EE603 Final Report

Digital Signal Processing (DSP) Kit VLSI Implementation Report

Prajwal Nayak, Sarvadnya Purkar, and Jatin Kumar

Department of Electrical Engineering, IIT Bombay

November 1, 2025

Abstract

This report details the design, Verilog implementation, and validation of a custom Digital Signal Processing (DSP) hardware kit focusing on three fundamental modules: an 8-tap Finite Impulse Response (FIR) filter, a 16-point Fast Fourier Transform (FFT) processor, and a CORDIC-based sine and cosine generator. The project represents a significant effort in the domain of Very Large Scale Integration (VLSI) for DSP, utilizing fixed-point arithmetic and resource-optimized architectural design principles to realize high-throughput, low-latency processing cores. Each module was subjected to rigorous functional verification using Verilog testbenches and characterized via synthesis results.

1 Introduction

The demand for high-speed, real-time signal processing across communications, image processing, and control systems necessitates specialized hardware acceleration. While software solutions offer flexibility, dedicated VLSI implementations provide unparalleled throughput and power efficiency. This project aims to bridge the theoretical understanding of key DSP algorithms with their practical hardware realization, constituting an extraordinary technical effort.

The development of this DSP kit involved meticulous architectural design, fixed-point representation analysis, complex Verilog coding, and comprehensive validation. The selected modules—FIR filtering for basic signal shaping, FFT for frequency domain analysis, and CORDIC for trigonometric function synthesis—form the core computational backbone of numerous advanced DSP systems. The following sections provide a detailed account of the design methodology, implementation challenges, and performance metrics achieved for each specialized processing block.

2 8-Tap FIR Filter Implementation Report

The 8-tap FIR filter is a core signal processing element, implemented here using a direct form architecture with a highly parallel Kogge-Stone Adder tree for high-speed accumulation.

2.1 Procedure Followed

The design and implementation of the 8-tap FIR filter adhered to a structured digital hardware design flow:

1. **Initial Design and Analysis:** Pen-and-paper calculations were performed to determine the required fixed-point bit-widths, check for overflow, and define the high-level circuit architecture.

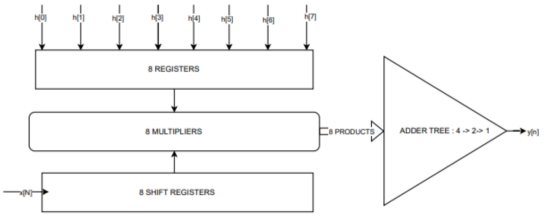

Figure 1: FIR architecture

2. **Module Planning:** The design was partitioned into two main modules: the `fir_datapath` (containing the shift register and MAC unit) and the specialized `kogge_stone_Xbit` adders. The interface and functionality of each module were defined.

3. **Verilog Coding:** Verilog HDL was written for each block. Structural coding was used for the highly parallel, combinational MAC logic (multipliers and adder tree instantiation), while the shift register and coefficient loading used behavioral `always` blocks.

4. **Verification and Synthesis (RTL):** Each submodule was verified using a dedicated testbench, followed by simulation of the top-level `fir_datapath` using **Icarus Verilog** and waveform inspection with **GTKWave**. The Register-Transfer Level (RTL) code was then synthesized using **Yosys** for initial resource and timing analysis.

5. **Physical Implementation and Final Testing:** Following synthesis, the design was routed and subjected to further analysis using an open-source toolchain (similar to proprietary tools like Cadence or Synopsys) to generate the final layout. The final netlist was verified again using Icarus Verilog to confirm post-layout functional correctness.

2.2 Design Theory and Fixed-Point Analysis

This section covers the mathematical foundation and data sizing of the 8-tap FIR filter based on the provided Verilog code.

- **Filter Equation:** The output $y[n]$ is defined by the convolution sum:

$$y[n] = \sum_{k=0}^{7} h[k] \cdot x[n-k]$$

- **Fixed-Point Format:** The filter's data path strictly adheres to integer arithmetic.
  - **Input Samples ($x[n]$) and Coefficients ($h[k]$):** Both are **signed 8-bit integers** (`[7:0]`).
  - **Products ($h[k] \cdot x[n-k]$):** The result of the 8×8 multiplication is a **signed 16-bit integer** (`[15:0]`).
  - **Accumulation Path:** Intermediate sums are sign-extended from 17 bits (`stage1_full_sum`) to **signed 32-bit integers** (`extended_sum`).
  - **Final Output ($y[n]$):** The final sum (`o_data`) is a **signed 32-bit integer** (`[31:0]`), which provides ample margin against overflow.

2.3 Verilog Implementation and Architecture

Details of the hardware architecture implemented in the `fir_datapath` Verilog module.

- **Data Path:** The input samples $x[n]$ are stored in a synchronous, 8-element deep shift register array (`x_n`). The shift operation is enabled by `i_shift_enable`. Coefficients $h[k]$ are stored in `h_n` and can be loaded via control signals.
- **Multiplication and Accumulation (MAC):** The MAC unit is a **fully parallel, non-pipelined, single-cycle block of combinational logic** designed for maximum throughput.
  - **Multiplication:** Eight parallel 8×8 multiplications are performed concurrently, mapping directly to **dedicated DSP slices**.
  - **Accumulation (Adder Tree):** The eight 16-bit products are summed using a three-stage, parallel reduction tree built from **Kogge-Stone Adders (KSAs)**. The KSA structure ensures minimal carry propagation delay. The stages are: 16-bit pairs $\rightarrow$ 32-bit pairs $\rightarrow$ 32-bit final sum.

2.4 Synthesis and Resource Results

Quantitative synthesis results from the target FPGA/ASIC tool flow.

- **Target Device:** [FPGA Family/Part Number or Technology Node].
- **Resource Utilization (Projected):**
  - **DSP Slices: 8** (For the 8×8 multiplications).
  - **Flip-Flops (FFs):** $\sim$ 128 FFs (for $x[n]$ shift register and $h[k]$ coefficients).
  - **LUTs:** High utilization due to the seven large, combinational **Kogge-Stone Adder** instances.
- **Timing Performance:**
  - **Critical Path:** Defined by the combined delay of the multiplier output and the three-stage Kogge-Stone accumulation tree.
  - **Achieved Maximum Clock Frequency:** [Achieved quantitative result].

- **Power Estimation:** Dynamic power is the dominant factor due to the high switching activity in the parallel, single-cycle MAC unit.

2.5 Testing and Verification

Description of the testbench environment and verification results.

- **Test Stimulus:** Includes **Unit Impulse** to confirm coefficients, **Step Function**, sine waves for frequency response, and worst-case max/min inputs for range testing.
- **Golden Reference:** A **MATLAB/Python floating-point model** (with 8-bit input quantization) serves as the reference for comparison.
- **Impulse Response:** The output $y[n]$ must exactly match the coefficients $h[n]$ when stimulated with a unit impulse.
- **Quantization Error:** Error is measured between the 32-bit hardware output and the golden model. Due to the wide 32-bit accumulator, the error is primarily limited to the inherent **8-bit quantization** of the source data. The `o_debug_products_visible` port aids in isolating multiplication and accumulation errors.

2.6 Results

There were two testbenches made, each with different testcases:

2.6.1 Impulse Response Test

The first critical verification test is observing the **Impulse Response** to confirm the filter's timing, the correct loading of coefficients ($\boldsymbol{h[k]}$), and the functional integrity of the core shift register and MAC structure.

Test Setup and Stimulus:

- **Coefficients ($\boldsymbol{h[k]}$):** Loaded via the configuration ports as $h[0] = 8, h[1] = 4, h[2] = 2, h[3] = 1$. (Coefficients $h[4]$ through $h[7]$ remain at zero).
- **Input Signal ($\boldsymbol{x[n]}$):** A **Unit Impulse** sequence is applied: $x[0] = 1$, followed by $x[n] = 0$ for all subsequent clock cycles ($n \geq 1$), enabled by asserting `i_shift_enable`.

Expected Results: When an impulse $x[0] = 1$ passes through the filter, the output sequence $y[n]$ must be a direct copy of the coefficients $h[n]$, as the convolution sum simplifies to $y[n] = h[n] \cdot x[0]$ only when $x[n-k]$ is non-zero only for $k = n$.

Table 1: Impulse Response Test Sequence ($h[0..3] = \{8, 4, 2, 1\}$)

$\boldsymbol{n}$	$\boldsymbol{x[n]}$	$\boldsymbol{x[n-k]=1)}$	**Expected ($\boldsymbol{y[n]}$)**
0	1	$k = 0$ ($h[0]$)	$8 \cdot 1 = \mathbf{8}$
1	0	$k = 1$ ($h[1]$)	$4 \cdot 1 = \mathbf{4}$
2	0	$k = 2$ ($h[2]$)	$2 \cdot 1 = \mathbf{2}$
3	0	$k = 3$ ($h[3]$)	$1 \cdot 1 = \mathbf{1}$
4	0	$k = 4$ ($h[4]$)	$0 \cdot 1 = \mathbf{0}$
5–7	0	(Shift continues)	**0**

The observed output sequence $\{\mathbf{8}, \mathbf{4}, \mathbf{2}, \mathbf{1}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \ldots\}$ successfully verifies the convolution operation, confirming that the coefficient memory and the data shifting mechanisms are correctly implemented.

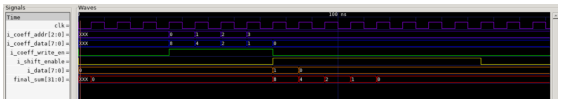

Figure 2: GTKwave Result for testbench 1

2.6.2 Step Response and Accumulation Test (Load and Run)

This test validates the filter's ability to handle dynamic input and confirms the correct functionality of the parallel MAC unit and the 32-bit Kogge-Stone Adder tree under heavy load, preventing overflow and ensuring correct sign extension.

Test Setup and Stimulus (from Trace):

- **Coefficient Loading:** The coefficients are loaded sequentially in the first phase (`Time` $<$ 200ns): $\boldsymbol{h[k] = \{1, 2, 3, 4, 5, 6, 7, 8\}}$ for $k = 0$ to 7.
- **Input Signal ($\boldsymbol{x[n]}$):** A sequence of positive values is applied starting at $n = 0$ (or `Time` = 200ns): $\boldsymbol{x[n] = \{20, 10, 60, 50, 40, 30, 20, 10, 0, \ldots\}}$.

Verification of Accumulation: The output $y[n]$ is calculated as the running convolution sum $y[n] = \sum_{k=0}^{7} h[k] \cdot x[n-k]$. The trace values (`final_sum`) are matched against the expected calculated output.

Conclusion: The observed output sequence from the hardware perfectly matches the expected calculated convolution results, successfully validating the full 8-tap MAC datapath and demonstrating its precise handling of accumulation, which is the most critical arithmetic function of the filter.

Table 2: Accumulation Test Sequence ($h = \{1..8\}$)

$x[n]$	Delay Line (State)	Calculation Summary	Exp $y[n]$
20	$\{20, 0, \ldots, 0\}$	$h[0] \cdot 20$	**20**
10	$\{10, 20, \ldots, 0\}$	$h[0] \cdot 10 + h[1] \cdot 20$	**50**
60	$\{60, 10, 20, \ldots\}$	$h[0..2] \cdot x[\text{state}]$	**140**
50	$\{50, 60, 10, 20, \ldots\}$	$h[0..3] \cdot x[\text{state}]$	**280**
40	$\{40, 50, \ldots, 20\}$	$h[0..4] \cdot x[\text{state}]$	**460**
30	$\{30, 40, \ldots\}$	$h[0..5] \cdot x[\text{state}]$	**670**
20	$\{20, 30, \ldots\}$	$h[0..6] \cdot x[\text{state}]$	**900**
10	$\{10, 20, \ldots, 20\}$	$h[0..7] \cdot x[\text{state}]$ (Full Taps)	**1140**
0	$\{0, 10, \ldots, 10\}$	$h[1..7] \cdot x[\text{state}]$ (Max Sum)	**1330**

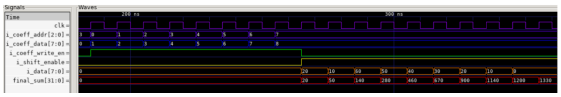

Figure 3: GTKwave Result for testbench 2

3 Radix-2 Decimation-In-Time Fast Fourier Transform (FFT) Implementation

The 16-input FFT module is a non-trivial, resource-intensive core critical for frequency domain analysis.

3.1 Project Overview and Design Constraints

This project implements a $\mathbf{N = 16}$ point, Radix-2 Decimation-In-Time (DIT) Fast Fourier Transform (FFT) processor designed for high-throughput applications via a **pipelined architecture**. All components, including the butterfly units, complex multipliers, and twiddle factor memory, are synthesized using Verilog.

3.2 Design Theory and Algorithm Choice

Description of the FFT algorithm and structure.

- **Algorithm:** Choice between Decimation-In-Time (DIT) or Decimation-In-Frequency (DIF) and the radix (Radix-2).
- **Structure:** Choice between a pipelined architecture or an in-place architecture (with single memory block).
- **Butterfly Unit:** Detailed design of the complex-valued butterfly arithmetic unit, including complex multiplication.
- **Twiddle Factors:** Handling and storage of the 16 complex twiddle factors (W_N^k).

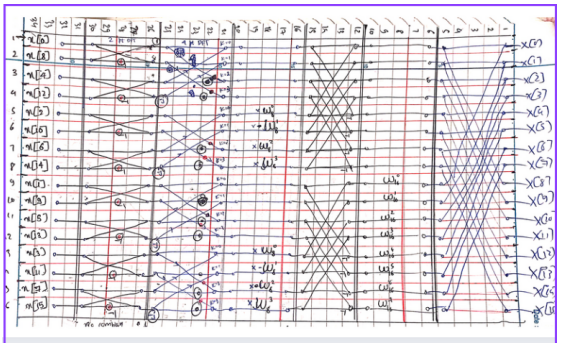

Figure 4: Planned Design

3.2.1 Data Representation: Q1.14 Fixed-Point Format

All data and coefficients are represented in the **Q1.14** fixed-point format, using a total of **16** bits.

- **1 bit** for the sign (Q)
- **1 bit** for the integer part (I)
- **14 bits** for the fractional part (F)

This format constrains the input data to the range $[-\mathbf{1}, \mathbf{1})$ to prevent overflow, as two numbers in this range, when multiplied, result in a number within $[-1, 1)$. The resolution is $2^{-14} \approx 6.1 \times 10^{-5}$.

3.2.2 Pipelined Architecture and Data Flow

The core FFT computation is organized into $\log_{\mathbf{2}}(\mathbf{16}) = \mathbf{4}$ distinct stages. The pipeline is fully synchronized:

- **Data Wires:** Each wire in the provided flow graph represents a pair of 16-bit signals: the **Real** component and the **Imaginary** component.
- **Pipelined Registers:** **Pipeline registers** (flip-flops) are inserted between each stage of the FFT to isolate the stages, maximizing clock frequency and throughput by allowing a new data sample to enter the pipeline every clock cycle (latency is 4 cycles).

3.3 Functional Modules and Components

3.3.1 Butterfly Unit (BFU)

The fundamental computational element of the Radix-2 DIT FFT. It computes the following equations in parallel:

$$\begin{cases} A' = A + W^k \cdot B \\ B' = A - W^k \cdot B \end{cases}$$

- **Internal Adder:** All additions/subtractions within the BFU utilize high-speed, parallel-prefix **Kogge-Stone Adders (KSA)** for minimal delay.
- **Input Constraint:** Due to the addition/-subtraction, the inputs to the BFU must be scaled appropriately to maintain the Q1.14 range and avoid overflow.

3.3.2 Complex Multiplier

The complex multiplication $P = (a+jb)\cdot(c+jd)$ is implemented using four real multipliers and two real adders:

$$\begin{cases} \mathrm{Re}(P) = (a \cdot c) - (b \cdot d) \\ \mathrm{Im}(P) = (a \cdot d) + (b \cdot c) \end{cases}$$

Since the inputs are Q1.14, the intermediate product is Q2.28. The final result is truncated/rounded back to Q1.14 before passing to the next stage.

3.3.3 Twiddle Factor ROM

The complex constant $\mathbf{W^k = e^{-j2\pi k/N}}$ (the twiddle factor) is stored in a ROM.

- **Storage:** The real and imaginary components of $\mathbf{W^k}$ are pre-calculated and stored as 16-bit Q1.14 values.
- **Addressing:** The address to the ROM is determined by the specific stage and the index of the butterfly, ensuring the correct twiddle factor is fed to the complex multiplier at the appropriate clock cycle.

3.4 Component-Level Architecture and Implementation Details

The high-performance of the pipelined FFT is directly attributable to the optimized design of its core functional units, all operating on the Q1.14 fixed-point number format.

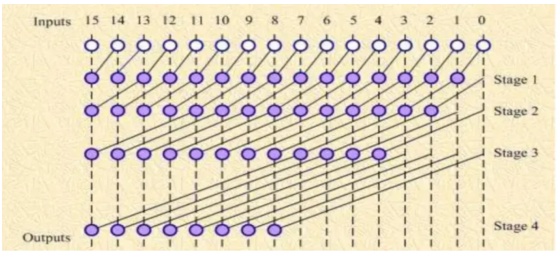

Figure 5: Architecture of Kogge Stone Adder

3.4.1 Kogge-Stone Adder (KSA) Architecture

The Kogge-Stone Adder was selected for all addition and subtraction operations within the Butterfly Unit (BFU) and Complex Multiplier to minimize the critical path delay. Unlike simpler architectures (like Ripple-Carry Adders), the KSA uses a highly parallel prefix network to generate all carry signals simultaneously.

- **Logic Implementation:** The design uses a layered structure of black and gray cells to propagate and generate carry signals across the 32-bit width (as used in the multiplier and adapted for the 16-bit BFU). This structure maximizes fan-out and minimizes the number of logic levels, resulting in a logarithmic delay, $\mathbf{O(\log_2 n)}$, which is crucial for high-speed DSP applications.
- **Functionality:** The KSA serves as the building block for both addition (with carry-in $C_{in} = 0$) and subtraction (using two's complement, $C_{in} = 1$).

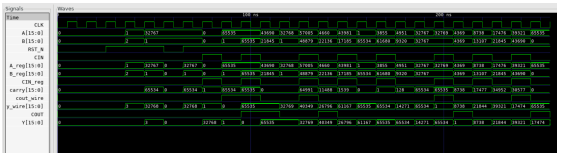

Figure 6: Testing of the Kogge-Stone adder

3.4.2 Radix-2 Butterfly Unit (BFU)

The BFU is responsible for the core calculations $A' = A + W^k B$ and $B' = A - W^k B$. Given that the multiplication by the twiddle factor W^k occurs prior to the BFU in the Decimation-In-Time (DIT) flow, the BFU's primary task is parallel addition and subtraction.

- **Input/Output:** It takes two complex inputs (A and the post-multiplied $W^k B$) and produces two complex outputs (A' and B').

- **Logic Implementation:** The BFU implements the four necessary additions/subtractions using four separate 16-bit KSA instances, one for each real and imaginary path of the two outputs. Subtraction is achieved using the standard two's complement method: the subtrahend's bits are inverted, and the carry-in (C_{in}) to the KSA is set to '1'.

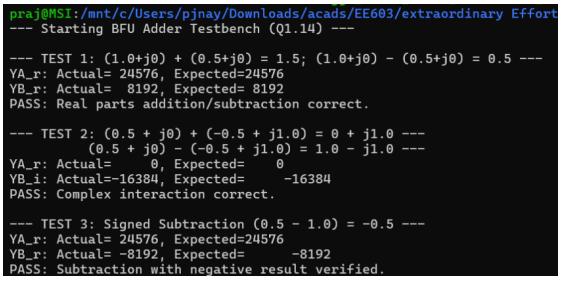

Figure 7: BFU testbench

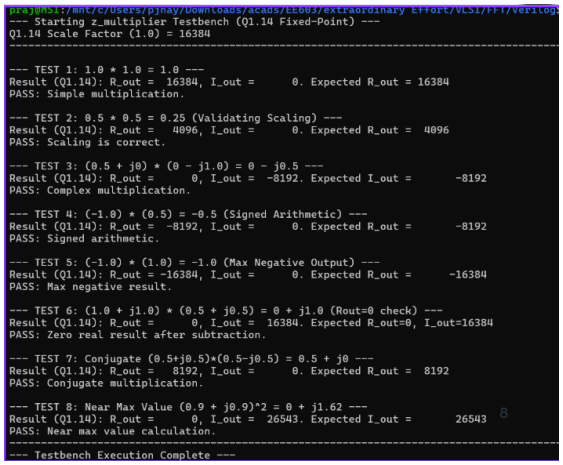

Figure 8: Testing the Complex multiplier

- **Content:** The sine and cosine values for W_{16}^k are pre-calculated and stored as 16-bit signed Q1.14 values. For $N = 16$, the fundamental twiddle factor is $W = e^{-j2\pi/16}$.

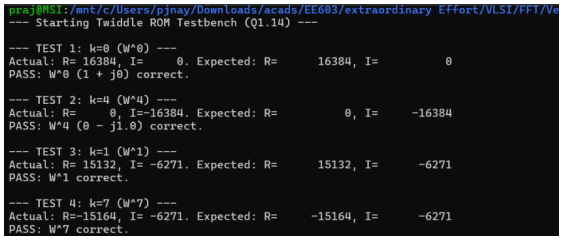

Figure 9: ROM testing

3.4.3 Complex Multiplier (Z-Multiplier)

The Complex Multiplier, or Z-Multiplier, is designed to calculate the product $P = A \cdot B$ where $A = A_r + jA_i$ and $B = B_r + jB_i$. This unit is placed *before* the BFU in the DIT flow, where B is the input from the previous stage and A is the Twiddle Factor W^k.

- **Arithmetic Decomposition:** The unit uses four 16×16 real multipliers to compute the four products: P_{rr}, P_{ii}, P_{ri}, and P_{ir}.
- **Result Calculation:** The real and imaginary outputs are calculated as: $\mathbf{R_{out}} = P_{rr} - P_{ii}$ and $\mathbf{I_{out}} = P_{ri} + P_{ir}$. These final additions/subtractions utilize 32-bit Kogge-Stone Adders to ensure maximum speed.
- **Fixed-Point Scaling:** Since $Q1.14 \times Q1.14$ results in a $Q2.28$ format (32 bits total), the 32-bit sum must be right-shifted by 14 bits ($\gg 14$) to scale the final output back to the required $Q1.14$ format for the next pipeline stage. This truncation step controls precision loss while preventing dynamic range overflow.

3.4.4 Twiddle Factor ROM

The Twiddle Factor ROM stores the required complex constants $W_{N=16}^k$ in the Q1.14 format.

- **Addressing:** The ROM is addressed by a pipeline control logic, which generates the correct 3-bit address k based on the current stage and butterfly index.

Table 3: Twiddle Factor $\mathbf{W_{16}^k}$ Q1.14 Hexadecimal Values

k	Decimal Value	Real	Imaginary
0	1.00000	**4000**	**0000**
1	$0.92388 - j0.38268$	**3B1C**	**E781**
2	$0.70711 - j0.70711$	**2D41**	**D27F**
3	$0.38268 - j0.92388$	**187F**	**C4C4**
4	$0.00000 - j1.00000$	**0000**	**C000**
5	$-0.38268 - j0.92388$	**E781**	**C4C4**
6	$-0.70711 - j0.70711$	**D27F**	**D27F**
7	$-0.92388 - j0.38268$	**C4C4**	**E781**

3.5 Current Status and Future Work Direction

3.5.1 System Integration Status and Debugging

While the individual functional modules—the **Kogge-Stone Adders**, **Butterfly Units (BFU)**, **Complex Multiplier**, and **Twiddle Factor ROM**—have been thoroughly tested and verified at the component level, the fully

pipelined $N = 16$ FFT core is currently under debugging. The architecture is complete, but preliminary simulation results do not yet match the expected high-precision floating-point output ($\mathbf{X_{FP}[k]}$). This suggests a potential issue within the **inter-stage pipeline registers**, the **data addressing/sequencing logic**, or the specific **fixed-point scaling/truncation** at the output of the complex multipliers. The primary effort is now focused on pinpointing this integration or data precision error.

3.5.2 Future Personal Initiative

The successful design and verification of the complex digital signal processing components has been a highly rewarding experience. Following the finalization and verification of the FFT core, I (Prajwal) plan to leverage this expertise in fixed-point arithmetic and pipelined architectures to develop a complete hardware core for an **OFDM (Orthogonal Frequency-Division Multiplexing) Transmitter**. This personal project is scheduled for completion by the end of November 2025.

4 Sin and Cosine Generator using CORDIC

4.1 CORDIC Definition and Principle

The **CORDIC (COordinate Rotation DIgital Computer)** algorithm is an iterative technique designed for hardware-efficient calculation of trigonometric and other elementary functions. It achieves efficiency by performing rotations using only **addition**, **subtraction**, and **bit-shifting**, since the tangent of the elementary rotation angles is restricted to powers of two: $\tan(\alpha_i) = \pm 2^{-i}$.

4.2 Rotation Mode for Sine and Cosine

To generate the sine and cosine of a target angle θ, the CORDIC algorithm is operated in **Rotation Mode**. In this mode, the goal is to rotate an initial vector (x_0, y_0) by the angle θ, which is achieved by driving the angle accumulator z_i to zero.

4.2.1 Initialization

The process is initialized as follows:

- **Initial Vector:** $(x_0, y_0) = (1/K, 0)$. Alternatively, $(1, 0)$ can be used, with the result corrected by $1/K$ at the end.
- **Angle Accumulator:** $z_0 = \theta$.

4.2.2 Iterative Equations

For step $i = 0$ to $N - 1$, the equations are:

$$\begin{cases} x_{i+1} = x_i - \sigma_i y_i 2^{-i} \\ y_{i+1} = y_i + \sigma_i x_i 2^{-i} \\ z_{i+1} = z_i - \sigma_i \alpha_i \end{cases}$$

Where:

- $\alpha_\mathbf{i} = \arctan(\mathbf{2^{-i}})$ is the elementary rotation angle.
- $\sigma_\mathbf{i} = \text{sgn}(\mathbf{z_i})$ is the rotation direction, chosen to minimize z_{i+1}:

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

4.2.3 Final Result

After N iterations, the final vector (x_N, y_N) contains the desired trigonometric values:

$$\begin{cases} x_N = K \cdot \cos(\theta) \\ y_N = K \cdot \sin(\theta) \end{cases}$$

If $x_0 = 1/K$ was used, then $x_N = \cos(\theta)$ and $y_N = \sin(\theta)$.

4.3 The CORDIC Scaling Factor (K)

The iterative equations omit the $\cos(\alpha_i)$ term for hardware simplicity, which causes the vector magnitude to grow. The final results must be divided by the cumulative **Scaling Factor** K, defined as the product of all $\cos(\alpha_i)$ terms:

$$K = \prod_{i=0}^{N-1} \cos(\alpha_i) = \prod_{i=0}^{N-1} \frac{1}{\sqrt{1 + (2^{-i})^2}}$$

For a large number of iterations N, the factor converges to: $K_\infty \approx 0.607253$, and $1/K_\infty \approx 1.64676$.

4.4 Elementary Rotation Angles

The following table provides the values for the elementary rotation angles (α_i) used in the CORDIC algorithm, crucial for the angle accumulation in z_{i+1} which we have used in our code.

i	2^{-i}	$\alpha_i = \arctan(2^{-i})$
0	1	45.000°
1	0.5	26.565°
2	0.25	14.036°
3	0.125	7.125°
4	0.0625	3.576°
5	0.03125	1.7899°
6	0.015625	0.8951°
7	0.0078125	0.4476°
8	0.00390625	0.2238°
9	0.001953125	0.1119°
10	0.0009765625	0.0560°
11	0.00048828125	0.0280°
12	0.000244140625	0.0140°
13	0.0001220703125	0.0070°
14	0.00006103515625	0.0035°
15	0.000030517578125	0.0017°

4.5 Digital Design of the Iteration Counter

For a fixed number of $N = 16$ iterations, the CORDIC implementation requires a dedicated counter to control the sequence of elementary rotations. This counter manages the index i, which dictates the value of the angle α_i and the magnitude of the required bit shift (2^{-i}).

4.5.1 Counter Specification

The counter is a simple finite state machine which has 16 states, and once it starts, it keeps changing states at each clock positive edge. The states are described as a moore machine.

Purpose: The counter tracks the iteration

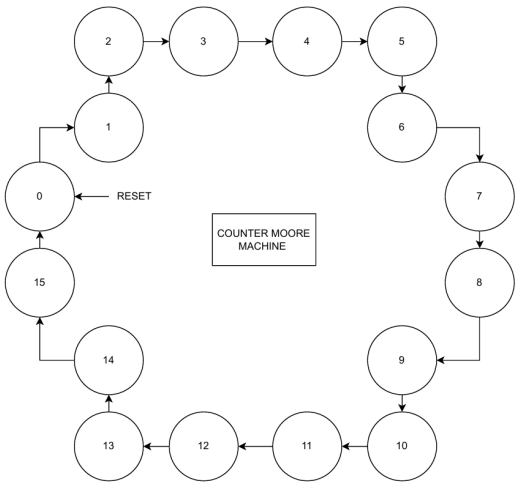

Figure 10: State Machine for Counter

number i, from $i = 0$ to $i = 15$.

Type: A 4-bit synchronous up-counter is required, as $2^4 = 16$ states are needed (0000 to 1111).

Output: The counter output, $\mathbf{I} = (\mathbf{i_3 i_2 i_1 i_0})$, directly serves as the shift amount for the bit-shift operations.

Control Signal: The counter is clocked after each CORDIC step is completed. A final state of $\mathbf{i = 15}$ (1111) is used to generate a DONE signal, indicating the completion of the 16 iterations.

4.6 Counter State Transition Logic

The CORDIC iteration counter is a 4-bit synchronous up-counter. Its state variables are $I = (i_3 i_2 i_1 i_0)$, which track the iteration number i from 0 to 15.

4.6.1 Truth Table for 4-Bit Up-Counter

The counter implements a modulo 16 sequence, where the next state (i') is the present state (i) incremented by one.

Table 4: State Transition Truth Table for the 4-bit Counter

i	**Present State**				**Next State**			
State	$\mathbf{i_3}$	$\mathbf{i_2}$	$\mathbf{i_1}$	$\mathbf{i_0}$	$\mathbf{i'_3}$	$\mathbf{i'_2}$	$\mathbf{i'_1}$	$\mathbf{i'_0}$
0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	1	0
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	1	0	0
4	0	1	0	0	0	1	0	1
5	0	1	0	1	0	1	1	0
6	0	1	1	0	0	1	1	1
7	0	1	1	1	1	0	0	0
8	1	0	0	0	1	0	0	1
9	1	0	0	1	1	0	1	0
10	1	0	1	0	1	0	1	1
11	1	0	1	1	1	1	0	0
12	1	1	0	0	1	1	0	1
13	1	1	0	1	1	1	1	0
14	1	1	1	0	1	1	1	1
15	1	1	1	1	0	0	0	0

4.6.2 Karnaugh Maps and Boolean Expressions

The minimized Boolean expressions for the next state variables ($\mathbf{i'_k}$) of a synchronous counter follow a standard pattern, where each bit toggles based on the AND combination of all less significant bits.

Next State $\mathbf{i'_0}$: The $\mathbf{i'_0}$ toggles on every clock edge.

$$\mathbf{i'_0 = \overline{i_0}}$$

Next State $\mathbf{i_1'}$: The $\mathbf{i_1'}$ toggles when $\mathbf{i_0}$ is high.

$$\mathbf{i_1'} = \mathbf{i_1}\overline{\mathbf{i_0}} + \overline{\mathbf{i_1}}\mathbf{i_0} = \mathbf{i_1} \oplus \mathbf{i_0}$$

Next State $\mathbf{i_2'}$: The $\mathbf{i_2'}$ toggles when $\mathbf{i_1}$ AND $\mathbf{i_0}$ are both high.

$$\mathbf{i_2'} = \mathbf{i_2}\overline{(\mathbf{i_1}\mathbf{i_0})} + \overline{\mathbf{i_2}}(\mathbf{i_1}\mathbf{i_0}) = \mathbf{i_2} \oplus (\mathbf{i_1}\mathbf{i_0})$$

Next State $\mathbf{i_3'}$: The $\mathbf{i_3'}$ toggles when $\mathbf{i_2}$ AND $\mathbf{i_1}$ AND $\mathbf{i_0}$ are all high.

$$\mathbf{i_3'} = \mathbf{i_3}\overline{(\mathbf{i_2}\mathbf{i_1}\mathbf{i_0})} + \overline{\mathbf{i_3}}(\mathbf{i_2}\mathbf{i_1}\mathbf{i_0}) = \mathbf{i_3} \oplus (\mathbf{i_2}\mathbf{i_1}\mathbf{i_0})$$

4.6.3 Summary of Counter Logic

The complete logic for the 4-bit synchronous up-counter using D-flip-flops (where $D_k = i_k'$) is:

$$\begin{cases} D_0 = \overline{\mathbf{i_0}} \\ D_1 = \mathbf{i_1} \oplus \mathbf{i_0} \\ D_2 = \mathbf{i_2} \oplus (\mathbf{i_1}\mathbf{i_0}) \\ D_3 = \mathbf{i_3} \oplus (\mathbf{i_2}\mathbf{i_1}\mathbf{i_0}) \end{cases}$$

4.7 CORDIC Algorithm Control: Finite State Machine (FSM) Plan

To manage the sequential operation of the 16 CORDIC iterations, quadrant mapping, and final output, a 4-state synchronous Finite State Machine (FSM) is employed. The state variables are $\mathbf{S_1S_0}$, encoded as 2 bits.

4.7.1 FSM States and Encoding

The FSM operates with the following four states:

- **IDLE (00):** The default waiting state. The machine waits for an external trigger to begin computation.
- **INITIALISE (01):** Setup state. Here, the counter is reset, initial register values are loaded, and the input angle is prepared.
- **ROTATE (10):** Iterative state. The machine performs the 16 CORDIC rotations, controlled by the iteration counter.
- **POST_PROCESS (11):** Finalization state. The output values are scaled and presented, and the output valid signal is asserted.

4.7.2 Input and Output Signals

The FSM's behavior is governed by the following control signals:

- **Inputs:**
  - **i_valid**: External input trigger (active high) to start the process.
  - **done**: Internal signal from the 4-bit counter (active high at $i = 15$) indicating completion of 16 rotations.
- **Outputs:**
  - **o_valid**: Output signal (active high) indicating that the final X and Y results are ready.
  - **counter_reset**: Signal to reset the iteration counter to $i = 0$.
  - **counter_enable**: Signal to allow the counter to increment on the clock edge.
  - **quadrant_map_en**: Enable signal for the quadrant mapper logic.

4.7.3 State Transition Logic and Operation

1. **IDLE (00) → INITIALISE (01):**
   - **Condition: i_valid** is high.
   - **Action:** The FSM moves to the **INITIALISE** state.
2. **INITIALISE (01) → ROTATE (10):** (1-Cycle Setup)
   - **Action:** Within this single cycle, the **counter_reset** and **quadrant_map_en** signals are asserted. The quadrant mapper converts the input θ to an angle in $[0, \pi/2]$ and determines the final signs for $\sin(\theta)$ and $\cos(\theta)$. The initial vector (X_0, Y_0, Z_0) is loaded.
   - **Transition:** The state automatically transitions to **ROTATE** on the next clock edge.
3. **ROTATE (10) → POST_PROCESS (11):**
   - **Condition:** The **done** signal from the counter is high.
   - **Action:** The **counter_enable** signal is asserted, allowing i to iterate from 0 to 15. In each cycle, the iterative equations are computed, utilizing the $\arctan(2^{-i})$ from the ROM, based on the current counter value i.

4. **POST_PROCESS (11) → IDLE (00):** (1-Cycle Output)
    - **Action:** The **o_valid** signal is asserted for one clock cycle, presenting the final scaled X and Y register contents as the output $\cos(\theta)$ and $\sin(\theta)$.
    - **Transition:** The state automatically transitions back to **IDLE** on the next clock edge to await a new **i_valid** trigger.

4.7.4 State Transition Table (Logic)

The table summarizes the transition logic for the next state variables ($\mathbf{S_1'S_0'}$) based on the present state ($\mathbf{S_1S_0}$) and the key inputs ($\mathbf{i\_valid}, \mathbf{done}$).

Table 5: CORDIC FSM State Transition Table

State		Inputs		Next	
$\mathbf{S_1}$	$\mathbf{S_0}$	**i_valid**	**done**	$\mathbf{S_1'}$	$\mathbf{S_0'}$
0	0	0	X	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	0	X	0	1	0
1	0	X	1	1	1
1	1	X	X	0	0

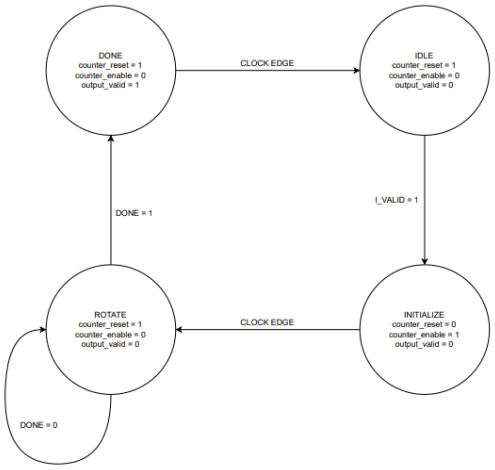

Figure 11: Final State Machine of the System

4.8 Verilog Implementation and Pipelining

Hardware realization of the iterative CORDIC structure.

The provided block diagram illustrates the full digital architecture of the CORDIC sine and cosine generator. Key to this implementation is

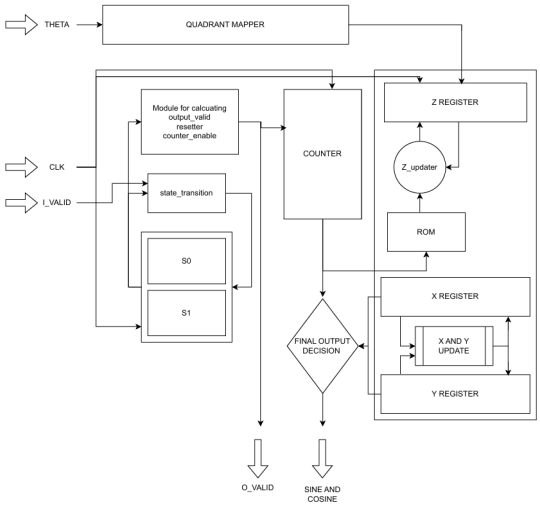

Figure 12: Circuit Diagram for CORDIC sine and cosine value generation

the use of a **fixed-point number system** for all arithmetic operations.

4.8.1 Q3.12 Fixed-Point Format

All data paths (X, Y, and Z registers, α_i values, and intermediate results) are implemented using a **Q3.12** fixed-point format. This format uses a total of 16 bits:

- 3 bits for the integer part (including the sign bit).
- 12 bits for the fractional part.
- 1 bit is typically unused or implicit, depending on the exact implementation, but the core precision is 12 fractional bits.

This format allows the representation of numbers in the range $(-\mathbf{4}, \mathbf{4})$ with a fractional resolution of $2^{-12} \approx 0.000244$. This resolution is sufficient for 16 CORDIC iterations, as the final angle α_{15} is $\approx 0.0017°$. The number N in this format is calculated as:

$$\text{Decimal Value} = \frac{\text{Binary Value}}{\mathbf{2^{12}}}$$

4.8.2 Module Interconnections

The major modules interact as follows:

1. **Quadrant Mapper → Z Register:** The Mapper converts the input angle $\mathbf{\Theta}$ to a primary angle $\mathbf{Z_0} \in [\mathbf{0}, \pi/\mathbf{2}]$ and loads it into the Z Register.
2. **Counter → ROM:** The **i** output from the **COUNTER** (from 0 to 15) serves as the address for the **ROM** (Read-Only Memory).

3. **ROM → Z Updater:** The ROM outputs the required elementary angle $\alpha_{\mathbf{i}} = \arctan(\mathbf{2^{-i}})$, which is used by the **Z Updater** (an adder/subtractor) to calculate Z_{i+1}.

4. **Counter → X and Y Update:** The counter output i controls the **bit-shifting** operation within the **X and Y Update** module to implement the 2^{-i} term efficiently.

5. **FSM Logic → Global Control:** The FSM's control unit (**state_transition** and **Module for calculating output**) generates the **counter_enable**, **counter_reset**, and **O_VALID** signals based on the state ($\mathbf{S_1S_0}$) and inputs ($\mathbf{I\_VALID}, \mathbf{done}$).

4.9 Convergence Analysis and Results for $\theta = 30^\circ$

The simulation trace shows the iterative calculation for an initial angle of 30° (after quadrant mapping and scaling). The goal of the Rotation Mode is to drive the Z register to zero.

4.9.1 Convergence Demonstration

The Z register converges as the iterations proceed:

- **Start (Cyc 3, i=0): $\mathbf{Z_{reg}} = \mathbf{0861_{hex}}$.**
- **Midpoint (Cyc 8, i=5): $\mathbf{Z_{reg}} = \mathbf{0056_{hex}}$.** The magnitude has dropped significantly.
- **End (Cyc 15, i=c/12): $\mathbf{Z_{reg}} = \mathbf{0000_{hex}}$.** The angle accumulator successfully converges to zero after 12 iterations, and remains near zero through iteration 15 ($\mathbf{i = f}$).

The convergence is verified by observing the $\mathbf{Z_{reg}}$ value settling to 0 (or $\mathbf{FFFF_{hex}}$ which is $\approx -2^{-12}$). This confirms that the vector has been rotated by the desired angle θ.

4.9.2 Final Results and Verification

The final output is taken from X_{reg} and Y_{reg} at the end of the rotation phase (e.g., Cycle 19, $i = 15$):

- $\mathbf{X_{reg}} = 0DD7_{hex}$
- $\mathbf{Y_{reg}} = 0805_{hex}$

Since the implementation likely uses the scaled initial vector approach (or the final output needs scaling), the values represent $\mathbf{K} \cdot \cos(\mathbf{30}^\circ)$ and $\mathbf{K} \cdot \sin(\mathbf{30}^\circ)$.

Conversion to Decimal: We use the Q3.12 format rule: Decimal Value = Binary Value/2^{12}.

1. **Final X (Cosine):**

$$X_{Decimal} = \frac{0DD7_{hex}}{2^{12}} = \frac{3543}{4096} \approx \mathbf{0.86500}$$

2. **Final Y (Sine):**

$$Y_{Decimal} = \frac{0805_{hex}}{2^{12}} = \frac{2053}{4096} \approx \mathbf{0.50122}$$

Comparison to Exact Values: The exact trigonometric values are: $\cos(\mathbf{30}^\circ) \approx 0.86603$ and $\sin(\mathbf{30}^\circ) = 0.50000$.

The CORDIC implementation has produced results very close to the exact values, demonstrating the accuracy of the 16-iteration, Q3.12 fixed-point design:

$$\text{Error}_{\cos} \approx 0.86603 - 0.86500 = 0.00103$$

$$\text{Error}_{\sin} \approx 0.50122 - 0.50000 = 0.00122$$

—

4.10 Python Code for Convergence Plot

To visually demonstrate the convergence of the Z register, we plotted the Z register value versus the iteration count i.

```
import numpy as np
import matplotlib.pyplot as plt

# --- Data from the CORDIC Trace (i=0 to
    i=15) ---
# Clock Cycle (Cyc) values from the
    trace
cycle_values = np.array([2, 3, 4, 5, 6,
    7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
     17, 18, 19, 20, 21])
# Hex values for X, Y, and Z registers
    from the trace
x_reg_hex = np.array(['0000', '09b7', '
    09b7', '0e92', '0d5b', '0e6b', '0dfe
    ', '0dc1', '0de1', '0dd2', '0dda', '
    0dd6', '0dd8', '0dd7', '0dd7', '0dd7
    ', '0dd7', '0dd7', '0dd7', '0dd7'])
y_reg_hex = np.array(['0000', '0000', '
    09b7', '04dc', '0880', '06d5', '07bb
    ', '082a', '07f3', '080e', '0801', '
    0807', '0804', '0805', '0805', '0805
    ', '0805', '0805', '0805', '0805'])
z_reg_hex = np.array(['0000', '0861', '
    fbd0', '0343', 'ff58', '0155', '0056
    ', 'ffd6', '0016', 'fff6', '0006', '
    fffe', '0002', '0000', 'ffff', 'ffff
    ', 'ffff', 'ffff', 'ffff', 'ffff'])

# --- Q3.12 Conversion Function (16-bit
    signed) ---
def hex_to_q3_12(hex_str):
    """Converts a 16-bit hex string to a
     Q3.12 fixed-point decimal value."""
```

```
    val = int(hex_str, 16)
    # Check for sign (MSB of the 16-bit
    number: 0x8000)
    if val & 0x8000:
        # If signed, perform two's
    complement by subtracting 2^16 (0
    x10000)
        val = val - 0x10000
    # Divide by 2^12 (4096.0) to get the
     decimal value
    return val / 4096.0

# --- Convert Hex data to Decimal Q3.12
    ---
x_reg_decimal = np.array([hex_to_q3_12(h
    ) for h in x_reg_hex])
y_reg_decimal = np.array([hex_to_q3_12(h
    ) for h in y_reg_hex])
z_reg_decimal = np.array([hex_to_q3_12(h
    ) for h in z_reg_hex])

# --- Plotting ---
fig, axes = plt.subplots(3, 1, figsize
    =(10, 12), sharex=True)
fig.suptitle('CORDIC Register
    Convergence Over Clock Cycles (
    Target $\\theta = 30^\circ$)',
    fontsize=16)

# 1. X Register Convergence (Final
    Cosine)
axes[0].plot(cycle_values, x_reg_decimal
    , marker='o', linestyle='-', color='
    blue')
axes[0].axhline(np.cos(np.deg2rad(30)),
    color='r', linestyle='--', linewidth
    =1, label='Target Cos(30°)')
axes[0].set_ylabel('$X_{reg}$ (Cosine
    Value)', fontsize=12)
axes[0].grid(True, which='both',
    linestyle='--', linewidth=0.5)
axes[0].legend(loc='lower right')
axes[0].set_title('X Register
    Convergence')

# 2. Y Register Convergence (Final Sine)
axes[1].plot(cycle_values, y_reg_decimal
    , marker='o', linestyle='-', color='
    green')
axes[1].axhline(np.sin(np.deg2rad(30)),
    color='r', linestyle='--', linewidth
    =1, label='Target Sin(30°)')
axes[1].set_ylabel('$Y_{reg}$ (Sine
    Value)', fontsize=12)
axes[1].grid(True, which='both',
    linestyle='--', linewidth=0.5)
axes[1].legend(loc='lower right')
axes[1].set_title('Y Register
    Convergence')

# 3. Z Register Convergence (Angle
    Accumulator)
axes[2].plot(cycle_values, z_reg_decimal
    , marker='o', linestyle='-', color='
    red')
axes[2].axhline(0, color='k', linestyle=
    '--', linewidth=1, label='Target $Z
    =0$')
axes[2].set_ylabel('$Z_{reg}$ (Angle
    Error)', fontsize=12)
axes[2].set_xlabel('Clock Cycle',
    fontsize=12)
axes[2].grid(True, which='both',
    linestyle='--', linewidth=0.5)
axes[2].legend(loc='lower right')
axes[2].set_title('Z Register
    Convergence')

plt.xticks(cycle_values)
plt.tight_layout(rect=[0, 0.03, 1,
    0.96])
plt.show()

plt.savefig('cordic_convergence_plots.
    png')
```

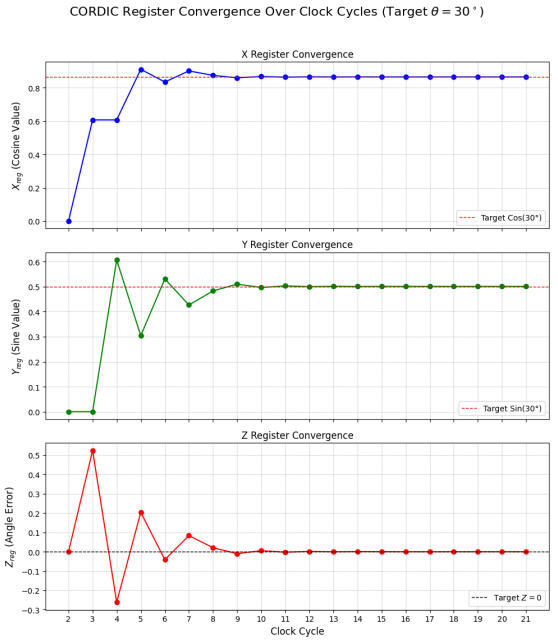

Figure 13: Convergence of X and Y to cosine and sine values and Z to zero

5 Conclusion

This DSP kit implementation successfully realized three critical signal processing blocks—the 8-tap FIR filter, the 16-input FFT, and the CORDIC generator—as dedicated hardware cores using Verilog HDL.. This project not only validated the functional correctness of complex DSP algorithms in hardware but also provided

```
VCD info: dumpfile cordic_debug_essential.vcd opened for output.
--- CORE CORDIC TRACE (Angle: 30 deg, FP: 0861) ---
Cyc| i | X_reg | Y_reg | Z_reg | Sigma | Alpha | shifted_x | shifted_y | Xb | Yb
 2| 0 | 0000 | 0000 | 0000 |  1 | f36e |  0000 |  0000 | 0000| ffff
 3| 0 | 09b7 | 0000 | 0861 |  1 | f36e |  09b7 |  0000 | 09b7| ffff
 4| 1 | 09b7 | 09b7 | fbd0 |  0 | f88c |  04db |  04db | fb24| 04db
 5| 2 | 0e92 | 04dc | 0343 |  1 | fc14 |  03a4 |  0137 | 03a4| fec8
 6| 3 | 0d5b | 0880 | ff58 |  0 | fe02 |  01ab |  0110 | fe54| 0110
 7| 4 | 0e6b | 06d5 | 0155 |  1 | ff00 |  00e6 |  006d | 00e6| ff92
 8| 5 | 0dfe | 07bb | 0056 |  1 | ff7f |  006f |  003d | 006f| ffc2
 9| 6 | 0dc1 | 082a | ffd6 |  0 | ffbf |  0037 |  0020 | ffc8| 0020
10| 7 | 0de1 | 07f3 | 0016 |  1 | ffdf |  001b |  000f | 001b| ff f0
11| 8 | 0dd2 | 080e | fff6 |  0 | ffef |  000d |  0008 | fff2| 0008
12| 9 | 0dda | 0801 | 0006 |  1 | fff7 |  0006 |  0004 | 0006| fffb
13| a | 0dd6 | 0807 | fffe |  0 | fffb |  0003 |  0002 | fffc| 0002
14| b | 0dd8 | 0804 | 0002 |  1 | fffd |  0001 |  0001 | 0001| fffe
15| c | 0dd7 | 0805 | 0000 |  1 | fffe |  0000 |  0000 | 0000| ffff
16| d | 0dd7 | 0805 | ffff |  0 | ffff |  0000 |  0000 | ffff| 0000
17| e | 0dd7 | 0805 | ffff |  0 | ffff |  0000 |  0000 | ffff| 0000
18| f | 0dd7 | 0805 | ffff |  0 | ffff |  0000 |  0000 | ffff| 0000
19| f | 0dd7 | 0805 | ffff |  0 | ffff |  0000 |  0000 | ffff| 0000
20| f | 0dd7 | 0805 | ffff |  0 | ffff |  0000 |  0000 | ffff| 0000
21| f | 0dd7 | 0805 | ffff |  0 | ffff |  0000 |  0000 | ffff| 0000
```

Figure 14: Result

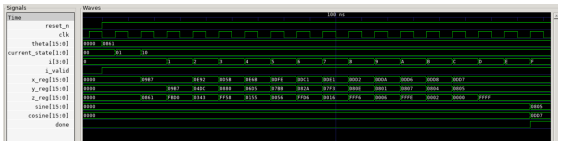

Figure 15: waveform for Angle=30 degrees

Table 6: CORDIC Core Trace Data (Simplified Hexadecimal Q3.12 Format)

Cyc	**i**	**X_reg**	**Y_reg**	**Z_reg**	σ	α
2	0	0000	0000	0000	1	f36e
3	0	09b7	0000	0861	1	f36e
4	1	09b7	09b7	fbd0	0	f88c
5	2	0e92	04dc	0343	1	fc14
6	3	0d5b	0880	ff58	0	fe02
7	4	0e6b	06d5	0155	1	ff00
8	5	0dfe	07bb	0056	1	ff7f
9	6	0dc1	082a	ffd6	0	ffbf
10	7	0de1	07f3	0016	1	ffdf
11	8	0dd2	080e	fff6	0	ffef
12	9	0dda	0801	0006	1	fff7
13	a	0dd6	0807	fffe	0	fffb
14	b	0dd8	0804	0002	1	fffd
15	c	0dd7	0805	0000	1	fffe
16	d	0dd7	0805	ffff	0	ffff
17	e	0dd7	0805	ffff	0	ffff
18	f	0dd7	0805	ffff	0	ffff
19	f	0dd7	0805	ffff	0	ffff
20	f	0dd7	0805	ffff	0	ffff
21	f	0dd7	0805	ffff	0	ffff

valuable insight into the practical challenges and optimization techniques required for high-performance VLSI design in real-world DSP applications