

RTL & TESTBENCH

by: Hazem Yasser Mahmoud Mohamed

Mohamed Ahmed Mohamed

mohamed anwar

GROUP 7

Rational Resampler

faced a lot of issues to achieve the novel architecture proposed in the golden model so we failed back to a more traditional approach

**a polyphase upsampler with 226 filter coeffs that satisfy the specs given followed by a down sampler with only 3 coeffs full pass as the full filtering happened in the UPSAMPLER

note : proposed filter in golden model has frequency doubling effect inherent to its structure not in this filter

so filter coeffs need to be adjusted for twice the cutoff just by changing the mem file

note : not all files are mentioned in this report but there are mem files for easy exchange of filters and vcd files for visualizing the result along with vvp files and txt log files along with python script to visualize and plot in python

polyphase resampler can further be optimized for area and resources by sharing DSP blocks instead of making it fully parallel it is a tradeoff between delay and utilization

the Notch takes most data delay to solve timing issues it was pipelined it went from -5~6 ns to less than 1 ns setup violation

because it is DFII iir filter it has FB so only pipelined to two sections so no further pipelining possible but still the DFE can be ran at 50 MHz so it is not a big deal

to fix input output wire delay we register I/O ports and modify DFE_top

RTL

polyphase_filter

```
`timescale 1ns / 1ps
```

```

module polyphase_filter #(
    parameter COEFF_FILE      = "decim_coeffs.mem", // File path for
coefficients
    parameter int DATA_WIDTH     = 16,
    parameter int COEFF_WIDTH    = 16,
    parameter int PHASES        = 8,   // Previously CONVERSION_FACTOR
    parameter int TAPS_PER_PHASE = 16,
    parameter bit IS_DECIMATION = 0    // 0 for Interpolation, 1 for
Decimation
) (
    input logic                      clk,
    input logic                      rst_n, // Added Reset
    input logic                      valid_i,
    input logic [DATA_WIDTH-1:0]       data_i,

    output logic                     valid_o,
    output logic [DATA_WIDTH-1:0]       data_o
);

// Calculate gain shift based on phases (ceil(log2(phases)))
localparam int GAIN_BITS = $clog2(PHASES);
localparam int TOTAL_TAPS = PHASES * TAPS_PER_PHASE;

// -----
// Coefficient Memory
// -----
logic signed [COEFF_WIDTH-1:0] coeff_rom [0:TOTAL_TAPS-1];

initial begin
    $readmemh(COEFF_FILE, coeff_rom);
end

// -----
// Internal Signals
// -----
// Phase Counters
int phase_counter;

// Registers
logic signed [DATA_WIDTH-1:0]      a_reg [0:TAPS_PER_PHASE-1]; // Input
Pipeline
logic signed [COEFF_WIDTH-1:0]      b_reg [0:TAPS_PER_PHASE-1]; // Coeff
Latch

// Pipeline Registers (MAC chain)

```

```

// Size matches VHDL: CONVERSION_FACTOR * TAPS_PER_PHASE
logic signed [2*DATA_WIDTH-1:0]      p_reg [0:TOTAL_TAPS-1];

// Accumulator for Decimation
logic signed [2*DATA_WIDTH-1:0]      product_sum;

// FSM State for Interpolation
typedef enum logic [1:0] {IDLE, GAP, PULSE} state_t;
state_t state;

logic active_cycle;

// -----
// Main Process
// -----
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        valid_o      <= 1'b0;
        data_o       <= '0;
        phase_counter <= 0;
        state         <= IDLE;
        product_sum   <= '0;
        active_cycle  <= 1'b0;

        // Reset Arrays
        for (int i = 0; i < TAPS_PER_PHASE; i++) a_reg[i] <= '0;
        for (int i = 0; i < TAPS_PER_PHASE; i++) b_reg[i] <= '0;
        for (int i = 0; i < TOTAL_TAPS; i++)      p_reg[i] <= '0;

    end else begin

        // Default Assignments
        valid_o <= 1'b0;
        active_cycle <= 1'b0;

        // =====
        // MODE 1: INTERPOLATION (IS_DECIMATION = 0)
        // =====
        if (!IS_DECIMATION) begin

            // --- FSM Control ---
            case (state)
                IDLE: begin
                    phase_counter <= 0;
                    if (valid_i) begin
                        active_cycle   <= 1'b1; // Process Phase 0

```

```

immediately
    phase_counter <= 1;      // Next is Phase 1
    state          <= GAP;
end

GAP: begin
    active_cycle <= 1'b0; // Wait state
    state         <= PULSE;
end

PULSE: begin
    active_cycle <= 1'b1;
    if (phase_counter == PHASES - 1) begin
        state          <= IDLE;
        phase_counter <= 0;
    end else begin
        phase_counter <= phase_counter + 1;
        state          <= GAP;
    end
end
endcase

// --- Processing Logic ---
if (active_cycle) begin
    valid_o <= 1'b1;

    // Filter Structure
    for (int m = 0; m < PHASES; m++) begin
        for (int t = 0; t < TAPS_PER_PHASE; t++) begin

            // 1. Input Latching (Only on Phase 0)
            // Note: Logic logic uses current_phase derived
from FSM previous state,
            // simpler to use the 'active_cycle' trigger
logic implied by VHDL.
            // If we are starting IDLE->GAP, phase is
effectively 0 for math.
            // If we are in PULSE, use current
phase_counter.

            // We use temporary variable for current
processing phase for clarity
            int current_p;
            if (state == IDLE) current_p = 0;
            else current_p = phase_counter; // In PULSE

```

```

state

    if (current_p == 0) begin
        a_reg[t] <= $signed(data_i);
    end

    // 2. Load Coefficient
    // Index = t * PHASES + current_p
    b_reg[t] <= coeff_rom[t*PHASES + current_p];

    // 3. MAC Operation
    // Index Mapping: t*PHASES + (PHASES-1) is the
    "Top" of the column for tap 't'
    if (t == TAPS_PER_PHASE - 1) begin
        p_reg[t*PHASES + (PHASES-1)] <= a_reg[t] *
    b_reg[t];
    end else begin
        p_reg[t*PHASES + (PHASES-1)] <= (a_reg[t] *
    b_reg[t]) + p_reg[(t+1)*PHASES];
    end

    // 4. Shift Pipeline
    if (m < PHASES - 1) begin
        p_reg[t*PHASES + m] <= p_reg[t*PHASES + m +
    1];
    end
end
end

// Output Scaling
data_o <= p_reg[0] [2*DATA_WIDTH-2-GAIN_BITS -:
DATA_WIDTH];
end
end

// =====
// MODE 2: DECIMATION (IS_DECIMATION = 1)
// =====
else begin

    // --- Control Logic ---
    if (valid_i) begin
        active_cycle <= 1'b1;
        if (phase_counter > 0)
            phase_counter <= phase_counter - 1;
    else

```

```

        phase_counter <= PHASES - 1;
    end

    // --- Processing Logic ---
    if (active_cycle) begin
        // Note: In decimation, we use the phase_counter state
*before* the update
            // inside the math loop effectively, but since we
updated it above non-blocking,
            // we need the logic to align.
            // The VHDL used variables to use "current" value before
update.
            // To match VHDL: if phase was 0, it wraps to 7.

            // We need the value BEFORE the decrement above took
effect?
            // No, standard coding: use a temporary or derived
logic.
            // Let's rely on the previous cycle value logic by using
immediate logic if needed.
            // Actually, simpler: Recalculate 'current'
conceptually.

        int current_p;
        // Reverse the decrement logic to find what the phase IS
for this data
        if (phase_counter == PHASES - 1) current_p = 0;
        else current_p = phase_counter + 1;

        // Wait, simpler approach:
        // If we just entered valid_i, the 'phase_counter'
register holds the CURRENT phase.
        // We decrement it for the NEXT cycle.
        // So we use 'phase_counter' (the value before the clock
edge update).

        // But in non-blocking assignments, reading
'phase_counter' reads the OLD value.
        // So simply using phase_counter here works perfectly.

    for (int m = 0; m < PHASES; m++) begin
        for (int t = 0; t < TAPS_PER_PHASE; t++) begin

            // 1. Always latch input
            a_reg[t] <= $signed(data_i);

```

```

        // 2. Load Coefficient
        b_reg[t] <= coeff_rom[t*PHASES + phase_counter];

        // 3. MAC
        if (t == TAPS_PER_PHASE - 1) begin
            p_reg[t*PHASES + (PHASES-1)] <= a_reg[t] *
b_reg[t];
        end else begin
            p_reg[t*PHASES + (PHASES-1)] <= (a_reg[t] *
b_reg[t]) + p_reg[(t+1)*PHASES];
        end

        // 4. Shift
        if (m < PHASES - 1) begin
            p_reg[t*PHASES + m] <= p_reg[t*PHASES + m +
1];
        end
    end
end

// Accumulator Logic
if (phase_counter > 0) begin
    product_sum <= p_reg[0] + product_sum;
end else begin
    // Phase 0 reached (End of Block)
    // Output Result
    // (p_reg[0] + product_sum)
    logic signed [2*DATA_WIDTH-1:0] final_val;
    final_val = p_reg[0] + product_sum;

    data_o <= final_val[2*DATA_WIDTH-2 -: DATA_WIDTH];
    valid_o <= 1'b1;
    product_sum <= '0;
end
end
end
end
endmodule

```

polyphase_resampler

```

`timescale 1ns / 1ps

module polyphase_resampler (
    input logic      clk,
    input logic      rst_n,
    input logic      i_valid,
    input logic signed [15:0] i_data,
    output logic     o_valid,
    output logic signed [15:0] o_data
);

// -----
// Parameters
// -----
localparam int DATA_WIDTH = 16;
localparam int COEFF_WIDTH = 16;

// Stage 1: Upsampler (Interpolate by 2)
localparam int L_FACTOR = 2;
localparam int L_TAPS_TOTAL = 226;
localparam int L_TAPS_PER_PHASE = L_TAPS_TOTAL / L_FACTOR; // 64

// Stage 2: Downampler (Decimate by 3)
localparam int M_FACTOR = 3;
localparam int M_TAPS_TOTAL = 3;
localparam int M_TAPS_PER_PHASE = M_TAPS_TOTAL / M_FACTOR; // 5

//
// -----
// Interconnect Signals
// -----
logic signed [DATA_WIDTH-1:0] s1_data;
logic                      s1_valid;

//
// -----
// 1. Upsampler (x2)
//   Input: 9 MHz -> Output: 18 MHz
//   Uses 128 Coeffs (64 taps per phase)
//
polyphase_filter #(
    .COEFF_FILE      ("interp_l2_226.mem"),

```

```

    .DATA_WIDTH      (DATA_WIDTH),
    .COEFF_WIDTH    (COEFF_WIDTH),
    .PHASES          (L_FACTOR),           // 2
    .TAPS_PER_PHASE (L_TAPS_PER_PHASE), // 64
    .IS_DECIMATION  (0)                  // Interpolation
) u_upsampler (
    .clk      (clk),
    .rst_n   (rst_n),
    .valid_i (i_valid),
    .data_i  (i_data),
    .valid_o (s1_valid),
    .data_o  (s1_data)
);

// =====
// 2. Downampler (/3)
//     Input: 18 MHz -> Output: 6 MHz
//     Uses 15 Coeffs (5 taps per phase), Full Pass
// =====

polyphase_filter #(
    .COEFF_FILE      ("decim_m3_pass.mem"),
    .DATA_WIDTH      (DATA_WIDTH),
    .COEFF_WIDTH    (COEFF_WIDTH),
    .PHASES          (M_FACTOR),           // 3
    .TAPS_PER_PHASE (M_TAPS_PER_PHASE), // 5
    .IS_DECIMATION  (1)                  // Decimation
) u_downampler (
    .clk      (clk),
    .rst_n   (rst_n),
    .valid_i (s1_valid), // Chained from Stage 1
    .data_i  (s1_data),
    .valid_o (o_valid),
    .data_o  (o_data)
);

endmodule

```

TestBench

tb_rational_resampler

```

`timescale 1ns / 1ps

module tb_rational_resampler;
// =====
// PARAMETERS & CONSTANTS
//
// =====

localparam int DATA_WIDTH = 16;
localparam time CLK_PERIOD = 10ns; // 100 MHz Clock

// Rational Resampling Config: 2/3
// Stage 1: Interpolation (L=2)
localparam int L_FACTOR = 2;
localparam int L_TAPS_TOTAL = 226;
localparam int L_TAPS_PER_PHASE = L_TAPS_TOTAL / L_FACTOR; // 64

// Stage 2: Decimation (M=3)
localparam int M_FACTOR = 3;
localparam int M_TAPS_TOTAL = 3;
localparam int M_TAPS_PER_PHASE = M_TAPS_TOTAL / M_FACTOR; // 5

//
// =====
// SIGNALS
//
// =====

logic clk;
logic rst_n;

// Stage 0: Input
logic s0_valid_i;
logic signed [DATA_WIDTH-1:0] s0_data_i;

// Stage 1: Intermediate (Output of x2 Upsampler)
logic s1_valid;
logic signed [DATA_WIDTH-1:0] s1_data;

// Stage 2: Final Output (Output of /3 Downampler)
logic s2_valid;
logic signed [DATA_WIDTH-1:0] s2_data;

// File Handle
integer fd;

```

```

// -----
// COMPONENT INSTANTIATION
// -----
// 1. Upsampler (x2) -> Uses 128 Coeffs (64 per phase)
polyphase_filter #(
    .COEFF_FILE      ("interp_l2_226.mem"),
    .DATA_WIDTH      (DATA_WIDTH),
    .PHASES          (L_FACTOR),           // 2
    .TAPS_PER_PHASE (L_TAPS_PER_PHASE), // 64
    .IS_DECIMATION   (0)                  // Interpolation
) u_upsampler (
    .clk(clk),
    .rst_n(rst_n),
    .valid_i(s0_valid_i),
    .data_i(s0_data_i),
    .valid_o(s1_valid),
    .data_o(s1_data)
);

// 2. Downampler (/3) -> Uses 15 Coeffs (5 per phase), Full Pass
polyphase_filter #(
    .COEFF_FILE      ("decim_m3_pass.mem"),
    .DATA_WIDTH      (DATA_WIDTH),
    .PHASES          (M_FACTOR),           // 3
    .TAPS_PER_PHASE (M_TAPS_PER_PHASE), // 5
    .IS_DECIMATION   (1)                  // Decimation
) u_downampler (
    .clk(clk),
    .rst_n(rst_n),
    .valid_i(s1_valid), // Chained from Stage 1
    .data_i(s1_data),
    .valid_o(s2_valid),
    .data_o(s2_data)
);

// -----
// CLOCK GENERATION
// -----
initial begin
    clk = 0;
    forever #(CLK_PERIOD/2) clk = ~clk;

```

```

end

// =====
// STIMULUS GENERATION
// =====

initial begin
    // Simulation Constants
    real FS_IN = 9.0e6;          // 9 MHz
    real F1     = 1.0e6;          // 1 MHz Tone
    real F2     = 4.0e6;          // 4 MHz Tone (Near Nyquist)
    real SCALE_FACTOR = 15000.0;
    int N_SAMPLES = 200;         // Number of input samples to generate

    real theta1 = 0.0;
    real theta2 = 0.0;
    real step1;
    real step2;
    real val_raw;
    int val_int;
    real PI = 3.141592653589793;

    // Reset Sequence
    rst_n = 0;
    s0_valid_i = 0;
    s0_data_i = 0;
    #100;
    rst_n = 1;
    @(posedge clk);

    $display("-----");
    $display("Generating Two-Tone Signal");
    $display("Tone 1: 1 MHz");
    $display("Tone 2: 4 MHz");
    $display("Fs In : 9 MHz");
    $display("L=2 (128 Taps), M=3 (15 Taps Pass)");
    $display("-----");

    // Open Log File
    fd = $fopen("resampler_output.txt", "w");

    // Calculate Steps
    step1 = 2.0 * PI * F1 / FS_IN;
    step2 = 2.0 * PI * F2 / FS_IN;

```

```

// Main Loop
for (int i = 0; i < N_SAMPLES; i++) begin

    // 1. Math Generation
    val_raw = $sin(theta1) + $sin(theta2);
    val_int = int'(val_raw * SCALE_FACTOR);

    // Saturation
    if (val_int > 32767) val_int = 32767;
    if (val_int < -32768) val_int = -32768;

    // 2. Drive Input
    s0_valid_i <= 1'b1;
    s0_data_i  <= val_int[15:0];

    @(posedge clk);
    // 3. Pipeline Gaps (Matching VHDL "wait for k in 1 to 15")
    // This slows down data input to allow processing time
    s0_valid_i <= 1'b0;
    // s0_data_i  <= '0; // if you remove it . it will cause like
zero hold
repeat(15) @(posedge clk);

    // 4. Update Phase
    theta1 = theta1 + step1;
    if (theta1 > 2.0*PI) theta1 = theta1 - 2.0*PI;

    theta2 = theta2 + step2;
    if (theta2 > 2.0*PI) theta2 = theta2 - 2.0*PI;
end

// End Simulation
#2000;
fclose(fd);
$display("Simulation Finished.");
$finish;
end

// -----
// LOGGING PROCESS
// -----
always @(posedge clk) begin
    if (rst_n) begin
        // Log INPUT

```

```

        if (s0_valid_i) begin
            $fdisplay(fd, "IN: %d", $signed(s0_data_i));
        end

        // Log INTERMEDIATE
        if (s1_valid) begin
            $fdisplay(fd, "MID: %d", $signed(s1_data));
        end

        // Log OUTPUT
        if (s2_valid) begin
            $fdisplay(fd, "OUT: %d", $signed(s2_data));
        end
    end
}

// =====
// WAVEFORM DUMPING (Required for Icarus Verilog)
// =====

initial begin
    $dumpfile("waveform_rational.vcd"); // Must match Makefile VCD2
variable
    $dumpvars(0, tb_rational_resampler);
end

endmodule

```

Makefile

```

# Variables
CC = iverilog
SIM = vvp
VIEWER = surfer
PYTHON = python3

# -g2012 is required for SystemVerilog
FLAGS = -g2012 -Wall

# Common Design File
DUT = polyphase_filter.sv

# ---- CONFIGURATION 2: Rational Resampler (Cascaded) ----

```

```

TB2_FILE = tb_rational_resampler.sv
# Output executable name
OUT2 = rational.vvp
# The VCD filename MUST match what is in $dumpfile inside the SV Testbench
VCD2 = waveform_rational.vcd
# Python script for plotting (Assumes script reads resampler_output.txt)
SCRIPT = plot_resampler.py

.PHONY: all help clean rational view_rational plot_rational

# Default Target
all: plot_rational view_rational

help:
    @echo "-----
    @echo "Available Commands:""
    @echo "  make rational      --> Compile and Run Simulation""
    @echo "  make view_rational  --> Run Sim & View Waveforms (Surfer)""
    @echo "  make plot_rational  --> Run Sim & Plot Data (Python)""
    @echo "  make clean          --> Delete all compiled files""
    @echo "-----"

#
=====
==

# OPTION 2: Rational Resampler (Cascaded)
#
=====

==

# 1. Compile
$(OUT2): $(DUT) $(TB2_FILE)
    @echo "--- Compiling Rational Resampler ---"
    $(CC) $(FLAGS) -o $(OUT2) $(DUT) $(TB2_FILE)

# 2. Run
rational: $(OUT2)
    @echo "--- Running Rational Simulation ---"
    $(SIM) $(OUT2)

# 3. View
view_rational: rational
    @echo "--- Opening Surfer (Rational) ---"
    @if [ -f $(VCD2) ]; then \
        $(VIEWER) $(VCD2) & \
    else \
        echo "Error: $(VCD2) not found. Did you add \$\$dumpfile to the

```

```

testbench?"; \
fi

# 4. Plot (New Target)
plot_rational: rational
    @echo "---- Running Python Analysis ---"
    $(PYTHON) $(SCRIPT);

#
=====
==

# Cleanup
#
=====

==

clean:
    @echo "---- Cleaning ---"
    rm -f *.vvp *.vcf *.out resampler_output.txt

```

Results

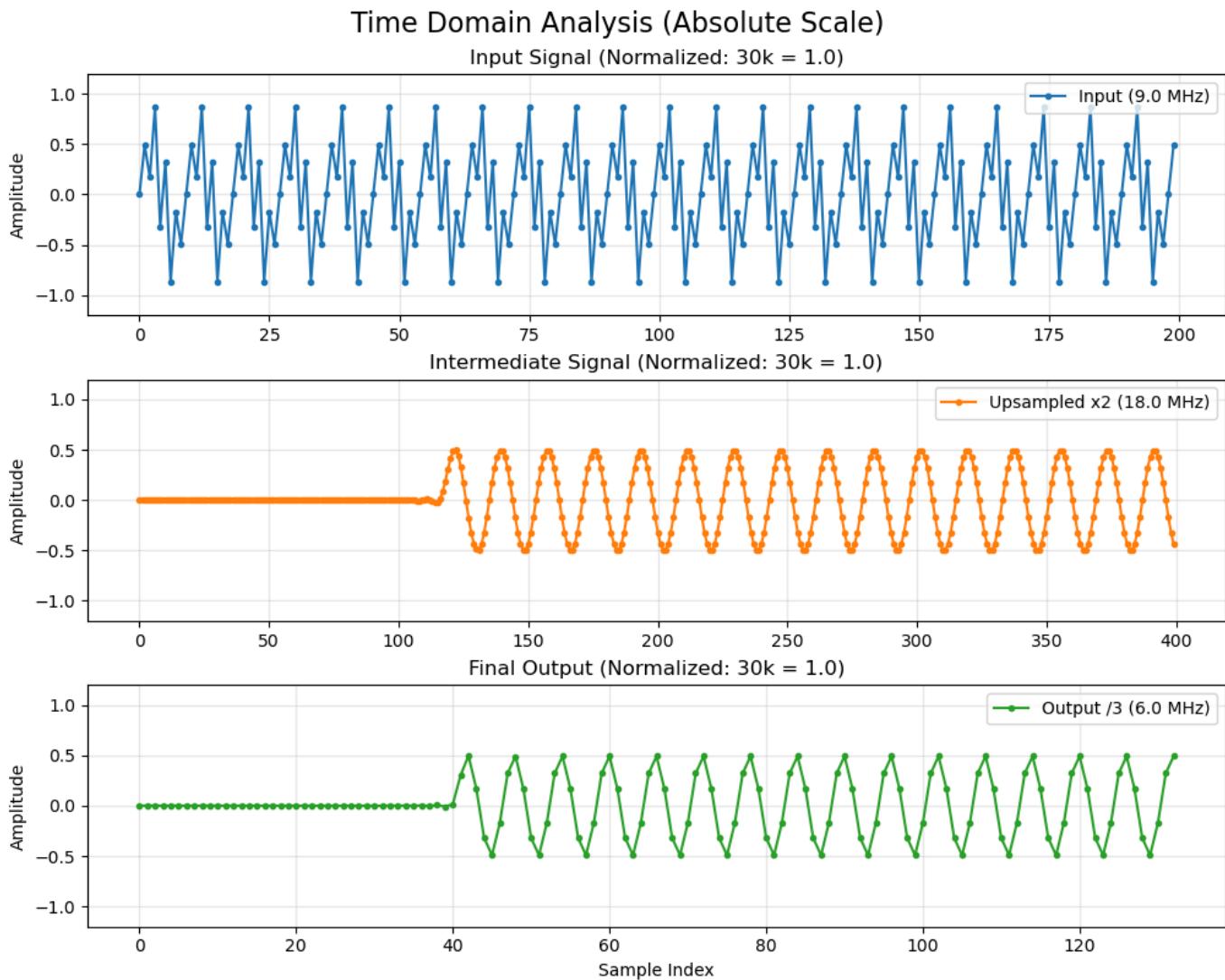
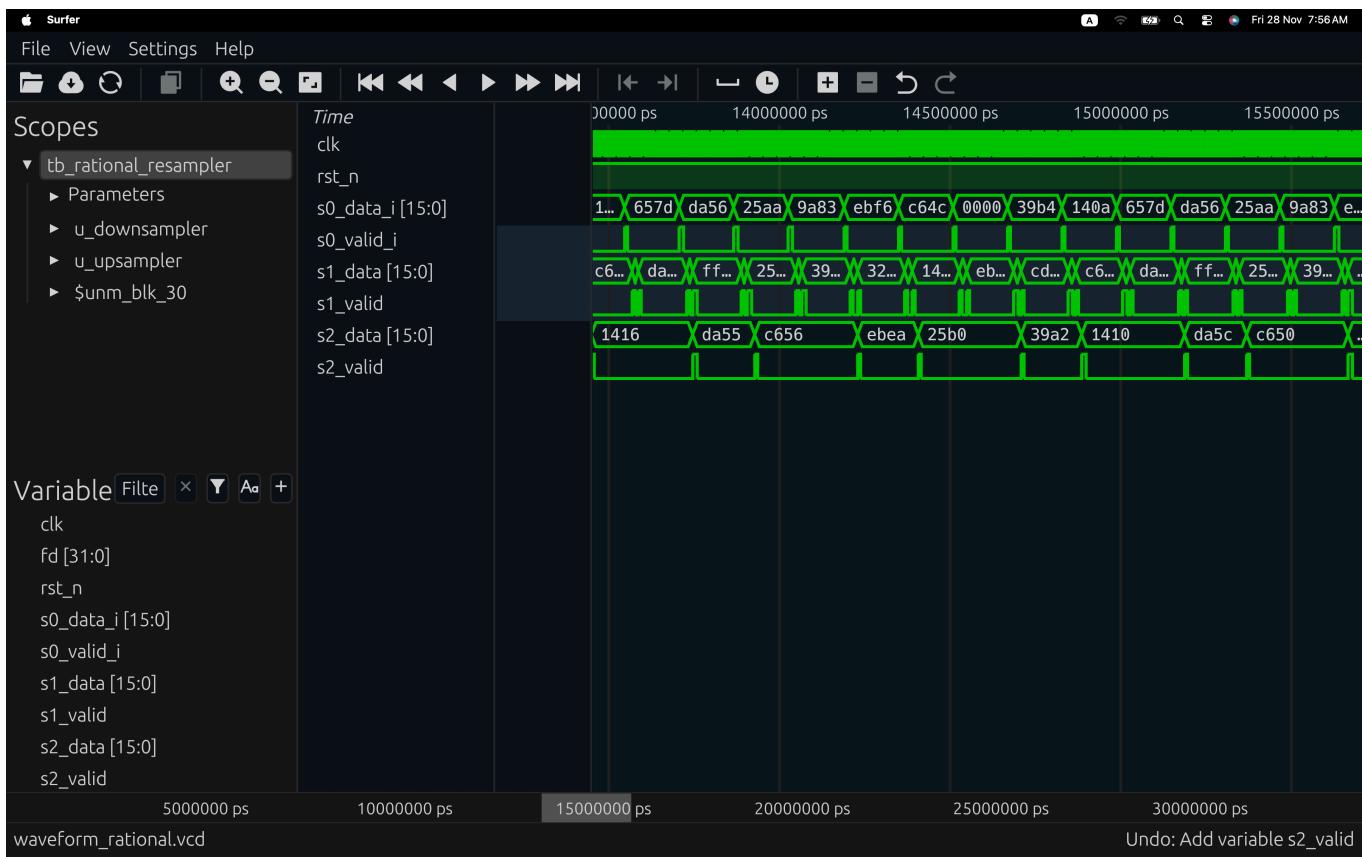
```

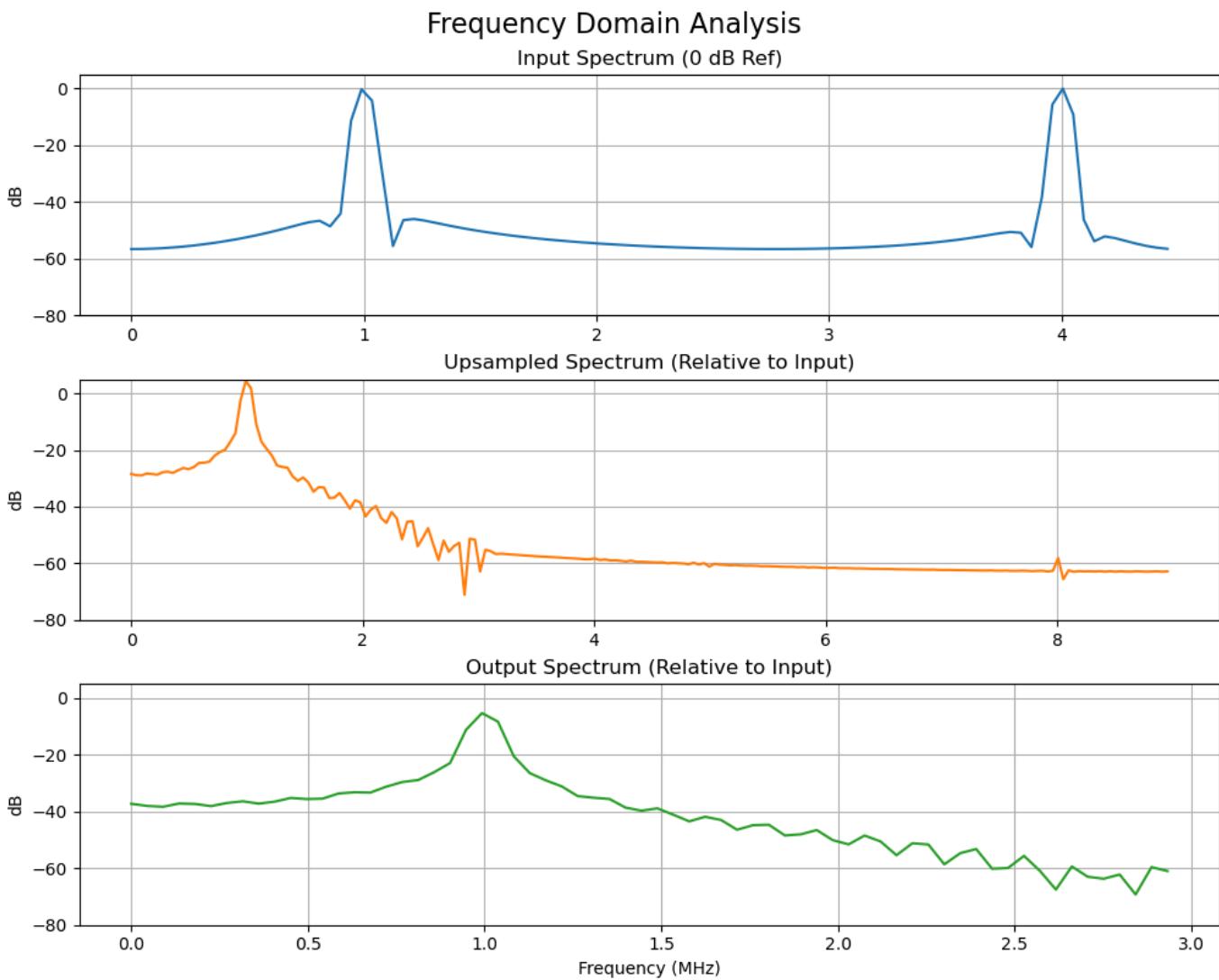
---- Running Rational Simulation ---
vvp rational.vvp
VCD info: dumpfile waveform_rational.vcd opened for output.

-----
Generating Two-Tone Signal
Tone 1: 1 MHz
Tone 2: 4 MHz
Fs In : 9 MHz
L=2 (128 Taps), M=3 (15 Taps Pass)

-----
Simulation Finished.
tb_rational_resampler.sv:163: $finish called at 34105000 (1ps)
---- Running Python Analysis ---
python3 plot_resampler.py;
Reading resampler_output.txt...
Samples Read -> In: 200, Mid: 400, Out: 133
Normalizing all signals by Fixed Factor: 30000.0

```





NOTCH filter

RTL

```

`timescale 1ns / 1ps

module notch_filter #(
    parameter int DATA_WIDTH = 16
) (
    input  logic                  clk,
    input  logic                  rst_n,
    input  logic                  valid_i,
    input  logic signed [DATA_WIDTH-1:0] data_i,

    output logic                 valid_o,
    output logic signed [DATA_WIDTH-1:0] data_o
);

    // 

```

```

// Coefficients (Q2.14)
//
localparam signed [15:0] B0 = 16'd15725;
localparam signed [15:0] B1 = 16'd25443;
localparam signed [15:0] B2 = 16'd15725;
localparam signed [15:0] A1 = 16'd25443;
localparam signed [15:0] A2 = 16'd15066;

//
// Internal State
//
logic signed [31:0] s1, s2;
logic signed [DATA_WIDTH-1:0] x_reg; // Latch input
logic signed [31:0] y_reg;           // Latch intermediate output

// State Machine
typedef enum logic { ST_CALC_OUT, ST_UPDATE_STATE } state_t;
state_t state;

//
// Processing Logic (Split into 2 Cycles)
//
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        valid_o <= 0;
        data_o <= 0;
        s1      <= 0;
        s2      <= 0;
        x_reg   <= 0;
        y_reg   <= 0;
        state   <= ST_CALC_OUT;
    end else begin
        valid_o <= 0; // Default low
    end
    case (state)
        // -----
        // State 0: Wait for Input -> Calculate Output (y)
        // -----

```

```

ST_CALC_OUT: begin
    if (valid_i) begin
        // 1. Latch Input
        x_reg <= data_i;

        // 2. Calculate Feed-Forward (Part A)
        //  $y[n] = b0*x[n] + s1[n-1]$ 
        // We register this result to break the timing path.
        y_reg <= (data_i * B0) + s1;

        // Move to next step immediately
        state <= ST_UPDATE_STATE;
    end
end

// -----
-----

// State 1: Calculate Feedback -> Update States (s1, s2)
// -----
-----
```

here.

```

ST_UPDATE_STATE: begin
    // 1. Output the result calculated in previous cycle
    data_o <= y_reg[29:14]; // Scale Q2.29 -> Q1.15
    valid_o <= 1;           // Signal valid output

    // 2. Calculate Feedback Terms (Using registered y_reg)
    // This splits the math load: Multiply A1/A2 happens
    // here.

    //  $s1[n] = b1*x - a1*y + s2[n-1]$ 
    s1 <= (x_reg * B1) - ((y_reg >> 14) * A1) + s2;

    //  $s2[n] = b2*x - a2*y$ 
    s2 <= (x_reg * B2) - ((y_reg >> 14) * A2);

    // Done, go back to wait for next sample
    state <= ST_CALC_OUT;
end
endcase
end
end

endmodule

```

TestBench

```
`timescale 1ns / 1ps

module tb_notch_filter;

// -----
// Header
//
// -----
// Author: Hazem Yasser Mahmoud Mohamed
// Description: Testbench for Notch Filter with Valid every 4 cycles

//
// -----
// Parameters
//
// -----
localparam int DATA_WIDTH = 16;
localparam real FS = 6.0e6;          // Sampling Frequency: 6 MHz
localparam int NUM_SAMPLES = 2048;   // Number of samples to simulate

//
// -----
// Signals
//
// -----
logic           clk;
logic           rst_n;
logic           valid_i;
logic signed [DATA_WIDTH-1:0] data_i;
logic           valid_o;
logic signed [DATA_WIDTH-1:0] data_o;

// File Handle
int fd;

// Simulation Variables
real t;
real val_1_0m, val_2_4m, val_total;
integer i;

// Temporary integer for conversion
int temp_val;
```

```

// -----
// DUT Instantiation
//
=====

notch_filter #(
    .DATA_WIDTH(DATA_WIDTH)
) u_dut (
    .clk      (clk),
    .rst_n   (rst_n),
    .valid_i (valid_i),
    .data_i  (data_i),
    .valid_o (valid_o),
    .data_o  (data_o)
);

// -----
// Clock Generation
//
=====

initial begin
    clk = 0;
    forever #5 clk = ~clk; // 100 MHz System Clock
end

// -----
// VCD Dump (Waveform Generation)
//
=====

initial begin
    $dumpfile("notch_filter.vcd");
    $dumpvars(0, tb_notch_filter);
end

// -----
// Stimulus Generation
//
=====

initial begin
    // 1. Initialize
    rst_n  = 0;
    valid_i = 0;
    data_i = 0;

```

```

fd      = $fopen("notch_io.txt", "w");

if (fd == 0) begin
    $display("Error: Could not open output file.");
    $finish;
end

// 2. Reset Sequence
repeat(10) @(posedge clk);
rst_n = 1;
repeat(10) @(posedge clk);

$display("Starting Simulation...");

// 3. Drive Data (Valid High for 1 cycle, Low for 3 cycles)
for (i = 0; i < NUM_SAMPLES; i++) begin

    // Calculate time 't'
    t = real'(i) / FS;

    // Tone 1: 1.0 MHz (Passband)
    val_1_0m = $sin(2.0 * 3.14159 * 1.0e6 * t);

    // Tone 2: 2.4 MHz (Notch Frequency)
    val_2_4m = $sin(2.0 * 3.14159 * 2.4e6 * t);

    // Combine: 1MHz + 0.2 DC + 2.4MHz
    val_total = val_1_0m + 0.2 + val_2_4m;

    // Scale to fit Q1.15 Fixed Point
    // Scaling factor 0.4 to keep within range (-1.0 to 1.0)
    temp_val = $rtoi(val_total * 0.4 * 32767.0);

    // Clamp to 16-bit range
    if (temp_val > 32767) temp_val = 32767;
    if (temp_val < -32768) temp_val = -32768;

    // --- DRIVE DATA ---
    valid_i <= 1'b1;
    data_i  <= temp_val[15:0];

    // Wait 1 clock cycle (Active Cycle)
    @(posedge clk);

    // --- IDLE GAP ---
    valid_i <= 1'b0;

```

```

        // Wait 3 clock cycles (Idle Cycles) -> Total period = 4 clocks
        repeat(3) @ (posedge clk);
    end

    @(posedge clk);
    valid_i = 0;
    data_i = 0;

    // Allow pipeline to flush
    repeat(50) @ (posedge clk);

    $display("Simulation Finished. Waveform dumped to
notch_filter.vcd");
    $display("Data written to notch_io.txt");
    $fclose(fd);
    $finish;
end

// =====
// Text Output (CSV Style)
// =====

always @ (posedge clk) begin
    if (rst_n) begin
        // Only write to file when valid_i or valid_o is active to save
        space/readability
        // Or keep it continuous to see the gaps. Keeping continuous
        based on previous code.
        $fdisplay(fd, "%d, %d, %d, %d", valid_i, data_i, valid_o,
data_o);
    end
end

endmodule

```

Makefile

```

#
// =====
===
# Makefile for Notch Filter Simulation
# Tools: Icarus Verilog (iverilog), VVP, GTKWave

```

```

#
=====
==

# Tools
COMPILER = iverilog
SIMULATOR = vvp
VIEWER = surfer
PYTHON = python3

# Files
# Assuming notch_filter.sv is in the same directory
SRC = notch_filter.sv tb_notch_filter.sv
OUT = notch_sim.out
VCD = notch_filter.vcd
TXT = notch_io.txt
SCRIPT = plot_notch_io.py

# Flags
# -g2012 enables SystemVerilog 2012 support
FLAGS = -g2012 -Wall

# Targets -----
-----

.PHONY: all compile run view plot clean

all: compile run

# 1. Compile Verilog
compile:
    @echo "Compiling SystemVerilog files..."
    $(COMPILER) $(FLAGS) -o $(OUT) $(SRC)

# 2. Run Simulation
run: compile
    @echo "Running Simulation..."
    $(SIMULATOR) $(OUT)

# 3. View Waveform (GTKWave)
view: run
    @echo "Opening Waveform Viewer..."
    $(VIEWER) $(VCD) &

# 4. Run Python Analysis
plot: run

```

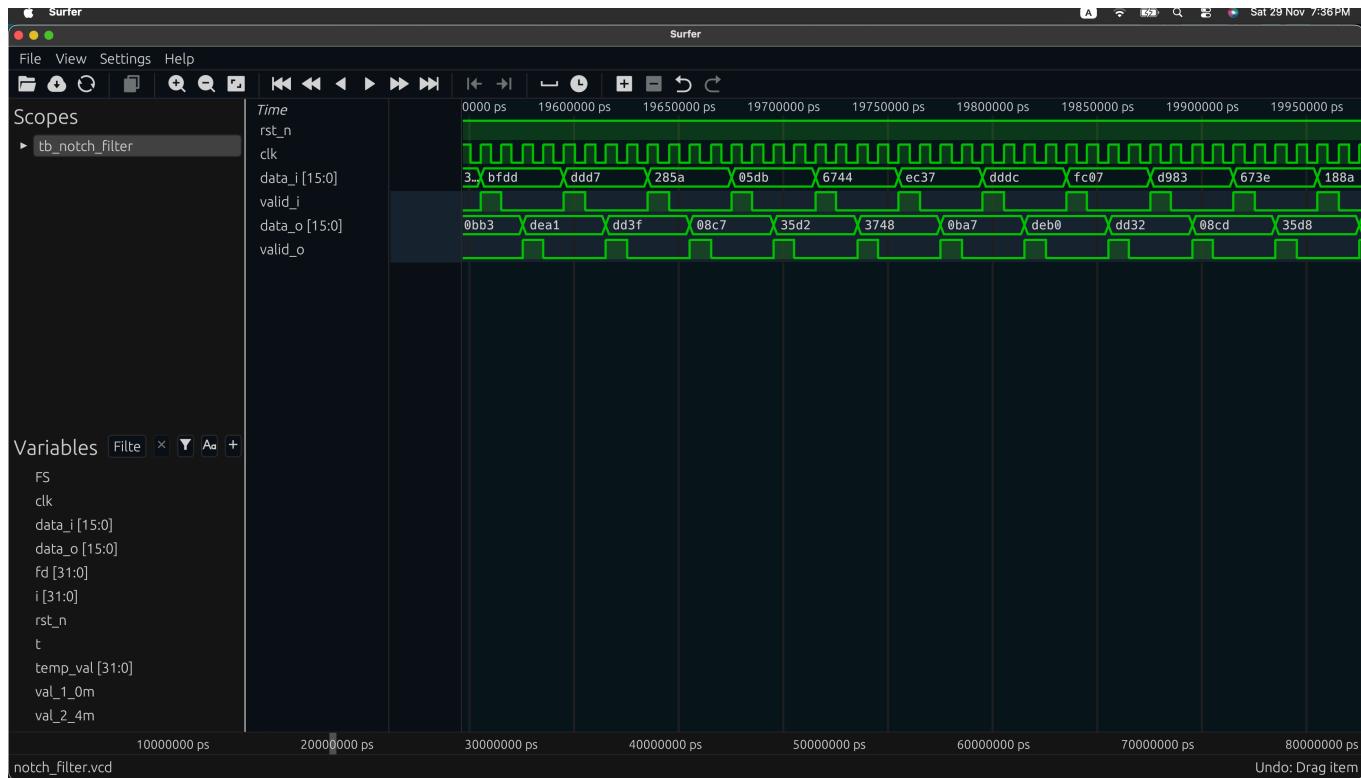
```

@echo "Running Python Analysis..."
$(PYTHON) $(SCRIPT)

# 5. Clean up
clean:
    @echo "Cleaning up..."
    rm -f $(OUT) $(VCD) $(TXT) *.png

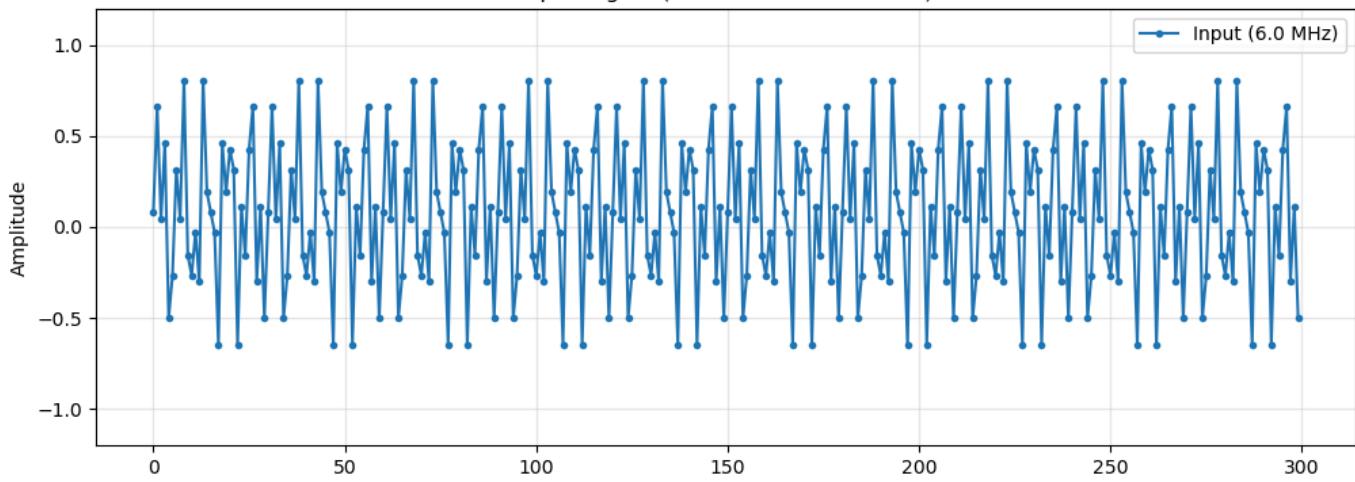
```

Results

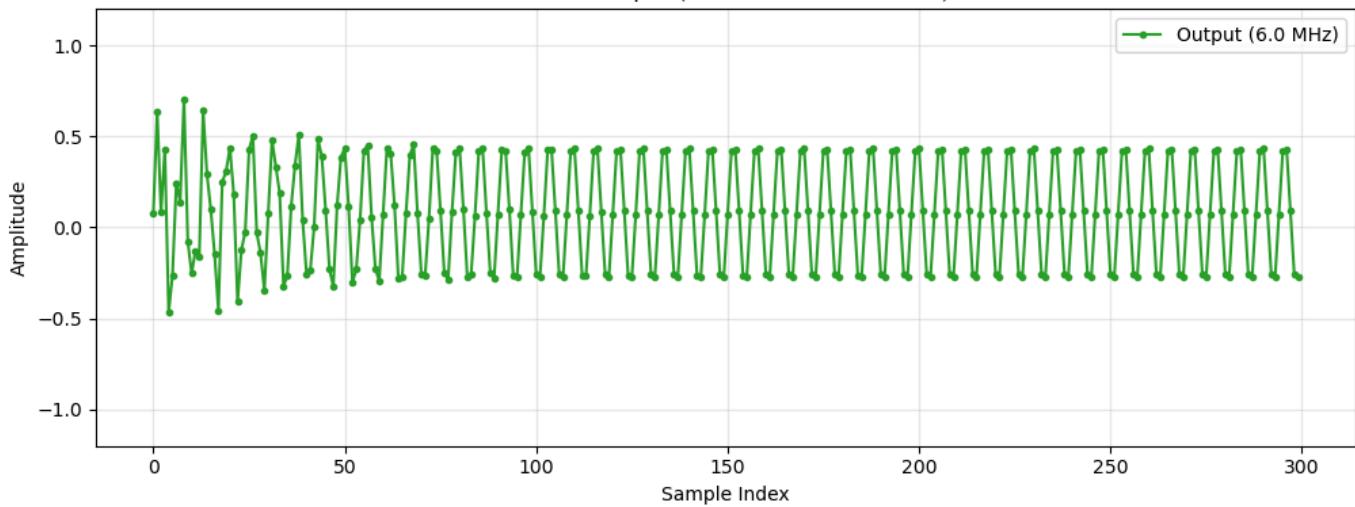


Time Domain Analysis (Absolute Scale)

Input Signal (Normalized: 32k = 1.0)



Filtered Output (Normalized: 32k = 1.0)



CIC

RTL

```

`timescale 1ns / 1ps

module cic_filter #(
    parameter int DATA_WIDTH = 16
) (
    input logic                 clk,
    input logic                 rst_n,
    input logic [4:0]           rate_i, // Decimation Rate: 1, 2, 4, 8,
16
    input logic                 valid_i,
    input logic signed [DATA_WIDTH-1:0] data_i,

    output logic                valid_o,
    output logic signed [DATA_WIDTH-1:0] data_o
);

```

```

//=====
// Parameters & Derived Types
//=====

// Max Decimation R = 16, Stages N = 5.
// Max Gain G = R^N = 16^5 = 1,048,576 (approx 2^20).
// Required Internal Width = Input Width + ceil(log2(G)) = 16 + 20 = 36
bits.
localparam int INTERNAL_WIDTH = 36;
localparam int STAGES = 5;

//=====
// Internal Signals
//=====

// Integrator Signals
// We sign-extend input to internal width
logic signed [INTERNAL_WIDTH-1:0] int_in;
logic signed [INTERNAL_WIDTH-1:0] integrators [0:STAGES-1];

// Comb Signals
logic signed [INTERNAL_WIDTH-1:0] comb_in;
logic signed [INTERNAL_WIDTH-1:0] combs [0:STAGES-1];
logic signed [INTERNAL_WIDTH-1:0] comb_delays [0:STAGES-1];

// Decimation Control
logic [4:0] count;
logic        comb_pulse; // Valid signal for the comb section (rate / R)

// Output Scaling
logic signed [INTERNAL_WIDTH-1:0] scaled_data;
int shift_amount;

//=====
// 1. Integrator Section (Running at High Rate)
//=====

// Sign extend input
assign int_in = {{ (INTERNAL_WIDTH-DATA_WIDTH){data_i[DATA_WIDTH-1]} }, 
data_i};

```

```

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        for (int i = 0; i < STAGES; i++) integrators[i] <= '0;
    end else if (valid_i) begin
        // Stage 0 accumulates Input
        // Standard CIC Integrator: y[n] = y[n-1] + x[n]
        // Relies on modulo arithmetic wrapping (2's complement)
        integrators[0] <= integrators[0] + int_in;

        // Subsequent stages accumulate previous stage
        for (int i = 1; i < STAGES; i++) begin
            integrators[i] <= integrators[i] + integrators[i-1];
        end
    end
end

// =====
// 2. Decimator (Rate Change)
// =====

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        count <= 0;
        comb_pulse <= 0;
    end else if (valid_i) begin
        if (rate_i <= 1) begin
            // If rate is 1, we treat it as always valid (Bypass logic
handles data)
            // But for comb logic, we don't pulse to save power/logic
            count <= 0;
            comb_pulse <= 0;
        end else begin
            // Count from 0 to R-1
            if (count == rate_i - 1) begin
                count <= 0;
                comb_pulse <= 1;
            end else begin
                count <= count + 1;
                comb_pulse <= 0;
            end
        end
    end else begin
        comb_pulse <= 0;
    end
end

```

```

        end
    end

    //

=====

// 3. Comb Section (Running at Low Rate)
//

=====

// Input to combs comes from the last integrator
assign comb_in = integrators[STAGES-1];

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        for (int i = 0; i < STAGES; i++) begin
            combs[i] <= '0;
            comb_delays[i] <= '0;
        end
        valid_o <= 0;
    end else if (comb_pulse) begin
        // Stage 0 Comb
        // y[m] = x[m] - x[m-1] (M=1 differential delay)
        combs[0]      <= comb_in - comb_delays[0];
        comb_delays[0] <= comb_in;

        // Subsequent Stages
        for (int i = 1; i < STAGES; i++) begin
            combs[i]      <= combs[i-1] - comb_delays[i];
            comb_delays[i] <= combs[i-1];
        end
        valid_o <= 1; // Pulse output valid
    end else begin
        // For Rate=1, valid logic is handled in the final block
        if (rate_i > 1) valid_o <= 0;
    end
end

//

=====

// 4. Output Scaling & Mux (Bypass Logic)
//


=====

// Gain G = R^N. To normalize, we right shift by log2(G) = N * log2(R).
// N=5.

always_comb begin
    case (rate_i)

```

```

    5'd2:    shift_amount = 5; // 2^5 = 32 (Shift 5)
    5'd4:    shift_amount = 10; // 4^5 = 1024 (Shift 10)
    5'd8:    shift_amount = 15; // 8^5 = 32k (Shift 15)
    5'd16:   shift_amount = 20; // 16^5 = 1M (Shift 20)
    default: shift_amount = 0;
endcase
end

// Truncation/Rounding logic could go here. For simplicity, we use
truncation (>>>).

// Note: We use the output of the last comb stage.
assign scaled_data = combs[STAGES-1] >>> shift_amount;

// Final Output Mux
always_comb begin
    if (rate_i <= 1) begin
        // Bypass Mode
        data_o = data_i;
        // valid_o follows valid_i directly in bypass
        // (Note: The sequential block above drives valid_o for
decimation,
        // we override it here for bypass combinationally or we need
distinct signals)
    end else begin
        // Decimation Mode
        // Clamp to output width (Saturation)
        // Although unity gain suggests it fits, transients might
overflow slightly.
        // Simple truncation:
        data_o = scaled_data[DATA_WIDTH-1:0];
    end
end

// Fix valid_o for Bypass:
// We need a clean valid signal.
// Let's create a specific `final_valid` wire
logic final_valid;
assign final_valid = (rate_i <= 1) ? valid_i : valid_o;

// Because valid_o is a reg in the always block, we can't assign it
continuously.
// Let's rename the register to `decim_valid` and assign valid_o.

// RE-WRITING Output Logic for clarity/correctness

endmodule

```

TestBench

```
`timescale 1ns / 1ps

module tb_cic_filter;

// Parameters
localparam int DATA_WIDTH = 16;
localparam real FS_IN = 6.0e6; // 6 MHz Input
localparam int R = 4;           // Testing Decimation by 4 (Output 1.5
MHz)

// Signals
logic clk;
logic rst_n;
logic [4:0] rate;
logic valid_i;
logic signed [DATA_WIDTH-1:0] data_i;
logic valid_o;
logic signed [DATA_WIDTH-1:0] data_o;

int fd;
real t;
real val_pass, val_stop, val_total;
int i;

// DUT
cic_filter #(
    .DATA_WIDTH(DATA_WIDTH)
) u_dut (
    .clk(clk),
    .rst_n(rst_n),
    .rate_i(rate),
    .valid_i(valid_i),
    .data_i(data_i),
    .valid_o(valid_o),
    .data_o(data_o)
);

// Clock
initial begin
    clk = 0;
```

```

        forever #5 clk = ~clk; // 100 MHz System Clock
    end

    // Simulation
    initial begin
        $dumpfile("cic_filter.vcd");
        $dumpvars(0, tb_cic_filter);

        fd = $fopen("cic_io.txt", "w");

        // Init
        rst_n = 0;
        valid_i = 0;
        data_i = 0;
        rate = R; // Set decimation rate

        repeat(10) @(posedge clk);
        rst_n = 1;
        repeat(10) @(posedge clk);

        $display("Starting CIC Simulation (R=%0d)...", R);

        // Drive 2048 Samples
        for (i = 0; i < 2048; i++) begin
            @(posedge clk);
            t = real'(i) / FS_IN;

            // 100 kHz (Passband)
            val_pass = $sin(2.0 * 3.14159 * 0.1e6 * t);

            // 2.0 MHz (Stopband for R=4, aliasing zone)
            // CIC should attenuate this significantly (sinc nulls at
            multiples of fs_out=1.5M?)
            // First null is at fs_in/R = 1.5MHz.
            // 2.0MHz is in the side lobe, should be attenuated.
            val_stop = $sin(2.0 * 3.14159 * 2.0e6 * t);

            val_total = (val_pass + val_stop) * 0.5; // Scale to fit

            data_i <= $rtoi(val_total * 32767.0);
            valid_i <= 1;
        end

        @(posedge clk);
        valid_i = 0;
    
```

```

        repeat(100) @(posedge clk);
        $fclose(fd);
        $display("Simulation Finished.");
        $finish;
    end

    // File Write
    always @(posedge clk) begin
        if (rst_n) begin
            $fdisplay(fd, "%d, %d, %d, %d", valid_i, data_i, valid_o,
data_o);
        end
    end
endmodule

```

Makefile

```

#
=====
==

# Makefile for CIC Filter Simulation
# Tools: Icarus Verilog (iverilog), VVP, Surfer/GTKWave
#
=====

==

# Tools
CC = iverilog
SIM = vvp
# Change VIEWER to gtkwave if you don't have surfer installed
VIEWER = surfer
PYTHON = python3

# Flags (-g2012 for SystemVerilog support)
FLAGS = -g2012 -Wall

# Files
DUT = cic_filter.sv
TB = tb_cic_filter.sv
OUT = cic_sim.out
VCD = cic_filter.vcd
TXT = cic_io.txt

```

```

SCRIPT = plot_cic.py

.PHONY: all compile run view plot clean help

# Default Target
all: plot view

help:
    @echo "-----"
    @echo "Available Commands:"
    @echo "  make compile      -> Compile SystemVerilog files"
    @echo "  make run          -> Run Simulation (generates .vcd and .txt)"
    @echo "  make view         -> Open Waveform Viewer"
    @echo "  make plot         -> Run Python Analysis Script"
    @echo "  make clean        -> Remove generated files"
    @echo "-----"

# 1. Compile
$(OUT): $(DUT) $(TB)
    @echo "--- Compiling CIC Filter ---"
    $(CC) $(FLAGS) -o $(OUT) $(DUT) $(TB)

compile: $(OUT)

# 2. Run
run: $(OUT)
    @echo "--- Running Simulation ---"
    $(SIM) $(OUT)

# 3. View
view: run
    @echo "--- Opening Waveform Viewer ---"
    @if [ -f $(VCD) ]; then \
        $(VIEWER) $(VCD) & \
    else \
        echo "Error: $(VCD) not found."; \
    fi

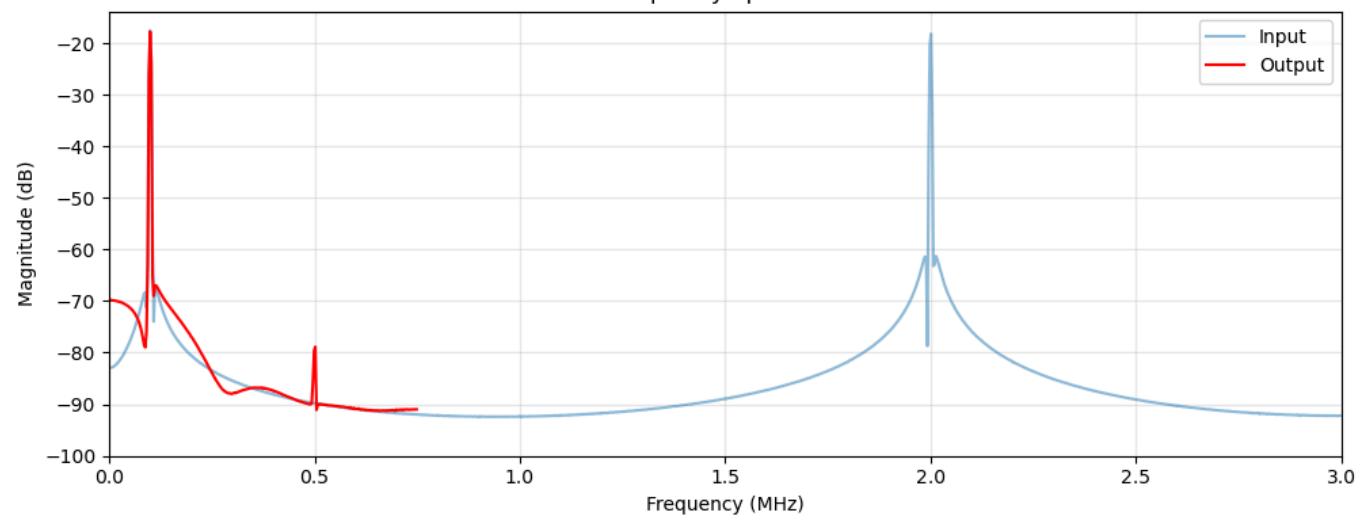
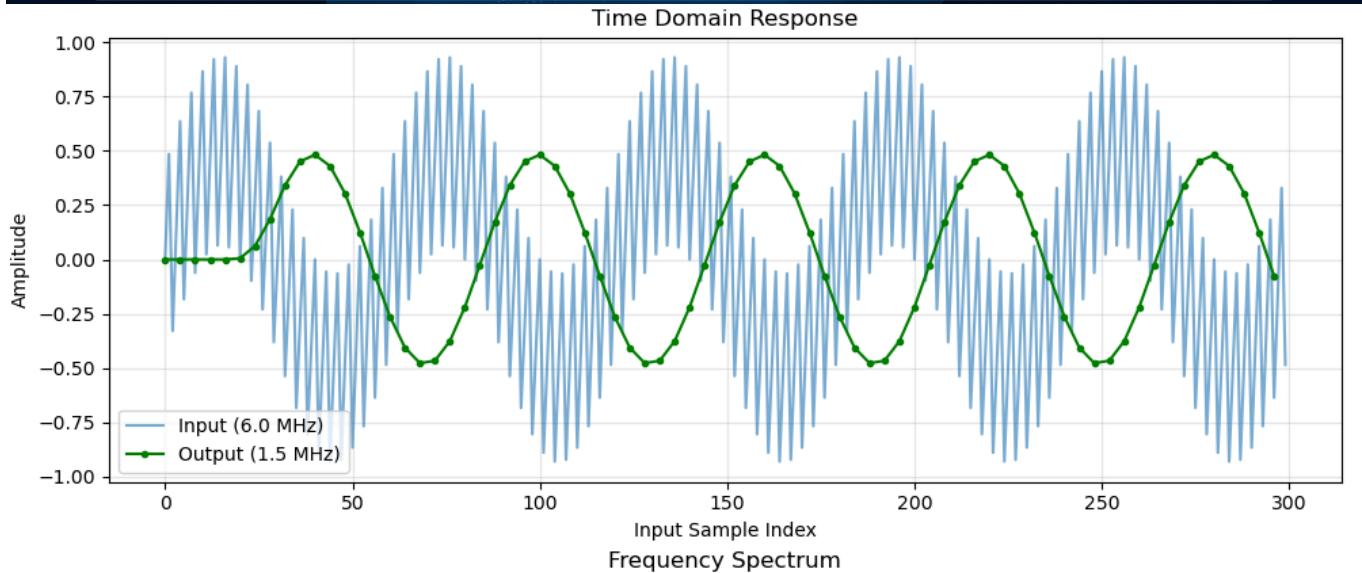
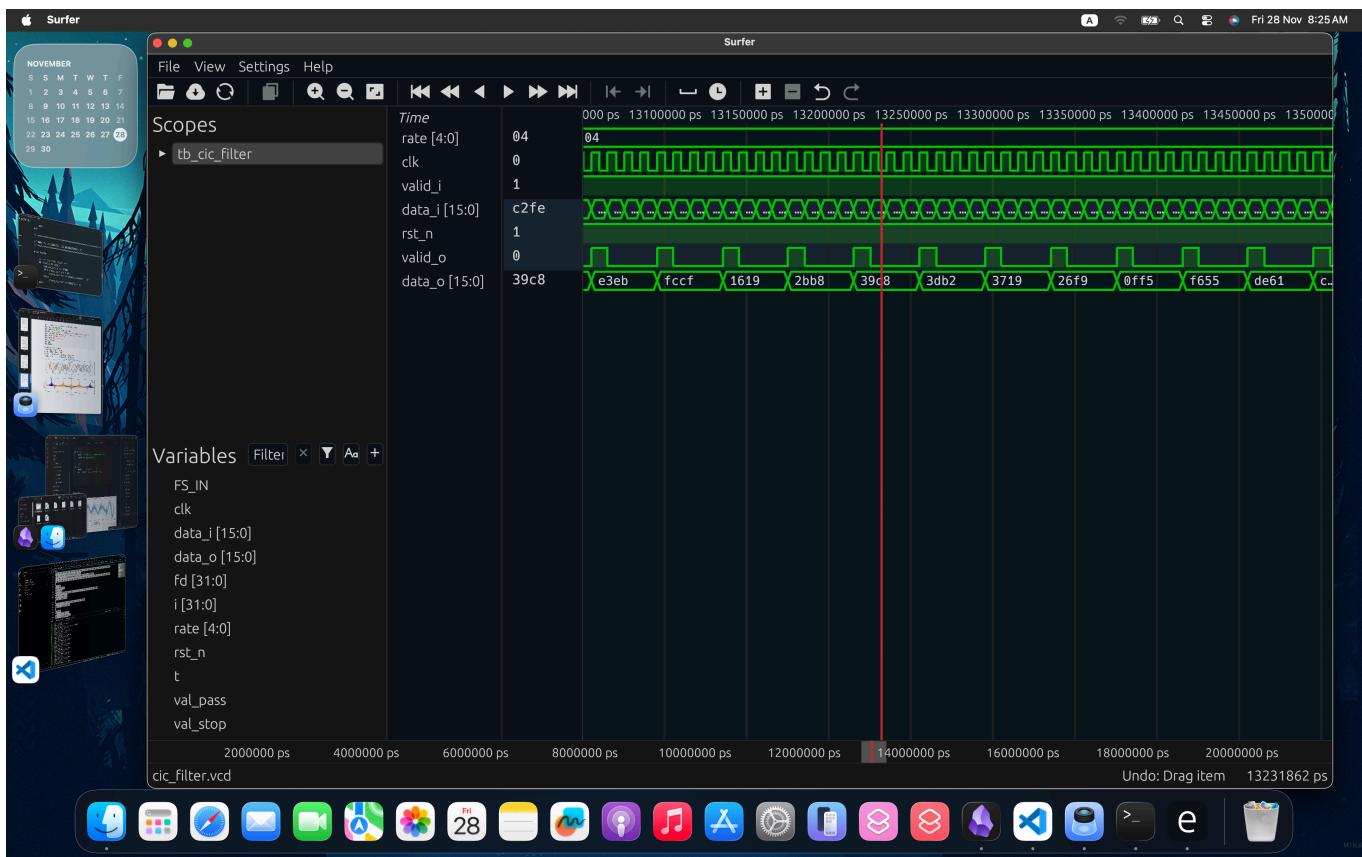
# 4. Plot
plot: run
    @echo "--- Running Python Analysis ---"
    $(PYTHON) $(SCRIPT); \

# Cleanup
clean:
    @echo "--- Cleaning ---"

```

```
rm -f $(OUT) $(VCD) $(TXT) *.png
```

Results



DEF TOP FILE

RTL

```
`timescale 1ns / 1ps

module dfe_top #(
    parameter int DATA_WIDTH = 16
) (
    input  logic                  clk,
    input  logic                  rst_n,
    // Configuration
    input  logic [4:0]             cic_rate_i,
    // Input Stream
    input  logic                  valid_i,
    input  logic signed [DATA_WIDTH-1:0] data_i,
    // Output Stream
    output logic                  valid_o,
    output logic signed [DATA_WIDTH-1:0] data_o
);

// =====
// 1. Input Pipeline Registers (CRITICAL FOR TIMING)
//     Isolates IO Pins from the heavy fanout of the filters.
// =====

logic signed [DATA_WIDTH-1:0] r_data_i;
logic                         r_valid_i;
logic [4:0]                   r_cic_rate;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_data_i  <= 0;
        r_valid_i <= 0;
        r_cic_rate <= 1;
    end else begin
        r_data_i  <= data_i;
        r_valid_i <= valid_i;
        r_cic_rate <= cic_rate_i;
    end
end
end
```

```

//=====
// Interconnect Signals
//=====

logic signed [DATA_WIDTH-1:0] w_resamp_data;
logic                      w_resamp_valid;
logic signed [DATA_WIDTH-1:0] w_notch_data;
logic                      w_notch_valid;

// Internal Output Signals
logic signed [DATA_WIDTH-1:0] w_final_data;
logic                      w_final_valid;

//=====

// Stage 1: Polyphase Resampler
//=====

polyphase_resampler u_stage1_resampler (
    .clk      (clk),
    .rst_n   (rst_n),
    .i_valid  (r_valid_i), // Connect to Pipelined Register
    .i_data   (r_data_i), // Connect to Pipelined Register
    .o_valid  (w_resamp_valid),
    .o_data   (w_resamp_data)
);

//=====

// Stage 2: Notch Filter
//=====

notch_filter #(
    .DATA_WIDTH (DATA_WIDTH)
) u_stage2_notch (
    .clk      (clk),
    .rst_n   (rst_n),
    .valid_i  (w_resamp_valid),
    .data_i   (w_resamp_data),
    .valid_o  (w_notch_valid),
    .data_o   (w_notch_data)
);

//=====
```

```

=====
// Stage 3: CIC Filter
//



=====

cic_filter #(
    .DATA_WIDTH (DATA_WIDTH)
) u_stage3_cic (
    .clk          (clk),
    .rst_n        (rst_n),
    .rate_i       (r_cic_rate), // Connect to Pipelined Register
    .valid_i      (w_notch_valid),
    .data_i       (w_notch_data),
    .valid_o      (w_final_valid),
    .data_o       (w_final_data)
);

//


=====

// 2. Output Pipeline Registers
//     Improves timing for signals leaving the chip to PMODs/LEDs
//


=====

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        valid_o <= 0;
        data_o  <= 0;
    end else begin
        valid_o <= w_final_valid;
        data_o  <= w_final_data;
    end
end

endmodule

```

TestBench

```

`timescale 1ns / 1ps

module tb_dfe_top;

// Parameters
localparam int DATA_WIDTH = 16;

```

```

localparam real FS_IN = 9.0e6;
localparam int NUM_SAMPLES = 4096;

// Signals
logic clk;
logic rst_n;
logic [4:0] cic_rate;
logic valid_i;
logic signed [DATA_WIDTH-1:0] data_i;
logic valid_o;
logic signed [DATA_WIDTH-1:0] data_o;

int fd;
real t;
real val_0_2m, val_1_0m, val_2_4m, val_4_0m, val_total;
integer i;
int temp_val;

// DUT
dfe_top #(
    .DATA_WIDTH(DATA_WIDTH)
) u_dut (
    .clk      (clk),
    .rst_n    (rst_n),
    .cic_rate_i (cic_rate),
    .valid_i   (valid_i),
    .data_i    (data_i),
    .valid_o   (valid_o),
    .data_o    (data_o)
);

// Clock (100 MHz)
initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

// VCD
initial begin
    $dumpfile("dfe_top.vcd");
    $dumpvars(0, tb_dfe_top);
end

// Stimulus
initial begin
    rst_n = 0;

```

```

valid_i  = 0;
data_i   = 0;
cic_rate = 4;

fd = $fopen("dfe_io.txt", "w");
if (fd == 0) begin
    $display("Error: Could not open output file.");
    $finish;
end

repeat(10) @(posedge clk);
rst_n = 1;
repeat(10) @(posedge clk);

$display("Starting DFE Simulation..."); 
$display("Input Fs: 9 MHz (Simulated Rate)");

// -----
// DATA DRIVER LOOP
// -----
for (i = 0; i < NUM_SAMPLES; i++) begin
    @(posedge clk);

    // 1. Calculate Signal Value
    t = real'(i) / FS_IN;
    val_0_2m = $sin(2.0 * 3.14159 * 0.2e6 * t);
    val_1_0m = $sin(2.0 * 3.14159 * 1.0e6 * t);
    val_2_4m = $sin(2.0 * 3.14159 * 2.4e6 * t);
    val_4_0m = $sin(2.0 * 3.14159 * 4.0e6 * t) * 0.5;

    val_total = val_0_2m + val_1_0m + val_2_4m + val_4_0m;
    temp_val = $rtoi(val_total * 0.25 * 32767.0); // Scale to Q1.15

    if (temp_val > 32767) temp_val = 32767;
    if (temp_val < -32768) temp_val = -32768;

    // 2. Drive Valid Pulse
    data_i  <= temp_val[15:0];
    valid_i <= 1'b1;

    // 3. Wait / Throttle
    // The Interpolator (L=2) needs 2 clock cycles to process 1
input.
    // If we drive every cycle, it resets and fails to interpolate.
    // We assert valid for 1 cycle, then deassert.
    @(posedge clk);

```

```

    valid_i <= 1'b0;

    // Wait an extra cycle to ensure the 2-phase filter finishes
    // (Total period = 2 clocks, Rate = 50 MHz vs 100 MHz clk)
    @(posedge clk);
end

@(posedge clk);
valid_i = 0;
data_i = 0;

repeat(200) @(posedge clk);
$fclose(fd);
$display("Simulation Finished.");
$finish;
end

// File Write
always @(posedge clk) begin
    if (rst_n) begin
        $fdisplay(fd, "%d, %d, %d, %d", valid_i, data_i, valid_o,
data_o);
    end
end
endmodule

```

FPGA constraints

constraints1.xdc

output given to led J input from PMOD JA & JB

```

## This file is a specific .xdc for the DFE Top Level on Nexys4 DDR
## -----
## -----


## Clock signal
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports {
clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{clk}];

```

```

## Reset (Active Low - CPU_RESETN)
set_property -dict { PACKAGE_PIN C12    IO_STANDARD LVCMOS33 } [get_ports {rst_n }];
#IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

## -----
-----

## Configuration & Controls (Switches)
## -----
-----


## CIC Decimation Rate (5 bits) -> Switches [4:0]
set_property -dict { PACKAGE_PIN J15    IO_STANDARD LVCMOS33 } [get_ports {cic_rate_i[0]}];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16    IO_STANDARD LVCMOS33 } [get_ports {cic_rate_i[1]}];
#IO_L3N_T0_DQS_EMCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13    IO_STANDARD LVCMOS33 } [get_ports {cic_rate_i[2]}];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15    IO_STANDARD LVCMOS33 } [get_ports {cic_rate_i[3]}];
#IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17    IO_STANDARD LVCMOS33 } [get_ports {cic_rate_i[4]}];
#IO_L12N_T1_MRCC_14 Sch=sw[4]

## Input Valid Signal -> Switch [15]
## Toggle this high to enable data processing
set_property -dict { PACKAGE_PIN V10    IO_STANDARD LVCMOS33 } [get_ports {valid_i }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

## -----
-----

## Output Data Visualization (LEDs)
## -----
-----


## Output Data (16 bits) -> LEDs [15:0]
set_property -dict { PACKAGE_PIN H17    IO_STANDARD LVCMOS33 } [get_ports {data_o[0]}];
#IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15    IO_STANDARD LVCMOS33 } [get_ports {data_o[1]}];
#IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13    IO_STANDARD LVCMOS33 } [get_ports {data_o[2]}];
#IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14    IO_STANDARD LVCMOS33 } [get_ports {data_o[3]}];
#IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18    IO_STANDARD LVCMOS33 } [get_ports {data_o[4]}];
#IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17    IO_STANDARD LVCMOS33 } [get_ports {data_o[5]}];
#IO_L18N_T2_A11_D27_14 Sch=led[5]

```

```

set_property -dict { PACKAGE_PIN U17    IO_STANDARD LVCMOS33 } [get_ports {
data_o[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16    IO_STANDARD LVCMOS33 } [get_ports {
data_o[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16    IO_STANDARD LVCMOS33 } [get_ports {
data_o[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15    IO_STANDARD LVCMOS33 } [get_ports {
data_o[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14    IO_STANDARD LVCMOS33 } [get_ports {
data_o[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16    IO_STANDARD LVCMOS33 } [get_ports {
data_o[11] }]; #IO_L15N_T2_DQS_DOUT_CS0_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15    IO_STANDARD LVCMOS33 } [get_ports {
data_o[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14    IO_STANDARD LVCMOS33 } [get_ports {
data_o[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12    IO_STANDARD LVCMOS33 } [get_ports {
data_o[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11    IO_STANDARD LVCMOS33 } [get_ports {
data_o[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

```

```

## Output Valid -> RGB LED 16 (Green Channel)
## Will light up green when valid output data is present
set_property -dict { PACKAGE_PIN M16    IO_STANDARD LVCMOS33 } [get_ports {
valid_o }]; #IO_L10P_T1_D14_14 Sch=led16_g

```

```

## -----
-----  

## Input Data Mapping (PMOD Headers)
## -----  

-----  

## Since data_i is 16 bits, we map it to PMOD JA (8 bits) and PMOD JB (8  

bits).  

## This is where you would connect your external signal source (ADC).  

## PMOD JA -> data_i[7:0]  

set_property -dict { PACKAGE_PIN C17    IO_STANDARD LVCMOS33 } [get_ports {
data_i[0] }]; #IO_L20N_T3_A19_15 Sch=ja[1]  

set_property -dict { PACKAGE_PIN D18    IO_STANDARD LVCMOS33 } [get_ports {
data_i[1] }]; #IO_L21N_T3_DQS_A18_15 Sch=ja[2]  

set_property -dict { PACKAGE_PIN E18    IO_STANDARD LVCMOS33 } [get_ports {
data_i[2] }]; #IO_L21P_T3_DQS_15 Sch=ja[3]  

set_property -dict { PACKAGE_PIN G17    IO_STANDARD LVCMOS33 } [get_ports {
data_i[3] }]; #IO_L18N_T2_A23_15 Sch=ja[4]  

set_property -dict { PACKAGE_PIN D17    IO_STANDARD LVCMOS33 } [get_ports {
data_i[4] }]; #IO_L16N_T2_A27_15 Sch=ja[7]

```

```

set_property -dict { PACKAGE_PIN E17    IO_STANDARD LVCMOS33 } [get_ports {
data_i[5] }]; #IO_L16P_T2_A28_15 Sch=ja[8]
set_property -dict { PACKAGE_PIN F18    IO_STANDARD LVCMOS33 } [get_ports {
data_i[6] }]; #IO_L22N_T3_A16_15 Sch=ja[9]
set_property -dict { PACKAGE_PIN G18    IO_STANDARD LVCMOS33 } [get_ports {
data_i[7] }]; #IO_L22P_T3_A17_15 Sch=ja[10]

## PMOD JB -> data_i[15:8]
set_property -dict { PACKAGE_PIN D14    IO_STANDARD LVCMOS33 } [get_ports {
data_i[8] }]; #IO_L1P_T0_AD0P_15 Sch=jb[1]
set_property -dict { PACKAGE_PIN F16    IO_STANDARD LVCMOS33 } [get_ports {
data_i[9] }]; #IO_L14N_T2_SRCC_15 Sch=jb[2]
set_property -dict { PACKAGE_PIN G16    IO_STANDARD LVCMOS33 } [get_ports {
data_i[10] }]; #IO_L13N_T2_MRCC_15 Sch=jb[3]
set_property -dict { PACKAGE_PIN H14    IO_STANDARD LVCMOS33 } [get_ports {
data_i[11] }]; #IO_L15P_T2_DQS_15 Sch=jb[4]
set_property -dict { PACKAGE_PIN E16    IO_STANDARD LVCMOS33 } [get_ports {
data_i[12] }]; #IO_L11N_T1_SRCC_15 Sch=jb[7]
set_property -dict { PACKAGE_PIN F13    IO_STANDARD LVCMOS33 } [get_ports {
data_i[13] }]; #IO_L5P_T0_AD9P_15 Sch=jb[8]
set_property -dict { PACKAGE_PIN G13    IO_STANDARD LVCMOS33 } [get_ports {
data_i[14] }]; #IO_0_15 Sch=jb[9]
set_property -dict { PACKAGE_PIN H16    IO_STANDARD LVCMOS33 } [get_ports {
data_i[15] }]; #IO_L13P_T2_MRCC_15 Sch=jb[10]

## -----
-----
## Configuration Properties
## -----
-----

set_property CFGBVS VCC0 [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]

```

constraints2.xdc

output given to PMOD JC & JC input from PMOD JA & JB

```

## This file is a specific .xdc for the DFE Top Level on Nexys4 DDR
## -----
-----
## Clock signal
set_property -dict { PACKAGE_PIN E3    IO_STANDARD LVCMOS33 } [get_ports {

```

```

clk }]; #I0_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{clk}];

## Reset (Active Low - CPU_RESETN)
set_property -dict { PACKAGE_PIN C12 IOSTANDARD LVCMOS33 } [get_ports {
rst_n }]; #I0_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

## -----
-----

## Configuration & Controls (Switches)
## -----
-----


## CIC Decimation Rate (5 bits) -> Switches [4:0]
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[0] }]; #I0_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[1] }]; #I0_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[2] }]; #I0_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[3] }]; #I0_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[4] }]; #I0_L12N_T1_MRCC_14 Sch=sw[4]

## Input Valid Signal -> Switch [15]
## Toggle this high to enable data processing
set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports {
valid_i }]; #I0_L21P_T3_DQS_14 Sch=sw[15]

## -----
-----

## Input Data Mapping (PMOD Headers JA & JB)
## -----
-----


## PMOD JA -> data_i[7:0] (Low Byte)
set_property -dict { PACKAGE_PIN C17 IOSTANDARD LVCMOS33 } [get_ports {
data_i[0] }]; #I0_L20N_T3_A19_15 Sch=ja[1]
set_property -dict { PACKAGE_PIN D18 IOSTANDARD LVCMOS33 } [get_ports {
data_i[1] }]; #I0_L21N_T3_DQS_A18_15 Sch=ja[2]
set_property -dict { PACKAGE_PIN E18 IOSTANDARD LVCMOS33 } [get_ports {
data_i[2] }]; #I0_L21P_T3_DQS_15 Sch=ja[3]
set_property -dict { PACKAGE_PIN G17 IOSTANDARD LVCMOS33 } [get_ports {
data_i[3] }]; #I0_L18N_T2_A23_15 Sch=ja[4]
set_property -dict { PACKAGE_PIN D17 IOSTANDARD LVCMOS33 } [get_ports {

```

```

data_i[4] }]; #IO_L16N_T2_A27_15 Sch=ja[7]
set_property -dict { PACKAGE_PIN E17    IO_STANDARD LVCMOS33 } [get_ports {
data_i[5] }]; #IO_L16P_T2_A28_15 Sch=ja[8]
set_property -dict { PACKAGE_PIN F18    IO_STANDARD LVCMOS33 } [get_ports {
data_i[6] }]; #IO_L22N_T3_A16_15 Sch=ja[9]
set_property -dict { PACKAGE_PIN G18    IO_STANDARD LVCMOS33 } [get_ports {
data_i[7] }]; #IO_L22P_T3_A17_15 Sch=ja[10]

## PMOD JB -> data_i[15:8] (High Byte)
set_property -dict { PACKAGE_PIN D14    IO_STANDARD LVCMOS33 } [get_ports {
data_i[8] }]; #IO_L1P_T0_AD0P_15 Sch=jb[1]
set_property -dict { PACKAGE_PIN F16    IO_STANDARD LVCMOS33 } [get_ports {
data_i[9] }]; #IO_L14N_T2_SRCC_15 Sch=jb[2]
set_property -dict { PACKAGE_PIN G16    IO_STANDARD LVCMOS33 } [get_ports {
data_i[10] }]; #IO_L13N_T2_MRCC_15 Sch=jb[3]
set_property -dict { PACKAGE_PIN H14    IO_STANDARD LVCMOS33 } [get_ports {
data_i[11] }]; #IO_L15P_T2_DQS_15 Sch=jb[4]
set_property -dict { PACKAGE_PIN E16    IO_STANDARD LVCMOS33 } [get_ports {
data_i[12] }]; #IO_L11N_T1_SRCC_15 Sch=jb[7]
set_property -dict { PACKAGE_PIN F13    IO_STANDARD LVCMOS33 } [get_ports {
data_i[13] }]; #IO_L5P_T0_AD9P_15 Sch=jb[8]
set_property -dict { PACKAGE_PIN G13    IO_STANDARD LVCMOS33 } [get_ports {
data_i[14] }]; #IO_0_15 Sch=jb[9]
set_property -dict { PACKAGE_PIN H16    IO_STANDARD LVCMOS33 } [get_ports {
data_i[15] }]; #IO_L13P_T2_MRCC_15 Sch=jb[10]

## -----
-----  

## Output Data Mapping (PMOD Headers JC & JD)
## -----
-----  

## NOTE: Output moved from LEDs to PMODs for Logic Analyzer connection.

## PMOD JC -> data_o[7:0] (Low Byte)
set_property -dict { PACKAGE_PIN K1    IO_STANDARD LVCMOS33 } [get_ports {
data_o[0] }]; #IO_L23N_T3_35 Sch=jc[1]
set_property -dict { PACKAGE_PIN F6    IO_STANDARD LVCMOS33 } [get_ports {
data_o[1] }]; #IO_L19N_T3_VREF_35 Sch=jc[2]
set_property -dict { PACKAGE_PIN J2    IO_STANDARD LVCMOS33 } [get_ports {
data_o[2] }]; #IO_L22N_T3_35 Sch=jc[3]
set_property -dict { PACKAGE_PIN G6    IO_STANDARD LVCMOS33 } [get_ports {
data_o[3] }]; #IO_L19P_T3_35 Sch=jc[4]
set_property -dict { PACKAGE_PIN E7    IO_STANDARD LVCMOS33 } [get_ports {
data_o[4] }]; #IO_L6P_T0_35 Sch=jc[7]
set_property -dict { PACKAGE_PIN J3    IO_STANDARD LVCMOS33 } [get_ports {
data_o[5] }]; #IO_L22P_T3_35 Sch=jc[8]

```

```

set_property -dict { PACKAGE_PIN J4      IO_STANDARD LVCMOS33 } [get_ports {
data_o[6] }; #IO_L21P_T3_DQS_35 Sch=jc[9]
set_property -dict { PACKAGE_PIN E6      IO_STANDARD LVCMOS33 } [get_ports {
data_o[7] }; #IO_L5P_T0_AD13P_35 Sch=jc[10]

## PMOD JD -> data_o[15:8] (High Byte)
set_property -dict { PACKAGE_PIN H4      IO_STANDARD LVCMOS33 } [get_ports {
data_o[8] }; #IO_L21N_T3_DQS_35 Sch=jd[1]
set_property -dict { PACKAGE_PIN H1      IO_STANDARD LVCMOS33 } [get_ports {
data_o[9] }; #IO_L17P_T2_35 Sch=jd[2]
set_property -dict { PACKAGE_PIN G1      IO_STANDARD LVCMOS33 } [get_ports {
data_o[10] }; #IO_L17N_T2_35 Sch=jd[3]
set_property -dict { PACKAGE_PIN G3      IO_STANDARD LVCMOS33 } [get_ports {
data_o[11] }; #IO_L20N_T3_35 Sch=jd[4]
set_property -dict { PACKAGE_PIN H2      IO_STANDARD LVCMOS33 } [get_ports {
data_o[12] }; #IO_L15P_T2_DQS_35 Sch=jd[7]
set_property -dict { PACKAGE_PIN G4      IO_STANDARD LVCMOS33 } [get_ports {
data_o[13] }; #IO_L20P_T3_35 Sch=jd[8]
set_property -dict { PACKAGE_PIN G2      IO_STANDARD LVCMOS33 } [get_ports {
data_o[14] }; #IO_L15N_T2_DQS_35 Sch=jd[9]
set_property -dict { PACKAGE_PIN F3      IO_STANDARD LVCMOS33 } [get_ports {
data_o[15] }; #IO_L13N_T2_MRCC_35 Sch=jd[10]

## -----
-----
## Output Valid Indicator
## -----
-----

## Output Valid -> RGB LED 16 (Green Channel)
set_property -dict { PACKAGE_PIN M16    IO_STANDARD LVCMOS33 } [get_ports {
valid_o }]; #IO_L10P_T1_D14_14 Sch=led16_g

## -----
-----
## Configuration Properties
## -----
-----

set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]

```

Results

```

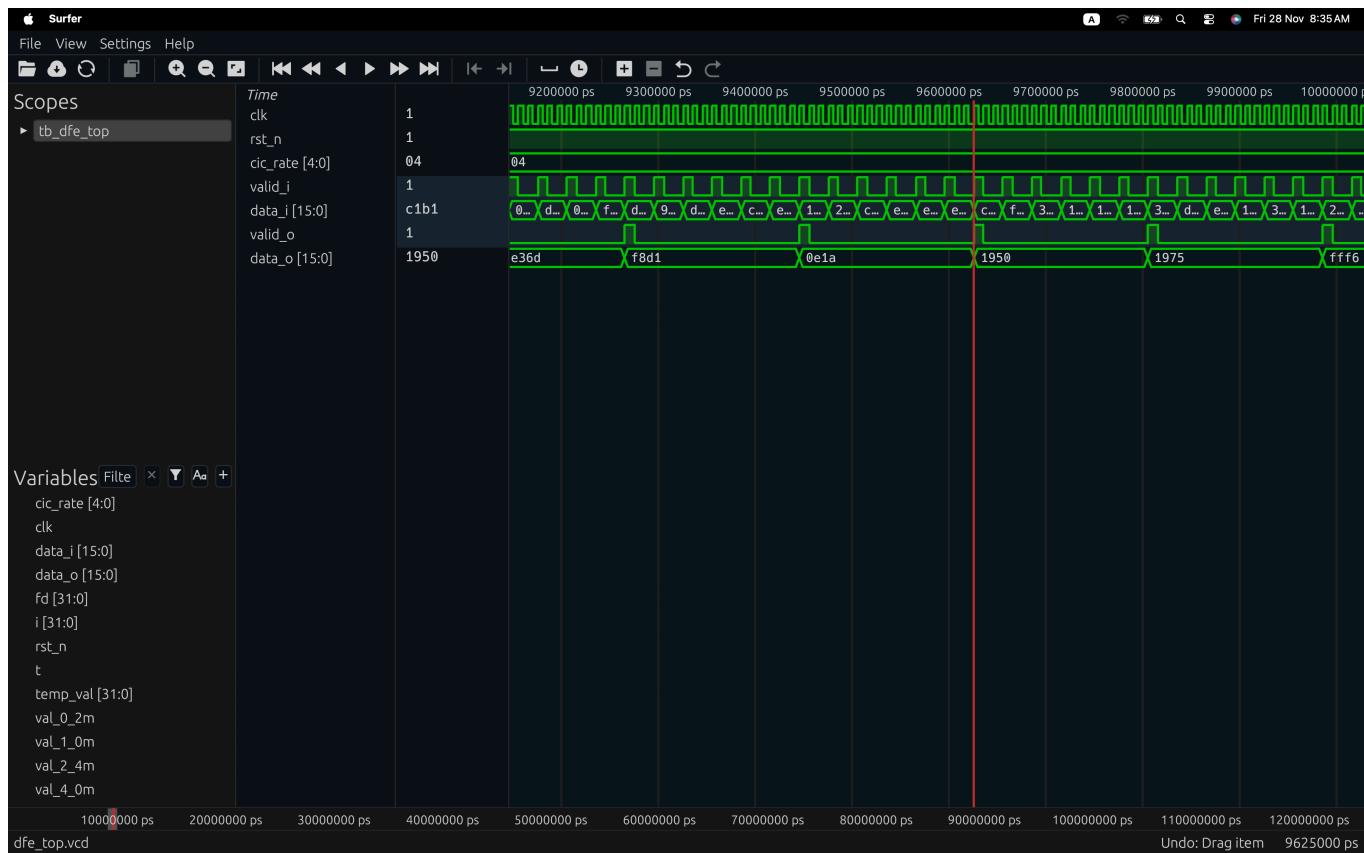
hazemysr@MacPro ~/Desktop/projects/DFE/DFE_top_level % make view_dfe
--- Running DFE Simulation ---

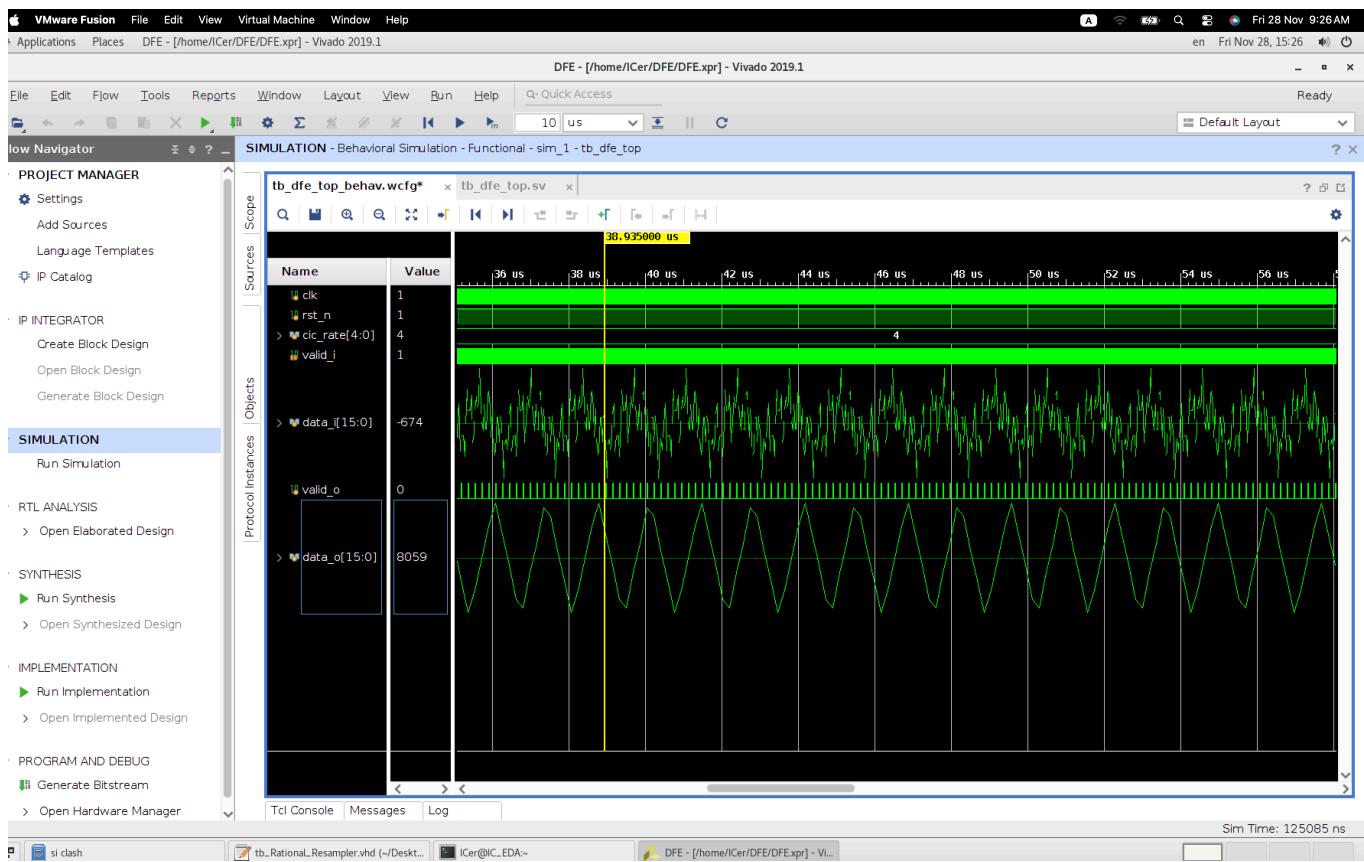
```

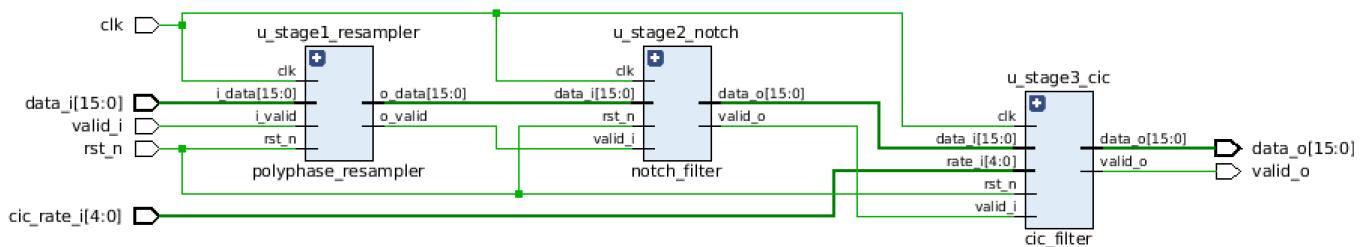
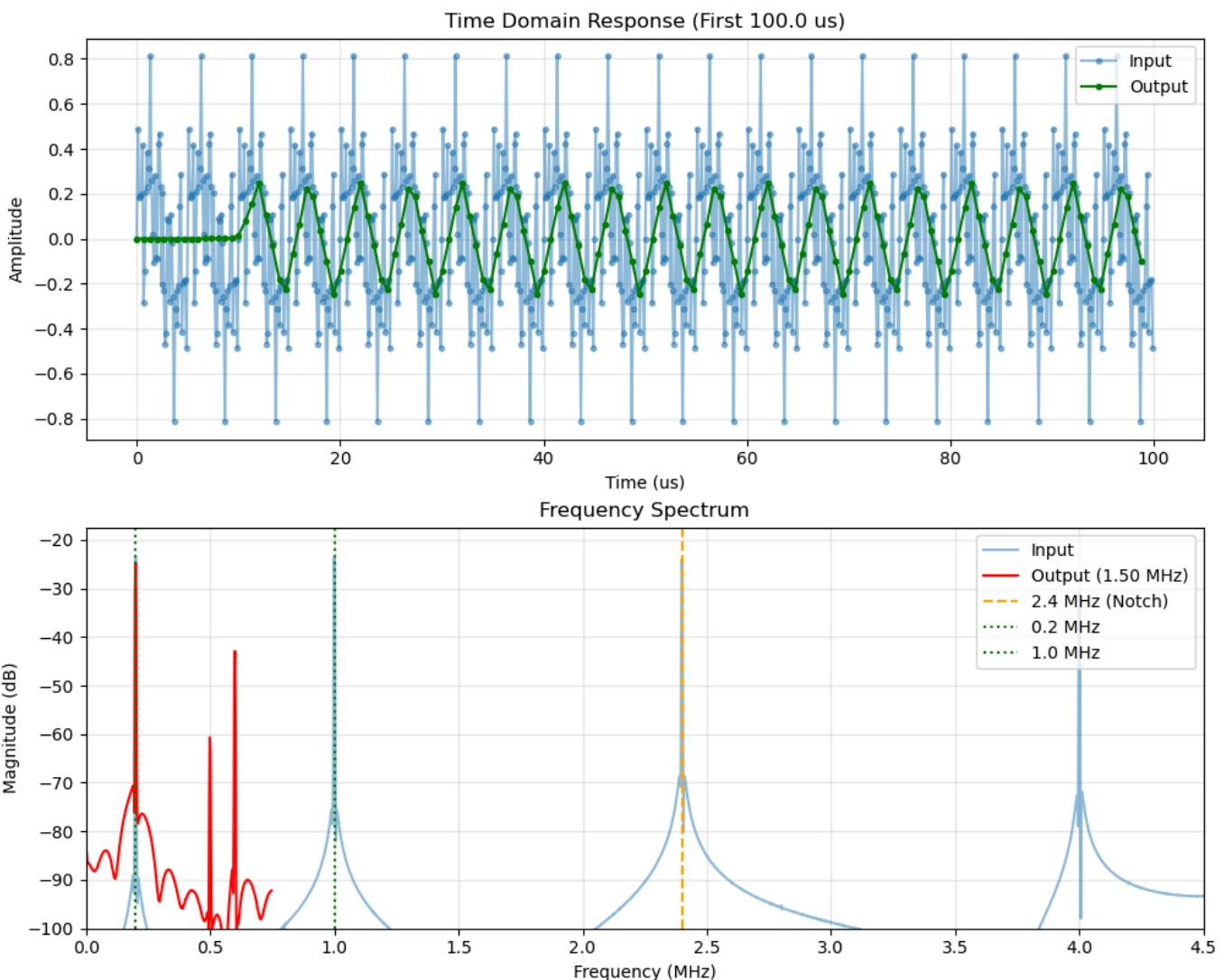
```

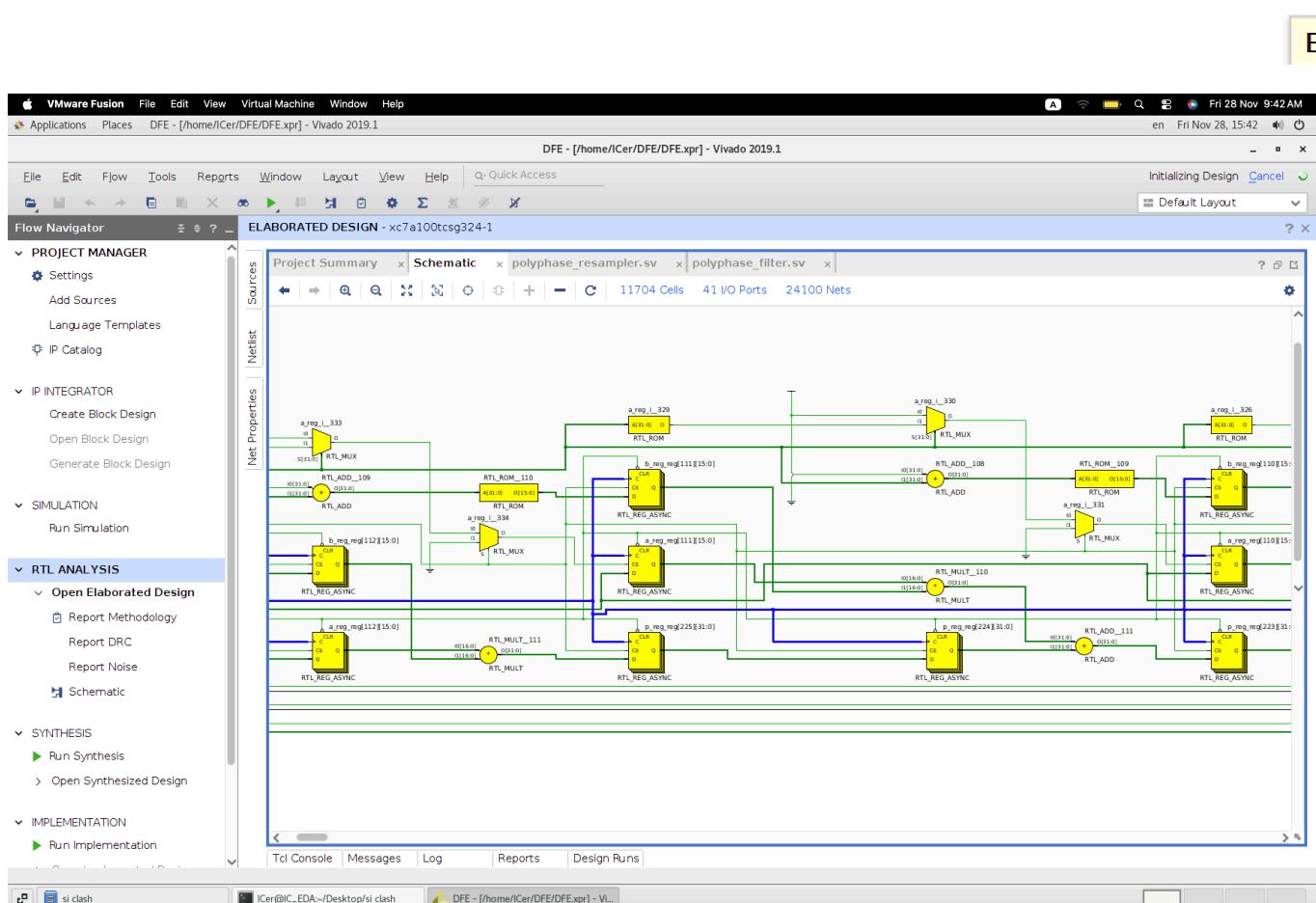
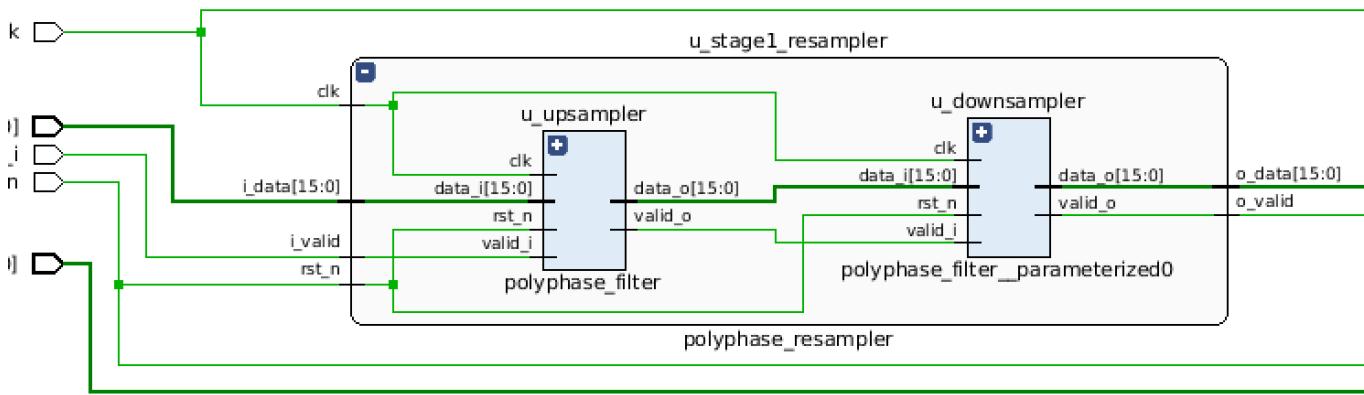
vvp dfe.vvp
VCD info: dumpfile dfe_top.vcd opened for output.
Starting DFE Simulation...
Input Fs: 9 MHz (Simulated Rate)
Simulation Finished.
tb_dfe_top.sv:112: $finish called at 125085000 (1ps)
---- Opening Waveform ----

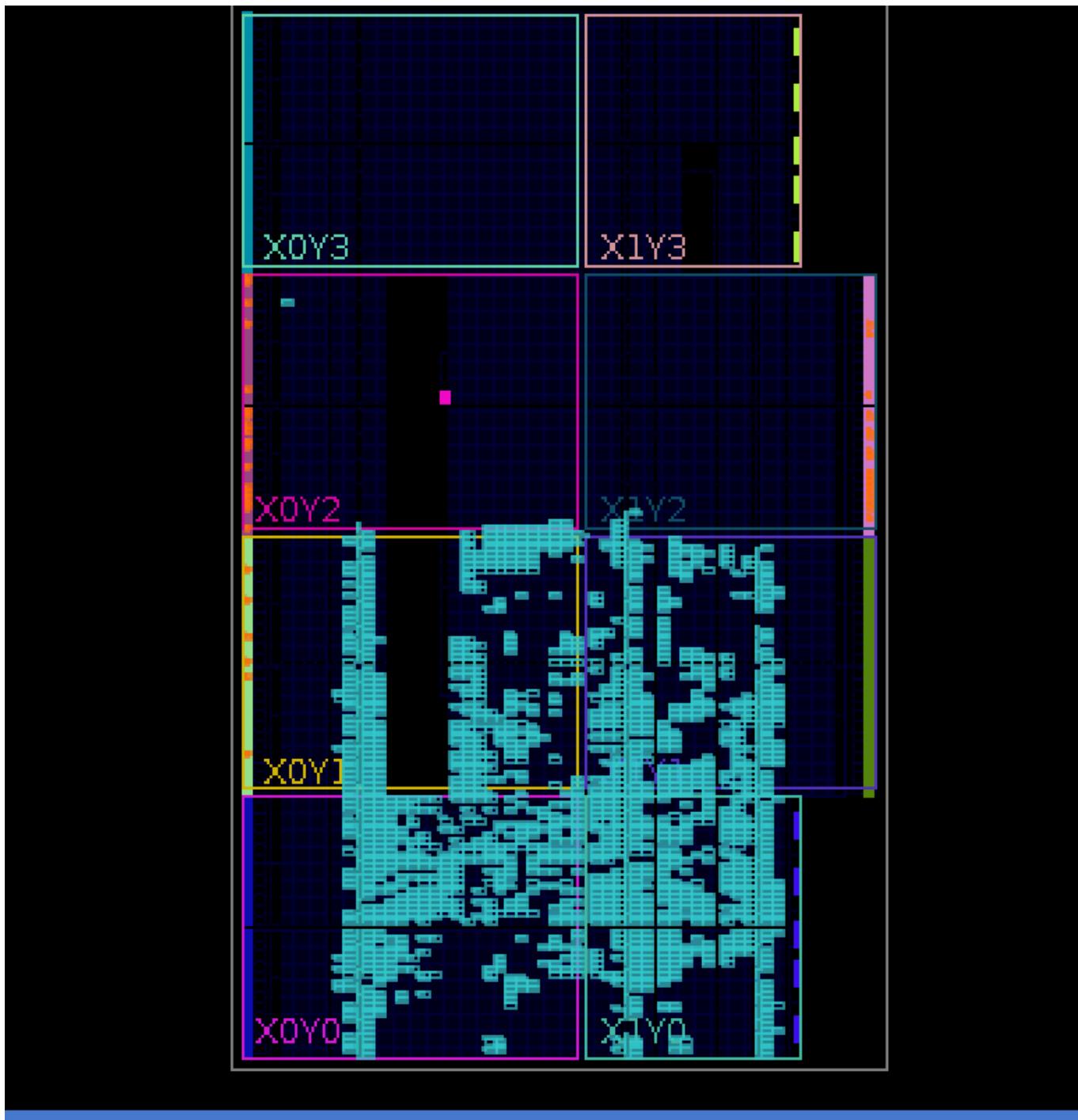
```



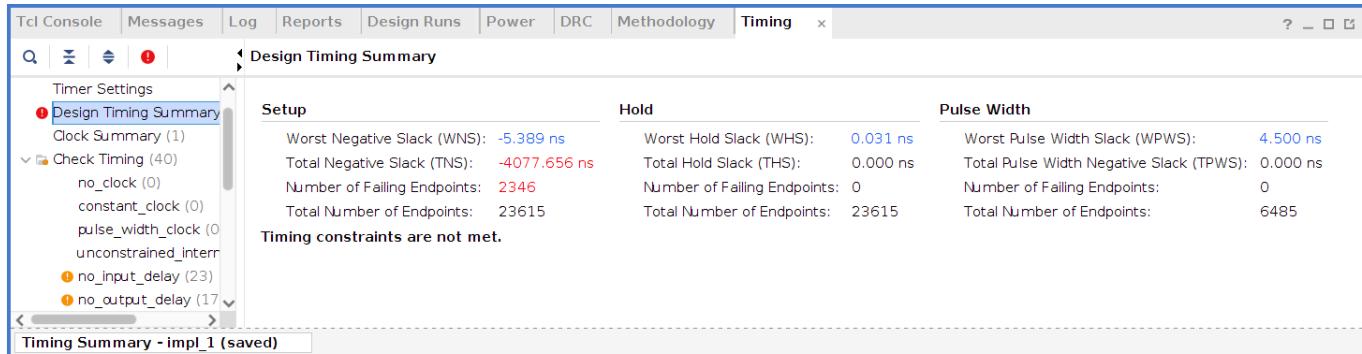








before NOTCH pipelining



after pipelining to further solve it you can run DFE at 50 MHz it won't cause any issue neither speed nor functionality it will even have lower power

Tcl Console | Messages | Log | Reports | Design Runs | DRC | Methodology | Power | Timing x ? - □ □

Design Timing Summary

Setup			Hold		Pulse Width	
Worst Negative Slack (WNS): -0.995 ns	Total Negative Slack (TNS): -457.866 ns	Number of Failing Endpoints: 1248	Worst Hold Slack (WHS): 0.042 ns	Total Hold Slack (THS): 0.000 ns	Number of Failing Endpoints: 0	Worst Pulse Width Slack (WPWS): 4.500 ns
Total Number of Endpoints: 23882	Timing constraints are not met.		Total Number of Endpoints: 23882	Total Pulse Width Negative Slack (TPWS): 0.000 ns	Total Number of Endpoints: 0	Total Number of Endpoints: 6559

Tcl Console | Messages | Log | Reports | Design Runs x Power | DRC | Methodology | Timing | ? - □ □

Summary

Settings

Summary (0.423 W, Margin)

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

Power Supply

- Utilization Details
 - Hierarchical (0.325 W)
 - Clocks (0.019 W)
 - Signals (0.09 W)
 - Data (0.086 W)
 - Clock Enable (0.004 W)
 - Set/Reset (0 W)
 - Logic (0.074 W)
 - DSP (0.128 W)
 - I/O (0.013 W)

Total On-Chip Power: 0.423 W

Design Power Budget: Not Specified

Power Budget Margin: N/A

Junction Temperature: 26.9°C

Thermal Margin: 58.1°C (12.6 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

Category	Power (W)	Percentage
Dynamic	0.325 W	77%
Clocks	0.019 W	(6%)
Signals	0.090 W	(28%)
Logic	0.074 W	(23%)
DSP	0.128 W	(39%)
I/O	0.013 W	(4%)

Effective thermal resistance. User selected package, airflow, heatsink and board characteristics are used with characterization and simulation to calculate.

impl_1 (saved)