REPORT 2

**by: Hazem Yasser Mahmoud Mohamed
Mohamed Ahmed Mohamed
Mohamed Naem

NOTES

- the RTL is still under development and is still being edited no syntax errors only functionality issues
- the code comprises of txt files of the filters coeffs & the verilog code of each stage & a mekefile to run the tb and produce & visualise the output with vcd viewer

hex files

```
hazemysr@MacPro ~/Desktop/projects/run % ls hex
coeffs_fixed_q15_hex.txt        comp_R4_hex.txt
notch_b_q14_hex.txt
comp_R16_hex.txt                comp_R8_hex.txt
comp_R2_hex.txt                 notch_a_q14_hex.txt
```

polyphase files v1

```
/*
 * Signed Multiplier Module
 * Performs signed multiplication with configurable bit widths
 */
module signed_mult #(
    parameter WIDTH_A = 16,
    parameter WIDTH_B = 16,
    parameter WIDTH_OUT = 32
)(
    input  wire signed [WIDTH_A-1:0] a,
    input  wire signed [WIDTH_B-1:0] b,
    output wire signed [WIDTH_OUT-1:0] product
);

    assign product = a * b;

endmodule
/*
```

```
 * FIR MAC (Multiply-Accumulate) Unit
 * Used as building block for polyphase filter branches
 */
module fir_mac #(
    parameter DATA_WIDTH = 16,
    parameter COEFF_WIDTH = 16,
    parameter ACC_WIDTH = 40
)(
    input  wire clk,
    input  wire rst_n,
    input  wire enable,
    input  wire clear_acc,
    input  wire signed [DATA_WIDTH-1:0] data_in,
    input  wire signed [COEFF_WIDTH-1:0] coeff,
    output reg  signed [ACC_WIDTH-1:0] acc_out
);

    wire signed [DATA_WIDTH+COEFF_WIDTH-1:0] product;

    // Multiply
    signed_mult #(
        .WIDTH_A(DATA_WIDTH),
        .WIDTH_B(COEFF_WIDTH),
        .WIDTH_OUT(DATA_WIDTH+COEFF_WIDTH)
    ) mult_inst (
        .a(data_in),
        .b(coeff),
        .product(product)
    );

    // Accumulate
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            acc_out <= {ACC_WIDTH{1'b0}};
        end else if (clear_acc) begin
            acc_out <= {ACC_WIDTH{1'b0}};
        end else if (enable) begin
            acc_out <= acc_out + {{(ACC_WIDTH-DATA_WIDTH-COEFF_WIDTH)
{product[DATA_WIDTH+COEFF_WIDTH-1]}}, product};
        end
    end

endmodule
/*
 * Polyphase Filter Branch
 * Implements one branch of the polyphase filter structure
```

```verilog
 * Processes decimated input stream through FIR filter
 */
module polyphase_branch #(
    parameter DATA_WIDTH = 16,
    parameter COEFF_WIDTH = 16,
    parameter NUM_TAPS = 38,      // Coefficients per branch
    parameter BRANCH_ID = 0,      // Branch identifier (0 to 5)
    parameter ACC_WIDTH = 40
)(
    input  wire clk,
    input  wire rst_n,
    input  wire enable,
    input  wire signed [DATA_WIDTH-1:0] data_in,
    input  wire signed [COEFF_WIDTH-1:0] coeff,
    output wire signed [DATA_WIDTH-1:0] data_out,
    output wire data_valid
);

    // Delay line for FIR filter
    reg signed [DATA_WIDTH-1:0] delay_line [0:NUM_TAPS-1];

    // MAC unit
    reg signed [ACC_WIDTH-1:0] accumulator;
    reg [5:0] tap_counter;
    reg computing;
    reg valid_out;

    integer i;

    // Shift register for input samples
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            for (i = 0; i < NUM_TAPS; i = i + 1)
                delay_line[i] <= {DATA_WIDTH{1'b0}};
        end else if (enable && !computing) begin
            // Shift in new sample
            delay_line[0] <= data_in;
            for (i = 1; i < NUM_TAPS; i = i + 1)
                delay_line[i] <= delay_line[i-1];
        end
    end

    // MAC computation
    wire signed [DATA_WIDTH+COEFF_WIDTH-1:0] product;
    assign product = delay_line[tap_counter] * coeff;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            accumulator <= {ACC_WIDTH{1'b0}};
            tap_counter <= 6'd0;
            computing <= 1'b0;
            valid_out <= 1'b0;
        end else if (enable && !computing) begin
            // Start computation
            computing <= 1'b1;
            tap_counter <= 6'd0;
            accumulator <= {ACC_WIDTH{1'b0}};
            valid_out <= 1'b0;
        end else if (computing) begin
            if (tap_counter < NUM_TAPS) begin
                accumulator <= accumulator + {{(ACC_WIDTH-DATA_WIDTH-
COEFF_WIDTH){product[DATA_WIDTH+COEFF_WIDTH-1]}}, product};
                tap_counter <= tap_counter + 1'b1;
            end else begin
                computing <= 1'b0;
                valid_out <= 1'b1;
            end
        end else begin
            valid_out <= 1'b0;
        end
    end

    // Output with rounding and saturation
    // Shift by 15 bits (Q1.15 format) and round
    wire signed [ACC_WIDTH-1:0] rounded = accumulator + (1 << 14);
    wire signed [DATA_WIDTH-1:0] truncated = rounded[29:14]; // Extract bits
[29:14] for Q1.15

    assign data_out = truncated;
    assign data_valid = valid_out;

endmodule
/*
 * Polyphase Resampler - Top Module
 * Rational resampler L=2, M=3 (9 MHz -> 6 MHz)
 * Implements parallel polyphase structure with 6 branches (M=3 main phases
× L=2 sub-phases)
 */
module polyphase_resampler #(
    parameter DATA_WIDTH = 16,
    parameter COEFF_WIDTH = 16,
    parameter NUM_TAPS_PER_BRANCH = 38,
    parameter L = 2,  // Interpolation factor
    parameter M = 3   // Decimation factor
)(
    input  wire clk,
    input  wire rst_n,
    input  wire signed [DATA_WIDTH-1:0] data_in,
    input  wire data_in_valid,
    output reg  signed [DATA_WIDTH-1:0] data_out,
    output reg  data_out_valid
);

    // State machine
    localparam IDLE = 0, PROCESS_M0 = 1, PROCESS_M1 = 2, PROCESS_M2 = 3,
OUTPUT = 4;
    reg [2:0] state, next_state;

    // Sample counter for M-way decimation
    reg [1:0] sample_count;

    // Coefficient ROM address control
    reg [7:0] coeff_addr;
    wire signed [COEFF_WIDTH-1:0] coeff_data;
    reg coeff_active;  // NEW: whether coefficients are being read

        // Coefficient address control
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            coeff_addr <= 8'd0;
            coeff_active <= 1'b0;
        end else begin
            // Start reading coeffs when any branch is enabled
            if (|branch_enable) begin
                coeff_active <= 1'b1;
            end else if (state == OUTPUT) begin
                coeff_active <= 1'b0;
                coeff_addr <= 8'd0; // Reset for next phase
            end

            // Increment coeff address while active
            if (coeff_active) begin
                if (coeff_addr < NUM_TAPS_PER_BRANCH*6 - 1)
                    coeff_addr <= coeff_addr + 1'b1;
                else
                    coeff_addr <= 8'd0; // Wrap around
            end
        end
    end
    coefficient_rom #(
        .COEFF_WIDTH(COEFF_WIDTH),
        .NUM_COEFFS(228)
    ) coeff_rom_inst (
        .clk(clk),
        .addr(coeff_addr),
        .coeff(coeff_data)
    );

    // 6 polyphase branches (3 main phases × 2 sub-phases)
    wire signed [DATA_WIDTH-1:0] branch_out [0:5];
    wire [5:0] branch_valid;
    reg [5:0] branch_enable;
    reg signed [DATA_WIDTH-1:0] branch_input;

    // Generate 6 branches
    genvar g;
    generate
        for (g = 0; g < 6; g = g + 1) begin : polyphase_branches
            polyphase_branch #(
                .DATA_WIDTH(DATA_WIDTH),
                .COEFF_WIDTH(COEFF_WIDTH),
                .NUM_TAPS(NUM_TAPS_PER_BRANCH),
                .BRANCH_ID(g)
            ) branch_inst (
                .clk(clk),
                .rst_n(rst_n),
                .enable(branch_enable[g]),
                .data_in(branch_input),
                .coeff(coeff_data),
                .data_out(branch_out[g]),
                .data_valid(branch_valid[g])
            );
        end
    endgenerate

    // Output accumulation for L=2 sub-phases
    reg signed [DATA_WIDTH:0] sum_phase0_l0, sum_phase0_l1;
    reg signed [DATA_WIDTH:0] sum_phase1_l0, sum_phase1_l1;
    reg signed [DATA_WIDTH:0] sum_phase2_l0, sum_phase2_l1;

    // Sample counter and state machine
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            sample_count <= 2'd0;
            state <= IDLE;
        end else begin
            state <= next_state;
            if (data_in_valid && state == IDLE) begin
                sample_count <= sample_count + 1'b1;
                if (sample_count == M-1)
                    sample_count <= 2'd0;
            end
        end
    end

    // Next state logic
    always @(*) begin
        next_state = state;
        branch_enable = 6'b000000;
        branch_input = data_in;

        case (state)
            IDLE: begin
                if (data_in_valid) begin
                    case (sample_count)
                        2'd0: next_state = PROCESS_M0;
                        2'd1: next_state = PROCESS_M1;
                        2'd2: next_state = PROCESS_M2;
                    endcase
                end
            end

            PROCESS_M0: begin
                branch_enable = 6'b000011; // Branches 0 and 1 (phase 0,
sub-phase 0 and 1)
                if (branch_valid[0] && branch_valid[1])
                    next_state = OUTPUT;
            end

            PROCESS_M1: begin
                branch_enable = 6'b001100; // Branches 2 and 3 (phase 1,
sub-phase 0 and 1)
                if (branch_valid[2] && branch_valid[3])
                    next_state = OUTPUT;
            end

            PROCESS_M2: begin
                branch_enable = 6'b110000; // Branches 4 and 5 (phase 2,
sub-phase 0 and 1)
                if (branch_valid[4] && branch_valid[5])
```

```
                    next_state = OUTPUT;
                end

                OUTPUT: begin
                    next_state = IDLE;
                end
            endcase
        end

        // Output generation with L=2 interleaving
        reg [1:0] output_phase;

        always @(posedge clk or negedge rst_n) begin
            if (!rst_n) begin
                data_out <= {DATA_WIDTH{1'b0}};
                data_out_valid <= 1'b0;
                output_phase <= 2'd0;
            end else if (state == OUTPUT) begin
                // Output L=2 samples for each input sample
                case (sample_count)
                    2'd0: begin
                        if (output_phase == 0) begin
                            data_out <= branch_out[0];
                            data_out_valid <= 1'b1;
                            output_phase <= 1;
                        end else begin
                            data_out <= branch_out[1];
                            data_out_valid <= 1'b1;
                            output_phase <= 0;
                        end
                    end
                    2'd1: begin
                        if (output_phase == 0) begin
                            data_out <= branch_out[2];
                            data_out_valid <= 1'b1;
                            output_phase <= 1;
                        end else begin
                            data_out <= branch_out[3];
                            data_out_valid <= 1'b1;
                            output_phase <= 0;
                        end
                    end
                    2'd2: begin
                        if (output_phase == 0) begin
                            data_out <= branch_out[4];
                            data_out_valid <= 1'b1;
                            output_phase <= 1;
                        end else begin
                            data_out <= branch_out[5];
                            data_out_valid <= 1'b1;
                            output_phase <= 0;
                        end
                    end
                endcase
            end else begin
                data_out_valid <= 1'b0;
            end
        end

    endmodule
```

notch filter

```
/*
 * Biquad IIR Filter - Direct Form II Transposed
 * Implements 2.4 MHz notch filter
 * Uses Q1.14 fixed-point coefficients
 */
module biquad_df2t #(
    parameter DATA_WIDTH = 16,
    parameter COEFF_WIDTH = 16,
    parameter STATE_WIDTH = 32
)(
    input  wire clk,
    input  wire rst_n,
    input  wire signed [DATA_WIDTH-1:0] data_in,
    input  wire data_in_valid,
    output reg  signed [DATA_WIDTH-1:0] data_out,
    output reg  data_out_valid
);

    // Biquad coefficients (Q1.14 format)
    // Load from external ROM or hardcode
    reg signed [COEFF_WIDTH-1:0] b0, b1, b2;
    reg signed [COEFF_WIDTH-1:0] a1, a2;

    // Coefficient arrays for loading
    reg signed [COEFF_WIDTH-1:0] b_coeffs [0:2];
    reg signed [COEFF_WIDTH-1:0] a_coeffs [0:1];

    // State variables (delay elements)
    reg signed [STATE_WIDTH-1:0] s1, s2;

    // Intermediate products
    wire signed [DATA_WIDTH+COEFF_WIDTH-1:0] b0_x, b1_x, b2_x;
    wire signed [STATE_WIDTH+COEFF_WIDTH-1:0] a1_y, a2_y;

    // Load coefficients from file
    initial begin
        $readmemh("notch_b_q14_hex.txt", b_coeffs);
        $readmemh("notch_a_q14_hex.txt", a_coeffs);
        b0 = b_coeffs[0];
        b1 = b_coeffs[1];
        b2 = b_coeffs[2];
        a1 = a_coeffs[0];
        a2 = a_coeffs[1];
    end

    // Multiply operations
    assign b0_x = b0 * data_in;
    assign b1_x = b1 * data_in;
    assign b2_x = b2 * data_in;

    // Output and state update
    reg signed [STATE_WIDTH-1:0] y_n;
    wire signed [STATE_WIDTH+COEFF_WIDTH-1:0] a1_y_prod, a2_y_prod;

    assign a1_y_prod = a1 * y_n;
    assign a2_y_prod = a2 * y_n;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            s1 <= {STATE_WIDTH{1'b0}};
            s2 <= {STATE_WIDTH{1'b0}};
            y_n <= {STATE_WIDTH{1'b0}};
            data_out <= {DATA_WIDTH{1'b0}};
            data_out_valid <= 1'b0;
        end else if (data_in_valid) begin
            // Direct Form II Transposed
            // y[n] = b0*x[n] + s1
            y_n = {{(STATE_WIDTH-DATA_WIDTH-COEFF_WIDTH)
{b0_x[DATA_WIDTH+COEFF_WIDTH-1]}}, b0_x} + s1;

            // s1 = b1*x[n] - a1*y[n] + s2
            s1 <= {{(STATE_WIDTH-DATA_WIDTH-COEFF_WIDTH)
{b1_x[DATA_WIDTH+COEFF_WIDTH-1]}}, b1_x}
                  - a1_y_prod[STATE_WIDTH+COEFF_WIDTH-1:COEFF_WIDTH]
                  + s2;

            // s2 = b2*x[n] - a2*y[n]
            s2 <= {{(STATE_WIDTH-DATA_WIDTH-COEFF_WIDTH)
{b2_x[DATA_WIDTH+COEFF_WIDTH-1]}}, b2_x}
                  - a2_y_prod[STATE_WIDTH+COEFF_WIDTH-1:COEFF_WIDTH];

            // Output with scaling (Q1.14 -> Q1.15)
            // Shift right by 14 bits and round
            data_out <= (y_n + (1 << 13)) >>> 14;
            data_out_valid <= 1'b1;
        end else begin
            data_out_valid <= 1'b0;
        end
    end

endmodule
```

cic

```
/*
 * CIC Integrator Stage
 * Single integrator for CIC filter
 */
module cic_integrator #(
    parameter WIDTH = 32
)(
    input  wire clk,
    input  wire rst_n,
    input  wire signed [WIDTH-1:0] data_in,
    input  wire data_in_valid,
    output reg  signed [WIDTH-1:0] data_out,
    output reg  data_out_valid
);

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            data_out <= {WIDTH{1'b0}};
            data_out_valid <= 1'b0;
        end else if (data_in_valid) begin
            data_out <= data_out + data_in;
            data_out_valid <= 1'b1;
        end else begin
            data_out_valid <= 1'b0;
        end
    end

endmodule
/*
 * CIC Comb Stage
 * Single comb (differentiator) for CIC filter
 */
module cic_comb #(
    parameter WIDTH = 32,
    parameter M = 1  // Differential delay
)(
    input  wire clk,
    input  wire rst_n,
    input  wire signed [WIDTH-1:0] data_in,
    input  wire data_in_valid,
    output reg  signed [WIDTH-1:0] data_out,
    output reg  data_out_valid
);

    // Delay line
    reg signed [WIDTH-1:0] delay_line [0:M-1];
    integer i;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            for (i = 0; i < M; i = i + 1)
                delay_line[i] <= {WIDTH{1'b0}};
            data_out <= {WIDTH{1'b0}};
            data_out_valid <= 1'b0;
        end else if (data_in_valid) begin
            // Comb: y[n] = x[n] - x[n-M]
            data_out <= data_in - delay_line[0];
            data_out_valid <= 1'b1;

            // Shift delay line
            for (i = 0; i < M-1; i = i + 1)
                delay_line[i] <= delay_line[i+1];
            delay_line[M-1] <= data_in;
        end else begin
            data_out_valid <= 1'b0;
        end
    end

endmodule
/*
```

```
 * CIC Decimator with Compensation FIR
 * Configurable decimation factor R = 1, 2, 4, 8, 16
 * N = 5 stages, M = 1 differential delay
 */
module cic_decimator #(
    parameter DATA_WIDTH = 16,
    parameter N = 5,              // Number of stages
    parameter M = 1,              // Differential delay
    parameter MAX_R = 16,         // Maximum decimation rate
    parameter CIC_WIDTH = 64      // Internal bit width
)(
    input  wire clk,
    input  wire rst_n,
    input  wire [4:0] decimation_rate,  // R = 1, 2, 4, 8, 16
    input  wire signed [DATA_WIDTH-1:0] data_in,
    input  wire data_in_valid,
    output reg  signed [DATA_WIDTH-1:0] data_out,
    output reg  data_out_valid
);

    // Calculate bit growth: B_growth = N * ceil(log2(R*M))
    // For R=16, M=1, N=5: B_growth = 5 * 4 = 20 bits
    localparam B_GROWTH = 20;

    // Bypass logic for R=1
    wire bypass_mode;
    assign bypass_mode = (decimation_rate == 5'd1);

    // ===== Integrator Section =====
    wire signed [CIC_WIDTH-1:0] integ_out [0:N];
    wire [N:0] integ_valid;

    // Convert input to wider format
    assign integ_out[0] = {{(CIC_WIDTH-DATA_WIDTH){data_in[DATA_WIDTH-1]}},
data_in};
    assign integ_valid[0] = data_in_valid;

    genvar i;
    generate
        for (i = 0; i < N; i = i + 1) begin : integrator_stages
            cic_integrator #(.WIDTH(CIC_WIDTH)) integ (
                .clk(clk),
                .rst_n(rst_n),
                .data_in(integ_out[i]),
                .data_in_valid(integ_valid[i]),
                .data_out(integ_out[i+1]),
                .data_out_valid(integ_valid[i+1])
            );
        end
    endgenerate

    // ===== Decimation =====
    reg [4:0] sample_counter;
    reg signed [CIC_WIDTH-1:0] decimated_data;
    reg decimated_valid;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            sample_counter <= 5'd0;
            decimated_data <= {CIC_WIDTH{1'b0}};
            decimated_valid <= 1'b0;
        end else if (bypass_mode) begin
            decimated_data <= integ_out[N];
            decimated_valid <= integ_valid[N];
            sample_counter <= 5'd0;
        end else if (integ_valid[N]) begin
            if (sample_counter == decimation_rate - 1) begin
                decimated_data <= integ_out[N];
                decimated_valid <= 1'b1;
                sample_counter <= 5'd0;
            end else begin
                sample_counter <= sample_counter + 1'b1;
                decimated_valid <= 1'b0;
            end
        end else begin
            decimated_valid <= 1'b0;
        end
    end

    // ===== Comb Section =====
    wire signed [CIC_WIDTH-1:0] comb_out [0:N];
    wire [N:0] comb_valid;

    assign comb_out[0] = decimated_data;
    assign comb_valid[0] = decimated_valid;

    generate
        for (i = 0; i < N; i = i + 1) begin : comb_stages
            cic_comb #(.WIDTH(CIC_WIDTH), .M(M)) comb (
                .clk(clk),
                .rst_n(rst_n),
                .data_in(comb_out[i]),
                .data_in_valid(comb_valid[i]),
                .data_out(comb_out[i+1]),
                .data_out_valid(comb_valid[i+1])
            );
        end
    endgenerate

    // ===== Compensation FIR Filter =====
    reg signed [DATA_WIDTH-1:0] comp_coeff [0:255];
    reg [7:0] comp_length;

    // Load compensation coefficients based on R
    initial begin
        comp_length = 8'd0;
    end

    always @(decimation_rate) begin
        case (decimation_rate)
            5'd2: begin
                $readmemh("comp_R2_hex.txt", comp_coeff);
                comp_length = 8'd32;
            end
            5'd4: begin
                $readmemh("comp_R4_hex.txt", comp_coeff);
                comp_length = 8'd32;
            end
            5'd8: begin
                $readmemh("comp_R8_hex.txt", comp_coeff);
                comp_length = 8'd32;
            end
            5'd16: begin
                $readmemh("comp_R16_hex.txt", comp_coeff);
                comp_length = 8'd32;
            end
            default: comp_length = 8'd0;
        endcase
    end

    // FIR filter delay line and computation
    reg signed [DATA_WIDTH-1:0] fir_delay [0:255];
    reg signed [CIC_WIDTH-1:0] fir_acc;
    reg [7:0] fir_tap_count;
    reg fir_computing;
    integer j;

    // Scale and truncate CIC output
    wire signed [DATA_WIDTH-1:0] cic_scaled;
    wire signed [CIC_WIDTH-1:0] cic_shifted = comb_out[N] >>> B_GROWTH;
    assign cic_scaled = cic_shifted[DATA_WIDTH-1:0];

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            for (j = 0; j < 256; j = j + 1)
                fir_delay[j] <= {DATA_WIDTH{1'b0}};
            fir_acc <= {CIC_WIDTH{1'b0}};
            fir_tap_count <= 8'd0;
            fir_computing <= 1'b0;
            data_out <= {DATA_WIDTH{1'b0}};
            data_out_valid <= 1'b0;
        end else if (comb_valid[N] && !fir_computing) begin
            // Shift in new sample
            for (j = 1; j < 256; j = j + 1)
                fir_delay[j] <= fir_delay[j-1];
            fir_delay[0] <= cic_scaled;

            if (bypass_mode || comp_length == 0) begin
                // Bypass compensation for R=1
                data_out <= cic_scaled;
                data_out_valid <= 1'b1;
            end else begin
                // Start FIR computation
                fir_computing <= 1'b1;
                fir_tap_count <= 8'd0;
                fir_acc <= {CIC_WIDTH{1'b0}};
            end
        end else if (fir_computing) begin
            if (fir_tap_count < comp_length) begin
                fir_acc <= fir_acc + (fir_delay[fir_tap_count] *
 comp_coeff[fir_tap_count]);
                fir_tap_count <= fir_tap_count + 1'b1;
            end else begin
                // Output FIR result
                // Shift right by 15 bits and truncate to output width
                data_out <= fir_acc[15 +: DATA_WIDTH];
                data_out_valid <= 1'b1;
                fir_computing <= 1'b0;
            end
        end else begin
            data_out_valid <= 1'b0;
        end
    end
```

```
endmodule
```

top modules and test bench

```
/*
 * DFE Top Module
 * Digital Front-End for RF receiver
 *
 * Pipeline:
 *   1. Polyphase Resampler: 9 MHz -> 6 MHz (L=2, M=3)
 *   2. Biquad Notch Filter: 2.4 MHz notch
 *   3. CIC Decimator + Compensation: 6 MHz -> (6/R) MHz
 *
 * Author: Group 7
 * Date: 2025
 */
module dfe_top #(
    parameter DATA_WIDTH = 16
)(
    input  wire clk,                        // System clock (must be >= 9
MHz)
    input  wire rst_n,                      // Active-low reset
    input  wire [4:0] decimation_rate,      // CIC decimation: 1, 2, 4, 8,
16
    input  wire signed [DATA_WIDTH-1:0] adc_data,
    input  wire adc_valid,
    output wire signed [DATA_WIDTH-1:0] dfe_out,
    output wire dfe_out_valid
);

    // ========================================
    // Stage 1: Polyphase Resampler (9 MHz -> 6 MHz)
    // ========================================
    wire signed [DATA_WIDTH-1:0] stage1_out;
    wire stage1_valid;

    // polyphase_resampler #(
    //     .DATA_WIDTH(DATA_WIDTH),
    //     .COEFF_WIDTH(16),
    //     .NUM_TAPS_PER_BRANCH(38),
    //     .L(2),
    //     .M(3)`
    // ) polyphase_inst (
    //     .clk(clk),
    //     .rst_n(rst_n),
    //     .data_in(adc_data),
    //     .data_in_valid(adc_valid),
    //     .data_out(stage1_out),
    //     .data_out_valid(stage1_valid)
    // );


    fir_resamp_top #(
        .word_size_in   (DATA_WIDTH),
        .word_size_out  (DATA_WIDTH),
        .coeff_word_size(16),       // your coeff file width
        .guard_bits     (4),        // keep as in the module or change if
desired
        .L              (2),
        .D              (3),
        .taps_per_branch(115)       // MUST match the number of fir_rom
modules provided
    ) fir_resamp_inst (
        .CLK  (clk),
        .RESET(~rst_n),      // fir_resamp_top uses active-high RESET;
dfe_top has active-low rst_n
        .DIN  (adc_data),
        .ND   (adc_valid),  // one-clock pulse per new input sample
(adc_valid must be 1 cycle)
        .DOUT (stage1_out),
        .RDY  (stage1_valid),
        .OV   ()
    );

    // ========================================
    // Stage 2: Biquad Notch Filter (2.4 MHz)
    // ========================================
    wire signed [DATA_WIDTH-1:0] stage2_out;
    wire stage2_valid;

    biquad_df2t #(
        .DATA_WIDTH(DATA_WIDTH),
        .COEFF_WIDTH(16),
        .STATE_WIDTH(32)
    ) notch_inst (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(stage1_out),
        .data_in_valid(stage1_valid),
        .data_out(stage2_out),
        .data_out_valid(stage2_valid)
    );

    // ========================================
    // Stage 3: CIC Decimator + Compensation
    // ========================================
    cic_decimator #(
        .DATA_WIDTH(DATA_WIDTH),
        .N(5),
        .M(1),
        .MAX_R(16),
        .CIC_WIDTH(64)
    ) cic_inst (
        .clk(clk),
        .rst_n(rst_n),
        .decimation_rate(decimation_rate),
        .data_in(stage2_out),
        .data_in_valid(stage2_valid),
        .data_out(dfe_out),
        .data_out_valid(dfe_out_valid)
    );

endmodule
/*
 * DFE Testbench
 * Tests the complete Digital Front-End with various input signals
 */
`timescale 1ns/1ps

module dfe_tb;

    // Parameters
    parameter DATA_WIDTH = 16;
    parameter CLK_PERIOD = 10;  // 100 MHz clock (much faster than signal
rates)

    // Signals
    reg clk;
    reg rst_n;
    reg [4:0] decimation_rate;
    reg signed [DATA_WIDTH-1:0] adc_data;
    reg adc_valid;
    wire signed [DATA_WIDTH-1:0] dfe_out;
    wire dfe_out_valid;

    // File handles for output
```

```verilog
    integer out_file;

    // Clock generation
    initial begin
        clk = 0;
        forever #(CLK_PERIOD/2) clk = ~clk;
    end

    // DUT instantiation
    dfe_top #(
        .DATA_WIDTH(DATA_WIDTH)
    ) dut (
        .clk(clk),
        .rst_n(rst_n),
        .decimation_rate(decimation_rate),
        .adc_data(adc_data),
        .adc_valid(adc_valid),
        .dfe_out(dfe_out),
        .dfe_out_valid(dfe_out_valid)
    );

    // Test stimulus
    initial begin
        // Initialize signals
        rst_n = 0;
        adc_data = 0;
        adc_valid = 0;
        decimation_rate = 5'd1;  // Start with R=1

        // Open output file
        out_file = $fopen("dfe_output.txt", "w");

        // Reset
        #(CLK_PERIOD*10);
        rst_n = 1;
        #(CLK_PERIOD*10);

        $display("========================================");
        $display("DFE Testbench Started");
        $display("========================================");

        // Test 1: Single tone at 1 MHz (input at 9 MHz sample rate)
        $display("\nTest 1: 1 MHz sine wave, R=1");
        decimation_rate = 5'd1;
        generate_sine_wave(1000000, 9000000, 1000);
        #(CLK_PERIOD*1000);
        // Test 2: Tone at 2.4 MHz (notch frequency)
        $display("\nTest 2: 2.4 MHz sine wave (notch test), R=1");
        decimation_rate = 5'd1;
        generate_sine_wave(2400000, 9000000, 1000);
        #(CLK_PERIOD*1000);

        // Test 3: Multi-tone with decimation R=2
        $display("\nTest 3: Multi-tone, R=2");
        decimation_rate = 5'd2;
        generate_multitone(9000000, 1000);
        #(CLK_PERIOD*2000);

        // Test 4: R=8 decimation
        $display("\nTest 4: Low frequency tone, R=8");
        decimation_rate = 5'd8;
        generate_sine_wave(50000, 9000000, 2000);
        #(CLK_PERIOD*5000);

        // Close file and finish
        $fclose(out_file);
        $display("\n========================================");
        $display("DFE Testbench Completed");
        $display("Output written to dfe_output.txt");
        $display("========================================");
        $finish;
    end

    // Monitor output
    always @(posedge clk) begin
        if (dfe_out_valid) begin
            $fwrite(out_file, "%d\n", dfe_out);
        end
    end

    // Task: Generate sine wave
    task generate_sine_wave;
        input integer freq;        // Frequency in Hz
        input integer fs;          // Sample rate in Hz
        input integer num_samples; // Number of samples
        integer i;
        real phase;
        real sample;
        begin
            adc_valid = 0;
            for (i = 0; i < num_samples; i = i + 1) begin
```

```
                phase  = 2.0 * 3.14159265359 * freq * i / fs;
                sample = 0.8 * $sin(phase);  // 0.8 amplitude

                adc_data  = $rtoi(sample * 32767);
                adc_valid = 1;   // Assert valid for one clock

                @(posedge clk);  // One cycle high
                adc_valid = 0;   // Deassert valid

                // Wait remaining cycles for 9 MHz sample rate (111.11 ns
period)
                // At 100 MHz clock, that's ~11 total cycles per sample, so
10 more after the valid pulse
                repeat(10) @(posedge clk);
            end
        end
    endtask

    // Task: Generate multi-tone signal
    task generate_multitone;
        input integer fs;          // Sample rate in Hz
        input integer num_samples; // Number of samples
        integer i;
        real phase1, phase2, phase3;
        real sample;
        begin
            for (i = 0; i < num_samples; i = i + 1) begin
                phase1 = 2.0 * 3.14159265359 * 500000 * i / fs;   // 0.5 MHz
                phase2 = 2.0 * 3.14159265359 * 1000000 * i / fs;  // 1.0 MHz
                phase3 = 2.0 * 3.14159265359 * 2400000 * i / fs;  // 2.4 MHz
(notch)
                sample = (0.3 * $sin(phase1) + 0.3 * $sin(phase2) + 0.3 *
$sin(phase3));
                adc_data = $rtoi(sample * 32767);
                adc_valid = 1;
                repeat(1) @(posedge clk);
                adc_valid = 0;

                repeat(10) @(posedge clk);
            end

        end
    endtask

    // Waveform dumping for simulation
    initial begin
```

```
        $dumpfile("dfe_tb.vcd");
        $dumpvars(0, dfe_tb);
    end

endmodule
```

makefile

- for running on macos using iverilog vvp and surfer to observe vcd files

```
# Tools
IVERILOG = iverilog
VVP = vvp
GTKWAVE = surfer

# Output and top-level module
OUT = dfe_sim
TOP = dfe_tb  # testbench top module

# Source files
SRC = \
  dfe_tb.v \
  dfe_top.v \
  polyphase_resampler.v \
  fir_mac.v \
  coefficient_rom.v \
  polyphase_branch.v \
  biquad_df2t.v \
  cic_decimator.v \
  cic_integrator.v \
  cic_comb.v \
  signed_mult.v

# Folder containing coefficient text files
TEXT_DIR = text

# Default target
all: wave

# Compile the design
compile:
    @echo "========================================"
    @echo " Compiling with Icarus Verilog..."
    @echo "========================================"
    $(IVERILOG) -o $(OUT) -s $(TOP) $(SRC)
```

```
# Copy coefficient files before running simulation
prepare_texts:
	@echo "========================================"
	@echo " Copying HEX text files from $(TEXT_DIR)..."
	@echo "========================================"
	@cp $(TEXT_DIR)/*_hex.txt . 2>/dev/null || true

# Run simulation
run: prepare_texts compile
	@echo "========================================"
	@echo " Running simulation..."
	@echo "========================================"
	$(VVP) $(OUT)

# Run simulation and open waveform in GTKWave
wave: run
	@echo "========================================"
	@echo " Opening waveform in GTKWave..."
	@echo "========================================"
	$(GTKWAVE) dfe_tb.vcd &

# Clean generated files
clean:
	@rm -f $(OUT) *.vcd *.vvp *_hex.txt
	@echo "Cleaned simulation files and copied HEX text files."
```