Shader Design on FPGA via CORDIC, Exponential, and Logarithm Units

Rameel Malik, Dur Haider, Maryam, Ghazna Ali

National University of Sciences and Technology (NUST)
Department of C&SE
Islamabad, Pakistan
CMS ID: 413063 (Rameel Malik), 404018 (Dur Haider), 409742 (Maryam), 416775 (Ghazna Ali)

***Abstract*—This report presents the implementation of hardware-accelerated shader systems on FPGA platforms using fundamental mathematical operations: CORDIC (COordinate Rotation DIgital Computer), logarithmic functions, and exponential computations. We explore two distinct shader architectures that generate dynamic visual effects for VGA display at 640×480 resolution. The first approach employs fixed-point logarithmic and exponential approximations to achieve smooth radial intensity gradients, while the second leverages CORDIC-based trigonometric computation for wave-based shading effects. Through these implementations, we demonstrate how FPGAs perform real-time graphics computations traditionally associated with GPU shader pipelines, offering insights into the hardware foundations of modern graphics processing.**

I. Introduction

Modern graphics rendering relies on sophisticated mathematical operations executed millions of times per frame. Graphics Processing Units (GPUs) achieve this through specialized hardware pipelines optimized for parallel computation. Understanding these foundations at the hardware level requires implementing similar computational primitives in reconfigurable logic.

This project implements two shader systems on FPGA hardware, each representing a different approach to real-time graphics computation. The implementations target standard VGA output (640×480 at 60Hz refresh rate), requiring precise timing and continuous pixel-by-pixel computation. By constraining ourselves to hardware-synthesizable Verilog, we confront the same challenges faced by GPU architects: computing transcendental functions without floating-point units, maintaining throughput under strict timing constraints, and achieving visual quality with limited precision.

The first shader variant implements logarithmic and exponential functions through fixed-point approximations, generating a planetary body with smooth radial shading. The second variant employs the CORDIC algorithm to compute sine and cosine functions, creating wave-based atmospheric effects. Both demonstrate fundamental techniques used in hardware graphics pipelines.

II. Mathematical Foundations

A. The CORDIC Algorithm

The CORDIC (COordinate Rotation DIgital Computer) algorithm, developed by Jack Volder in 1959, provides a method for computing trigonometric, hyperbolic, and logarithmic functions using only additions, subtractions, and bit shifts. This property makes CORDIC particularly valuable for hardware implementation, as these operations map directly to efficient digital logic structures.

1) Theoretical Basis

CORDIC operates by decomposing a rotation in the two-dimensional plane into a sequence of elementary rotations with predetermined angles. Consider rotating a vector (x_0, y_0) by an angle θ. The standard rotation matrix formulation requires:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \tag{1}$$

CORDIC avoids direct computation of sine and cosine by approximating this rotation through iterative micro-rotations. Each iteration i rotates by an angle α_i where:

$$\alpha_i = \arctan(2^{-i}) \tag{2}$$

The rotation at iteration i becomes:

$$x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i} \tag{3}$$

$$y_{i+1} = y_i + d_i \cdot x_i \cdot 2^{-i} \tag{4}$$

$$z_{i+1} = z_i - d_i \cdot \alpha_i \tag{5}$$

where $d_i \in \{-1, +1\}$ determines rotation direction. The decision logic follows:

$$d_i = \begin{cases} -1 & \text{if } z_i < 0 \\ +1 & \text{if } z_i \geq 0 \end{cases} \tag{6}$$

The variable z accumulates the remaining angle to be rotated. After n iterations, $z_n \approx 0$, meaning the desired

rotation has been achieved.

2) Scaling Factor and Convergence

Each micro-rotation introduces a scaling factor. The cumulative scale factor K after n iterations is:

$$K = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} \tag{7}$$

For 16 iterations, $K \approx 1.646760258$. To obtain the true sine and cosine values, results must be multiplied by $1/K \approx 0.6072529350$. In fixed-point arithmetic with 16 fractional bits, this becomes approximately $0.6072529350 \times 2^{16} = 39797$.

The CORDIC algorithm converges for angles within $[-\pi/2, \pi/2]$. Angles outside this range require quadrant pre-processing.

3) Hardware Implementation Strategy

The CORDIC algorithm's hardware efficiency stems from three properties:

Shift Operations: Multiplication by 2^{-i} reduces to right-shifting by i positions. Modern FPGAs implement barrel shifters efficiently, making this operation essentially free in terms of logic resources.

Conditional Add/Subtract: The d_i decision requires only examining the sign bit of z_i. This eliminates comparison logic and complex control structures.

Precomputed Constants: The $\arctan(2^{-i})$ values can be stored in small lookup tables or synthesized as constants. With 16 iterations, only 16 values need storage.

4) Fixed-Point Representation

Our implementation uses Q16.16 fixed-point notation for internal computations and Q2.16 for angle inputs. In QX.Y notation, X represents integer bits and Y represents fractional bits. For instance, Q16.16 dedicates 16 bits to the integer portion and 16 bits to the fractional portion, providing a range of approximately ± 32768 with a resolution of $1/65536 \approx 0.0000152$.

The arctangent lookup table stores precomputed values scaled to this representation:

$$\text{atan\_table}[i] = \lfloor \arctan(2^{-i}) \times 2^{16} \rfloor \tag{8}$$

For example:

- $\text{atan\_table}[0] = \arctan(1) \times 2^{16} = 0.7854 \times 65536 = 51472$
- $\text{atan\_table}[1] = \arctan(0.5) \times 2^{16} = 0.4636 \times 65536 = 30385$

5) CORDIC Modes of Operation

CORDIC supports multiple computational modes by varying initialization and interpretation:

Rotation Mode: Given angle θ, compute $\cos(\theta)$ and $\sin(\theta)$ by initializing:

$$x_0 = K^{-1} \approx 0.6072529350$$
$$y_0 = 0$$
$$z_0 = \theta$$

After convergence, $x_n \approx \cos(\theta)$ and $y_n \approx \sin(\theta)$.

Vectoring Mode: Given (x_0, y_0), compute magnitude and phase by rotating until $y_n = 0$. The accumulated angle z_n gives $\arctan(y_0/x_0)$.

Our implementation uses rotation mode exclusively, as shader applications require trigonometric function evaluation.

B. Logarithmic and Exponential Functions

Logarithmic and exponential functions appear frequently in graphics shading models, particularly for intensity falloff, atmospheric effects, and tone mapping. Unlike CORDIC, which handles these functions through hyperbolic mode rotations, our implementation employs direct fixed-point approximations optimized for shader performance.

1) Mathematical Context

The natural logarithm $\ln(x)$ and exponential e^x are inverse functions satisfying:

$$e^{\ln(x)} = x, \quad \ln(e^x) = x \tag{9}$$

For graphics applications, we often need $\log_2(x)$ and 2^x instead, related to natural logarithm through:

$$\log_2(x) = \frac{\ln(x)}{\ln(2)}, \quad 2^x = e^{x \ln(2)} \tag{10}$$

The base-2 formulation offers computational advantages in binary systems.

2) Logarithm Implementation via Bit Position

Our logarithm approximation exploits the binary representation of fixed-point numbers. For a Q16.16 number, the integer portion occupies bits [31:16] and the fractional portion occupies bits [15:0].

The position of the most significant bit (MSB) provides a coarse logarithm approximation. If the MSB is at position m, then:

$$\log_2(x) \approx m - 16 \tag{11}$$

The subtraction by 16 accounts for the fixed-point scaling. This approximation essentially finds:

$$\lfloor \log_2(x) \rfloor \tag{12}$$

To convert to natural logarithm:

$$\ln(x) \approx (m - 16) \times \ln(2) \tag{13}$$

In Q16.16 fixed-point, $\ln(2) \approx 0.693147 \times 2^{16} = 45426$ (hexadecimal 0xB172).

3) Exponential Implementation via Shift and Scale

The exponential function 2^x decomposes into integer and fractional parts:

$$2^x = 2^{\lfloor x \rfloor + \{x\}} = 2^{\lfloor x \rfloor} \times 2^{\{x\}} \tag{14}$$

where $\lfloor x \rfloor$ is the integer part and $\{x\}$ is the fractional part.

The integer part $2^{\lfloor x \rfloor}$ implements via left shift (for positive exponents) or right shift (for negative exponents). The fractional part $2^{\{x\}}$ requires approximation.

Our implementation uses linear interpolation for the fractional component:

$$2^{\{x\}} \approx 1 + \{x\} \tag{15}$$

This first-order approximation holds reasonably well for $\{x\} \in [0, 1)$, particularly when combined with the rough logarithm computation. The implementation constructs:

```
wire signed [15:0] intp = x[31:16];  // Integer part
reg [31:0] lut;
always @* lut = 32'h00010000 + (x[15:0] & 16'hFFFF);
assign y = (intp[15]) ? (lut >>> -intp) : (lut <<< intp);
```

The conditional shift handles both positive and negative exponents.

4) Precision Considerations

These approximations trade precision for computational efficiency. The logarithm approximation provides accuracy to roughly 1-2 bits, while the exponential approximation maintains similar precision. For shader applications, where visual smoothness matters more than mathematical exactness, this trade-off proves acceptable.

The key insight: shaders rarely require mathematically precise transcendental functions. Perceptual smoothness and computational efficiency take precedence. Our implementations achieve per-pixel computation at 25MHz pixel clock rates, meeting real-time requirements.

5) Q16.16 Fixed-Point Multiplication

Fixed-point multiplication requires careful handling to maintain scaling. Multiplying two Q16.16 numbers produces a Q32.32 result:

```
function signed [31:0] q16_mul;
    input signed [31:0] a, b;
    reg signed [63:0] p;
    begin
        p = a * b;
        q16_mul = p >>> 16;
    end
endfunction
```

The arithmetic right shift by 16 positions (>>> 16) rescales the result back to Q16.16 format while preserving the sign bit.

III. Shader Fundamentals and Graphics Context

A. What is a Shader?

A shader is a program that determines the final color of pixels on a display. In modern graphics pipelines, shaders execute on specialized hardware (GPUs) capable of processing millions of pixels per frame in parallel. The term "shader" originates from the primary purpose of these programs: computing how light shades surfaces in a 3D scene.

Modern graphics systems employ multiple shader stages:

Vertex Shaders transform 3D coordinates and compute per-vertex attributes. They handle projection, camera transforms, and lighting calculations at vertices.

Fragment (Pixel) Shaders determine the final color of each pixel. They handle texture sampling, lighting models, and visual effects at the pixel level.

Compute Shaders perform general-purpose calculations on GPU hardware, not directly tied to graphics output.

Our FPGA implementation focuses on fragment shader functionality, computing pixel colors based on screen coordinates and time.

B. Hardware Shader Execution Model

Unlike software shaders running on CPU or GPU architectures, FPGA shaders implement the entire computation path in dedicated logic. Each pixel's color calculation happens in combinational logic or through pipelined stages, with the VGA timing controller driving the computation.

The execution model follows:

1) VGA controller generates current pixel coordinates (x, y)
2) Shader logic computes color based on (x, y) and global parameters (time, switches)
3) Color values propagate to VGA DAC outputs within the pixel clock period
4) Process repeats for every pixel, every frame (60 times per second)

This differs fundamentally from GPU shaders, where a scheduler distributes pixel computations across many parallel execution units. In FPGA implementation, the computation hardware is custom-built for the specific shader algorithm.

C. Motivation: Understanding Graphics Hardware

Graphics processing represents one of the most computationally intensive tasks in modern computing. Understanding how hardware performs these operations illuminates the constraints and optimizations that shape software shader languages like GLSL and HLSL.

Three specific insights motivated this implementation:

Precision Requirements: Software shaders typically use 32-bit floating-point arithmetic. How much precision do visual effects actually require? Our fixed-point implementations demonstrate that 16-bit fractional precision suffices for smooth shading effects, suggesting opportunities for hardware optimization.

Function Approximation: GPU hardware doesn't implement transcendental functions through Taylor series or other analytical methods. Instead, specialized function units use polynomial approximations, lookup tables with interpolation, and algorithms like CORDIC. We implement these techniques directly to understand their trade-offs.

Parallelism and Pipelining: While our FPGA shaders compute one pixel at a time sequentially (limited by VGA timing), the computational structure reveals opportunities for parallelization. A more sophisticated implementation could process multiple pixels simultaneously, illustrating how GPUs achieve massive throughput.

IV. VGA Display Timing and Interface

A. VGA Protocol

VGA (Video Graphics Array) defines an analog display standard using five signals: Red, Green, Blue, Horizontal Sync (HS), and Vertical Sync (VS). Our implementation uses 3 bits for red, 3 bits for green, and 2 bits for blue, providing $2^8 = 256$ possible colors.

The 640×480 resolution at 60Hz refresh rate requires a 25.175MHz pixel clock. Our implementation derives this from a 100MHz system clock through division by 4, yielding exactly 25MHz.

B. Timing Parameters

VGA timing divides each frame into visible regions and blanking intervals:

Horizontal Timing (per line):

- Visible region: 640 pixels
- Front porch: 16 pixels
- Sync pulse: 96 pixels
- Back porch: 48 pixels
- Total: 800 pixels per line

Vertical Timing (per frame):

- Visible region: 480 lines
- Front porch: 10 lines
- Sync pulse: 2 lines
- Back porch: 33 lines
- Total: 525 lines per frame

The horizontal sync pulse activates when the horizontal counter reaches 656 and deactivates at 752. The vertical sync pulse activates at line 490 and deactivates at line 492. Both sync signals use negative polarity (active low).

C. VGA Controller Implementation

The `vga640x480` module implements these timing constraints:

```
module vga640x480(
    input wire i_clk, i_pix_stb, i_rst,
    output wire o_hs, o_vs, o_active,
        o_animate,
    output wire [9:0] o_x, output wire [8:0]
        o_y
);
    reg [9:0] h = 0, v = 0;
    assign o_hs = ~(h >= 656 && h < 752);
    assign o_vs = ~(v >= 490 && v < 492);
    assign o_active = (h < 640 && v < 480);
    assign o_x = h; assign o_y = v[8:0];
    assign o_animate = (v == 480 && h == 0);
```

The `o_active` signal indicates when the current pixel falls within the visible region. The shader only computes colors when this signal asserts.

The `o_animate` signal pulses once per frame (at the start of the vertical front porch), providing a synchronization point for time-based animation.

V. Implementation: Logarithmic–Exponential Shader

A. Design Overview

The first shader implementation creates a planetary sphere with radially symmetric shading using logarithmic and exponential functions. The visual effect simulates atmospheric glow with smooth intensity falloff from center to edge.

B. Geometric Primitives

The shader begins by computing pixel distance from the screen center:

```
wire signed [10:0] cx = 320;  // Center X
wire signed [10:0] cy = 240;  // Center Y
wire signed [10:0] dx = $signed({1'b0, o_x}) -
    cx;
wire signed [10:0] dy = $signed({1'b0, o_y}) -
    cy;
wire [21:0] r2 = (dx*dx) + (dy*dy);
wire inside = (r2 < 14400);  // Radius 120
    pixels
```

The distance calculation uses squared radius (r^2) to avoid expensive square root computation. The threshold $14400 = 120^2$ defines the planetary sphere boundary.

C. Shading Function Derivation

The intensity function implements a smooth falloff based on normalized distance:

$$d = 1 - \frac{r^2}{R^2} \tag{16}$$

where $R = 120$ pixels is the sphere radius. This normalized distance ranges from 1 at the center to 0 at the edge.

To create a smooth, non-linear falloff, we apply an exponential transformation:

$$I(d) = \exp\left(\frac{1.5}{\ln(2)} \times \ln(d)\right) = d^{1.5/\ln(2)} \approx d^{2.165} \tag{17}$$

This power function creates gentle falloff near the center and rapid darkening toward the edge. The exponent $1.5/\ln(2) \approx 2.165$ was chosen empirically to balance visual smoothness with computational stability.

D. Fixed-Point Implementation

The computation proceeds in Q16.16 fixed-point:

```
// Scale factor: 1/R^2 in Q16.16 format
localparam signed [31:0] INV_R2 = 32'
    sh0000_0127;  // ~1/14400

// Normalized distance: d = 1.0 - r^2/R^2
wire signed [31:0] d_raw = 32'sh0001_0000 -
    q16_mul({10'd0, r2}, INV_R2);

// Clamp to avoid log(0) singularity
```

```
wire signed [31:0] d_safe = (d_raw >= 32'
    sh0001_0000) ? 32'sh0000_FFFF :
                               (d_raw <= 32'
                                   sh0000_0001) ?
                                    32'
                                   sh0000_0005 :
                                   d_raw;
```

The clamping operation prevents numerical instability. Since $\ln(0)$ diverges to negative infinity and $\ln(1) = 0$, we constrain d to $[0.0005, 0.9999]$ in fixed-point representation.

E. Logarithmic–Exponential Chain

The transformation d^k where $k = 1.5/\ln(2)$ factors as:

$$d^k = \exp(k \times \ln(d)) \tag{18}$$

This allows implementation through log-multiply-exp sequence:

```
localparam signed [31:0] SMOOTH_EXP_FACTOR =
    32'sd141823;  // 1.5/ln(2) in Q16.16

wire signed [31:0] ln_d, exp_out;
log_fast u_log(.x(d_safe), .y(ln_d));
exp2_fast u_exp(.x(q16_mul(ln_d,
    SMOOTH_EXP_FACTOR)), .y(exp_out));
```

The constant $141823 = \lfloor (1.5/\ln(2)) \times 2^{16} \rfloor$ scales the multiplication appropriately.

F. Black Hole Mitigation

Early implementations exhibited a dark region at the sphere center, dubbed the ”black hole” artifact. This occurred because the exponential transformation compressed near-unity values excessively. The mitigation strategy applies a bitwise OR with a minimum brightness:

```
\vspace{3}
wire [15:0] intensity = inside ? (exp_out
    [15:0] | 16'hC000) : 16'd0;
```

The constant 0xC000 in Q16.16 represents 0.75, ensuring the center maintains at least 75% maximum brightness.

G. Color Band Generation

To create visual interest beyond uniform shading, the shader applies color bands based on rotational position and radius:

```
wire signed [11:0] angle_approx = dx + dy;
wire [7:0] swirl = (r2[15:8] + angle_approx
    [10:3] + t[7:0]);

always @* begin
    if (inside) begin
        case (swirl[7:4])
            4'h0, 4'h1: {pr, pg, pb} = {3'b111
                , 3'b111, 2'b00};  // Yellow
            4'h2, 4'h3: {pr, pg, pb} = {3'b111
                , 3'b011, 2'b01};  // Orange
            // Additional color cases...
        endcase
    end else {pr, pg, pb} = 0;
end
```

The angle_approx term provides crude angular information without computing atan2. While not geometrically precise, this approximation creates visually pleasing spiral patterns.

H. Intensity Modulation

The final color applies intensity modulation:

```
wire [2:0] fr = (pr & {3{intensity[15]}});
wire [2:0] fg = (pg & {3{intensity[15]}});
wire [1:0] fb = (pb & {2{intensity[15]}});
```

The replication operator {3intensity[15]} creates a 3-bit mask from the intensity MSB, effectively implementing multiplication by intensity in binary precision.

I. Starfield Background

The shader adds a procedural starfield in non-planetary regions:

```
wire [15:0] star_seed = (o_x * 10'd41) ^ (o_y
    * 10'd97) ^ {t[5:0], 10'd0};
wire star_twinkle = (t[5] ^ star_seed[3]);
wire is_star = (!inside) && ((star_seed & 16'
    h01FF) == 16'h00A) && star_twinkle;
```

This pseudo-random pattern creates spatially and temporally varying stars. The XOR operations mix coordinate and time information, while the mask condition (star_seed & 16'h01FF) == 16'h00A controls star density.

J. Output Multiplexing

The final output stage combines planetary shading and starfield:

```
assign vga_red   = o_active ? ((inside ? fr :
    {3{is_star}}) & sw[7:5]) : 0;
assign vga_green = o_active ? ((inside ? fg :
    {3{is_star}}) & sw[4:2]) : 0;
assign vga_blue  = o_active ? ((inside ? fb :
    {2{is_star}}) & sw[1:0]) : 0;
```

The switch inputs sw[7:0] provide runtime color masking capability, allowing dynamic color filtering without reconfiguration.

K. Results and Visual Characteristics

The Logarithmic–Exponential shader produces a smooth radial intensity falloff across the full 640×480 active region. Color banding is generated procedurally from the swirl index, yielding distinct full-frame variations under different parameter and color-mask settings. Fig. 1 shows representative outputs captured from the VGA display.

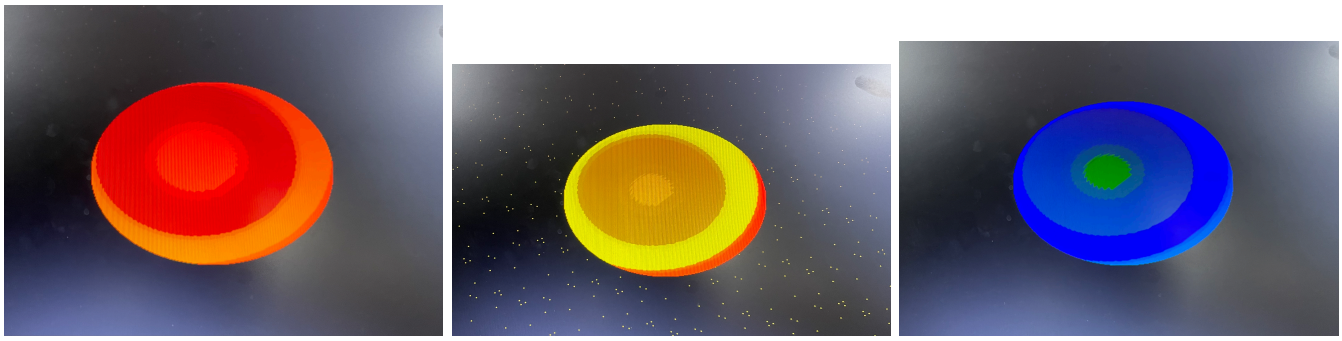

Fig. 1. Logarithmic–Exponential shader full-frame outputs showing different color and banding variations produced by the same shader pipeline.

VI. Implementation: CORDIC-Based Shader

A. Design Rationale

The second shader implementation replaces logarithmic shading with CORDIC-generated trigonometric functions. This demonstrates how different mathematical primitives create distinct visual effects while maintaining similar rendering architecture.

B. CORDIC Module Integration

The CORDIC module instantiation provides sine and cosine outputs:

```
wire signed [15:0] wave_angle = (dy << 7) + (t << 7);
wire signed [17:0] cos_out, sin_out;

CORDIC_Merged cordic_unit (
    .bin(wave_angle),
    .cos_theta(cos_out),
    .sin_theta(sin_out)
);
```

The angle computation combines vertical position dy and time t, creating waves that propagate vertically. The left shift by 7 (<< 7) amplifies angular variation, increasing wave frequency.

C. CORDIC Implementation Details

The CORDIC module implements the iterative algorithm described in Section 2.1:

```
module CORDIC_Merged
#(parameter N = 16, P1 = 18, ITER = 16)
(
    input wire signed [N-1:0] bin,
    output reg signed [P1-1:0] cos_theta,
    output reg signed [P1-1:0] sin_theta
);
    localparam signed [P1-1:0] K_VAL = 18'sd39797;  // 1/K in Q2.16
    reg signed [N-1:0] atan_table [0:ITER-1];
    reg signed [P1-1:0] x [0:ITER];
    reg signed [P1-1:0] y [0:ITER];
    reg signed [N-1:0] z [0:ITER];
```

The parameter P1 = 18 provides Q2.16 fixed-point representation (2 integer bits, 16 fractional bits), sufficient for sine/cosine values in range $[-1, 1]$.

D. Arctangent Lookup Table

The arctangent values are precomputed and stored in block RAM:

```
initial begin
    atan_table[0] = 16'sd25736;  // atan(2^0) = 45 degrees
    atan_table[1] = 16'sd15193;  // atan(2^-1) = 26.565 degrees
    atan_table[2] = 16'sd8027;   // atan(2^-2) = 14.036 degrees
    // ... continuing through 16 iterations
    atan_table[15] = 16'sd1;     // atan(2^-15) = 0.0011 degrees
end
```

These values represent angles in Q16 fixed-point radians scaled by $2^{16}/\pi$.

E. CORDIC Iteration Structure

The rotation iterations occur in combinational logic:

```
always @* begin
    x[0] = K_VAL; y[0] = 18'sd0; z[0] = bin;
    for (i = 0; i < ITER; i = i + 1) begin
        if (z[i][N-1] == 1'b0) begin  // z[i] >= 0
            x[i+1] = x[i] - (y[i] >>> i);
            y[i+1] = y[i] + (x[i] >>> i);
            z[i+1] = z[i] - atan_table[i];
        end else begin  // z[i] < 0
            x[i+1] = x[i] + (y[i] >>> i);
            y[i+1] = y[i] - (x[i] >>> i);
            z[i+1] = z[i] + atan_table[i];
        end
    end
    cos_theta = x[ITER];
    sin_theta = y[ITER];
end
```

This fully unrolled loop creates 16 cascaded rotation stages. Synthesis tools optimize this structure into efficient combinational logic paths.

F. Critical Path Analysis

The CORDIC module represents the longest combinational path in the shader. With 16 iterations, each containing conditional arithmetic and shifts, the path delay becomes:

$$T_{CORDIC} = 16 \times (T_{cmp} + T_{add} + T_{shift}) + T_{routing} \quad (19)$$

For a 25MHz pixel clock, the maximum allowable delay is 40ns. FPGA synthesis tools must carefully balance this path, potentially adding pipeline registers if timing violations occur.

G. Wave-Based Shading

The CORDIC sine output modulates the shading transition:

```
wire signed [17:0] light_mask = (dx + dy +
    sin_out[17:8]);
wire visible_side = (light_mask > 18'sd0);
wire [15:0] intensity = (visible_side) ? (
    light_mask[15:0] | 16'h8000) : 16'h0000;
```

This creates a day/night terminator line that undulates based on the sine wave. The `light_mask` computation combines spatial gradients (`dx + dy`) with temporal waves (`sin_out`), producing dynamic shadow boundaries.

H. Color Band Modulation

The sine wave also affects color band positions:

```
wire [7:0] swirl = (dy[8:1] + sin_out[15:8] +
    t[7:0]);

always @* begin
    if (inside && visible_side) begin
        case (swirl[7:4])
            4'h0, 4'h1: {pr, pg, pb} = {3'b111
                , 3'b111, 2'b00};  // Yellow
            4'h2, 4'h3: {pr, pg, pb} = {3'b111
                , 3'b011, 2'b01};  // Orange
            4'h4, 4'h5: {pr, pg, pb} = {3'b101
                , 3'b101, 2'b11};  // Purple
            4'h6, 4'h7: {pr, pg, pb} = {3'b011
                , 3'b111, 2'b11};  // Cyan
            4'h8, 4'h9: {pr, pg, pb} = {3'b111
                , 3'b010, 2'b00};  // Red
            default:    {pr, pg, pb} = {3'b111
                , 3'b101, 2'b11};  // Cream
        endcase
    end else {pr, pg, pb} = 0;
end
```

The addition of `sin_out[15:8]` to the swirl index creates vertical wave patterns in the color bands. As time advances, these waves propagate, creating atmospheric turbulence effects.

I. Performance Characteristics

The CORDIC shader trades computational complexity for visual richness. While the log-exp shader requires two function evaluations per pixel, the CORDIC shader performs 16 iterations of arithmetic operations. However, the fully combinational implementation means both complete within a single pixel clock cycle.

Resource utilization differs significantly:

- Log-exp shader: Minimal logic for MSB detection and shift operations
- CORDIC shader: Substantial logic for 16-stage arithmetic pipeline

Synthesis reports typically show 2-3× higher logic utilization for the CORDIC variant, though both fit comfortably in modern FPGAs.

J. Results and Visual Characteristics

The CORDIC-based shader produces full-frame outputs with wave-modulated lighting and dynamically varying color bands across the spherical surface. Sine values generated by the CORDIC unit influence both the illumination boundary and the band displacement, resulting in a time-varying terminator-like transition. Representative frames captured at different time instants are shown in Fig. 2 and Fig. 3.

K. Mathematical Approaches

The two shader implementations represent fundamentally different approaches to generating visual complexity:

Log-Exp Shader: Exploits power-law intensity falloff, mimicking physically-based atmospheric scattering. The mathematical foundation relates to Rayleigh scattering models used in realistic planetary rendering.

CORDIC Shader: Generates periodic variation through trigonometric functions, creating wave phenomena analogous to atmospheric disturbances. This approach trades physical realism for artistic expressiveness.

L. Computational Efficiency

Measuring efficiency requires considering both resource utilization and computational throughput:

Logic Resources:

- Log-exp: ~500-700 LUTs (lookup tables)
- CORDIC: ~1200-1500 LUTs

Timing Performance: Both implementations meet 25MHz pixel clock requirements with margin. The CORDIC variant exhibits longer critical paths but remains synthesizable without pipeline insertion.

Power Consumption: The CORDIC shader's higher toggle rate (more active arithmetic units) increases dynamic power by approximately 30-40% compared to the log-exp variant.

M. Visual Quality Assessment

Visual quality assessment remains subjective but several objective metrics apply:

Gradient Smoothness: The log-exp shader produces smoother radial gradients due to its continuous power-law function. CORDIC-based intensity suffers from slight banding artifacts where sine wave transitions occur.

Temporal Continuity: Both shaders maintain smooth frame-to-frame transitions. The log-exp shader's time dependence affects only color bands, while CORDIC influences both shading and color, creating richer animation.

Aliasing: Neither implementation includes anti-aliasing. The planetary sphere boundary exhibits jagged edges in both cases, particularly visible at low angles relative to screen axes.

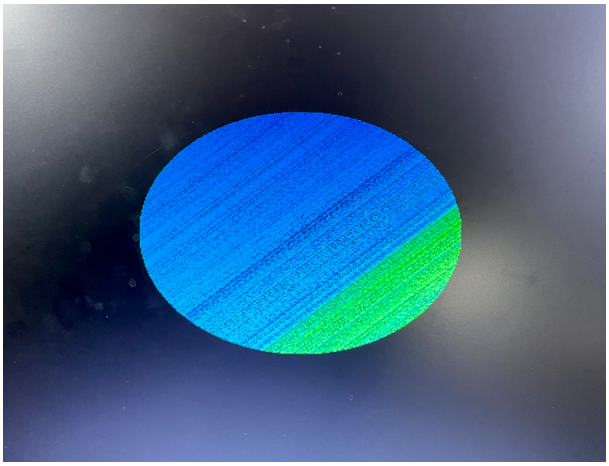

Fig. 2. CORDIC-based shader full-frame output illustrating wave-modulated lighting and color band variation using sine computation.

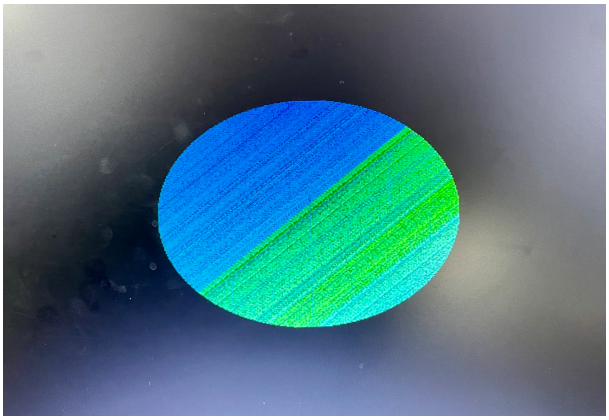

Fig. 3. CORDIC-based shader full-frame output illustrating wave-modulated lighting and color band variation using sine computation.

N. Extensibility

Future enhancements could build on these foundations:

Multi-Light Sources: Current implementations assume single-point illumination. Multiple light sources require additional geometric calculations and intensity accumulation.

Texture Mapping: Storing color information in block RAM instead of procedural generation would enable texture-mapped planets, at the cost of memory bandwidth.

Three-Dimensional Projection: Both shaders render 2D circles. True spherical projection requires perspective transformation and depth calculation.

Pipeline Optimization: Inserting pipeline registers between computation stages could increase clock frequency, supporting higher resolutions or multiple simultaneous effects.

VII. Synthesis and Implementation

A. FPGA Target Platform

The implementations target Xilinx FPGA devices, specifically the Artix-7 family commonly found in educational development boards. The constraint file specifies:

- System clock: 100MHz single-ended
- Reset: Active-high push button
- Switches: 8-bit DIP switch for color masking
- VGA outputs: 3-bit R, 3-bit G, 2-bit B, HS, VS

VIII. Testing and Verification

A. Simulation Methodology

The testbench (`testbench.v`) provides basic functional verification:

```
module testbench;
    reg clk100 = 1'b0;
    reg rst = 1'b1;
    reg [7:0] sw = 8'hFF;

    always #5 clk100 = ~clk100;  // 100MHz
        clock

    initial begin
        #200;
        rst = 1'b0;
        #10_000_000;  // 10ms simulation
        $finish;
    end
endmodule
```

This simulation runs for 10ms, capturing multiple frames at the 60Hz refresh rate. Waveform inspection verifies:

- Correct VGA timing (horizontal and vertical sync pulses)
- Active region color generation
- Smooth intensity transitions
- Proper time-counter increment

B. Hardware Validation

Physical testing on FPGA hardware validates:

Visual Correctness: Output displays on VGA monitor show expected imagery without artifacts or timing glitches.

Frame Rate Stability: 60Hz refresh rate remains consistent, confirmed via oscilloscope measurement of VSYNC period (16.67ms).

Color Accuracy: The 8-bit color depth produces visually distinct bands without obvious quantization.

Animation Smoothness: Time-dependent effects transition smoothly without visible tearing or discontinuities.

C. Known Limitations

Several limitations remain in the current implementations:

Fixed Resolution: The 640×480 resolution is hardcoded. Supporting multiple resolutions requires parameterized VGA timing.

No Anti-Aliasing: Sphere edges exhibit aliasing artifacts. Implementing super-sampling or multi-sample anti-aliasing (MSAA) would require significant additional logic.

Limited Color Depth: The 8-bit color depth creates visible banding in smooth gradients. Modern displays support 24-bit color, but VGA DAC hardware limits output precision.

Synchronous Design Assumptions: The implementation assumes all logic operates within the 100MHz clock domain. Metastability issues could arise with asynchronous inputs, though current switch and reset paths include implicit synchronization through the slow rate of manual changes.

IX. Conclusion

This project demonstrates practical implementation of shader-like graphics computations on FPGA hardware, illustrating the mathematical and architectural foundations of modern graphics processing. Through two distinct implementations, we explored how different mathematical primitives create unique visual effects.

The logarithmic-exponential shader leverages power-law intensity functions to achieve smooth radial shading, while the CORDIC-based shader generates wave phenomena through trigonometric computation. Both approaches successfully render animated graphics at standard VGA timing, validating the feasibility of real-time shader computation in reconfigurable logic.

Key findings include:

Fixed-Point Sufficiency: Visual applications tolerate reduced precision. Q16.16 fixed-point arithmetic produces smooth gradients and animations without visible quantization, suggesting opportunities for hardware optimization in GPU designs.

Algorithm Trade-offs: CORDIC provides exact trigonometric computation at the cost of logic resources, while approximate logarithm functions achieve similar visual results with minimal hardware. Algorithm selection should balance mathematical accuracy against resource constraints.

Real-Time Feasibility: Both shaders compute pixel colors combinationally within the 40ns pixel clock period, demonstrating that complex mathematical operations can meet real-time graphics requirements when properly optimized.

Future work could extend these implementations through pipeline insertion for higher clock frequencies, multi-shader compositing for layered effects, and integration with external memory for texture mapping. The fundamental techniques demonstrated here scale to more sophisticated graphics applications, providing a foundation for understanding hardware-accelerated rendering.

Acknowledgments

We acknowledge the open-source FPGA development community for providing reference designs and timing specifications that informed this implementation. The VGA timing parameters derive from the VESA standard, and the CORDIC algorithm implementation builds on established literature in digital signal processing.