INDIAN INSTITUTE OF TECHNOLOGY, ROPAR

PROJECT PRESENTATION (CP-302)

A Configurable Direct Digital Frequency Synthesizer Based on LUT and Rotation: Reproduction and Custom Hardware Implementation Using Verilog and Python-Based Simulation

SUBMITTED TO :	SUBMITTED BY :
Dr. Devarshi Das	Amitoj Singh
Dr. Neeraj Goel	2022EEB1295

Abstract

- Reproduced the DDFS architecture from IEEE TCAS-I 2019.
- Implemented using Verilog HDL with modular hierarchy.
- Simulated sinusoidal outputs using Python.
- Verified performance metrics: frequency output, SFDR.

Introduction

DDFS: Key for radar, comms, and agile RF systems.

Paper proposes LUT + Rotation (LUT-ROT) based DDFS.

Highlights:

- Multi-Bit Rotation Tree (MBRT)
- Mode-configurable operation
- High SFDR, low power

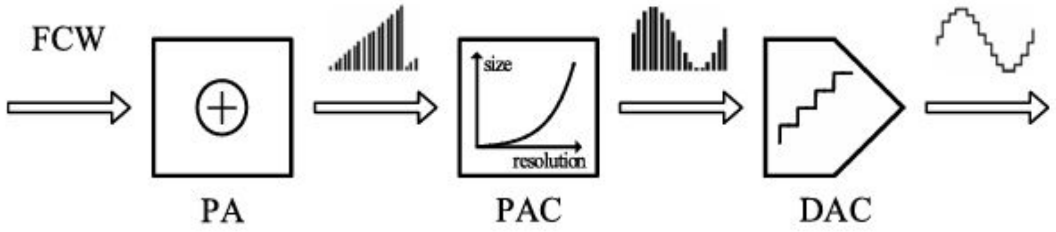


Fig. 1 - Conventional DDFS architecture [1]

Reference paper summary

Hybrid LUT + Rotation: Reduced ROM + latency.

MBRT: Low power, short pipeline.

Modes: Tunable trade-offs (speed vs power).

Achieved:

- 102-dBc SFDR
- 2.2-GHz frequency
- 6.9 mW/GHz efficiency

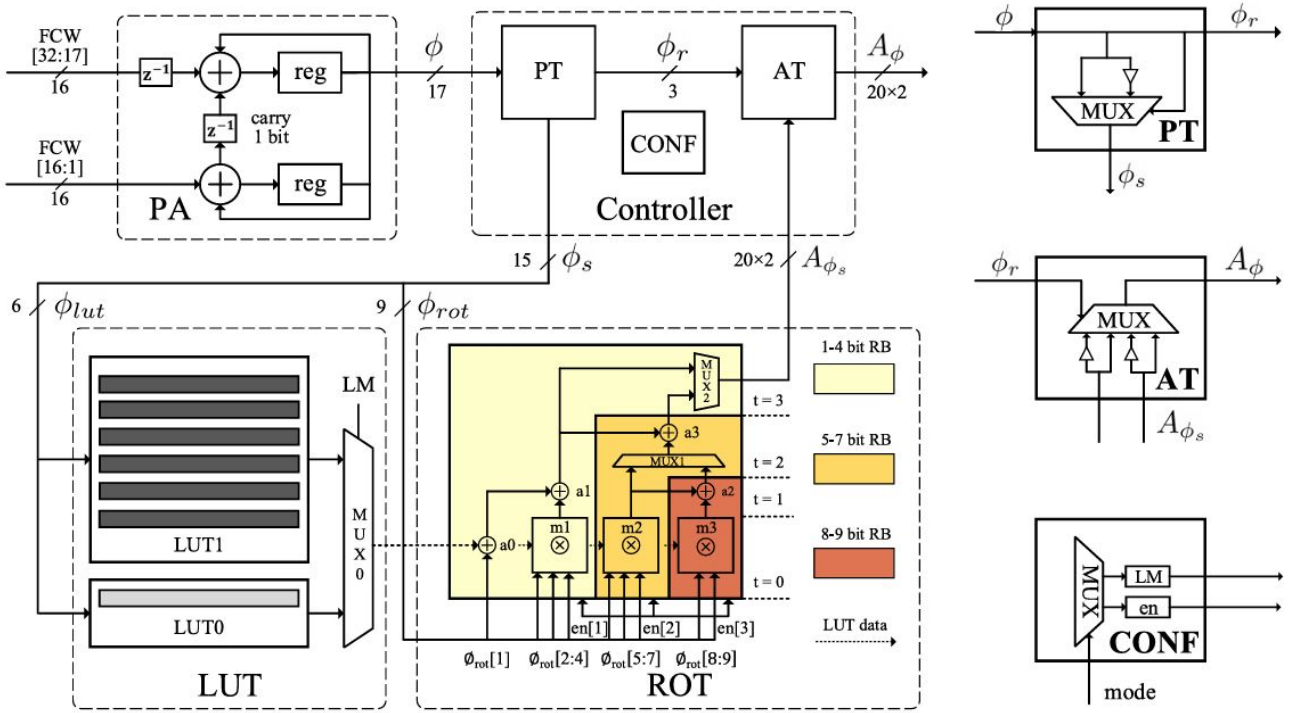
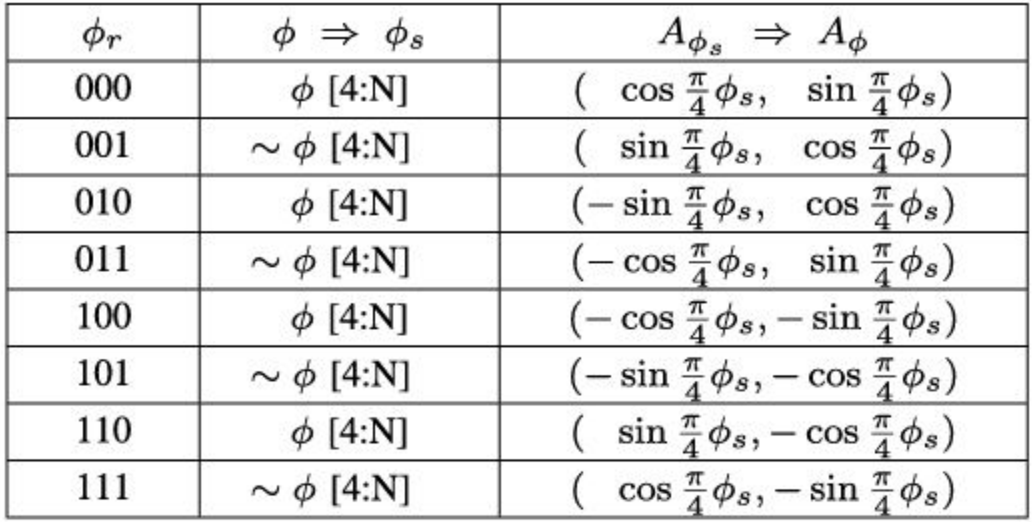


Fig. 2 - Final proposed circuit implementations [1]



ϕ_r	$\phi \Rightarrow \phi_s$	$A_{\phi_s} \Rightarrow A_\phi$
000	ϕ [4:N]	$(\cos\frac{\pi}{4}\phi_s, \sin\frac{\pi}{4}\phi_s)$
001	$\sim\phi$ [4:N]	$(\sin\frac{\pi}{4}\phi_s, \cos\frac{\pi}{4}\phi_s)$
010	ϕ [4:N]	$(-\sin\frac{\pi}{4}\phi_s, \cos\frac{\pi}{4}\phi_s)$
011	$\sim\phi$ [4:N]	$(-\cos\frac{\pi}{4}\phi_s, \sin\frac{\pi}{4}\phi_s)$
100	ϕ [4:N]	$(-\cos\frac{\pi}{4}\phi_s, -\sin\frac{\pi}{4}\phi_s)$
101	$\sim\phi$ [4:N]	$(-\sin\frac{\pi}{4}\phi_s, -\cos\frac{\pi}{4}\phi_s)$
110	ϕ [4:N]	$(\sin\frac{\pi}{4}\phi_s, -\cos\frac{\pi}{4}\phi_s)$
111	$\sim\phi$ [4:N]	$(\cos\frac{\pi}{4}\phi_s, -\sin\frac{\pi}{4}\phi_s)$

Fig. 3 - Phase and Amplitude transformations [1]

Verilog Implementation Review

PA_16bit.v: 32-bit phase accumulator (pipelined).

lut.v: ROM with xc, yc, xp, yp values (Python-generated).

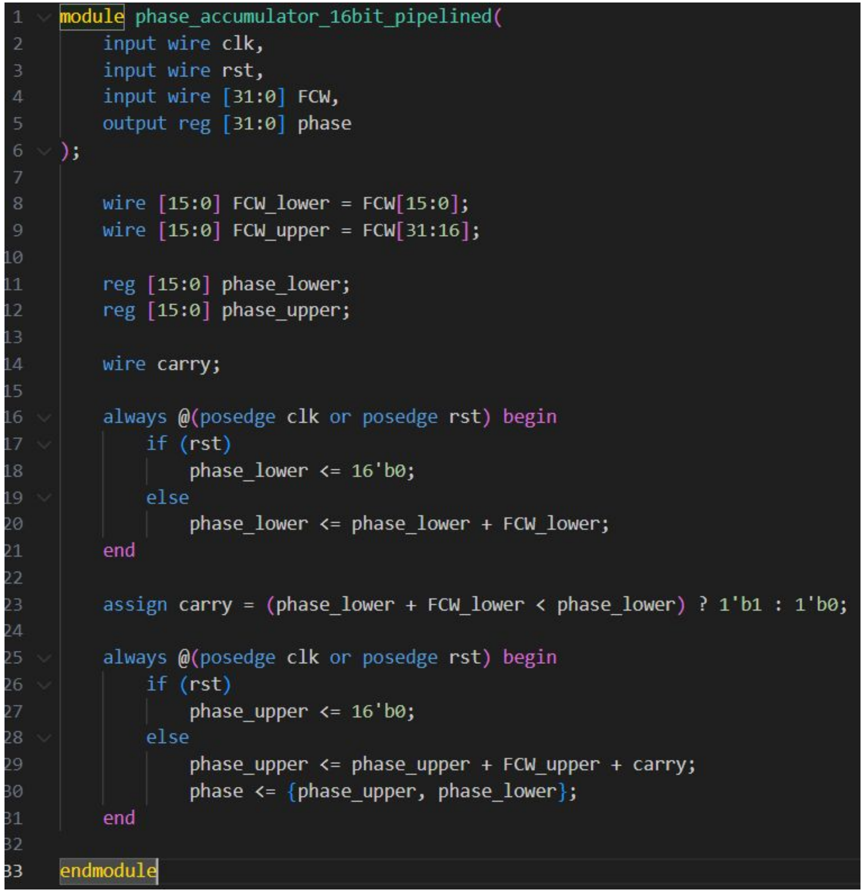
rot_new.v: MBRT rotation logic.

PT_transformed.v / at.v: π/4 symmetry transforms.

conf.v: Mode and block enable controls.

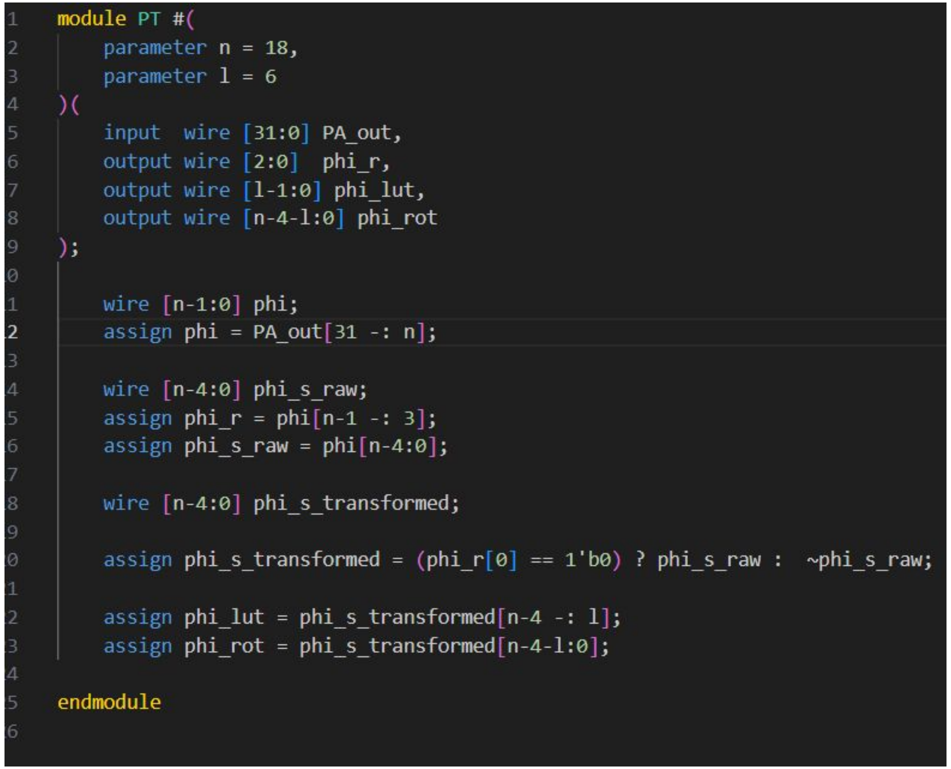
top.v / top_tb.v: System integration and testbench.

PHASE ACCUMULATOR



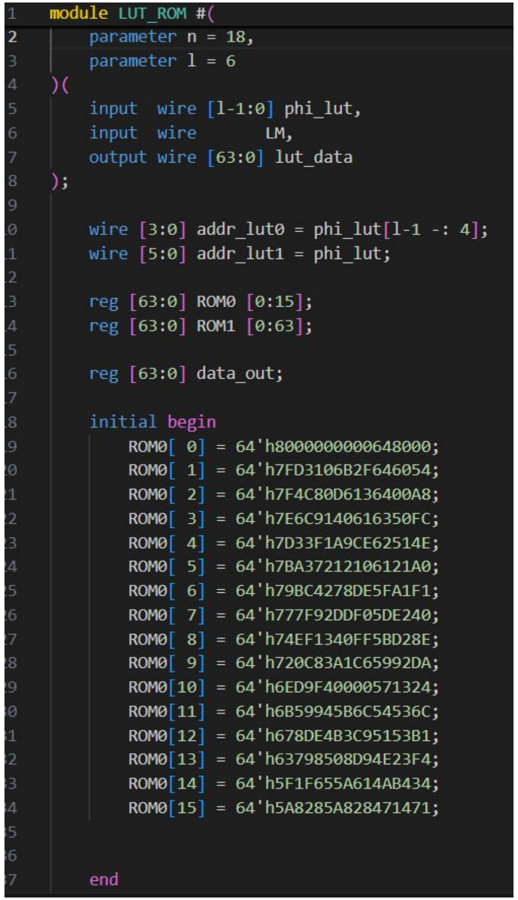
```
module phase_accumulator_16bit_pipelined(
    input wire clk,
    input wire rst,
    input wire [31:0] FCW,
    output reg [31:0] phase
);

    wire [15:0] FCW_lower = FCW[15:0];
    wire [15:0] FCW_upper = FCW[31:16];

    reg [15:0] phase_lower;
    reg [15:0] phase_upper;

    wire carry;

    always @(posedge clk or posedge rst) begin
        if (rst)
            phase_lower <= 16'b0;
        else
            phase_lower <= phase_lower + FCW_lower;
    end

    assign carry = (phase_lower + FCW_lower < phase_lower) ? 1'b1 : 1'b0;

    always @(posedge clk or posedge rst) begin
        if (rst)
            phase_upper <= 16'b0;
        else
            phase_upper <= phase_upper + FCW_upper + carry;
            phase <= {phase_upper, phase_lower};
    end

endmodule
```

PHASE TRANSFORMER

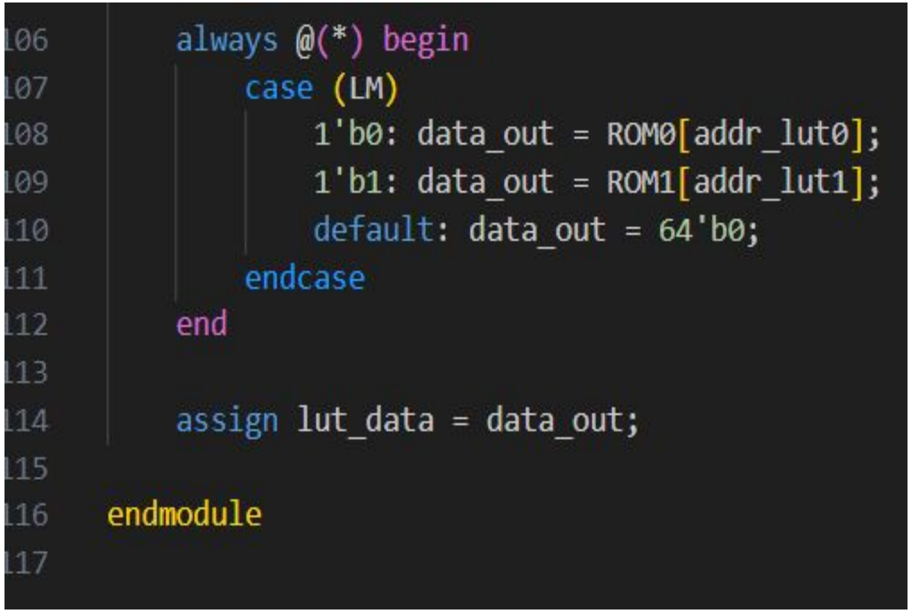


```
module PT #(
    parameter n = 18,
    parameter l = 6
)(
    input  wire [31:0] PA_out,
    output wire [2:0]  phi_r,
    output wire [l-1:0] phi_lut,
    output wire [n-4-l:0] phi_rot
);

    wire [n-1:0] phi;
    assign phi = PA_out[31 -: n];

    wire [n-4:0] phi_s_raw;
    assign phi_r = phi[n-1 -: 3];
    assign phi_s_raw = phi[n-4:0];

    wire [n-4:0] phi_s_transformed;

    assign phi_s_transformed = (phi_r[0] == 1'b0) ? phi_s_raw :  ~phi_s_raw;

    assign phi_lut = phi_s_transformed[n-4 -: l];
    assign phi_rot = phi_s_transformed[n-4-l:0];

endmodule
```

LUT

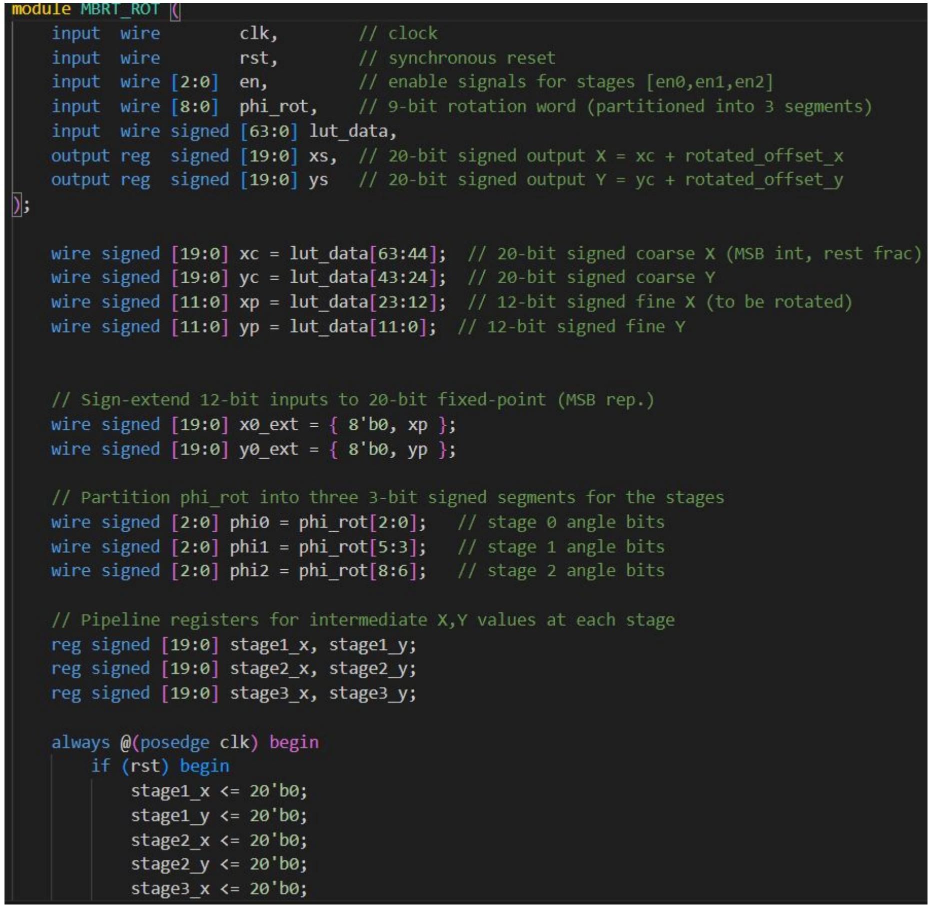


```
module LUT_ROM #(
    parameter n = 18,
    parameter l = 6
)(
    input  wire [l-1:0] phi_lut,
    input  wire         LM,
    output wire [63:0] lut_data
);

    wire [3:0] addr_lut0 = phi_lut[l-1 -: 4];
    wire [5:0] addr_lut1 = phi_lut;

    reg [63:0] ROM0 [0:15];
    reg [63:0] ROM1 [0:63];

    reg [63:0] data_out;

    initial begin
        ROM0[ 0] = 64'h8000000000648000;
        ROM0[ 1] = 64'h7FD3106B2F646054;
        ROM0[ 2] = 64'h7F4C80D6136400A8;
        ROM0[ 3] = 64'h7E6C9140616350FC;
        ROM0[ 4] = 64'h7D33F1A9CE62514E;
        ROM0[ 5] = 64'h7BA37212106121A0;
        ROM0[ 6] = 64'h79BC4278DE5FA1F1;
        ROM0[ 7] = 64'h777F92DDF05DE240;
        ROM0[ 8] = 64'h74EF1340FF5BD28E;
        ROM0[ 9] = 64'h720C83A1C65992DA;
        ROM0[10] = 64'h6ED9F40000571324;
        ROM0[11] = 64'h6B59945B6C54536C;
        ROM0[12] = 64'h678DE4B3C95153B1;
        ROM0[13] = 64'h63798508D94E23F4;
        ROM0[14] = 64'h5F1F655A614AB434;
        ROM0[15] = 64'h5A8285A828471471;


    end
```

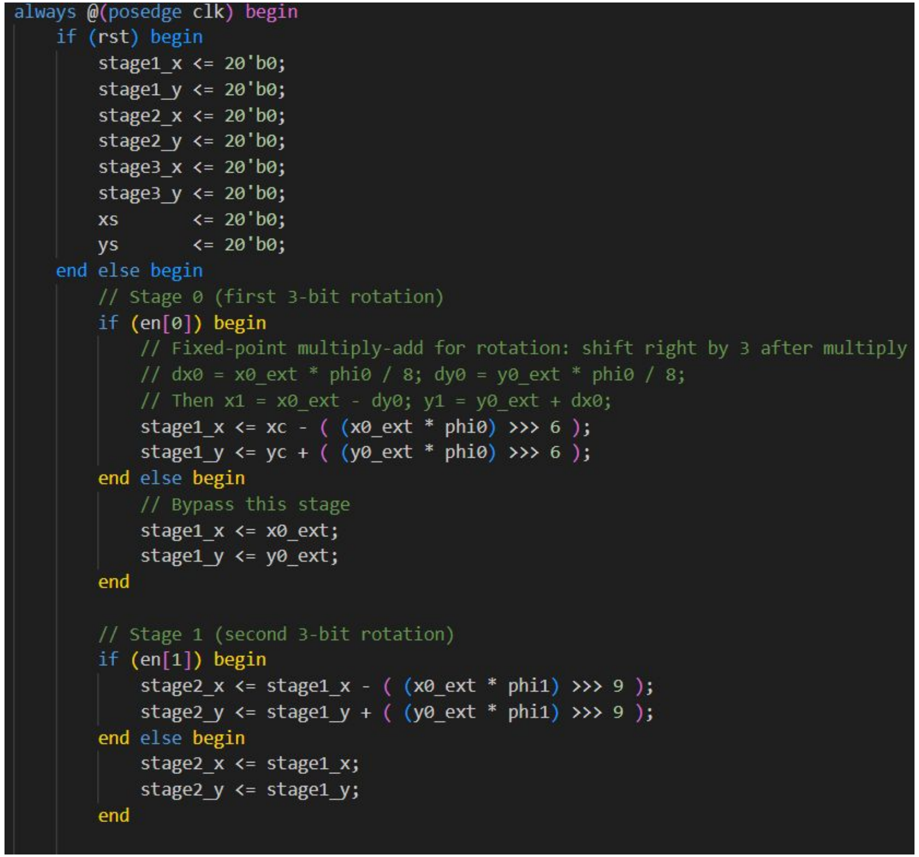


```
    always @(*) begin
        case (LM)
            1'b0: data_out = ROM0[addr_lut0];
            1'b1: data_out = ROM1[addr_lut1];
            default: data_out = 64'b0;
        endcase
    end

    assign lut_data = data_out;

endmodule

```

ROT

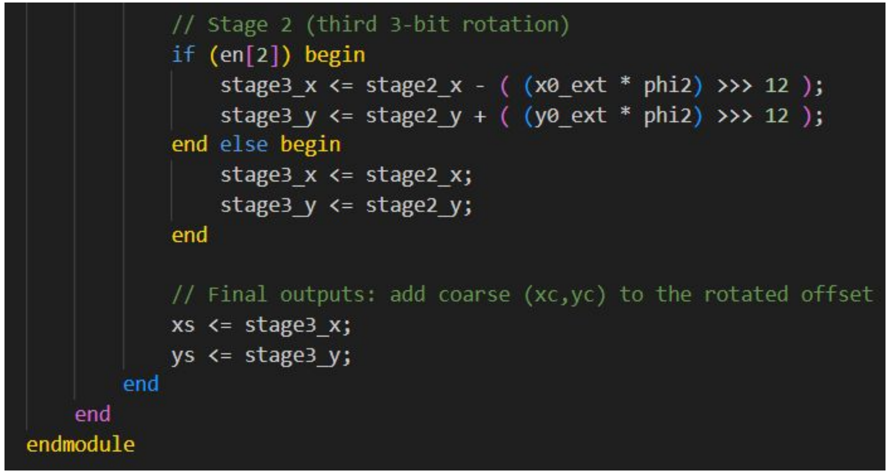


```
module MBRT_ROT (
    input  wire        clk,        // clock
    input  wire        rst,        // synchronous reset
    input  wire [2:0]  en,         // enable signals for stages [en0,en1,en2]
    input  wire [8:0]  phi_rot,    // 9-bit rotation word (partitioned into 3 segments)
    input  wire signed [63:0] lut_data,
    output reg  signed [19:0] xs,  // 20-bit signed output X = xc + rotated_offset_x
    output reg  signed [19:0] ys   // 20-bit signed output Y = yc + rotated_offset_y
);


    wire signed [19:0] xc = lut_data[63:44];  // 20-bit signed coarse X (MSB int, rest frac)
    wire signed [19:0] yc = lut_data[43:24];  // 20-bit signed coarse Y
    wire signed [11:0] xp = lut_data[23:12];  // 12-bit signed fine X (to be rotated)
    wire signed [11:0] yp = lut_data[11:0];  // 12-bit signed fine Y


    // Sign-extend 12-bit inputs to 20-bit fixed-point (MSB rep.)
    wire signed [19:0] x0_ext = { 8'b0, xp };
    wire signed [19:0] y0_ext = { 8'b0, yp };

    // Partition phi_rot into three 3-bit signed segments for the stages
    wire signed [2:0] phi0 = phi_rot[2:0];   // stage 0 angle bits
    wire signed [2:0] phi1 = phi_rot[5:3];   // stage 1 angle bits
    wire signed [2:0] phi2 = phi_rot[8:6];   // stage 2 angle bits

    // Pipeline registers for intermediate X,Y values at each stage
    reg signed [19:0] stage1_x, stage1_y;
    reg signed [19:0] stage2_x, stage2_y;
    reg signed [19:0] stage3_x, stage3_y;

    always @(posedge clk) begin
        if (rst) begin
            stage1_x <= 20'b0;
            stage1_y <= 20'b0;
            stage2_x <= 20'b0;
            stage2_y <= 20'b0;
            stage3_x <= 20'b0;
```



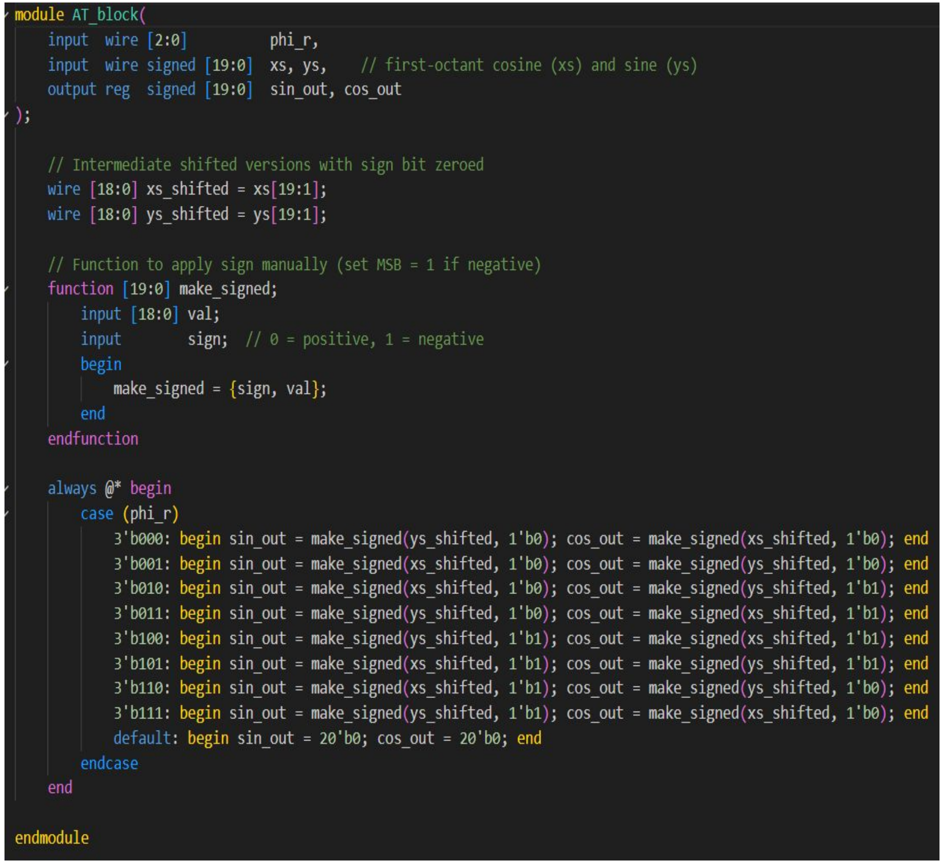
```
always @(posedge clk) begin
    if (rst) begin
        stage1_x <= 20'b0;
        stage1_y <= 20'b0;
        stage2_x <= 20'b0;
        stage2_y <= 20'b0;
        stage3_x <= 20'b0;
        stage3_y <= 20'b0;
        xs       <= 20'b0;
        ys       <= 20'b0;
    end else begin
        // Stage 0 (first 3-bit rotation)
        if (en[0]) begin
            // Fixed-point multiply-add for rotation: shift right by 3 after multiply
            // dx0 = x0_ext * phi0 / 8; dy0 = y0_ext * phi0 / 8;
            // Then x1 = x0_ext - dy0; y1 = y0_ext + dx0;
            stage1_x <= xc - ( (x0_ext * phi0) >>> 6 );
            stage1_y <= yc + ( (y0_ext * phi0) >>> 6 );
        end else begin
            // Bypass this stage
            stage1_x <= x0_ext;
            stage1_y <= y0_ext;
        end

        // Stage 1 (second 3-bit rotation)
        if (en[1]) begin
            stage2_x <= stage1_x - ( (x0_ext * phi1) >>> 9 );
            stage2_y <= stage1_y + ( (y0_ext * phi1) >>> 9 );
        end else begin
            stage2_x <= stage1_x;
            stage2_y <= stage1_y;
        end
```

ROT



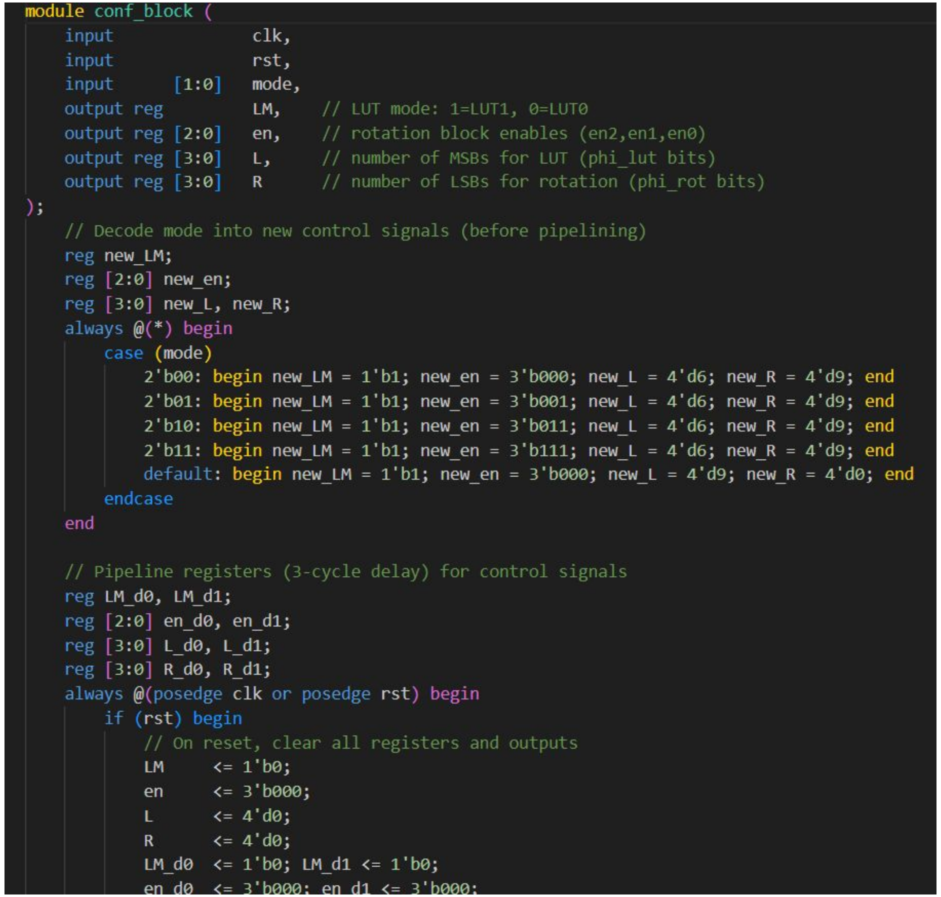
```
            // Stage 2 (third 3-bit rotation)
            if (en[2]) begin
                stage3_x <= stage2_x - ( (x0_ext * phi2) >>> 12 );
                stage3_y <= stage2_y + ( (y0_ext * phi2) >>> 12 );
            end else begin
                stage3_x <= stage2_x;
                stage3_y <= stage2_y;
            end

            // Final outputs: add coarse (xc,yc) to the rotated offset
            xs <= stage3_x;
            ys <= stage3_y;
        end
    end
endmodule
```

AMPLITUDE TRANSFORMER

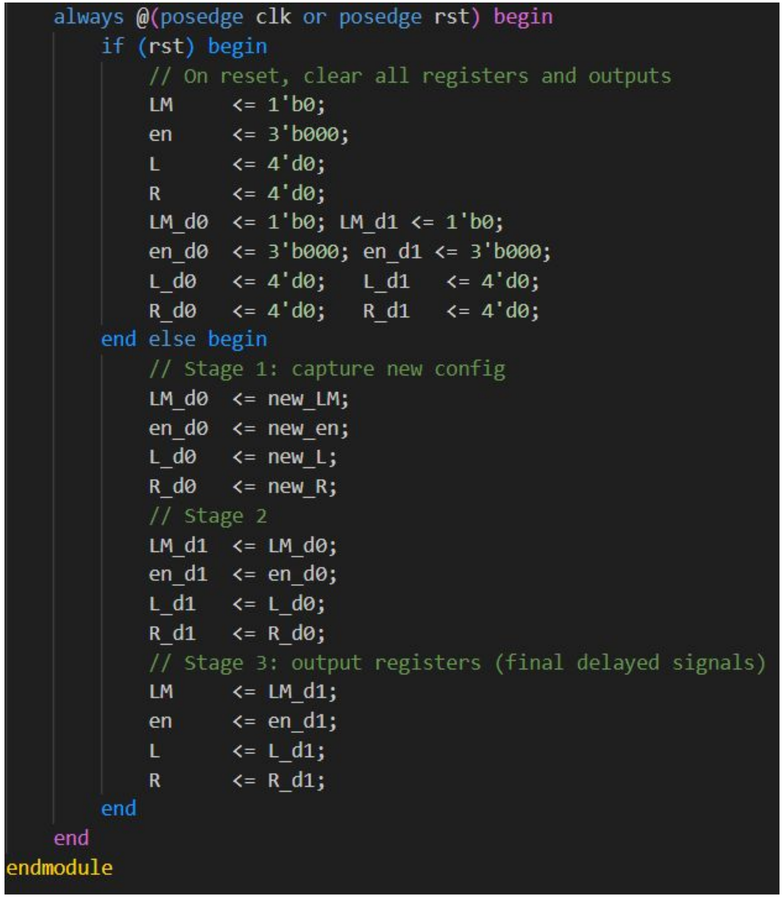


```
module AT_block(
    input  wire [2:0]         phi_r,
    input  wire signed [19:0] xs, ys,    // first-octant cosine (xs) and sine (ys)
    output reg  signed [19:0] sin_out, cos_out
);

    // Intermediate shifted versions with sign bit zeroed
    wire [18:0] xs_shifted = xs[19:1];
    wire [18:0] ys_shifted = ys[19:1];

    // Function to apply sign manually (set MSB = 1 if negative)
    function [19:0] make_signed;
        input [18:0] val;
        input        sign;  // 0 = positive, 1 = negative
        begin
            make_signed = {sign, val};
        end
    endfunction

    always @* begin
        case (phi_r)
            3'b000: begin sin_out = make_signed(ys_shifted, 1'b0); cos_out = make_signed(xs_shifted, 1'b0); end
            3'b001: begin sin_out = make_signed(xs_shifted, 1'b0); cos_out = make_signed(ys_shifted, 1'b0); end
            3'b010: begin sin_out = make_signed(xs_shifted, 1'b0); cos_out = make_signed(ys_shifted, 1'b1); end
            3'b011: begin sin_out = make_signed(ys_shifted, 1'b0); cos_out = make_signed(xs_shifted, 1'b1); end
            3'b100: begin sin_out = make_signed(ys_shifted, 1'b1); cos_out = make_signed(xs_shifted, 1'b1); end
            3'b101: begin sin_out = make_signed(xs_shifted, 1'b1); cos_out = make_signed(ys_shifted, 1'b1); end
            3'b110: begin sin_out = make_signed(xs_shifted, 1'b1); cos_out = make_signed(ys_shifted, 1'b0); end
            3'b111: begin sin_out = make_signed(ys_shifted, 1'b1); cos_out = make_signed(xs_shifted, 1'b0); end
            default: begin sin_out = 20'b0; cos_out = 20'b0; end
        endcase
    end

endmodule
```

CONF

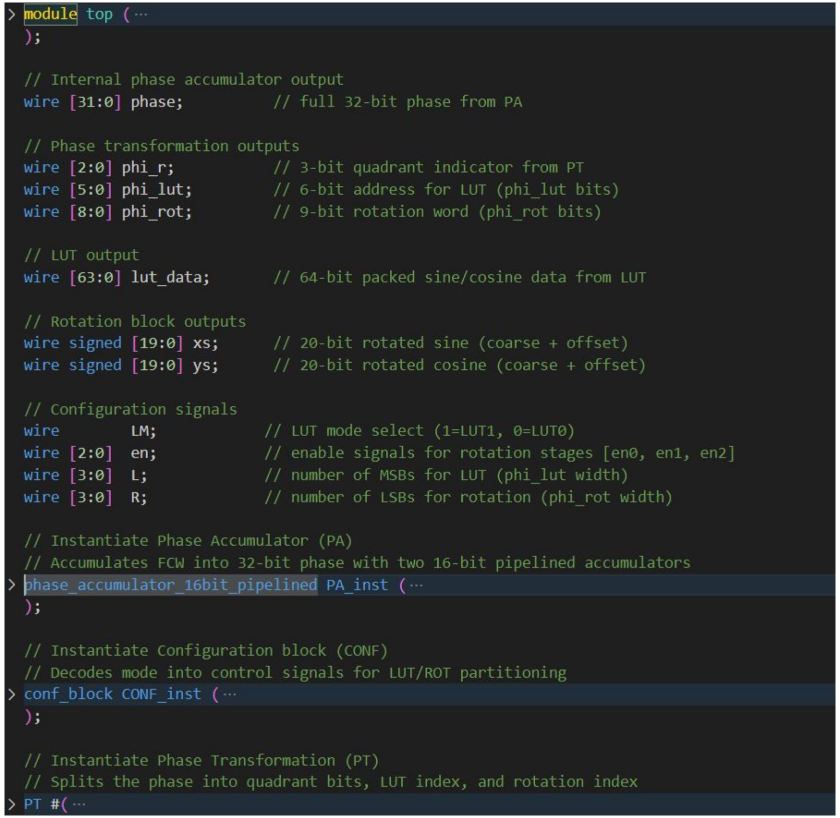


```verilog
module conf_block (
    input              clk,
    input              rst,
    input       [1:0]  mode,
    output reg         LM,    // LUT mode: 1=LUT1, 0=LUT0
    output reg [2:0]   en,    // rotation block enables (en2,en1,en0)
    output reg [3:0]   L,     // number of MSBs for LUT (phi_lut bits)
    output reg [3:0]   R      // number of LSBs for rotation (phi_rot bits)
);
    // Decode mode into new control signals (before pipelining)
    reg new_LM;
    reg [2:0] new_en;
    reg [3:0] new_L, new_R;
    always @(*) begin
        case (mode)
            2'b00: begin new_LM = 1'b1; new_en = 3'b000; new_L = 4'd6; new_R = 4'd9; end
            2'b01: begin new_LM = 1'b1; new_en = 3'b001; new_L = 4'd6; new_R = 4'd9; end
            2'b10: begin new_LM = 1'b1; new_en = 3'b011; new_L = 4'd6; new_R = 4'd9; end
            2'b11: begin new_LM = 1'b1; new_en = 3'b111; new_L = 4'd6; new_R = 4'd9; end
            default: begin new_LM = 1'b1; new_en = 3'b000; new_L = 4'd9; new_R = 4'd0; end
        endcase
    end

    // Pipeline registers (3-cycle delay) for control signals
    reg LM_d0, LM_d1;
    reg [2:0] en_d0, en_d1;
    reg [3:0] L_d0, L_d1;
    reg [3:0] R_d0, R_d1;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            // On reset, clear all registers and outputs
            LM     <= 1'b0;
            en     <= 3'b000;
            L      <= 4'd0;
            R      <= 4'd0;
            LM_d0  <= 1'b0; LM_d1 <= 1'b0;
            en_d0  <= 3'b000; en_d1 <= 3'b000;
```

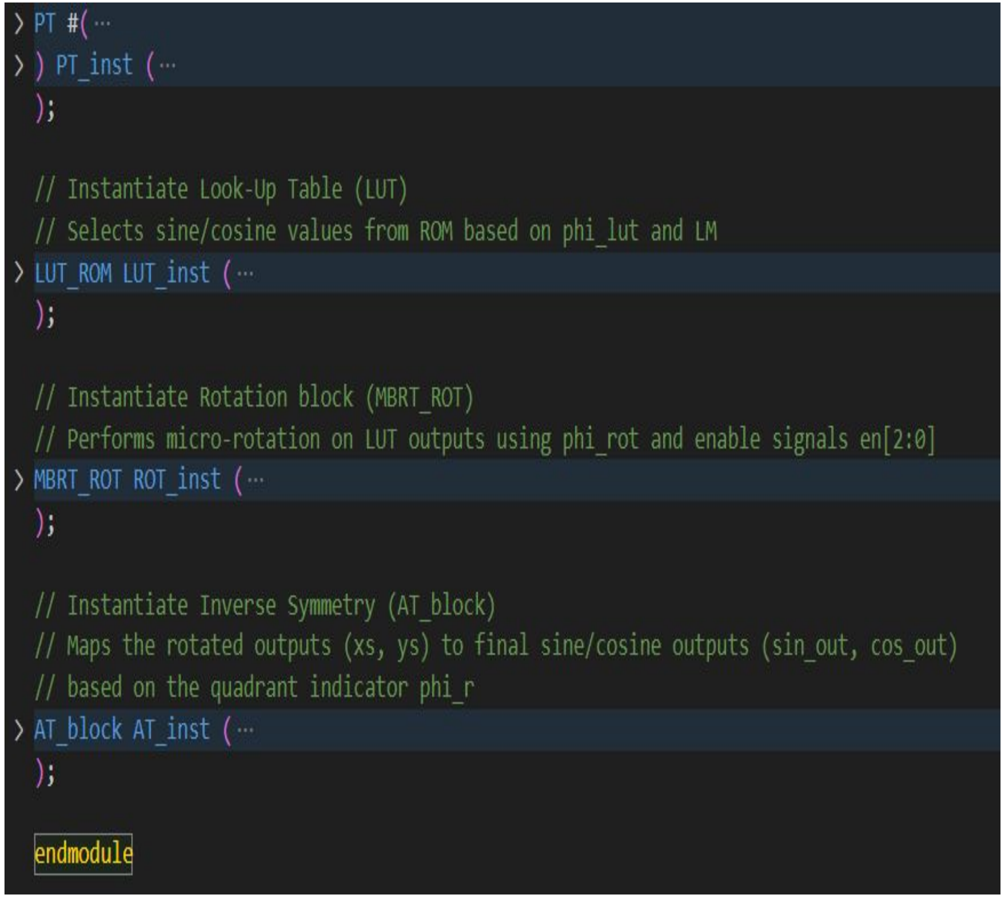


```verilog
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            // On reset, clear all registers and outputs
            LM     <= 1'b0;
            en     <= 3'b000;
            L      <= 4'd0;
            R      <= 4'd0;
            LM_d0  <= 1'b0; LM_d1 <= 1'b0;
            en_d0  <= 3'b000; en_d1 <= 3'b000;
            L_d0   <= 4'd0;   L_d1   <= 4'd0;
            R_d0   <= 4'd0;   R_d1   <= 4'd0;
        end else begin
            // Stage 1: capture new config
            LM_d0  <= new_LM;
            en_d0  <= new_en;
            L_d0   <= new_L;
            R_d0   <= new_R;
            // Stage 2
            LM_d1  <= LM_d0;
            en_d1  <= en_d0;
            L_d1   <= L_d0;
            R_d1   <= R_d0;
            // Stage 3: output registers (final delayed signals)
            LM     <= LM_d1;
            en     <= en_d1;
            L      <= L_d1;
            R      <= R_d1;
        end
    end
endmodule
```

TOP



```
> module top (···
);

// Internal phase accumulator output
wire [31:0] phase;          // full 32-bit phase from PA

// Phase transformation outputs
wire [2:0] phi_r;           // 3-bit quadrant indicator from PT
wire [5:0] phi_lut;         // 6-bit address for LUT (phi_lut bits)
wire [8:0] phi_rot;         // 9-bit rotation word (phi_rot bits)

// LUT output
wire [63:0] lut_data;       // 64-bit packed sine/cosine data from LUT

// Rotation block outputs
wire signed [19:0] xs;      // 20-bit rotated sine (coarse + offset)
wire signed [19:0] ys;      // 20-bit rotated cosine (coarse + offset)

// Configuration signals
wire        LM;            // LUT mode select (1=LUT1, 0=LUT0)
wire [2:0]  en;            // enable signals for rotation stages [en0, en1, en2]
wire [3:0]  L;             // number of MSBs for LUT (phi_lut width)
wire [3:0]  R;             // number of LSBs for rotation (phi_rot width)

// Instantiate Phase Accumulator (PA)
// Accumulates FCW into 32-bit phase with two 16-bit pipelined accumulators
> phase_accumulator_16bit_pipelined PA_inst (···
);

// Instantiate Configuration block (CONF)
// Decodes mode into control signals for LUT/ROT partitioning
> conf_block CONF_inst (···
);

// Instantiate Phase Transformation (PT)
// Splits the phase into quadrant bits, LUT index, and rotation index
> PT #(···
```



```
> PT #(···
> ) PT_inst (···
);

// Instantiate Look-Up Table (LUT)
// Selects sine/cosine values from ROM based on phi_lut and LM
> LUT_ROM LUT_inst (···
);

// Instantiate Rotation block (MBRT_ROT)
// Performs micro-rotation on LUT outputs using phi_rot and enable signals en[2:0]
> MBRT_ROT ROT_inst (···
);

// Instantiate Inverse Symmetry (AT_block)
// Maps the rotated outputs (xs, ys) to final sine/cosine outputs (sin_out, cos_out)
// based on the quadrant indicator phi_r
> AT_block AT_inst (···
);

endmodule
```

Python LUT generator

Generated 64 samples from 0 to $\pi/4$.

Computed:

- $xc = \cos(\theta)$, $yc = \sin(\theta)$
- $xp = (\pi/4)\cdot\cos(\theta)$, $yp = (\pi/4)\cdot\sin(\theta)$

Scaled to fixed-point:

- xc, yc $\rightarrow$ 20-bit
- xp, yp $\rightarrow$ 12-bit

Exported to ROM for Verilog LUT.

Simulation Results

FCW inputs tested using **top_tb.v**.

Python:

- Converts output to float
- Plots sine/cos waveforms and smoothens them
- Performs FFT using smoothed values

Output frequency matched predictions:

- $f\_out = (FCW \times f\_clk) / 2^n$

Link to codes - link

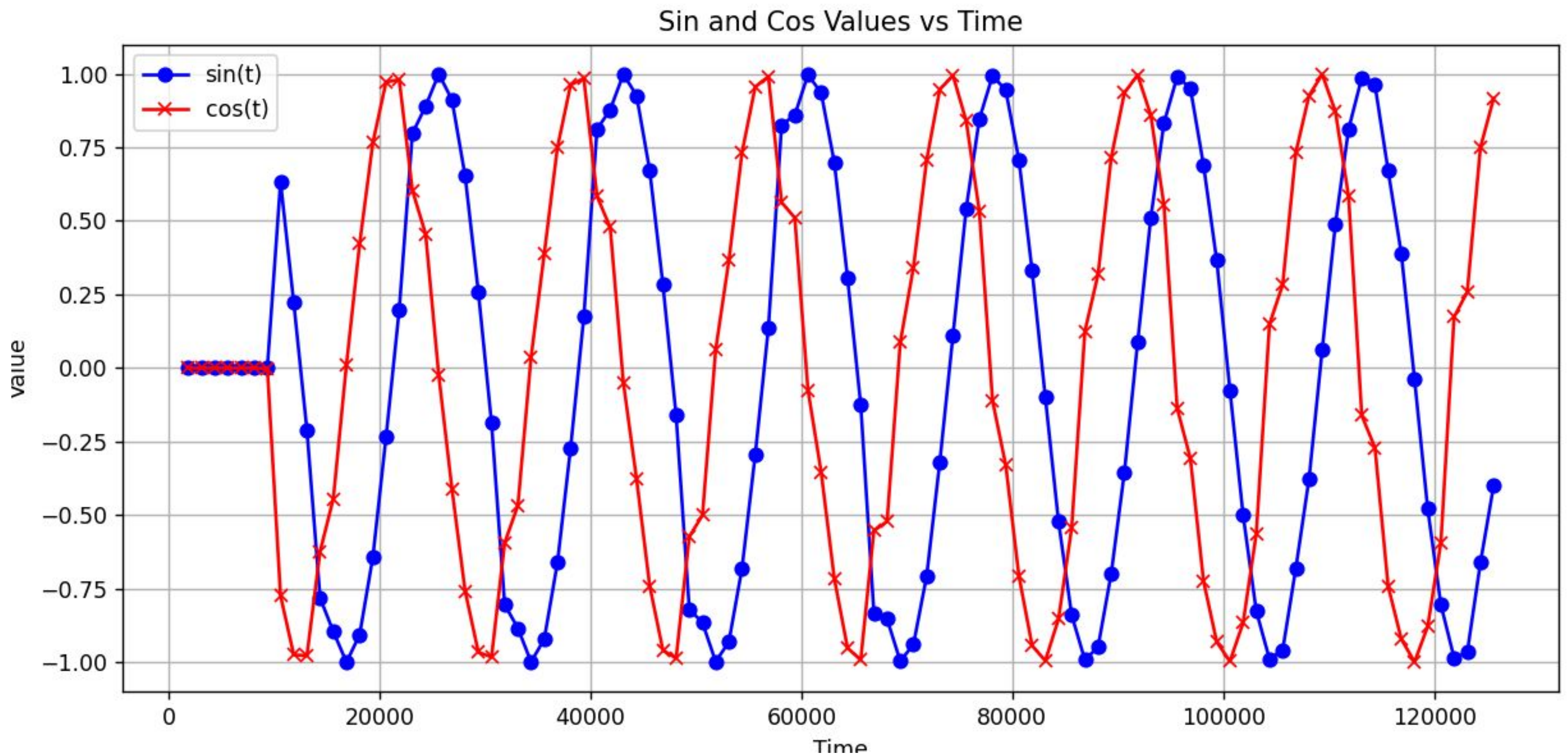


Fig. 4 - Output obtained

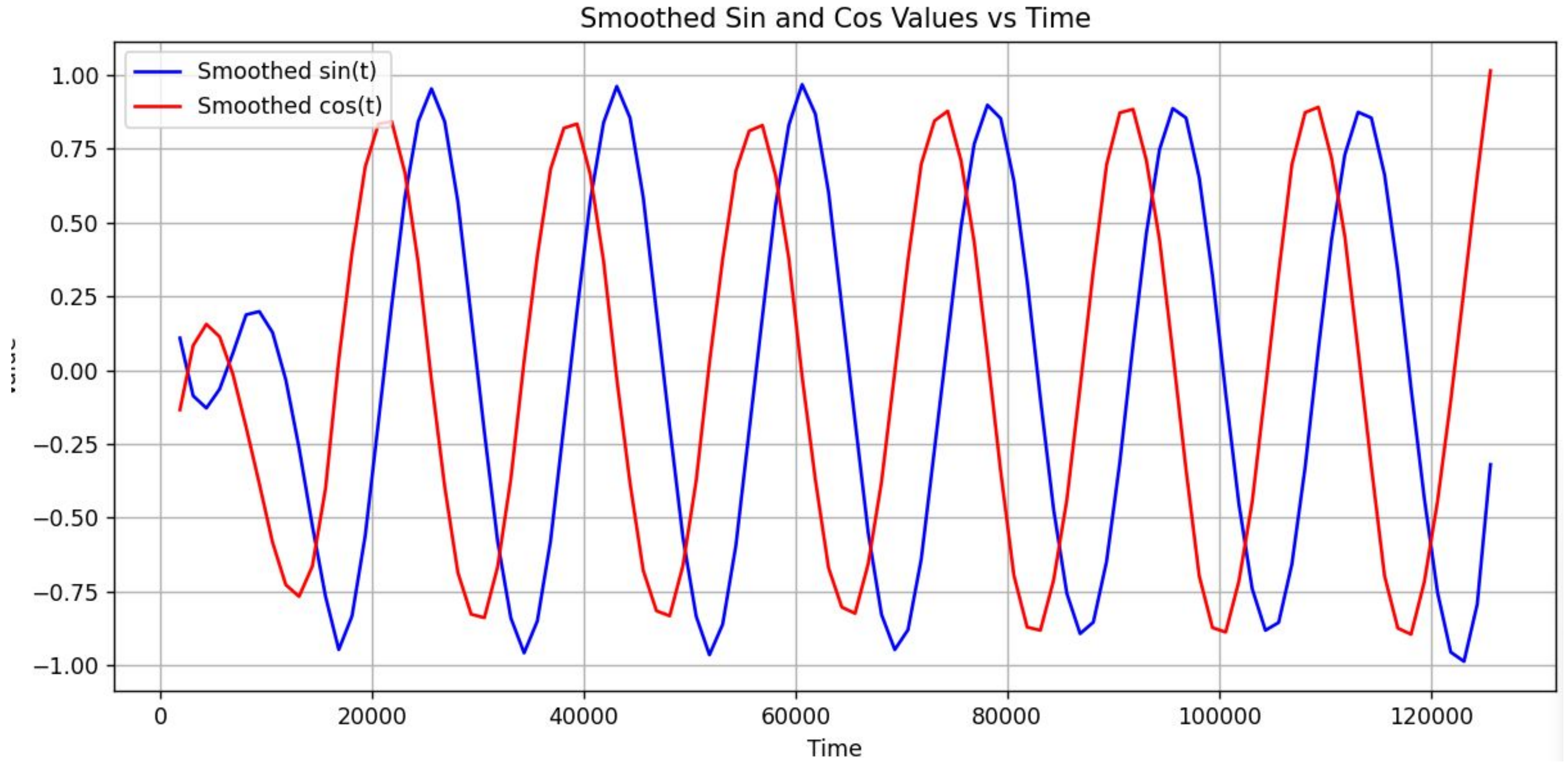


Fig. 5 - Output obtained

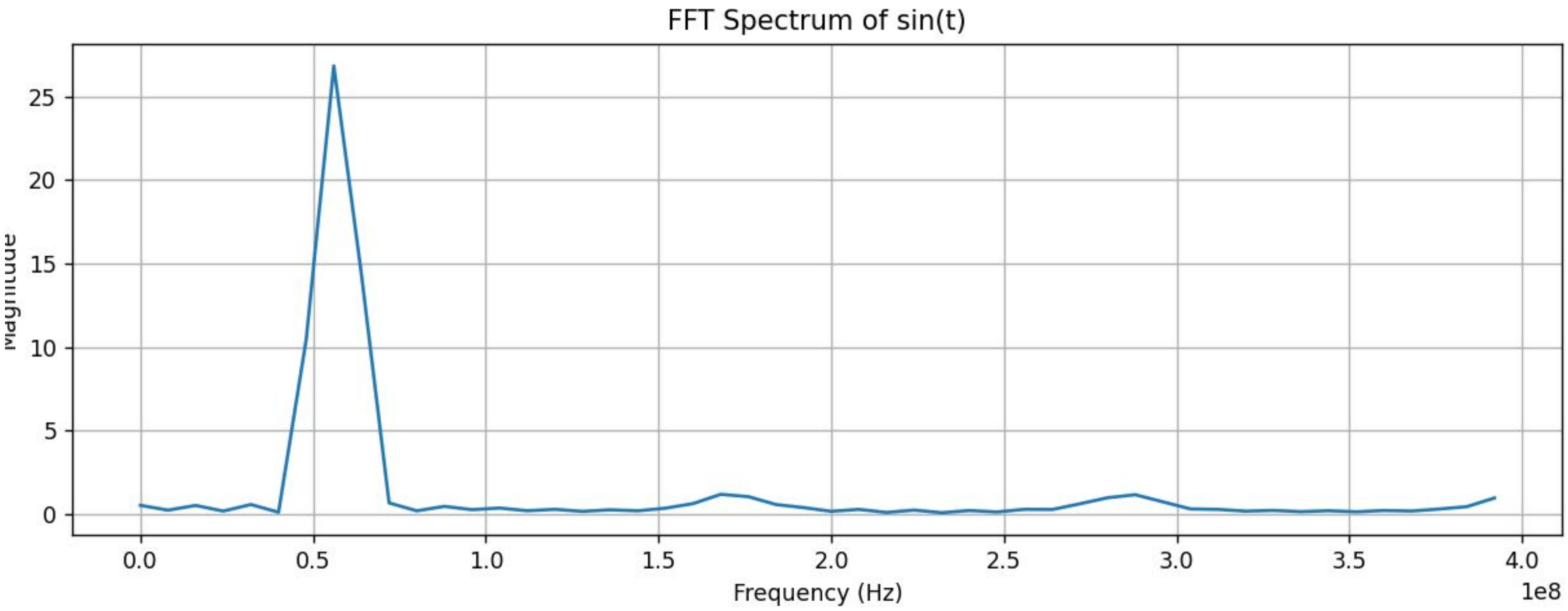


Fig. 6 - Output obtained

Module Analysis

PA_16bit.v: Pipelined accumulator (fast, low delay).

lut.v: Compact, fast amplitude lookup.

rot_new.v: Tree-based rotation:

- $xs = xc - \varphi \cdot yp$
- $ys = yc + \varphi \cdot xp$

PT_transformed / at.v: Angle compression logic.

conf.v: Dynamic mode and LUT control.

Discussions

Output frequency accurate; SFDR lower than expected.

Potential reasons:

- Lower LUT precision
- Approximation in rotation
- FFT resolution limits

Strength: Successfully mimics architecture at lower scale.

Future Work

Use BRAMs for LUTs.

Enhance bit resolution post-PA.

Vary FCW to study SFDR trends.

FPGA + DAC implementation.

Optimize area, power, and speed.

Conclusion

- Reproduced LUT-ROT DDFS architecture.
- Verified correct frequency synthesis.
- Shows potential for custom RF synthesizer design.

References

[1] Yang et al., *“A 2.2-GHz Configurable Direct Digital Frequency Synthesizer Based on LUT and Rotation,”* IEEE TCAS-I, 2019.

Thank you!