```
//////////////////////VERILOG DESIGN code ://////////////////////////
// 16th-Order (17-Tap) Low-Pass FIR Filter
module fir_filter #(
   parameter DATA_WIDTH = 16,
   parameter COEFF_WIDTH = 16
) (
   input                   clk,
   input                   reset,
   input    signed [DATA_WIDTH-1:0] data_in,
   output reg signed [DATA_WIDTH-1:0] data_out
);

   // Filter Coefficients (Quantized, 16-bit, from MATLAB)
   localparam signed [COEFF_WIDTH-1:0] B0  = -25;
   localparam signed [COEFF_WIDTH-1:0] B1  = -87;
   localparam signed [COEFF_WIDTH-1:0] B2  = -13;
   localparam signed [COEFF_WIDTH-1:0] B3  = 264;
   localparam signed [COEFF_WIDTH-1:0] B4  = 353;
   localparam signed [COEFF_WIDTH-1:0] B5  = -168;
   localparam signed [COEFF_WIDTH-1:0] B6  = -1092;
   localparam signed [COEFF_WIDTH-1:0] B7  = -815;
   localparam signed [COEFF_WIDTH-1:0] B8  = 5446;
   localparam signed [COEFF_WIDTH-1:0] B9  = -815;
   localparam signed [COEFF_WIDTH-1:0] B10 = -1092;
   localparam signed [COEFF_WIDTH-1:0] B11 = -168;
   localparam signed [COEFF_WIDTH-1:0] B12 = 353;
   localparam signed [COEFF_WIDTH-1:0] B13 = 264;
   localparam signed [COEFF_WIDTH-1:0] B14 = -13;
   localparam signed [COEFF_WIDTH-1:0] B15 = -87;
   localparam signed [COEFF_WIDTH-1:0] B16 = -25;

   localparam PRODUCT_WIDTH = DATA_WIDTH + COEFF_WIDTH;

   // 1. Input Data Shift Register (x[n], x[n-1], ..., x[n-16])
   reg signed [DATA_WIDTH-1:0] x_regs [0:16];
```

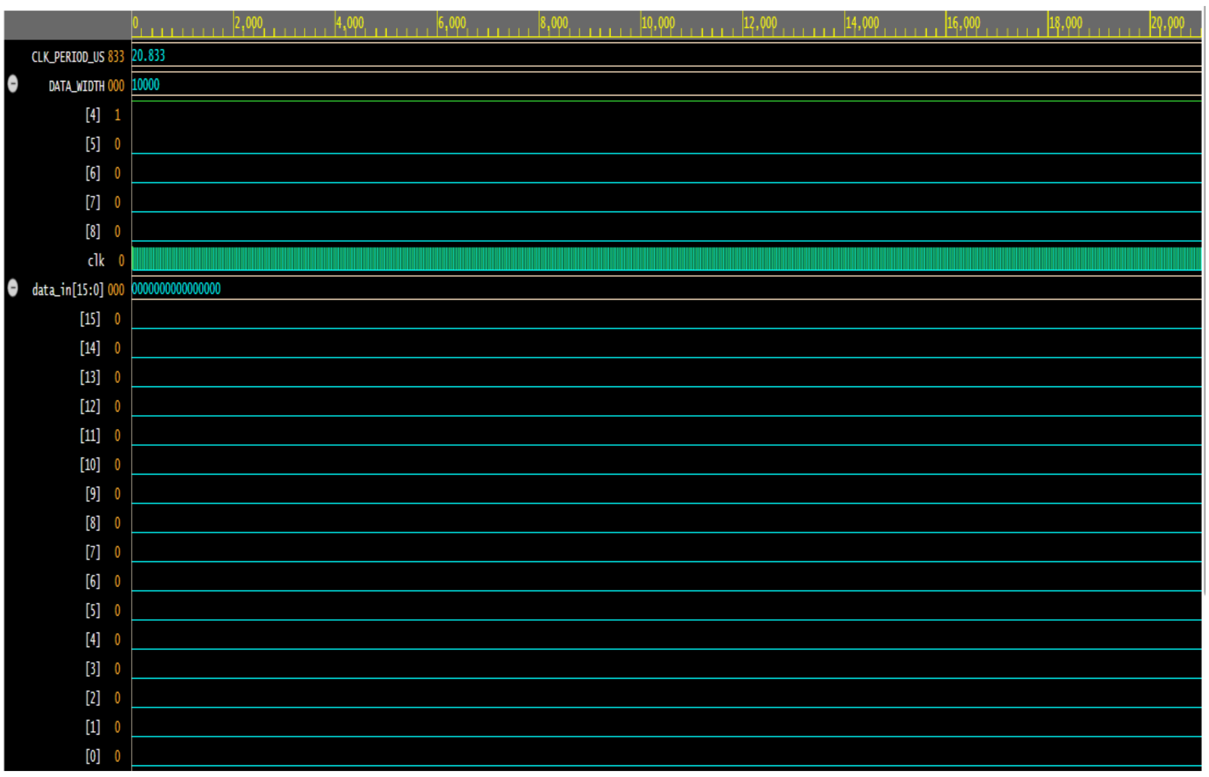
```
always @(posedge clk) begin
    if (reset) begin
        for (int i = 0; i <= 16; i++) begin
            x_regs[i] <= 0;
        end
    end else begin
        x_regs[0] <= data_in;
        for (int i = 1; i <= 16; i++) begin
            x_regs[i] <= x_regs[i-1];
        end
    end
end

// 2. Multiplier Stage (b_k * x[n-k]) - Pipelined
reg signed [PRODUCT_WIDTH-1:0] mul_products [0:16];
always @(posedge clk) begin
    mul_products[0]  <= x_regs[0]  * B0;
    mul_products[1]  <= x_regs[1]  * B1;
    mul_products[2]  <= x_regs[2]  * B2;
    mul_products[3]  <= x_regs[3]  * B3;
    mul_products[4]  <= x_regs[4]  * B4;
    mul_products[5]  <= x_regs[5]  * B5;
    mul_products[6]  <= x_regs[6]  * B6;
    mul_products[7]  <= x_regs[7]  * B7;
    mul_products[8]  <= x_regs[8]  * B8;
    mul_products[9]  <= x_regs[9]  * B9;
    mul_products[10] <= x_regs[10] * B10;
    mul_products[11] <= x_regs[11] * B11;
    mul_products[12] <= x_regs[12] * B12;
    mul_products[13] <= x_regs[13] * B13;
    mul_products[14] <= x_regs[14] * B14;
    mul_products[15] <= x_regs[15] * B15;
    mul_products[16] <= x_regs[16] * B16;
end
```

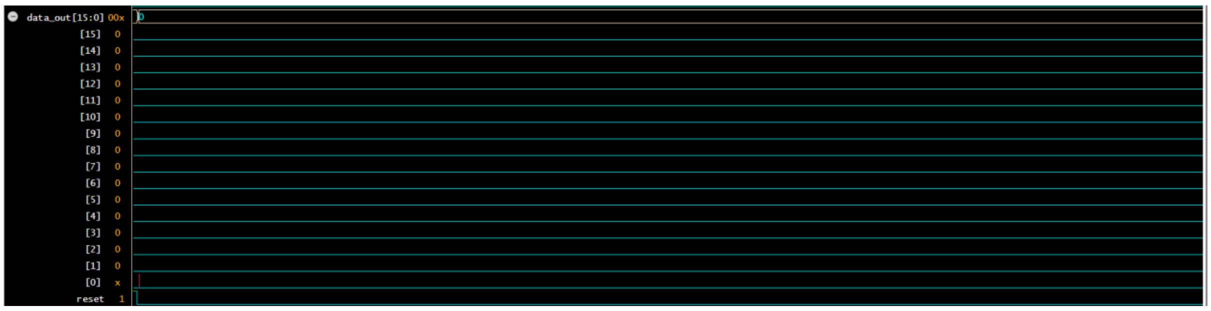
```
// 3. Pipelined Adder Tree

// Stage 1: 17 inputs -> 9 outputs

reg signed [PRODUCT_WIDTH:0] add_stage1 [0:8];

always @(posedge clk) begin

    add_stage1[0] <= mul_products[0]  + mul_products[1];

    add_stage1[1] <= mul_products[2]  + mul_products[3];

    add_stage1[2] <= mul_products[4]  + mul_products[5];

    add_stage1[3] <= mul_products[6]  + mul_products[7];

    add_stage1[4] <= mul_products[8]  + mul_products[9];

    add_stage1[5] <= mul_products[10] + mul_products[11];

    add_stage1[6] <= mul_products[12] + mul_products[13];

    add_stage1[7] <= mul_products[14] + mul_products[15];

    add_stage1[8] <= mul_products[16]; // Pass through

end


// Stage 2: 9 inputs -> 5 outputs

reg signed [PRODUCT_WIDTH+1:0] add_stage2 [0:4];

always @(posedge clk) begin

    add_stage2[0] <= add_stage1[0] + add_stage1[1];

    add_stage2[1] <= add_stage1[2] + add_stage1[3];

    add_stage2[2] <= add_stage1[4] + add_stage1[5];

    add_stage2[3] <= add_stage1[6] + add_stage1[7];

    add_stage2[4] <= add_stage1[8]; // Pass through

end


// Stage 3: 5 inputs -> 3 outputs

reg signed [PRODUCT_WIDTH+2:0] add_stage3 [0:2];

always @(posedge clk) begin

    add_stage3[0] <= add_stage2[0] + add_stage2[1];

    add_stage3[1] <= add_stage2[2] + add_stage2[3];

    add_stage3[2] <= add_stage2[4]; // Pass through

end


// Stage 4: 3 inputs -> 2 outputs

reg signed [PRODUCT_WIDTH+3:0] add_stage4 [0:1];
```

```
    always @(posedge clk) begin
        add_stage4[0] <= add_stage3[0] + add_stage3[1];
        add_stage4[1] <= add_stage3[2]; // Pass through
    end

    // Stage 5: 2 inputs -> 1 final sum
    reg signed [PRODUCT_WIDTH+4:0] final_sum;
    always @(posedge clk) begin
        final_sum <= add_stage4[0] + add_stage4[1];
    end

    // 4. Output Stage
    // Truncate the full-precision sum back to the output data width.
    always @(posedge clk) begin
        if (reset) begin
            data_out <= 0;
        end else begin
            // Right shift by 15 to remove fractional bits from Q15 coefficient format
            data_out <= final_sum >>> 15;
        end
    end

endmodule



//////////////////////////VERILOG TESTBENCH CODE////////////////////////////////

`timescale 1ns/1ps

module tb_fir_filter;

    // Parameters
    localparam DATA_WIDTH = 16;
    // Clock period for 48 kHz sampling frequency (1 / 48000 Hz = 20.833 us)
    localparam CLK_PERIOD_US = 20.833;
```

```
// Signals
reg                     clk;
reg                     reset;
reg  signed [DATA_WIDTH-1:0]   data_in;
wire signed [DATA_WIDTH-1:0]   data_out;

// Instantiate the DUT (Device Under Test)
fir_filter #(
    .DATA_WIDTH(DATA_WIDTH)
) dut (
    .clk(clk),
    .reset(reset),
    .data_in(data_in),
    .data_out(data_out)
);

// Clock generator
always #((CLK_PERIOD_US/2)/1000) clk = ~clk;

// Test stimulus
initial begin
    // Initialize signals and apply reset
    clk = 0;
    reset = 1;
    data_in = 0;
    #((CLK_PERIOD_US)/1000 * 5); // Wait for a few cycles
    reset = 0;

    // Generate a composite sine wave: 2kHz (passband) + 10kHz (stopband)
    // Amplitude is scaled to fit within 16-bit signed range
    for (integer i = 0; i < 1000; i = i + 1) begin
        // Current time in seconds for sine calculation
        real time_sec = i * (CLK_PERIOD_US * 1e-6);
```

```
            // 2kHz sine wave with amplitude ~10000
            real sin_2k = 10000 * $sin(2 * 3.14159 * 2000 * time_sec);

            // 10kHz sine wave with amplitude ~10000
            real sin_10k = 10000 * $sin(2 * 3.14159 * 10000 * time_sec);

            data_in = sin_2k + sin_10k;
            #((CLK_PERIOD_US)/1000);
        end

        $display("Simulation finished.");
        $finish;
    end

    // Optional: Dump waveform for viewing in a simulator like GTKWave or Vivado
    initial begin
        $dumpfile("fir_waveform.vcd");
        $dumpvars(0, tb_fir_filter);
    end

endmodule
```


0
2,000
4,000
6,000
8,000
10,000
12,000
14,000
16,000
18,000
20,000
CLK_PERIOD_US 833
20.833
DATA_WIDTH 000
10000
[4] 1
[5] 0
[6] 0
[7] 0
[8] 0
clk 0
data_in[15:0] 000
0000000000000000
[15] 0
[14] 0
[13] 0
[12] 0
[11] 0
[10] 0
[9] 0
[8] 0
[7] 0
[6] 0
[5] 0
[4] 0
[3] 0
[2] 0
[1] 0
[0] 0


data_out[15:0] 00x
0
[15] 0
[14] 0
[13] 0
[12] 0
[11] 0
[10] 0
[9] 0
[8] 0
[7] 0
[6] 0
[5] 0
[4] 0
[3] 0
[2] 0
[1] 0
[0] x
reset 1