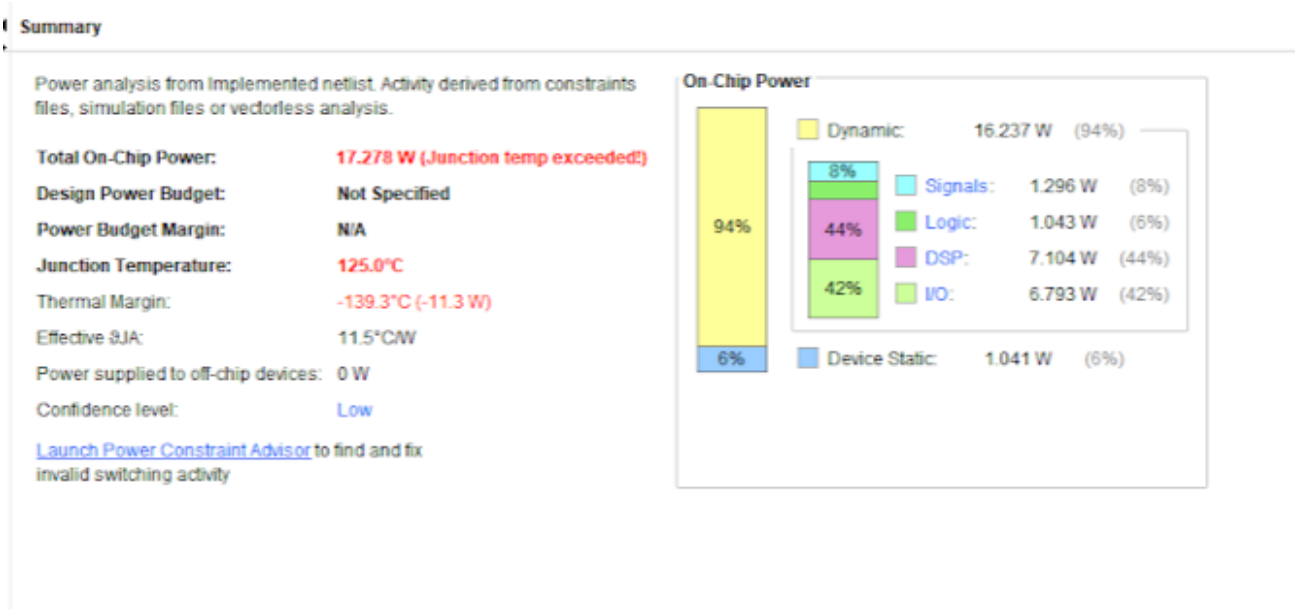
Utilization Report :

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	219	53200	0.41
LUTRAM	14	17400	0.08
FF	212	106400	0.20
DSP	8	220	3.64
IO	10	200	5.00
BUFG	3	32	9.38

On Chip Power Report :



Applications :

There are several useful uses for FIR and IIR filter implementation on FPGA platforms, like the ZedBoard, in real-time digital signal processing. These filters are crucial components of audio processing systems because they improve speech or music signals, reduce noise, and equalize them. They aid in the removal of artifacts and enhancement of signal clarity in biomedical signal processing by filtering physiological signals such as ECG and EEG. Furthermore, FPGA-based filtering provides high-speed performance and low latency, which makes it appropriate for software-defined radio, radar, and real-time communication systems. The approach is adaptable and effective for embedded DSP activities because the filter coefficients can be changed to create adaptive systems that can react dynamically to changing input conditions.

Future Scope :

This project sets the stage for more advanced FPGA-based digital signal processing systems. In the future, the filter design can be expanded to support stereo input/output or multi-channel audio processing for more complex applications. By adding machine learning, the system could adjust filter settings automatically based on the signal. The design can also be improved to handle more complex signals with higher-order or multiband filters. A GUI interface or HDMI can be used to show input and output waveforms in real time, making the system easier to use. This approach also opens up opportunities for creating low-power DSP solutions for smart electronics, portable communication devices, and wearable health gadgets.

Challenges Faced :

1. During the implementation, the input to both FIR and IIR filters was 12-bit data from the PMOD MIC. However, while processing the signal, we had to use 32-bit data because of repeated addition and multiplication operations inside the loops. This helped prevent data loss and overflow. Since the final output also needed to be 12-bit for the PMOD DA2, we applied techniques like bit slicing and truncation to convert the 32-bit processed data back to 12-bit, ensuring the output was accurate and within the required range.

2. Interfacing the PMOD MIC and PMOD DA2 with the ZedBoard was one of the more challenging parts of the project. It required going through a lot of documentation and resources to understand how to properly connect and communicate with these modules. One key learning was about the external connection pin constraints, which had to be carefully considered to ensure proper signal routing and functionality. This part of the project greatly improved our understanding of hardware interfacing and low-level signal handling on FPGA platforms.

Result / Conclusion :

The FIR and IIR filters were successfully implemented on the ZedBoard FPGA, allowing for real-time signal processing with very little delay. The filters worked well with input signals from the PMOD MIC and produced accurate outputs through the PMOD DA2. When compared to software-based filters, the FPGA filters showed faster response times and better efficiency. The ability to adjust the filter settings easily made the system flexible for different types of signals. The system also used less power and worked quickly, making it suitable for real-time applications like audio processing and communication. Overall, the project achieved its goals and provided a reliable solution for filtering signals in real time.

References :

- https://digilent.com/reference/pmod/pmodda2/start
- https://digilent.com/reference/pmod/pmodmic3/start
- https://www.youtube.com/@aleksandarhaber
- https://www.youtube.com/@Vipinkmenon
- https://github.com/suoglu/Pmod/tree/master/Pmods/DA2

Verilog Code :

1. Datapath

```
`timescale 1ns / 1ps
module top(input clk,
    input rst,
    output [1:0] SDATA,
    output SCLK,
    output SYNC,

    input miso,
    output  sclk_m,
    output  ss,
    output  data_valid
    );



wire [11:0] value0,value1;
wire[11:0] data_out;
wire update;

   DA2_Top da2
(.clk(clk),.rst(rst),.update(1'b1),.value0(value0),.value1(value1),.SDATA(SDATA),.SCLK(SCLK),.SYNC(SYNC
));

   spi_receiver
spi(.clk(clk),.rst(rst),.miso(miso),.sclk(sclk_m),.ss(ss),.data_out(data_out),.data_valid(data_valid));

   fir f (.input_data(data_out),.clk(clk),.rst(rst),.output_data(value0));

   iir r (.clk(clk),.din(data_out),.dout(value1));
endmodule
```

2. DA2_Top

```
module DA2_Top (
    input clk, rst, update,
   input [11:0] value0,value1,
    output [1:0] SDATA,
    output SYNC,SCLK
);
wire SCLK_en;

da2_dual uut1(
  clk,
  rst,
  SCLK,
  SDATA,
  SYNC,
  SCLK_en,
  2'b00,
  2'b00,
  //Channel modes: 00 Enabled
  value0,
  value1,
  update);

 clkDiv25en uut2(
  clk,
  rst,
  SCLK_en,
  SCLK);



endmodule
```

3. da2_dual

```
`timescale 1ns / 1ps

module da2_dual(
  input clk,
  input rst,
  //Serial data line
  input SCLK,
  output [1:0] SDATA,
  output reg SYNC,
  //Enable clock
  output reg working,
  //Output value and mode
  input [1:0] chmode0,
  input [1:0] chmode1,
  //Channel modes: 00 Enabled, Power off modes: 01 1kOhm, 10 100kOhm, 11 High-
Z input [11:0] value0,
  input [11:0] value1,
  //Control signals
  input update
  /*output reg start*/);
  reg count;
  reg contCount;
  reg [3:0] counter; //Count edges
  wire [15:0] SDATAbuff_cont0, SDATAbuff_cont1;
  reg [15:0] SDATAbuff0, SDATAbuff1;

  //Handle SDATA buffer
  initial begin
  SDATAbuff0[15]<=1'b0;
  SDATAbuff1[15]<=1'b0;
  //start<=1'b0;
  end
  assign SDATAbuff_cont0 = {2'd0, chmode0, value0};
  assign SDATAbuff_cont1 = {2'd0, chmode1, value1};
  assign SDATA = {SDATAbuff1[15], SDATAbuff0[15]};
  always@(posedge SCLK or posedge SYNC) begin
    /*if(count==4'b0100)
    start<=1'b1;
    else
    start<=1'b0;*/
    if(SYNC) begin
      SDATAbuff0 <= SDATAbuff_cont0;
      SDATAbuff1 <= SDATAbuff_cont1;
    end else begin
      SDATAbuff0 <= (count) ? {SDATAbuff0[14:0], 1'b0} : SDATAbuff0;
      SDATAbuff1 <= (count) ? {SDATAbuff1[14:0], 1'b0} : SDATAbuff1;
    end
  end

  //count
  always@(posedge clk or posedge rst) begin
    if(rst) begin
        count <= 1'b0;
    end else case(count)
      1'b0: count <= SCLK & contCount;
      1'b1:  count <= (counter != 4'd0) | contCount;
    endcase
  end

  //contCount
  always@(posedge clk or posedge SYNC) begin
    if(SYNC) begin
      contCount <= 1'b1;
    end else begin
      contCount <= working & contCount & (counter != 4'd15);
    end
  end


  //working
  always@(posedge clk or posedge rst) begin
    if(rst) begin
        working <= 1'b0;
    end else case(working)
      1'b0: working <= SYNC;
      1'b1: working <= (counter != 4'd0) | contCount;
    endcase
  end

  //SYNC
  always@(posedge clk or posedge rst) begin
    if(rst) begin
        SYNC <= 1'b0;
    end else case(SYNC)
      1'b0: SYNC <= update & ~(contCount | count);
      1'b1: SYNC <= 1'b0;
    endcase
  end

  //Count SCLK
  always@(negedge SCLK or posedge SYNC) begin
    if(SYNC) begin
      counter <= 4'd0;
    end else begin
      counter <= counter + {3'd0, count};
    end
  end
endmodule//da2
```

```
`timescale 1ns / 1ps

module da2_dual(
  input clk,
  input rst,
  //Serial data line
  input SCLK,
  output [1:0] SDATA,
  output reg SYNC,
  //Enable clock
  output reg working,
  //Output value and mode
  input [1:0] chmode0,
  input [1:0] chmode1,
   //Channel modes: 00 Enabled, Power off modes: 01 1kOhm, 10 100kOhm,
11 High-Z
  input [11:0] value0,
  input [11:0] value1,
  //Control signals
  input update
  /*output reg start*/);
  reg count;
  reg contCount;
  reg [3:0] counter; //Count edges
  wire [15:0] SDATAbuff_cont0, SDATAbuff_cont1;
  reg [15:0] SDATAbuff0, SDATAbuff1;

  //Handle SDATA buffer
  initial begin
  SDATAbuff0[15]<=1'b0;
  SDATAbuff1[15]<=1'b0;
  //start<=1'b0;
  end
  assign SDATAbuff_cont0 = {2'd0, chmode0, value0};
  assign SDATAbuff_cont1 = {2'd0, chmode1, value1};
  assign SDATA = {SDATAbuff1[15], SDATAbuff0[15]};
  always@(posedge SCLK or posedge SYNC) begin
    /*if(count==4'b0100)
    start<=1'b1;
    else
```

```
  start<=1'b0;*/
  if(SYNC) begin
    SDATAbuff0 <= SDATAbuff_cont0;
    SDATAbuff1 <= SDATAbuff_cont1;
  end else begin
    SDATAbuff0 <= (count) ? {SDATAbuff0[14:0], 1'b0} : SDATAbuff0;
    SDATAbuff1 <= (count) ? {SDATAbuff1[14:0], 1'b0} : SDATAbuff1;
  end
end

//count
always@(posedge clk or posedge rst) begin
  if(rst) begin
      count <= 1'b0;
  end else case(count)
    1'b0: count <= SCLK & contCount;
    1'b1:  count <= (counter != 4'd0) | contCount;
  endcase
end

//contCount
always@(posedge clk or posedge SYNC) begin
  if(SYNC) begin
    contCount <= 1'b1;
  end else begin
    contCount <= working & contCount & (counter != 4'd15);
  end
end


//working
always@(posedge clk or posedge rst) begin
  if(rst) begin
      working <= 1'b0;
  end else case(working)
    1'b0: working <= SYNC;
    1'b1: working <= (counter != 4'd0) | contCount;
  endcase
end
```

```
  //SYNC
  always@(posedge clk or posedge rst) begin
    if(rst) begin
       SYNC <= 1'b0;
    end else case(SYNC)
      1'b0: SYNC <= update & ~(contCount | count);
      1'b1: SYNC <= 1'b0;
    endcase
  end

  //Count SCLK
  always@(negedge SCLK or posedge SYNC) begin
    if(SYNC) begin
      counter <= 4'd0;
    end else begin
      counter <= counter + {3'd0, count};
    end
  end
endmodule
```

4. clkDiv25en

```
`timescale 1ns / 1ps

module clkDiv25en(
  input clk,
  input rst,
  input en,
  output reg SCLK);
  reg clk_m;

  //50 MHz
  always@(posedge clk) begin
    if(~en) begin
      clk_m <= 1'b0;
    end else begin
      clk_m <= ~clk_m;
    end
  end
  //25 MHz
  always@(posedge clk_m or negedge en) begin
    if(~en) begin
      SCLK <= 1'b0;
    end else begin
      SCLK <= ~SCLK;
    end
  end

endmodule//clkDiv25
```

5. Spi_reciever

```
`timescale 1ns / 1ps
module spi_receiver (
    input clk,              // 100 MHz system clock
    input rst,
    input miso,
    output reg sclk,
    output reg ss,
    output reg [11:0] data_out,
    output reg data_valid
);

    reg [4:0] bit_cnt;
    reg [15:0] shift_reg;
    reg [7:0] clk_div;
    parameter CLK_DIVIDER = 50;  // 100 MHz / (2*50) = 1 MHz SPI
clock
    reg sclk_en;

    always @(posedge clk) begin
        if (rst) begin
            clk_div <= 0;
            sclk <= 1;
            sclk_en <= 0;
        end else begin
            if (clk_div == CLK_DIVIDER) begin
                clk_div <= 0;
                sclk <= ~sclk;
                sclk_en <= 1;
            end else begin
                clk_div <= clk_div + 1;
                sclk_en <= 0;
            end
        end
    end

    always @(negedge sclk) begin
        if (rst) begin
            bit_cnt <= 0;
            shift_reg <= 0;
            ss <= 1;
            data_out <= 0;
            data_valid <= 0;
        end else if (sclk_en) begin
            ss <= 0;
            shift_reg <= {shift_reg[14:0], miso};
            bit_cnt <= bit_cnt + 1;

            if (bit_cnt == 15) begin
                data_out <= shift_reg[11:0]; // Capture lower 12 bits
                data_valid <= 1;
                bit_cnt <= 0;
                ss <= 1;  // Deselect after transfer
            end else begin
                data_valid <= 0;
            end
        end
    end
endmodule
```

```
`timescale 1ns / 1ps
module spi_receiver (
    input clk,            // 100 MHz system clock
    input rst,
    input miso,
    output reg sclk,
    output reg ss,
    output reg [11:0] data_out,
    output reg data_valid
);

    reg [4:0] bit_cnt;
    reg [15:0] shift_reg;
    reg [7:0] clk_div;
    parameter CLK_DIVIDER = 50;  // 100 MHz / (2*50) = 1 MHz SPI clock

    reg sclk_en;

    always @(posedge clk) begin
        if (rst) begin
            clk_div <= 0;
            sclk <= 1;
            sclk_en <= 0;
        end else begin
            if (clk_div == CLK_DIVIDER) begin
                clk_div <= 0;
                sclk <= ~sclk;
                sclk_en <= 1;
            end else begin
                clk_div <= clk_div + 1;
                sclk_en <= 0;
            end
        end
    end

    always @(negedge sclk) begin
        if (rst) begin
            bit_cnt <= 0;
            shift_reg <= 0;
            ss <= 1;
```

```
            data_out <= 0;
            data_valid <= 0;
        end else if (sclk_en) begin
            ss <= 0;
            shift_reg <= {shift_reg[14:0], miso};
            bit_cnt <= bit_cnt + 1;

            if (bit_cnt == 15) begin
                data_out <= shift_reg[11:0]; // Capture lower 12 bits
                data_valid <= 1;
                bit_cnt <= 0;
                ss <= 1;  // Deselect after transfer
            end else begin
                data_valid <= 0;
            end
        end
    end
endmodule
```

6. FIR

```
`timescale 1ns / 1ps
module fir(
    input signed [11:0] input_data,       // Q5.6 input
    input clk,
    input rst,
    output signed [11:0] output_data      // Q5.6 output
);

    parameter n1 = 8;    // Coefficient width (Q1.6)
    parameter n2 = 12;   // Input width (Q5.6)
    parameter n3 = 24;   // Accumulator width (Q6.12 max sum width)

    // Fixed-point Q1.6 coefficients (scaled by 64)
    wire signed [n1-1:0] coeff [0:7];

assign coeff[0] = 8'b00000000; //  0
assign coeff[1] = 8'b11111111; // -1
assign coeff[2] = 8'b00000110; //  6
assign coeff[3] = 8'b00011100; // 28
assign coeff[4] = 8'b00011100; // 28
assign coeff[5] = 8'b00000110; //  6
assign coeff[6] = 8'b11111111; // -1
assign coeff[7] = 8'b00000000; //  0


    // Sample delay line: Q5.6
    reg signed [n2-1:0] samples [0:7];

    // MAC result: Q6.12 (intermediate fixed-point result)
    reg signed [n3-1:0] mac_result;

    always @(posedge clk) begin
        if (rst) begin
            samples[0] <= 0; samples[1] <= 0; samples[2] <= 0; samples[3] <=
0;          samples[4] <= 0; samples[5] <= 0; samples[6] <= 0; samples[7] <=
0;          mac_result <= 0;
        end else begin
            // Multiply and accumulate (fixed-point Q6.12)
            mac_result <=  coeff[0] * input_data +
                           coeff[1] * samples[0] +
                           coeff[2] * samples[1] +
                           coeff[3] * samples[2] +
                           coeff[4] * samples[3] +
                           coeff[5] * samples[4] +
                           coeff[6] * samples[5] +
                           coeff[7] * samples[6];

            // Shift register update
            samples[0] <= input_data;
            samples[1] <= samples[0];
            samples[2] <= samples[1];
            samples[3] <= samples[2];
            samples[4] <= samples[3];
            samples[5] <= samples[4];
            samples[6] <= samples[5];
        end
    end

    // Scale result back from Q6.12 ? Q5.6 by right-shifting 6 bits
    assign output_data = mac_result[17:6];

endmodule
```

```
`timescale 1ns / 1ps
module fir(
    input signed [11:0] input_data,       // Q5.6 input
    input clk,
    input rst,
    output signed [11:0] output_data      // Q5.6 output
);

    parameter n1 = 8;    // Coefficient width (Q1.6)
    parameter n2 = 12;   // Input width (Q5.6)
    parameter n3 = 24;   // Accumulator width (Q6.12 max sum width)

    // Fixed-point Q1.6 coefficients (scaled by 64)
    wire signed [n1-1:0] coeff [0:7];

assign coeff[0] = 8'b00000000; //  0
assign coeff[1] = 8'b11111111; // -1
assign coeff[2] = 8'b00000110; //  6
assign coeff[3] = 8'b00011100; // 28
assign coeff[4] = 8'b00011100; // 28
assign coeff[5] = 8'b00000110; //  6
assign coeff[6] = 8'b11111111; // -1
assign coeff[7] = 8'b00000000; //  0


    // Sample delay line: Q5.6
    reg signed [n2-1:0] samples [0:7];

    // MAC result: Q6.12 (intermediate fixed-point result)
    reg signed [n3-1:0] mac_result;

    always @(posedge clk) begin
        if (rst) begin
            samples[0] <= 0; samples[1] <= 0; samples[2] <= 0; samples[3] <= 0;
            samples[4] <= 0; samples[5] <= 0; samples[6] <= 0; samples[7] <= 0;
            mac_result <= 0;
        end else begin
            // Multiply and accumulate (fixed-point Q6.12)
            mac_result <=  coeff[0] * input_data +
                        coeff[1] * samples[0] +
```

```
                    coeff[2] * samples[1] +
                    coeff[3] * samples[2] +
                    coeff[4] * samples[3] +
                    coeff[5] * samples[4] +
                    coeff[6] * samples[5] +
                    coeff[7] * samples[6];

            // Shift register update
            samples[0] <= input_data;
            samples[1] <= samples[0];
            samples[2] <= samples[1];
            samples[3] <= samples[2];
            samples[4] <= samples[3];
            samples[5] <= samples[4];
            samples[6] <= samples[5];
        end
    end

    // Scale result back from Q6.12 ? Q5.6 by right-shifting 6 bits
    assign output_data = mac_result[17:6];

endmodule
```

7. IIR

```
`timescale 1ns / 1ps
module iir(
    input clk,
    input signed [11:0] din,
    output reg signed [11:0] dout
    );

    // Fixed-point Q1.6 Coefficients (8-bit)
    reg signed [7:0] a1 = -8'sd71;
    reg signed [7:0] a2 =  8'sd25;
    reg signed [7:0] b0 =  8'sd5;
    reg signed [7:0] b1 =  8'sd9;
    reg signed [7:0] b2 =  8'sd5;

    // Internal delayed signals
    reg signed [15:0] rx = 0;
    reg signed [15:0] rx_1 = 0;
    reg signed [15:0] rx_2 = 0;
    reg signed [15:0] ry_1 = 0;
    reg signed [15:0] ry_2 = 0;

    wire signed [31:0] sum;
    wire signed [11:0] scaled;

    always @(posedge clk) begin
        rx <= din;
        rx_1 <= rx;
        rx_2 <= rx_1;
        ry_1 <= sum >>> 6;  // Scale back from Q1.6
        ry_2 <= ry_1;

        // Saturation logic
        if (scaled > 12'sd2047)
            dout <= 12'sd2047;
        else if (scaled < -12'sd2048)
            dout <= -12'sd2048;
        else
            dout <= scaled;
    end

    assign sum = (rx * b0) + (rx_1 * b1) + (rx_2 * b2)
               + (ry_1 * -a1) + (ry_2 * -a2);

    assign scaled = sum[17:6];  // Extract 12-bit scaled
endmodule
```

MATLAB Code :

1. **Code for Coefficients of FIR Filter** :

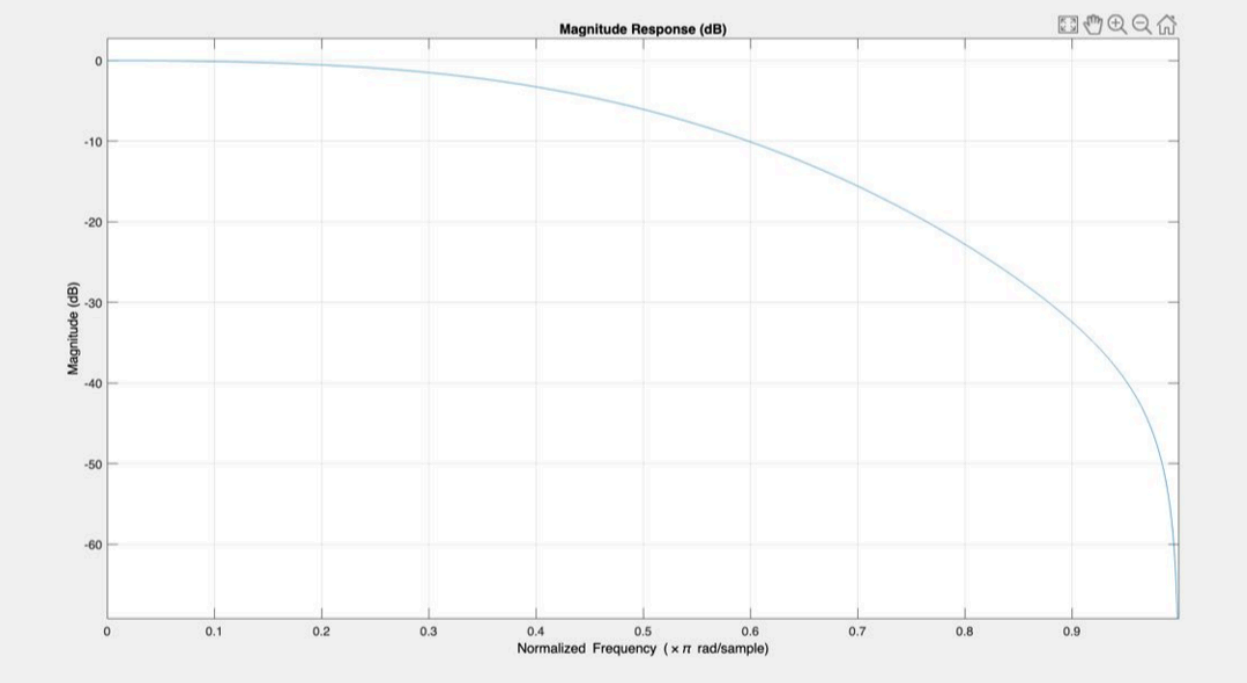
```
%% FIR Low-Pass Filter Specs
Fs = 16000;          % Sampling Frequency (Hz)
Fc = 4000;           % Cutoff Frequency (Hz)
N = 7;              % Filter order (results in N+1 taps)

% Normalize cutoff (0 to 1 scale, relative to Nyquist)
Wn = Fc / (Fs/2);

% Design FIR using Hamming window
h = fir1(N, Wn, 'low', hamming(N+1));

% Plot filter response
fvtool(h, 1);
title('Low-Pass FIR Filter Response');

%% Create Filter Object for HDL Coder
firFilt = dsp.FIRFilter('Numerator', h);


%% Generate HDL Code (Verilog or VHDL)
generatehdl(firFilt, ...
    'Name', 'LowPassFIR', ...
    'TargetLanguage', 'Verilog', ...
    'GenerateHDLTestbench', true, ...
    'OptimizeForHDL', true);
```

Coefficients for FIR

1x8 double

	1	2	3	4	5	6	7	8	9
1	-0.0052	-0.0229	0.0968	0.4313	0.4313	0.0968	-0.0229	-0.0052	
2									
3									
4									
5									
6									
7									
8									
9									
10									

Response of the FIR Filter



2. **Code for Coefficients of FIR Filter** :

```
% Parameters
Fs = 48000;              % Sampling frequency (Hz)
Fc = 5000;               % Cutoff frequency (Hz)
order = 2;               % Filter order

% Design a Butterworth low-pass filter
[b, a] = butter(order, Fc/(Fs/2));  % Normalized frequency

% Fixed-point scaling (Q1.15 format)
scalingFactor = 2^15;
b_fixed = round(b * scalingFactor);
a_fixed = round(a(2:end) * scalingFactor);  % Skip a0 (assumed to be 1)

% Create tables for export
coeffs_floating = table((0:length(b)-1)', b', [1; a(2:end)'], ...
    'VariableNames', {'Index', 'b_float', 'a_float'});
coeffs_fixed = table((0:length(b)-1)', b_fixed', [0; a_fixed'], ...
    'VariableNames', {'Index', 'b_fixed_Q15', 'a_fixed_Q15'});

% Save to Excel
filename = 'iir_coefficients.xlsx';
writetable(coeffs_floating, filename, 'Sheet', 1, 'Range', 'A1');
writetable(coeffs_fixed, filename, 'Sheet', 2, 'Range', 'A1');

disp(['Coefficients saved to ', filename]);
```

Coefficients for IIR Filter

	1 Index	2 b_fixed_Q15	3 a_fixed_Q15	4
1	0	2367	0	
2	1	4734	-36347	
3	2	2367	13047	
4				
5				
6				

Response of the IIR Filter