///////////////////////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



