UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Department of Information Engineering

Master Degree in Electronic Engineering

Design of 32-Bit Floating Point CORDIC Co-Processor

Master's Candidate

Shakil Hosen

2070907

Supervisor

Prof. Daniele Vogrig

University of Padova

Academic Year

2024/2025

To my Family and Friends

Abstract

This thesis centres on the design of a CORDIC co-processor tailored for the 32-bit floating-point data format, specifically to compute sine and cosine values. The co-processor complies with the IEEE-754 standard for single-precision floating-point numbers and employs the CORDIC algorithm, which necessitates multiple additions and multiplications per iteration. A specialized floating-point adder has been created to implement the CORDIC algorithm and a binary subtractor has been used for the multiplication operation. Various floating point adder architectures have been evaluated to identify the most optimized structure. The final proposal includes both pipelined and serial architectures, synthesized for 130nm (UMC) and 22nm (Global Foundry) technology nodes. The co-processor core is modelled in VHDL, providing a robust solution for efficient computation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The CORDIC (Coordinate Rotation Digital Computer) co-processor is a hardware-based implementation of the CORDIC algorithm[1], an iterative method developed to compute a wide range of mathematical functions. These include trigonometric, hyperbolic, and logarithmic functions, as well as various arithmetic operations such as calculating vector magnitudes and converting between polar and rectangular coordinates. The algorithm's strength lies in its use of simple operations like shifts, additions, and subtractions, rather than complex multiplications or divisions, which make it particularly well-suited for low power and cost-effective hardware implementation. This lucrative characteristic allow scientist, integrated circuit designer and researcher to use CORDIC in Signal processing [2], Biomedical imaging [3], Robotics and control [4], Communications [5], Encryption and Decryption [6] etc. This project focuses specifically on the design of a floating-point CORDIC (FP-CORDIC) unit to generate sine and cosine functions in rotation mode. Such a design can be effectively implemented as a high-speed direct digital synthesizer (DDS), as reported in [7] and [8].

The CORDIC processor has been specifically designed to handle floating-point data encoded in compliance with the IEEE 754 single-precision standard. The IEEE 754 floating-point number system is a globally recognized standard for representing and processing real numbers in digital computing. Established by the Institute of Electrical and Electronics Engineers (IEEE) in 1985, called IEEE754-1985 standard[9]. The latest version of this standard is IEEE754-2019[10]. This IEEE standard adopted by the International standard organisation (ISO) as ISO/IEC 60559:2020. The IEEE754 standard format offers a uniform and efficient framework for handling a vast spectrum of numerical values, from tiny fractions to extremely large numbers. This system strikes a balance between **precision**, **range**, and **computational performance**, making it indispensable in fields like scientific computing, graphics processing etc.

The sections in this chapter are organized as follows:

- **Section 1.1** discusses the arithmetic of the CORDIC algorithm.
- **Section 1.2** is dedicated to describing the IEEE 754-2019 floating-point number format.
- **Section 1.3** explores one of the figures of merit, specifically the precision of floating-point CORDIC.

1.1 CORDIC Algorithm

CORDIC algorithm[1] derived from 2D rotation transformation equation as follows,

$$x_R = x_A \cos\theta - y_A \sin\theta \qquad (1.1)$$

$$y_R = x_A \sin\theta + y_A \cos\theta \qquad (1.2)$$

The equation 1 and 2 can write in matrix form as,

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = R(\theta) \begin{bmatrix} x_A \\ y_A \end{bmatrix} \qquad (1.3)$$

Were, rotational matrix, $R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$

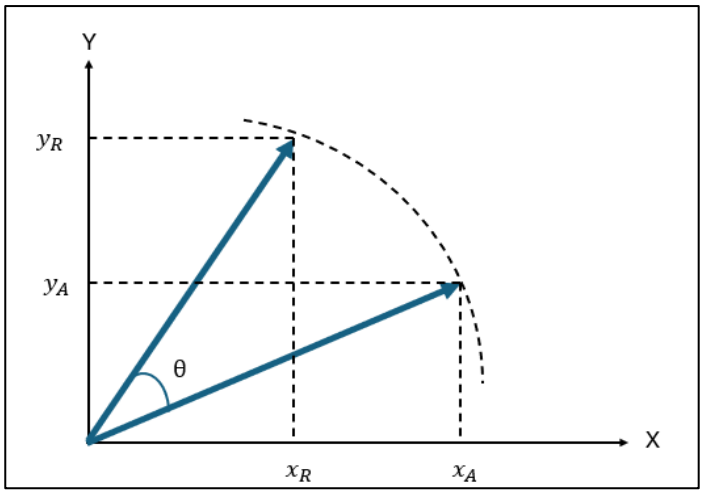

Figure 1.1: Rotation of the vector and corresponding coordinates

The angle of the rotation θ can be divided into sequence of micro angle φ_k such that

$$\theta = \sum_{k=0}^{N-1} \varphi_k$$

$$R(\theta) = \prod_{k=0}^{N-1} R(\varphi_k)$$

So, equation 1.1 and 1.2 can write as,

$$x_{k+1} = x_k \cos\varphi_k - y_k \sin\varphi_k \qquad (1.3)$$

$$y_{k+1} = x_k \sin\varphi_k + y_k \cos\varphi_k \qquad (1.4)$$

Factorizing the $\cos\varphi_k$, we find

$$x_{k+1} = \cos\varphi_k \,(x_k - y_k \tan\varphi_k) \qquad (1.5)$$

$$y_{k+1} = \cos\varphi_k \,(x_k \tan\varphi_k + y_k) \qquad (1.6)$$

The micro angle φ_k need to choose such that,

$$\tan\varphi_k = 2^{-k}$$

$$\varphi_k = \tan^{-1}(2^{-k})$$

This choice enhances the power of the CORDIC algorithm by enabling the shifter to take the place of the multiplier. However, it requires the values of φ_k to be precomputed and stored. The vector undergoes iterative rotations, and during the kth iteration, it rotates by an angle φ_k. This process continues until the cumulative rotation of the vector equals the desired angle θ (Figure1.2).

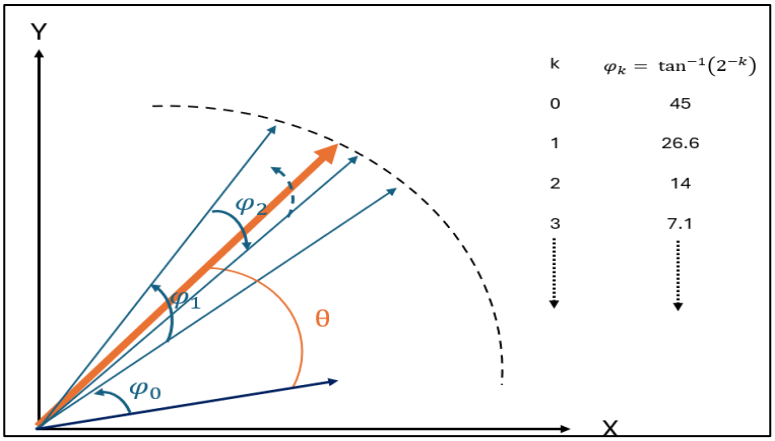

Figure 1.2: Iterative rotation of the vector with angle φ_k

But still the multiplication required due to the cos term. Normalizing the equation 1.5 and 1.6 with respect to $\cos\varphi_k$ and it found

$$\tilde{x}_{k+1} = (\tilde{x}_k - \tilde{y}_k d_k 2^{-k}) \qquad (1.7)$$

$$y_{k+1} = (\tilde{x}_k d_k 2^{-k} + \tilde{y}_k) \qquad (1.8)$$

Were, $d_k = \begin{cases} -1, & \varphi_k < 0 \ \text{(i.e Clockwise rotation)} \\ +1, & \varphi_k \geq 0 \ \text{(i.e counterClockwise rotation)} \end{cases}$

Due to the scaling the vector rotation is not purely circular i.e. the magnitude of the rotated vector is not constant (Figure 1.3). For N rotation extension, the vector magnitude,

$$M_N = M_0 \prod_{k=0}^{N-1} \frac{1}{\cos\varphi_k} = M_0 \prod_{k=0}^{N-1} \sqrt{(1 + 2^{-2k})} \qquad (1.9)$$

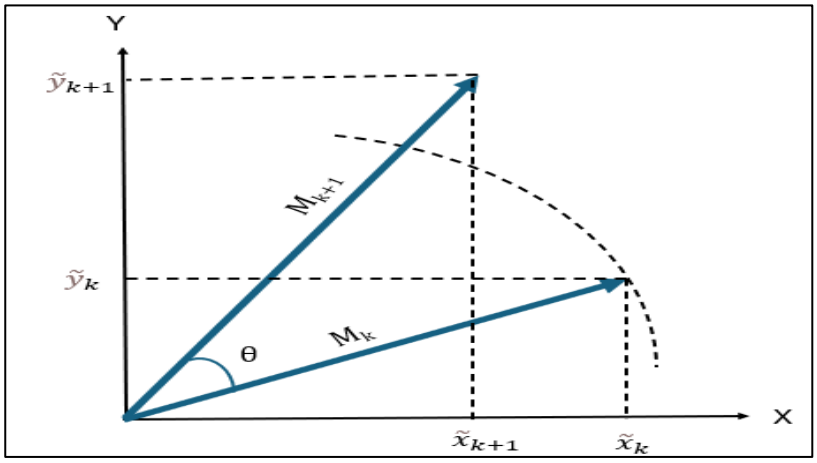

Figure 1.3: Rotation extension(pseudo-rotation) of vector

To get the 100% accurate value it requires infinite iterations. So, the asymptotic value of the magnitude scaling factor (sometimes called CORDIC gain[11]) is,

$$K = \lim_{N\to\infty} \prod_{k=0}^{N-1} \sqrt{(1 + 2^{-2k})} = 1.6468$$

Only a few iterations are needed for the scaling factor to converge closely to the constant *K*. Table 1.1 displays the values of K_N for various iterations.

Table 1-1: CORDIC gain for iterations 1 to 15

Iteration (N)	*Gain(K_N)*
1	1.414213562373095
2	1.581138830084190
3	1.629800601300662
4	1.642484065752237
5	1.645688915757255
6	1.646492278712479
7	1.646693254273644
8	1.646743506596901
9	1.646756070204878
10	1.646759211139820
11	1.646759996375618
12	1.646760192684695
13	1.646760241761973
14	1.646760254031292
15	1.646760257098622

The algorithm can be operated in two modes

- Rotation mode
- Vectoring mode

1.1.1 Rotation Mode

In the rotation mode the vector is rotate over a targeted angle. During each iteration it perform,

$$x_{k+1} = x_k - y_k d_k 2^{-k} \quad (1.10)$$

$$y_{k+1} = x_k d_k 2^{-k} + y_k \quad (1.11)$$

$$z_{k+1} = z_k - d_k \varphi_k \quad (1.12)$$

$$d_k = \begin{cases} -1, & z_k < 0 \\ +1, & z_k \geq 0 \end{cases} \quad (1.13)$$

I. The operation starts with loading initial vector coordinates in (x_0, y_0) and angle θ in z_0

II. Compute (x_{k+1}, y_{k+1}, z_{k+1})

III. Stop when z_{k+1}=0 (or $z_{k+1} < \varepsilon$) else go to 2

After N iteration we get,

$$x_N = K_N (x_0 \cos\theta - y_0 \sin\theta) \quad (1.14)$$

$$y_N = K_N (x_0 \sin\theta + y_0 \cos\theta) \quad (1.15)$$

$$z_N = 0 \ (\text{or } z_N < \varepsilon) \quad (1.16)$$

In rotation mode, the algorithm can be used to rotate a vector to an angle, convert from polar (r, θ) to cartesian coordinate and compute sin and cosine of an angle. In this work, the CORDIC processor has designed to compute sin and cosine i.e. it's the implementation of CORDIC algorithm in rotational mode. To simulate and verify we need to start with the initial conditions,

$$x_0 = \frac{1}{K}\ , \qquad y_0 = 0, \qquad z_0 = 0$$

After N iteration we get,

$$x_N = \cos\theta \qquad (1.17)$$

$$y_N = \sin\theta \qquad (1.18)$$

1.1.2 Vectoring Mode

In vectoring mode, vector (x_0, y_0) is rotate so that y_k reduced to zero. In this mode the operation on x, y and z is same as the equation 1.10, 1.11 and 1.12 respectively a but d_k change as,

$$d_k = \begin{cases} 1, & z_k \le 0 \\ -1, & y_k > 0 \end{cases} \qquad (1.19)$$

I. Start with leading initial vector coordinate (x_0, y_0) and 0 in z_0

II. Compute (x_{k+1}, y_{k+1}, z_{k+1})

III. Stop when y_{k+1}=0 (or $y_{k+1} < \varepsilon$) else go to 2

The CORDIC algorithm in rotation mode can be used to compute the magnitude and angle of a vector and inverse trigonometric functions.

1.2 IEEE754-2019 (Floating-point) Format

The IEEE 754-2019[10] standard defines three binary floating-point formats with lengths of 32, 64, and 128 bits. In this project, the design focuses on the 32-bit single-precision format. To encode a real number, it is represented in the form:

$$(-1)^{sign} \times 2^{exponent} \times significand$$

The encoding maps the number into a 32-bit binary string. The leftmost bit represents the **sign (s)**, where 0 indicates a positive number and 1 indicates a negative number. The next 8 bits are dedicated to the **biased exponent (E)**, which is calculated by adding 127 to the actual exponent (i.e., E=exponent+127E=exponent+127) to avoid the need for a separate sign bit for the exponent. The exponent field (E) ranges from 1 to 254, representing all finite numbers except zero. The exponent values 0 and 255 are reserved for special cases: 0 represents **zero**, and 255 represents **infinity (∞)**. Table 1.2 shows the encoding of the special values.

The **significand** (also called the mantissa) is normalized by removing the implicit leading 1 before the radix point (binary point). The remaining fractional part of the significand is converted into a 23-bit binary number, which forms the **mantissa** of the floating-point number. This mantissa is represented by the last 23 bits of the 32-bit string.

In summary, the 32-bit single-precision floating-point format consists of[10]:

- **1 bit** for the sign (s),
- **8 bits** for the biased exponent (E),
- **23 bits** for the mantissa (M) (fractional part of the significand).

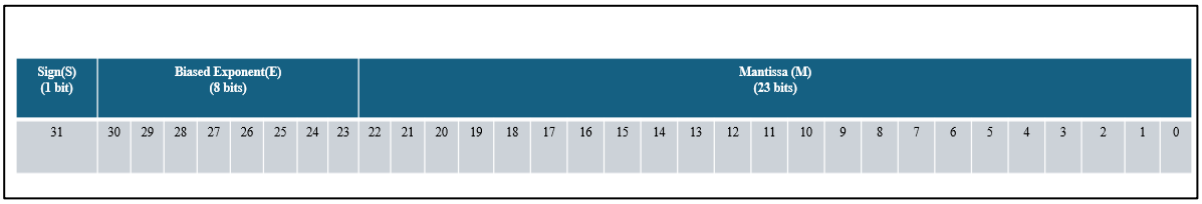

Figure 1.4: 32-bits (single precision) floating-point format

Table 1-2: Special Values

Special Value	Sign	Exponent	Mantissa
+0	0	00000000	00000000000000000000000
-0	1	00000000	00000000000000000000000
+∞	0	11111111	00000000000000000000000
-∞	1	11111111	00000000000000000000000
Not a Number (NaN)	×	1111111	00000000000000000000001 to 11111111111111111111111

1.3 CORDIC Algorithm Precision for Floating-point Data

Precision refers to the level of detail or exactness with which a value is represented or measured. In case of numerical computations, precision refers to the number of accurate digits (decimal precision) or bits (binary precision) after the radix point. For single precision (32-bits) floating-point number the precision is 24 (including hidden 1 of the mantissa) [10]. The resolution of floating-point number termed as **interval machine epsilon (ε)**. Which define as if we represent any real number between 2^n to 2^{n+1} where $n \in \mathbb{N}$**,** the minimum interval of two number that can represent identically in single precision IEEE754 is 2^{n-1}. For example, real numbers between 1 and 2 the decimal precision is 2^{-23} and numbers between 2 and 4 the decimal precision is 2^{-22}. Besides the smallest possible representable normalized number is about 1.18×10^{-38}. To represent the number smaller than that with reliable accuracy the subnormal version of this number format can be use. But in this project the CORDIC and corresponding building blocks

is designed only for the normal range because limited number of iterations has been used which use only small portion of the huge range ($\pm 3.402823 \times 10^{38}$) of the number system.

In case of CORDIC, to get the M bit precision k iterations required where k ≥ M [12]. As So, Using IEEE754 format we can reach up to maximum precision 23 mantissa bits(binary) i.e. 7 digits (decimal point). Besides, in CORDIC residual error is defined by residual angle[12]. Theoretically, the residual angle after k iterations is,

$$\theta_R = \tan^{-1}(2^{-k}) \cong 2^{-k}\ radian = \left(2^{-k}\ \frac{180}{\pi}\right)^{\circ} \qquad (1.20)$$

So, binary precision, (not in IEEE754 format but represent in Binary)

$$P_B = \log_2 {\theta_R}^c \quad bits \qquad (1.21)$$

Decimal precision,

$$P_D = \log_{10} {\theta_R}^c \quad digits \qquad (1.22)$$

Figure 1.5 shows the characteristics of the residue angle and precision with increasing number of iteration(k).

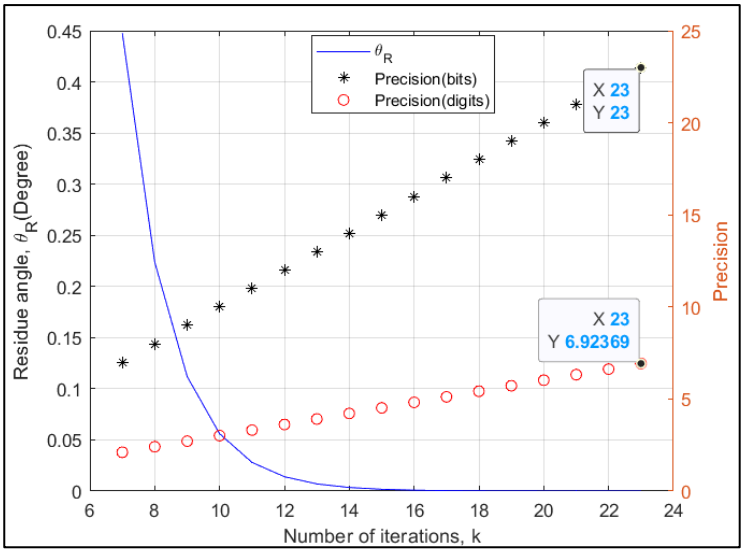

Figure 1.5: Residual angle and precision Vs number of iterations

1.4 Project organization and Design Tools:

In this project, the design process follows a bottom-up approach. The flow begins with designing the fundamental building blocks, followed by architectural optimization. This iterative process of lower-level optimization ultimately leads to the most efficient final design. Initially, the floating-point adder and shifter were designed and optimized as building blocks. Using these components, the final CORDIC processor was then constructed.

At each stage, standard design steps were followed, including behavioural verification, logic synthesis, pre-layout timing analysis, layout design, and post-layout timing analysis.

1.4.1 Model Creation and Behavioural verification

The behavioural and optimized RTL model was coded in **VHDL**. Additionally, a **MATLAB** behavioural model was developed for both the individual building blocks and the final CORDIC processor. In the **MATLAB** model, the built-in function called `single` was used to convert real numbers to single-precision IEEE 754 format, providing both input and output decoding.

Behavioural verification was performed using **Synopsys' verification and debugging tool, Verdi**. In the testbench, two functions were created: one to convert real numbers to IEEE 754 float32 format and another to convert IEEE 754 float32 back to real numbers. Finally, the output was printed to a text file using the `std.text` library.

1.4.2 Logic Synthesis

Logic synthesis was carried out using **Synopsys Design Suite, specifically through Design Vision**. Design Vision is a comprehensive tool that integrates both **Design Compiler** and **PrimeTime**. Design Compiler was responsible for logic mapping, optimization, and design rule checks (DRC), while PrimeTime handled the post-synthesis timing analysis. For this project, timing constraints were intentionally set pessimistically, meaning they were over-constrained to ensure robust performance under various conditions. The architecture in this project was implemented using two different technology nodes, each representing distinct libraries:

- **UMC13**: A 130nm planar MOS technology from United Microelectronics Corporation (UMC).
- **GF22**: A 22nm fully depleted silicon-on-insulator (FDX) technology from GlobalFoundries.

The operating environment was defined for both the **worst-case** and **best-case** conditions to ensure the design is thoroughly tested across a broad range of operating scenarios. This approach enhances the robustness and reliability of the design, making it more suitable for real-

world variations and edge cases. By evaluating the design under these extreme conditions, we can ensure its performance remains stable across diverse situations.

Timing analysis was performed for both **setup time** and **hold time**. The **setup time analysis** was conducted under the worst-case scenario, while the **hold time analysis** was carried out based on the best-case conditions defined in the library. This comprehensive analysis guarantees that the design will function correctly and meet timing requirements under all expected operating conditions.

The operating corners for the worst-case, best-case, and typical-case scenarios were selected by editing the **Design Compiler (DC) setup file**. This setup file specifies key conditions, including voltage, temperature, and transistor characteristics, that affect the synthesis process. By modifying the file to reflect these different scenarios, we ensured that the design was evaluated under a range of conditions, confirming its reliability in varying environments.

- **For the UMC13 Library:**
  - **Worst Case**:
    fsc0h_d_generic_core_ss1p08v125c
    (Transistor P-N: Slow-Slow, Supply Voltage: 1.08 V, Temperature: 125°C)
  - **Best Case**:
    fsc0h_d_generic_core_ff1p32vm40c
    (Transistor P-N: Fast-Fast, Supply Voltage: 1.32 V, Temperature: -40°C)
  - **Typical Case**:
    fsc0h_d_generic_core_tt1p2v25c
    (Transistor P-N: Typical-Typical, Supply Voltage: 1.2 V, Temperature: 25°C)

- **For the GF22 Library:**
  - **Worst Case**:
    SSG_0P81V_0P00V_0P00V_0P00V_125C
    (Transistor P-N: Slow-Slow, Supply Voltage: 0.81 V, Temperature: 125°C)
  - **Best Case**:
    FFG_0P945V_0P00V_0P00V_0P00V_M40C
    (Transistor P-N: Fast-Fast, Supply Voltage: 0.945 V, Temperature: -40°C)

- **Typical Case**:
  TT_0P90V_0P00V_0P00V_0P00V_25C
  (Transistor P-N: Typical-Typical, Supply Voltage: 0.9 V, Temperature: 25°C)

The **pre-layout logic synthesis** and **timing analysis** were performed with a focus on maximizing the **setup time**. The clock frequency was then determined after thorough optimization to ensure that the highest possible frequency could be achieved, with a positive slack margin of at least 15% to 20% of the clock period.
In addition, **hold time analysis** was conducted, but no optimization was applied for minor hold time violations or small positive slack margins. **Power analysis** was performed based on the worst-case conditions to ensure efficient power consumption under extreme scenarios.

1.4.3 Physical Design

After logic synthesis, the constraints and design specifications were saved, marking the beginning of the physical design process. The **MMMC (Multimode Multi-Corner)** file outlined all the operating conditions for timing analysis. The design flow then followed standard steps, starting with **Floor Planning**, where the layout and placement of blocks were established. This was followed by **Power Plane Design**, which ensured efficient power distribution across the entire chip.

Next, the **Placement** stage involved positioning the cells and components in optimal locations to minimize area and wire lengths, while satisfying timing and power constraints. **Clock Tree Synthesis (CTS)** was then performed to ensure the clock signal was evenly distributed across the design, minimizing clock skew and improving overall performance.

Once the placement and clock tree were set, **Routing** was carried out to connect the components, while ensuring minimal delay and congestion. After routing, **Post-Layout Verification** was done to check for any violations or errors, ensuring the design met timing, area, and power requirements.
The entire physical design process was executed using the **Cadence Design Suite**, specifically the **Innovus Implementation System**, which provided powerful tools for timing closure, routing, and verification, streamlining the implementation and ensuring the design met all specifications.

Chapter 2

State of the Art

Hardware implementation of CORDIC iteration step in rotation mode basically the implementation of equation 1.10, 1.11, 1.12 and 1.13. Those are,

$$x_{k+1} = x_k - y_k d_k 2^{-k} \quad (2.1)$$

$$y_{k+1} = x_k d_k 2^{-k} + y_k \quad (2.2)$$

$$z_{k+1} = z_k - d_k \varphi_k \quad (2.3)$$

$$d_k = \begin{cases} -1, & z_k < 0 \\ +1, & z_k \geq 0 \end{cases} \quad (2.4)$$

To implement the equations mentioned earlier, a floating-point adder/subtractor and a shifter or multiplier are required[13][14](Figure 2.1). Numerous architectures for floating-point CORDIC have been proposed in the literature. For instance,[12] provides the theoretical foundation for floating-point CORDIC, while [15], [16],[17] offer comprehensive reviews of CORDIC implementations. The CORDIC algorithm has been widely utilized for the computation of various mathematical functions, including trigonometric [18], [14], [19], exponential functions[20], and logarithmic functions[21], among others. These implementations have been tailored for different domains, mostly FPGA[14], [18], [19], and ASIC[22]. The following section discusses the Architecture choice and corresponding building blocks including adder/subtractor, shifter/multiplier.

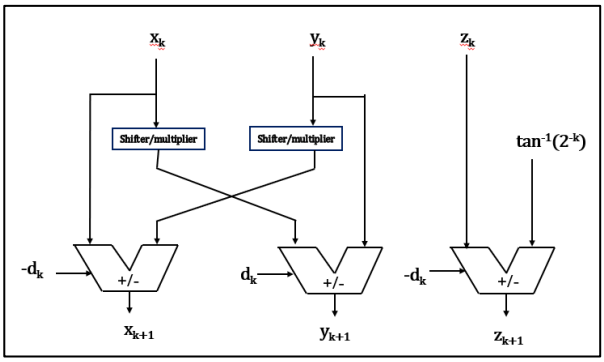

Figure 2.1: Each iterative stage of CORDIC

The sections in this chapter are organized as follows:

- **Section 2.1** for Specification and design strategy
- **Section 2.2** Architecture Choice
- **Section 2.3** Background of Building blocks

2.1 Specification and design strategy:

As mentioned in the previous chapter, the primary objective of this project is to implement a pure CORDIC architecture for single-precision IEEE 754 floating-point number systems. ASIC implementation can be optimized based on different design goals such as speed, area, or power consumption. In this project, one architecture has been designed and implemented with a focus on achieving high speed, while another architecture has been chosen to target reduced area and lower power consumption.

2.1.1 Define Specification

The implementation begins with clearly defined target specifications. Traditionally, the speed of a system is measured in terms of **throughput**. Throughput refers to the rate at which a system processes data and produces outputs — that is, how much work the system can perform per unit of time. For an architecture operating at a clock frequency f, and requiring **CPO (cycles per output)** clock cycles to generate each output, the throughput is defined as:

$$Throughput = \frac{f}{CPO}$$

Here, f is the clock frequency, and CPO represents the number of clock cycles needed to produce one output result.

Another important speed-related metric is **latency**, which measures the time delay from input to output — essentially the time it takes to process and generate a single output. It is expressed as:

$$\text{Latency} = CPO \text{ cycle}$$

Thus, while **throughput** indicates how often new outputs are generated, **latency** reflects the internal delay for each output.

2.1.2 Design strategy:

Achieving all three key design objectives—high speed, low area, and low power—in a single floating-point CORDIC architecture is extremely challenging due to inherent trade-offs among these parameters. To address this, the project adopted a dual-architecture strategy inspired by existing state-of-the-art designs. The first architecture was selected based on its potential to deliver high computational throughput, making it suitable for speed-critical applications. However, to make it more practical for hardware implementation, significant optimizations were carried out to reduce its combinational and sequential area, as well as to keep power consumption within a reasonable range. On the other hand, the second architecture was chosen

for its area- and power-efficient characteristics. Although it inherently had lower latency, its speed performance was enhanced through architectural refinements such as simplified data paths and reduced pipeline overhead. This two-pronged approach allowed the project to explore and demonstrate how architectural modifications and optimization techniques can help meet specific design targets depending on application needs—whether they are speed-dominant or resource-constrained.

2.2 Architecture Choice:

To design a CORDIC processor, the first and most critical design decision is the selection of the underlying architecture. Since its introduction in 1956, various CORDIC (COordinate Rotation Digital Computer) architectures have been proposed, each improving performance for specific application domains. Over the years, several extensions and variants of the original CORDIC algorithm have emerged, including scale-free CORDIC[23], Redundant arithmetic CORDIC[24], T-CORDIC (CORDIC-Tailor)[25], DCORDIC (differential CORDIC)[23], Higher radix CORDIC[26]. However, this project focuses on implementing the Conventional CORDIC architecture, due to its simplicity, predictable iteration structure. Specifically, the thesis concentrates on realizing this architecture in IEEE 754 32-bit single-precision floating-point format, a widely used standard in modern digital signal processing and scientific computation. In the context of floating-point implementations, CORDIC architectures can generally be categorized into two major classes:

1. Global floating-point implementation
2. Local Floating-point implementation

Global floating-point implementation [27]**:**

In the global floating-point approach, the input floating-point numbers are first converted into fixed-point representation. All CORDIC computations—such as iterative shifts and additions—are then carried out using simple fixed-point hardware, typically binary shifters and adders. Once the computation is complete, the result is converted back into IEEE 754 floating-point format at the output stage. This method significantly simplifies the hardware design by avoiding the complexity of floating-point arithmetic within the CORDIC pipeline. As a result, it generally offers lower latency and reduced hardware complexity. However, the major drawback of this approach is its **limited precision**, since the intermediate computations are performed in fixed-point. This precision loss undermines the key advantages of using IEEE 754 floating-

point representation, such as dynamic range and numerical stability. Therefore, while global floating-point implementation is efficient in terms of speed and resource usage, it compromises accuracy—making it less suitable for applications where numerical precision is critical.

Local floating-point implementation:

Unlike the global approach, the local floating-point implementation does not involve any data format conversion. All computations are performed directly in IEEE 754 floating-point format throughout the CORDIC pipeline. This requires the use of floating-point adders and subtractors, along with floating-point-compatible shift operations to realize the 2^{-i} scaling factor used in each iteration. While this approach offers **superior numerical precision** by fully preserving the benefits of floating-point representation, it comes with notable drawbacks: significantly **larger hardware area**, **higher latency**, and **increased power consumption** due to the complexity of floating-point units.

Despite these challenges, this project targets the local floating-point implementation to exploit its accuracy advantages. The complexity is treated as a design challenge, and efforts have been made to **optimize the architecture**—particularly by reducing latency through pipelining and minimizing area through resource sharing and architectural improvements. The goal is to balance performance, area, and power while maintaining high computational precision.

From a hardware perspective, **three fundamental CORDIC architectures** are widely discussed in the literature:

1. Pipelined Architecture
2. Serial Architecture
3. Hybrid Architecture

In a **pipelined CORDIC**, the maximum throughput achievable is f samples/second when the **clocks per output (CPO)** equals 1, indicating a fully pipelined design where one output is produced every clock cycle. In this project, no strict frequency constraint is imposed; instead, the pipelined architecture is optimized to operate at **the highest possible clock frequency**, maximizing throughput. In contrast, a **serial CORDIC architecture** performs one iteration per clock cycle. To increase the output rate in serial designs, the strategy is to **maximize the clock frequency f**.

2.2.1 Pipelined Architecture of CORDIC

The **pipelined architecture** of a floating-point CORDIC, as illustrated in Figure 2.2 and discussed in[14] processes three parallel inputs: **X**, **Y**, and **Z**. Each input passes through multiple pipeline stages, where **each stage performs one CORDIC iteration**. This fully unrolled configuration is commonly referred to as an **unrolled CORDIC architecture** [13]. At every stage, a fixed precomputed **micro-angle** (α) is either added or subtracted from the Z-path depending on the sign of Z. Since these micro-angles are hardwired into the architecture, **no runtime ROM lookup** is required, reducing memory complexity.

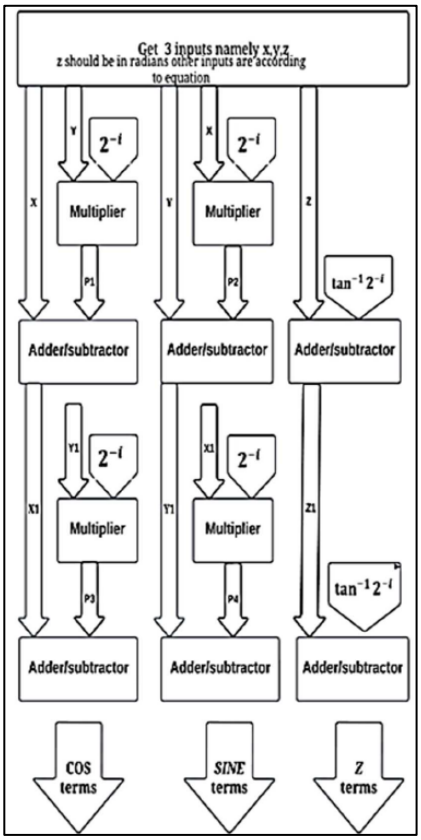

Figure 2.2: Pipelined structure of CORDIC to generate Sin and Cosine[14]

This architecture enables **maximum throughput**, as new input data can enter the pipeline at each clock cycle, and **parallel execution** of multiple input sets occurs simultaneously across the stages. As a result, this architecture is well-suited for **high-performance applications** that require continuous and rapid trigonometric computations. The unrolled implementation was specifically chosen for this project because it provides the highest possible throughput. In this setup, the CPO (Cycle Per Output) is 1, meaning one output is generated per clock cycle, and the latency is equal to the number of CORDIC iterations (N). Therefore, the throughput is:

Throughput = f

Although this implementation uses a multiplier, efforts were made to realize the multiplication operation in a time- and area-efficient manner. One of the main challenges in this architecture is the floating-point adder/subtractor, which requires considerable processing time.

2.2.2 Serial Architecture of CORDIC

The serial architecture of CORDIC processes one iteration per clock cycle (for fixed-point data systems), ensuring that the algorithm is computed step-by-step. In this architecture, the precomputed micro-angles are stored in a ROM lookup table, and during each iteration, a micro-angle is added or subtracted from the residual angle. Additionally, the output from each iteration is stored in a register. For this project, the serial architecture is inspired by [13](Figure 2.4), which serves as the foundation due to its minimal device requirements.

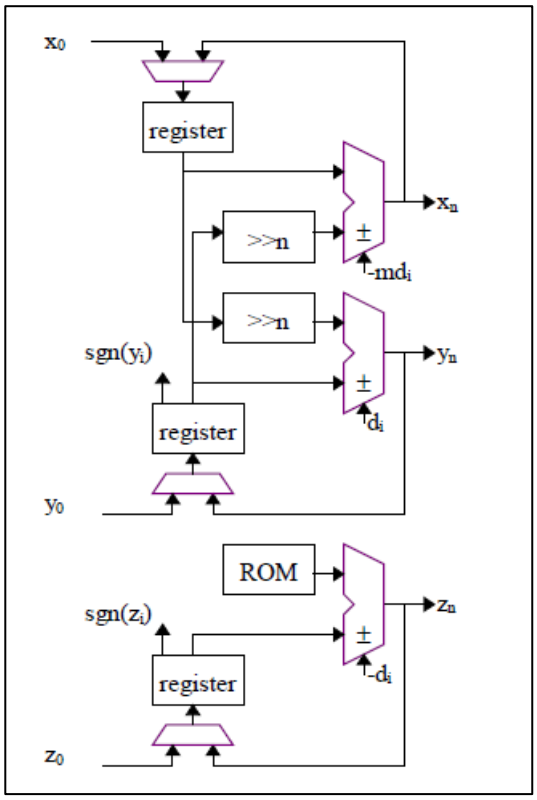

Figure 2.3: Serial Architecture of CORDIC[13]

This architecture was selected for its low power consumption and reduced area requirements, as it utilizes only a single CORDIC iteration unit. In this architecture, the throughput is given by:

$$Throughput = \frac{f}{CPO}$$

Efforts were made to increase the value of f (the operating frequency) and decrease the value of CPO (Clock Per Output) to maximize throughput.

2.3 Background of Building Blocks:

To implement Floating-Point CORDIC (FP-CORDIC), two fundamental units are required: the floating-point adder and the shifting/multiplication logic for the IEEE 754 number system. Various FP adder/subtractor algorithms and architectures have been discussed in the literature such as reversable architecture[28], tuneable FP-adder[29], area efficient FP-adder [30].

2.3.1 Floating-point (FP) Adder/Subtractor

The floating-point (FP) adder/subtractor is responsible for performing the addition or subtraction of two floating-point numbers. The design of the adder/subtractor in this project is primarily based on the approach described in [30] , with several important modifications. While the algorithm outlined in [30] serves as the foundation, the architecture has been tailored to optimize throughput, latency, and area efficiency for the FP-CORDIC implementation.

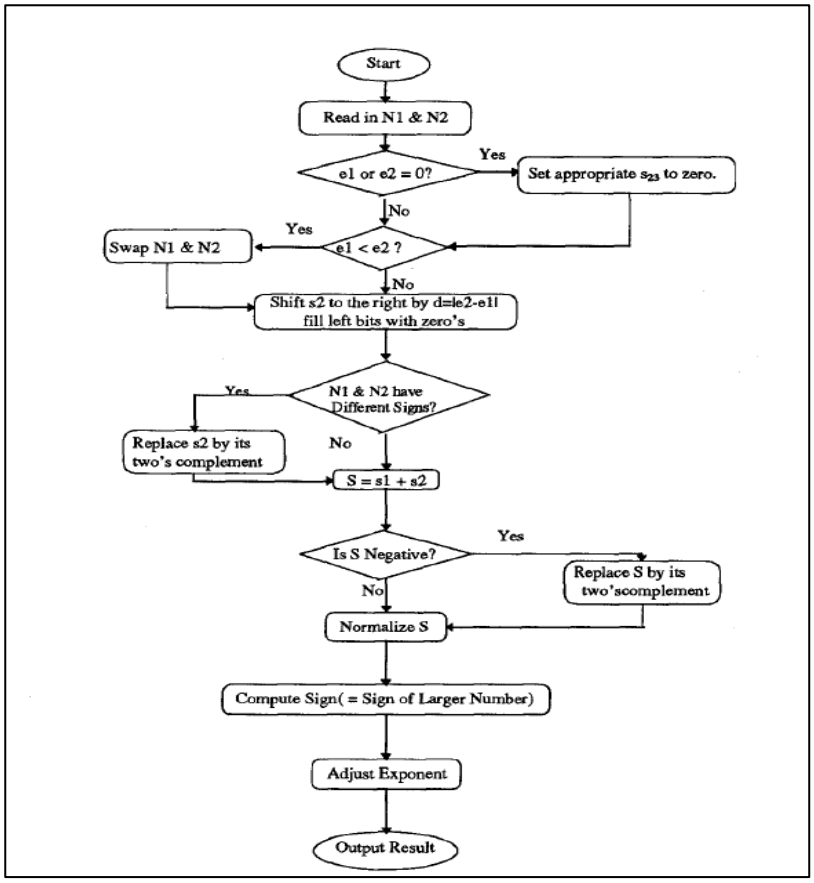

Figure 2.4: Floating point addition algorithm[30]

2.3.2 Shifter/Multiplier

The CORDIC algorithm involves multiplying by 2^{-k}. For fixed-point data, this multiplication is efficiently carried out using a bitwise shift operation [1], as shifting the binary representation

to the right by k positions is equivalent to multiplying by 2^{-k}. This approach simplifies the implementation and reduces computational complexity, making it ideal for hardware designs. In [14] the floating-point implementation of CORDIC used a Vedic multiplier [31] , which is based on a digit-serial multiplication approach. which is based on a digit-serial multiplication method. However, in this project, the shifting or multiplication operation is performed using 8-bit binary subtraction. This method not only simplifies the hardware implementation but also boosts efficiency, resulting in a more streamlined and resource-effective design.

Chapter 3

Design Building Blocks

The adder/subtractor and shifter/multiplier are the basic building block of CORDIC co-processor. In fixed point number system those operation is strait forward where the binary adder/subtractor use to add/subtract. Besides the shifter use to multiply any number with number in the form of 2^i. But the adding and subtraction is not straight forward for the floating-point number in IEEE754 format. This chapter is dedicated to the design, simulation, pre-layout (gate-level) synthesis and static time analysis of those Building blocks. As adder is the crucial part of an floating point system to get highest throughput the pipeline architecture of the a.
The sections in this chapter are organized as follows:

- **Section 3.1** Algorithm of floating-point Addition/Subtraction.
- **Section 3.2** Pipelined Architecture of floating-point Adder/Subtractor.
- **Section 3.3** Serial Architecture of floating-point Adder/Subtractor.
- **Section 3.4** Scaling of IEEE-754 Floating-Point Values by Powers of Two

3.1 Algorithm of Floating-point Adder/Subtractor

The floating-point addition algorithm can be derived from the mapping mechanism of real numbers to the IEEE 754 format. Consider two numbers represented as:

$$(-1)^{SA} \times 2^{EA} \times MA \qquad \text{(A)}$$

$$(-1)^{SB} \times 2^{EB} \times MB \qquad \text{(B)}$$

Here ***MA, MB*** are the mantissa which represent the significant of real counterpart. ***EA*** and ***EB*** are the biased exponent of floating-point number. ***SA, SB*** represent the sign bits of those number.

The floating-point addition and subtraction process begins by **de-normalizing** the input mantissas. If both input exponents (***EA*** and ***EB***) are non-zero, a leading '1' is concatenated to the left of each mantissa (***MA*** and ***MB***). This restores the hidden bit that was omitted during IEEE 754 normalization, making each mantissa effectively 24 bits long, and the entire representation 33 bits including the concatenated digit. If either exponent is zero, both mantissas are set to zero, indicating that one of the inputs represents a zero value.

Next, the two exponent values are **compared** to determine which one is greater. The larger exponent is stored as the shared exponent, referred to as ***bigEX,*** which will be used for the result. The difference between the exponents is calculated, and the mantissa corresponding to the smaller exponent is right shifted by this difference to align both operands to the same exponent level. This **alignment** ensures the mantissas represent numbers on the same scale, which is essential for meaningful addition or subtraction. At this stage, the sign bits of the inputs are stored. For subtraction, the sign of the second operand is inverted to enable the same hardware unit to handle both addition and subtraction uniformly.

Since IEEE 754 uses sign-magnitude format, but arithmetic operations are easier in **two's complement**, the mantissas are converted accordingly. If both operands have the same sign, no changes are made. However, if their signs differ, the mantissa of the negative operand (where the sign bit is 1) is converted to its two's complement form. This prepares the operands for correct arithmetic computation.

Once converted, both mantissas are extended to 25 bits by concatenating their respective sign bits to the left. These extended forms are then added together using a standard adder, producing a result called ***SUMMAN***, which is also 25 bits long. The sign and magnitude of the sum are then determined. If the original operands had the same sign, the result sign (***Ssum***) is the same as the input sign, and the sum remains unchanged. If the operands had different signs, the **most significant bit (MSB)** of ***SUMMAN*** is used to detect whether the result is negative. If it is, the **two's complement** of the result is taken, and the MSB is used to set the final sign of the result. In the case of same-sign addition, overflow may occur. If the MSB of ***SUMMAN*** is 1, indicating **overflow**, the result is shifted right by one bit and the exponent is incremented by one to maintain accuracy.

After arithmetic operations, **normalization** is performed to ensure the result follows IEEE 754 format. If the shared exponent ***bigEX*** is zero, this indicates the result is zero. In such cases, the final sign, exponent, and mantissa are all set to zero. Otherwise, the position of the first leading '1' within the 24 least significant bits of ***SUMMAN*** is identified. The result is then shifted left by that number of positions, and the exponent is reduced accordingly to maintain the value of the result. Finally, the sign bit ***(Ssum)***, adjusted exponent ***(Esum)***, and the 23 least significant bits of the normalized mantissa ***(Msum)*** are combined to form the final floating-point output.

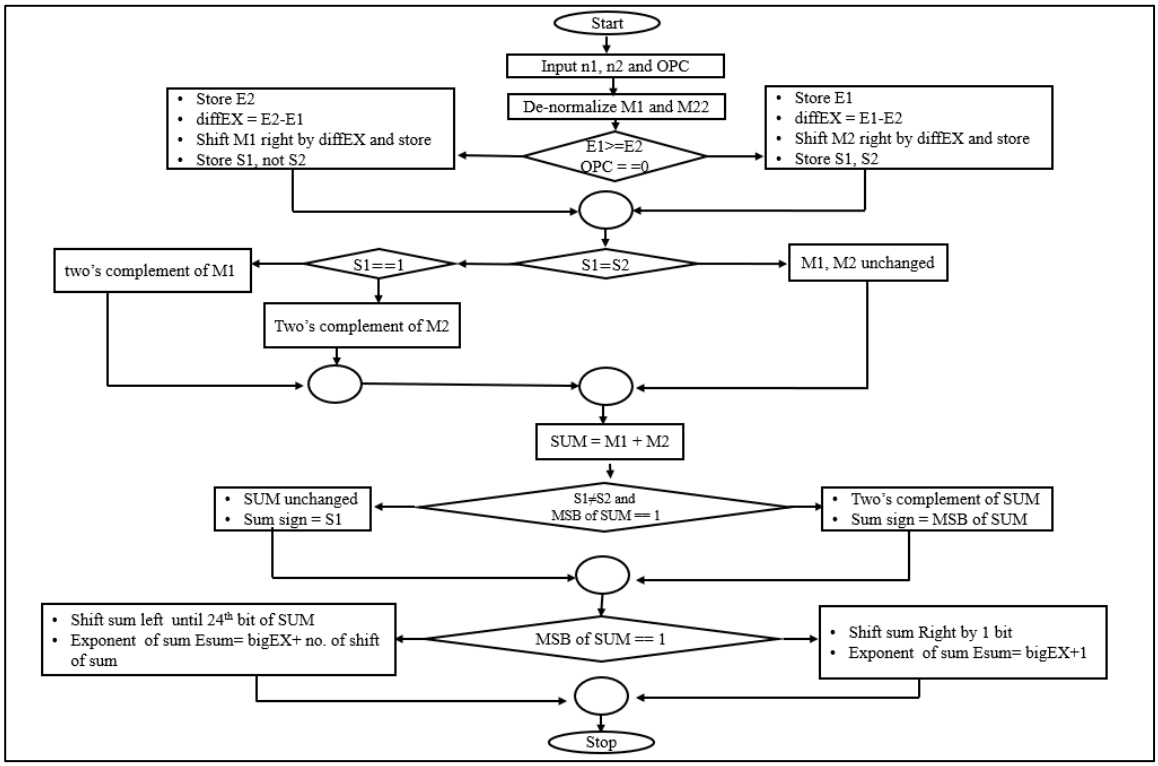

Figure 3.1: Flow chat of floating-point addition/subtraction

3.2 Pipelined Architecture of Floating-point Adder

As outlined in the preceding section, the floating-point adder implementation necessitates a multiple sequential operation. As the early estimation the floating-point adder performance is the hindrance of the speed of overall CORDIC architecture, the pipeline architecture of the floating-point adder has been implemented to facilitate the CORDIC architecture with deep pipelining which enhance the throughput. The design methodology employs an iterative workflow, rigorously adhering to ASIC design best practices—including behavioural verification, gate-level synthesis, and static timing analysis.

The pipelined architecture is strategically optimized to maximize throughput while minimizing area overhead. Leveraging a bottom-up design approach, synthesis is executed with intentionally over-constrained timing parameters to guarantee seamless integration within the CORDIC co-processor. Furthermore, the proposed maximum clock frequency incorporates a deliberate design margin, pre-emptively mitigating potential hold-time violations during physical synthesis. This foresight ensures that any timing discrepancies remain within manageable limits, readily correctable through post-layout optimization techniques.

The architectural design follows these steps:

1.Create RTL model based on Behavioural (model -1)

2. Simulate and debug the Behavioural model
3. Gate level synthesize of the model-1, time, and inferred device analysis
4. Create RTL targeting structural optimization and re-distribute the work (model-2)
5. Simulate and debug the model-2
6. Gate level synthesis of model-2 and compare with model 1

3.2.1 Create RTL based on Behavioural model(model-01)

The design process initiates with a systematic decomposition of the floating-point adder into functional units. This partitioning strictly follows the computational sequence defined in the algorithm, with each discrete operation implemented as an independent pipeline stage. The resulting architecture comprises six precisely defined processing stages:

1. Denormalization
2. Alignment
3. Two's Complement Conversion (Mantissa)
4. Mantissa Addition
5. Two's Complement Conversion (Sum)
6. Normalization

Denormalization

The denormalization unit serves as the first stage in the floating-point adder pipeline, responsible for properly reconstructing the mantissa's implicit leading bit according to IEEE754 standards. The unit begins by examining the 8-bit exponent field of each input operand using a dedicated comparator circuit. When the comparator detects all exponent bits as zero - indicating either a true zero value or a denormal number - it triggers the concatenation of a '0' bit with the original 23-bit mantissa. For all other cases (non-zero exponents), the unit prepends a '1' bit to the mantissa, effectively restoring the hidden bit that was removed during floating-point encoding. This process generates a **33-bit denormalized output**, where the most significant bit represents the restored leading digit and the least significant 23 bits maintain the original mantissa value. Notably, the architecture preserves the exponent field in its original encoded form, eliminating the need for exponent denormalization. (Figure 3.2)

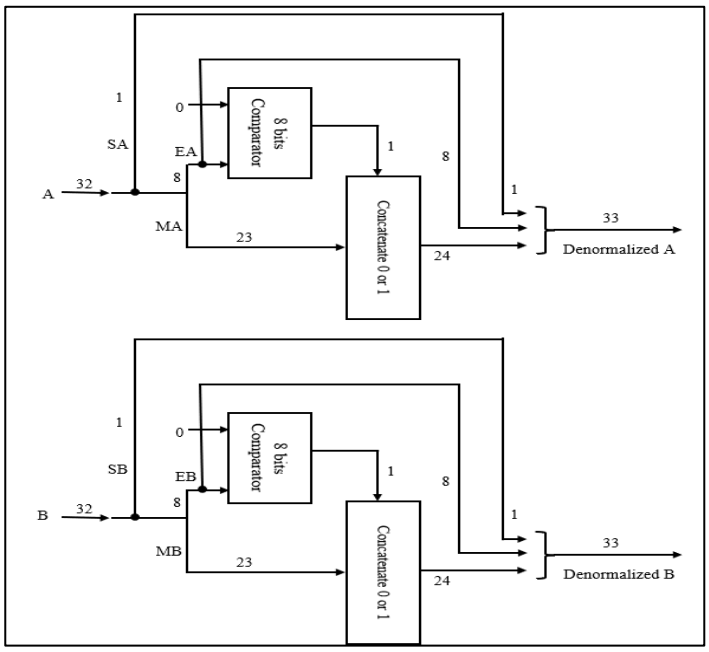

Figure 3.2: Denormalization Unit (model-1 and model-2)

Alignment

The alignment unit shown in figure 3.3, performs three key operations to prepare floating-point operands for arithmetic processing: exponent comparison and selection, exponent difference calculation, and mantissa alignment. An 8-bit comparator evaluates the input exponents (*EA, EB*), controlling two 8-bit multiplexers (MUX8) that select the larger (*bigEX*) and smaller (*smallEX*) exponents through complementary selection. Simultaneously, two 24-bit MUX(MUX24) choose the corresponding mantissas, while an 8-bit subtractor computes the exponent difference (*diffEX = bigEX – smallEX*) to determine the shift amount for a barrel right shifter, which aligns the smaller-exponent mantissa by |***diffEX***| bits. The operation control signal (OPC) determines whether the unit performs addition or subtraction by conditionally inverting the sign bit of operand *B (BSIGN)* via a 1-bit MUX when OPC = 1. The unit outputs the aligned operands as *bigEX* (8-bit), *AMAN_S* (24-bit), and *BMAN_S* (24-bit), along with their processed sign bits *ASIGN* (1-bit) and *BSIGN* (1-bit), ensuring proper preparation for subsequent arithmetic stages.

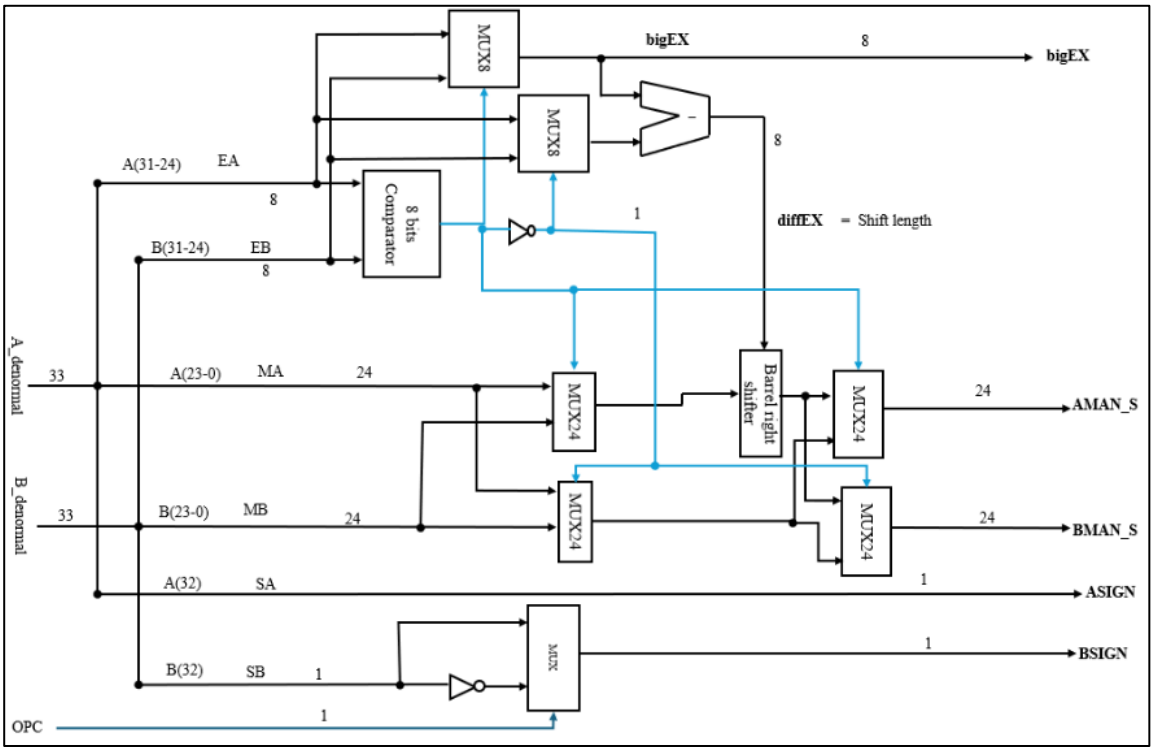

Figure 3.3: Alignment Unit

Two's Complement Conversion (Mantissa)

The two's complement conversion unit processes the aligned 24-bit mantissas (*AMAN_S*, *BMAN_S*) and their 1-bit sign inputs *(ASIGN, BSIGN)* to perform conditional two's complement conversion. In a sign comparison blocks first checks if *ASIGN* equals *BSIGN*. When signs are identical, the mantissas pass through unchanged (*AMAN_COMPLI = AMAN_S*, *BMAN_COMPLI = BMAN_S*). For differing signs, the system selectively converts the negative operand: if *ASIGN* is '1' (indicating *A* is negative), *AMAN_S* undergoes two's complement conversion while *BMAN_S* remains unmodified; conversely, when *BSIGN* is '1', *BMAN_S* is converted while *AMAN_S* stays unchanged. The two's complement operation is implemented through bit inversion followed by addition of '1' using a dedicated 24-bit adder. Two 24-bit multiplexers, controlled by the sign comparison output, select between original and converted mantissa values. The final output stage concatenates a sign bit extension - preserving the original sign when operands have opposite signs or appending '0' when signs match. This unit ensures proper **2's complement signed** representation for subsequent arithmetic operations. The complete implementation requires: (1) a sign comparator, (2) a 23-bit adder for complement operations, (3) two 24-bit multiplexers for mantissa selection, and (4) output formatting logic for sign-bit handling (shown in figure 3.4).

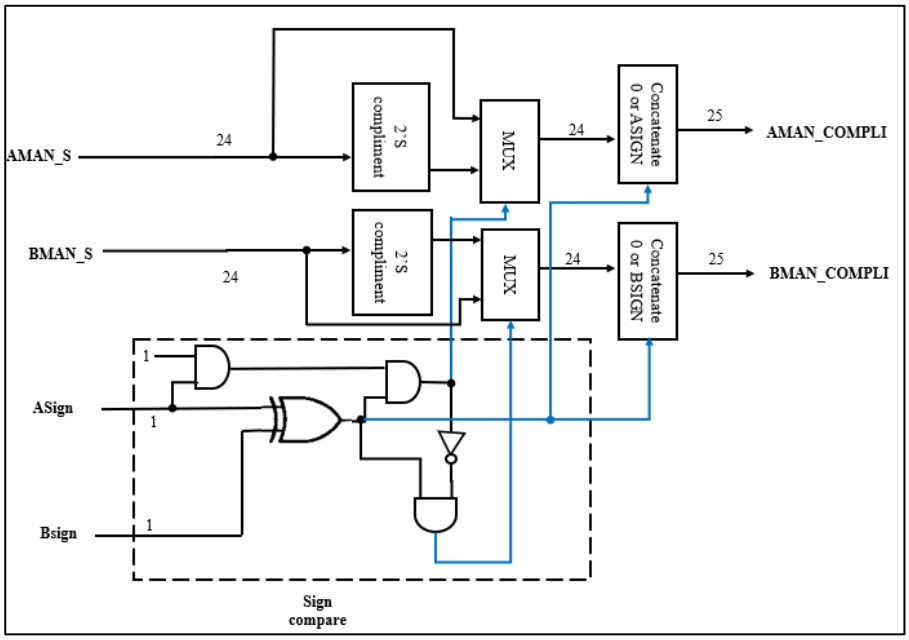

Figure 3.4: Twos compliment conversion (Mantissa)

Add Mantissas

The mantissa addition unit is implemented as a 25-bit unsigned adder that performs the fundamental arithmetic operation on the processed mantissa inputs. It takes the two 25-bit conditioned mantissa values (`AMAN_COMPLI and BMAN_COMPLI)` from the previous two's complement conversion stage as inputs, which have been properly formatted to represent signed values in two's complement form. The adder computes the raw sum of these two operands and produces a 25-bit output (`SUM_MAN`) representing the unnormalized mantissa of the final floating-point result. This 25-bit output contains the mantissa part of arithmetic result before any normalization or final two's complement processing, preserving all necessary precision for the subsequent pipeline stages. The 25-bit width is chosen to accommodate potential overflow and preserve the sign bit during addition, ensuring that no significant bits are lost prior to normalization. When both the addend and augend have the same sign, the most significant bit (*MSB*) of the result indicates overflow. Conversely, when the operands have opposite signs, the *MSB* represents the sign of the final sum.

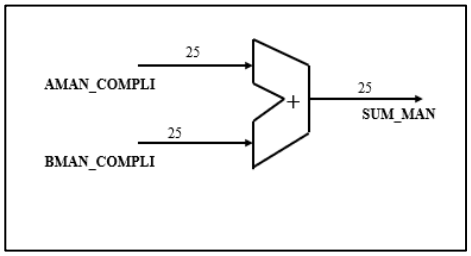

Figure 3.5: Add mantissa (model-1)

Two's Complement Conversion (Sum)

This unit performs three critical operations to finalize the floating-point sum as illustrate in figure 3.6. First, it determines the final sign (*SUM_SIGN*) through sign comparison logic - when input signs (*ASIGN* and *BSIGN*) match, *SUM_SIGN* equals *ASIGN*; otherwise, it takes the leading bit of *SUM_MAN* as the result sign. Second, the unit conditionally converts the sum mantissa using two's complement: a multiplexer selects either the original *SUM_MAN* or its two's complement (created by bit inversion followed by +1 addition), choosing the complement only when signs differ and the sum is negative. Third, overflow handling occurs when input signs match and *SUM_MAN*'s leading bit is '1'; in this case, a 1-bit right shifter corrects the mantissa while an adder adjusts the exponent *(bigEX + 1)*. The unit outputs the processed 25-bit mantissa (*SUMMAN_COMPLIMENT*), adjusted 8-bit exponent (*bigEX*), and 1-bit sign (*SUM_SIGN*), completing all necessary preparations for the final normalization stage.

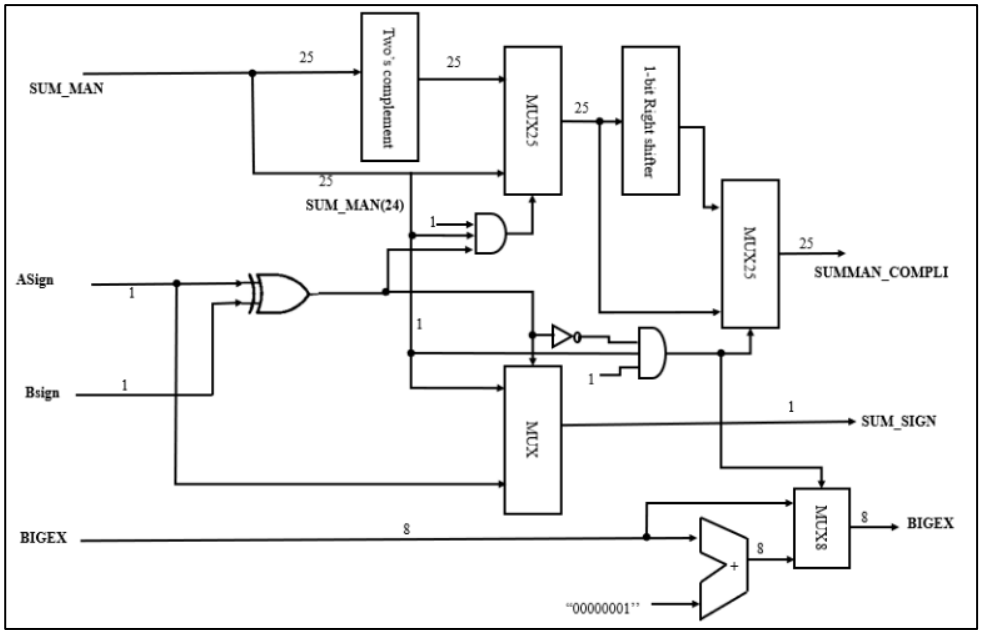

Figure 3.6: Twos complement conversion (SUM) (model-1)

Normalization

The normalization unit performs the final processing of the floating-point sum by locating the leading '1' in the 25-bit *SUMMAN_COMPLIMENT*, determining its bit position (*first_1)* through a leading zero counter (implemented as a for-loop in the RTL code). This position value drives two parallel operations: a barrel left shifter aligns the mantissa by left-shifting *SUMMAN_COMPLIMENT* by |*first_1*| bit to produce the normalized 23-bit mantissa (*Msum*), while an 8-bit subtractor calculates the normalized exponent (*Esum*) by subtracting *first_1*

from *bigEX*. Special cases are handled through multiplexer logic - when |*first_1*| exceeds 23 (indicating an effective zero result) or when *bigEX* is zero (denoting zero inputs), *Esum* is forced to zero. The final floating-point result combines the preserved sign bit (*Ssum = SUMSIGN*), normalized exponent (*Esum*), and normalized mantissa (*Msum*) in standard IEEE 754 format. (as shown in figure 3.7)

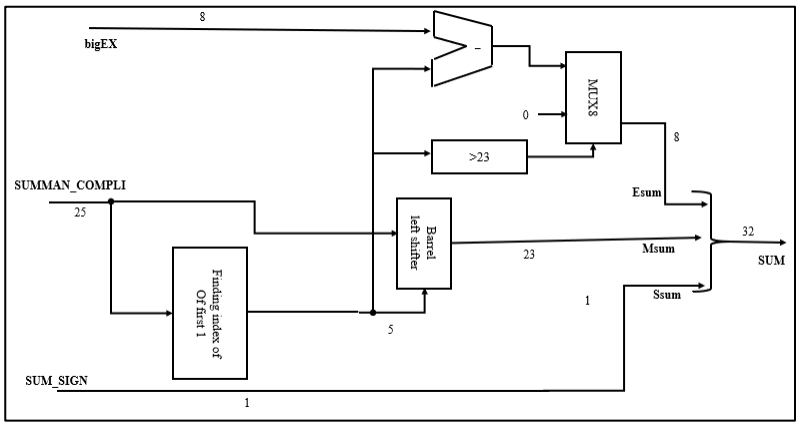

Figure 3.7: Normalization Unit (model-1)

Register placement and top Hierarchy

The pipelined architecture incorporates register stages between each computational block to maximize throughput while ensuring proper synchronization (depicted in the figure 3.8). Registers (*REG1−REG6*) are strategically placed after all six major operations: *Denormalization (Step 1), Alignment (Step 2), Two's Complement Conversion of Mantissas (Step 3), Mantissa Addition (Step 4), Two's Complement conversion of Sum (Step 5),* and *Normalization (Step 6).* Each register contains

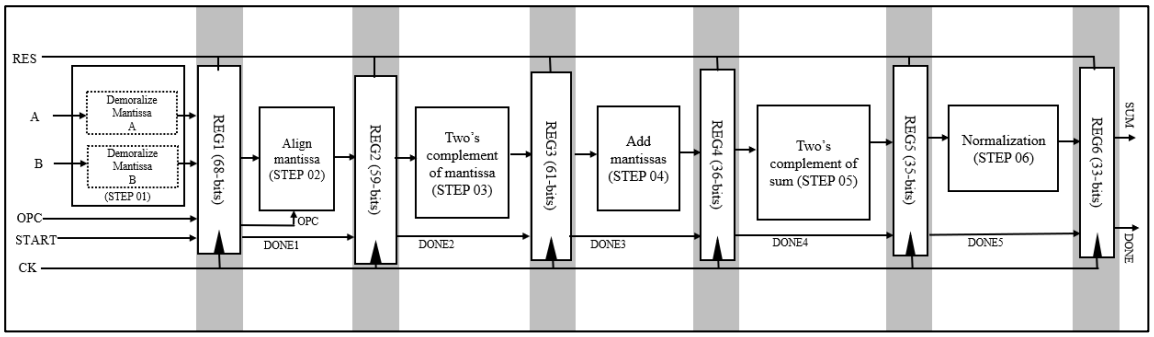

Figure 3.8: Pipelined Floating-point adder/subtractor (model-1) top hierarchy

dedicated control logic, including a 1-bit *DONE* flag that signals operation completion and triggers the subsequent stage by propagating a *START* signal to the next register, enforcing strict sequential execution. Crucially, all registers reserve an 8-bit field for *BIGEX* storage,

maintaining exponent consistency throughout the pipeline. The *BIGEX* value is initially determined during *alignment* (Step 2), potentially updated during *overflow handling* (Step 5), and finally adjusted during *normalization* (Step 6), ultimately becoming the exponent component (*Esum*) of the result. The pipeline registers (*REG1–REG6*) follow a standardized bit allocation scheme as detailed in Table 3-1.

Table 3-1 : Allocated registers for corresponding Data and their length (i.e. flipflop) (model-1)

Register block	**Data stored and its length**	**Total length (bits)**
REG1	*A*: 33-bits *B*: 33-bits *OPC*: 1-bit *DONE1*: 1-bit	68
REG2	*BigEX*: 8-bits *AMAN_S*: 24-bits *AMAN_S*: 24-bits *ASIGN*: 1-bit *BSIGN*: 1-bit *DONE2*: 1-bit	59
REG3	*BigEX*: 8-bits *AMAN_COMPLI*: 25-bits *AMAN_COMPLI:* 25-bits *ASIGN*: 1-bit *BSIGN*: 1-bit *DONE3*: 1-bit	61
REG4	*BigEX*: 8-bits *SUM_MAN*: 25-bits *ASIGN*: 1-bit *BSIGN*: 1-bit *DONE4*: 1-bit	36
REG5	*BigEX*: 8-bits *SUMMAN_COMPLI*:25-bits *SUM_SIGN*: 1-bit *DONE5*: 1-bit	35
REG6	*SUM*: 32-bits *DONE*: 1-bit	33

3.2.2 Gate level synthesizes of the model-1:

The gate-level synthesis of the behavioural model was conducted to analyse propagation delays, measure area utilization, and identify the specific low-level components inferred by the synthesizer. This process aimed to pinpoint critical timing paths and resource-intensive modules, providing actionable insights for architectural refinements. By examining the synthesized netlist, the propagation delays of individual pipeline stages were quantified. Additionally, the synthesis tool's component inference patterns were analysed—such as whether it implemented faster adders or multiplexer-based shifters—to understand how the **RTL code mapped to hardware primitives**. These findings revealed opportunities to optimize the design by modifying the RTL description to steer the synthesizer toward **lower-latency** components or redistributing logic across pipeline stages. The experiment established a data-driven foundation for iteratively enhancing the design's speed through targeted code adjustments, ultimately **reducing critical path delays** while maintaining **area efficiency**. This synthesis-guided approach enables systematic **trade-off** analysis between timing performance and resource utilization in subsequent design iterations.

Set constrains:

As previously stated, the synthesis of building blocks employed deliberately conservative timing constraints, intentionally overestimated to account for worst-case scenarios and avoid optimistic assumptions. In strict adherence to **Synopsys® Design Compiler™ (DC)** guidelines, constraints were pessimistically defined to ensure robust timing closure. Specifically, delay calculations were configured to prioritize maximum setup time conditions, with the primary synthesis objective being setup time resolution. The target clock period was proposed with a **15–20% timing slack margin** to accommodate process variations and post-layout uncertainties. All synthesis results discussed in this chapter adhere to this rigorously constrained methodology, ensuring consistent and reliable timing analysis across all design stages. Table 3-2 illustrates all the constrains and operating environment imposed during the synthesis.

Table 3-2: Constrains and operating condition for synthesis

Constrain on clock:		
	UMC13	**GF22**
Clock latency (estimated delay of the clock tree):	0.3 ns	0.05 ns
Clock skew (estimated setup and hold time):	0.06 ns	0.02 ns
Clock transition (approximated due to the large fanout at clock port):	0.003 ns	0.0005 ns
Input delay:	0.6 ns	0.06 ns
Output delay:	0.6 ns	0.06 ns
Operating environment:		
Operating condition:	-min (best case) -max (worst case)	-min (best case) -max (worst case)
Wire load model:	not sated	not sated
Wire load mode:	enclosed	enclosed
Driving cell: Buffer	`BUFEHD`	`SC8T_BUFX1_CSC20R`
Output load:	NAND `(ND3EHD{I3})`	capacitor `(5 fF)`
Maximum area:	15000	500
DRC (Design Rule Checking)	kept default	kept default
Don't touch network:	clock *(CK)*	clock *(CK)*

Time and area analysis:

The design was compiled using a top-down methodology, with comparative trials of design budgeting revealing minimal impact on component inference patterns. Timing analysis evaluated both setup time (under worst-case conditions for maximum path delay) and hold time (under best-case conditions for minimum path delay). For **UMC 130nm** technology, the minimum achievable clock period was **6 ns**, with a **16% setup slack (0.95 ns)** and **1.2% hold slack (0.07 ns)**. In contrast, the **GF22** technology achieved a **1.1 ns** clock period, delivering **16.4% setup slack (0.18 ns**) and **0.9% hold slack (0.01 ns)**. As detailed in Figure 3.9, the synthesis report categorizes combinational logic area per pipeline stage for both technologies, with Gray-highlighted regions representing non-combinational elements (flip-flops). Notably, area calculations exclude register reset logic and startup circuitry, focusing exclusively on core functional blocks.

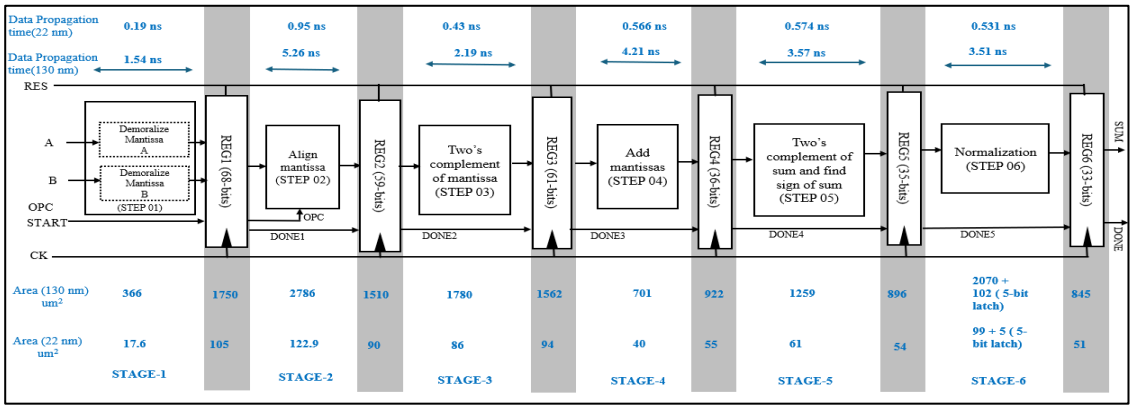

Figure 3.9: Propagation time and occupied area of each stage (model-1)

As illustrated in Figure 3.9, the Alignment Unit occupies the largest cell area and longest propagation time, establishing it as the critical bottleneck for overall throughput while the Two's Complement Conversion stage demonstrates the shortest propagation delay. To identify the root cause of timing constraints, a detailed analysis was conducted to evaluate the low-level components (e.g., **binary adders**) inferred by the synthesis tool that contribute to increased propagation delays. The following section provides a comprehensive discussion of these findings, focusing on how synthesized logic structures impact both timing performance and area utilization across pipeline stages.

Analysis of Mapping:

As depicted in Figure 3.10, the **denormalization unit** is synthesized as a flattened combinational logic structure composed of *AND* and *NOR* gates. This implementation achieves minimal propagation delay due to its optimized gate-level arrangement, enabling efficient restoration of the IEEE 754 mantissa's implicit leading bit.

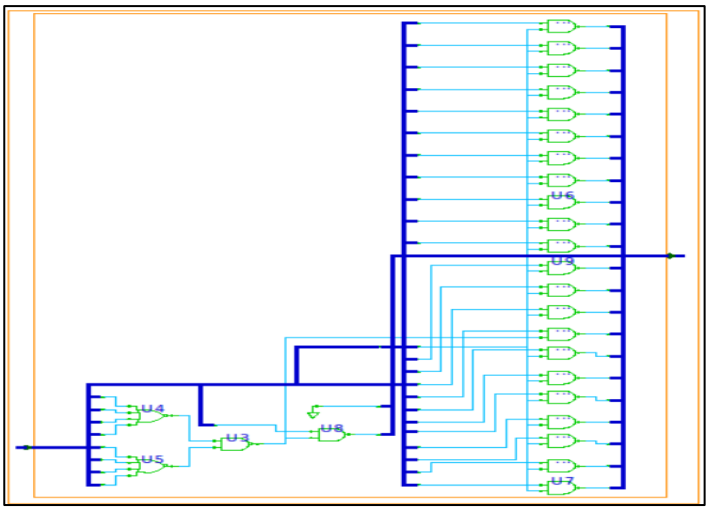

Figure 3.10: Denormalization Unit inferred by synthesized

The **alignment unit** is synthesized as a combinational logic block comprising *AND, OR, NOT* gates, an 8-bit subtractor, and a barrel right shifter. Among these components, the barrel shifter and subtractor dominate area utilization, with the subtractor—mapped as a **ripple-carry adder (RCA)** as shown in Figure 3.11 —being the primary contributor to both propagation delay and area overhead. The RCA's cascaded full-adder structure introduces significant carry propagation latency. Additionally, the standalone comparator implementation further increases area consumption and sequential multiple operations like **comparison => Subtraction => Shifting** increases the overall input to output propagation time of this stage.

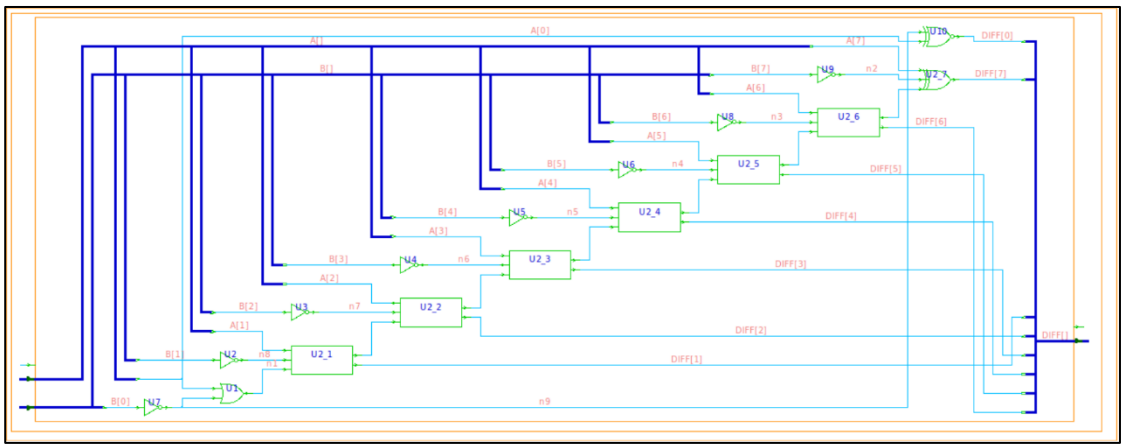

Figure 3.11: Inferred 9-bit ripple carry adder

The **Two's Complement conversion units** are synthesized as a combinational circuit comprising two binary adders, a sign comparator (implemented via an *XOR* gate), and a multiplexer (*MUX*). Notably, the adders are synthesized as **25-bit Ripple Carry Adders (RCA)**, (as illustrated in Figure 3.12) that introduces significant propagation delay due to their sequential carry propagation mechanism.

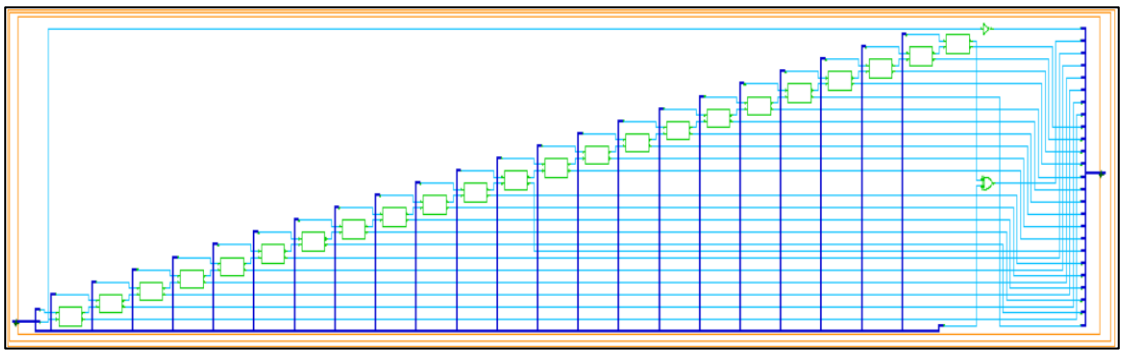

Figure 3.12: 24-bit RCA inferred in twos complement unit

The **Add Mantissa unit** is synthesized as a 25-bit Ripple Carry Adders (RCA) (shown in Figure 3.13), exhibiting the propagation delays in the design: **4.21 ns for UMC13 (130nm) and 0.566 ns for GF22 (22 nm)** technologies. As a single operation mantissa addition require the highest propagation time. This implementation leverages a cascaded full-adder structure, where sequential carry propagation creates a critical timing path. The ripple-carry architecture, while area-efficient, introduces significant latency due to its linear dependency chain.

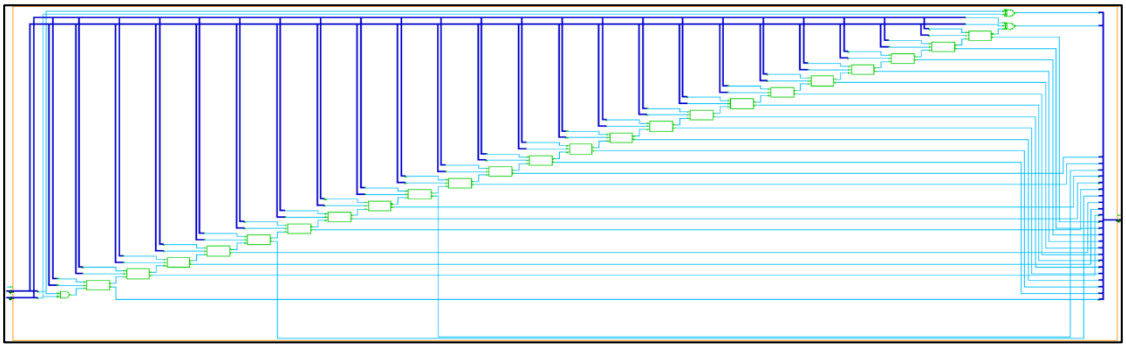

Figure 3.13: 25-bit RCA inferred to add mantissa

The n**ormalization unit** incorporates a leading '1' `(first_1)` detection block (coded as for-loop in the RTL), which synthesizes into a priority encoder and a **5-bit latch** (shown in red circle of Figure 3.14), introducing significant propagation delays due to its iterative search mechanism. This structure creates a critical timing path, as the **priority encoder's** sequential logic and latch-based storage extend the unit's latency. Additionally, the 8-bit exponent subtractor, implemented as a **ripple-carry adder (RCA)**, further exacerbates timing constraints due to its cascaded carry propagation.

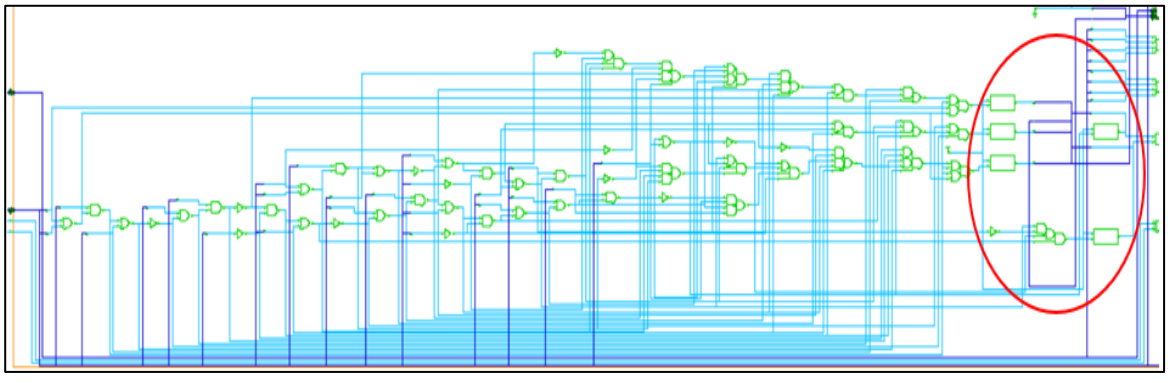

Figure 3.14: Latch inferred at the end of PENC

The synthesis analysis conclusively demonstrates that all adder and subtractor modules are persistently implemented as **ripple-carry adders (RCAs)**, even after aggressive logic

flattening and constraint adjustments. This structural limitation arises from the synthesizer's inability to infer optimized arithmetic components (e.g., carry-lookahead adders, prefix structures) under the current hierarchical design paradigm. While hierarchical decomposition enhances modularity, it inadvertently restricts the tool's capacity to reconfigure critical arithmetic paths for timing efficiency.

3.2.3 Optimized RTL and work-load re-distribution (Model-2)

This implementation focuses on restructuring pipeline stages to help the synthesizer infer components with lower propagation delays. Initially, all addition and subtraction operations were performed using **ripple-carry adders (RCAs)**, which resulted in high latency. Additionally, the alignment unit's comparator and subtractor were implemented as separate components, increasing the area overhead. The normalization stage's latch also contributed significantly to both area and timing inefficiencies. Repetitive tasks—such as performing sign comparisons across multiple stages—further added to the inefficiency.

A key challenge was estimating propagation delay early and targeting its reduction. Based on Figure 3.9, it is evident that **Stage 3 has the lowest propagation delay** (excluding the denormalization unit in count), measured at 2.19 ns for 130 nm and 0.43 ns for 22 nm technology.

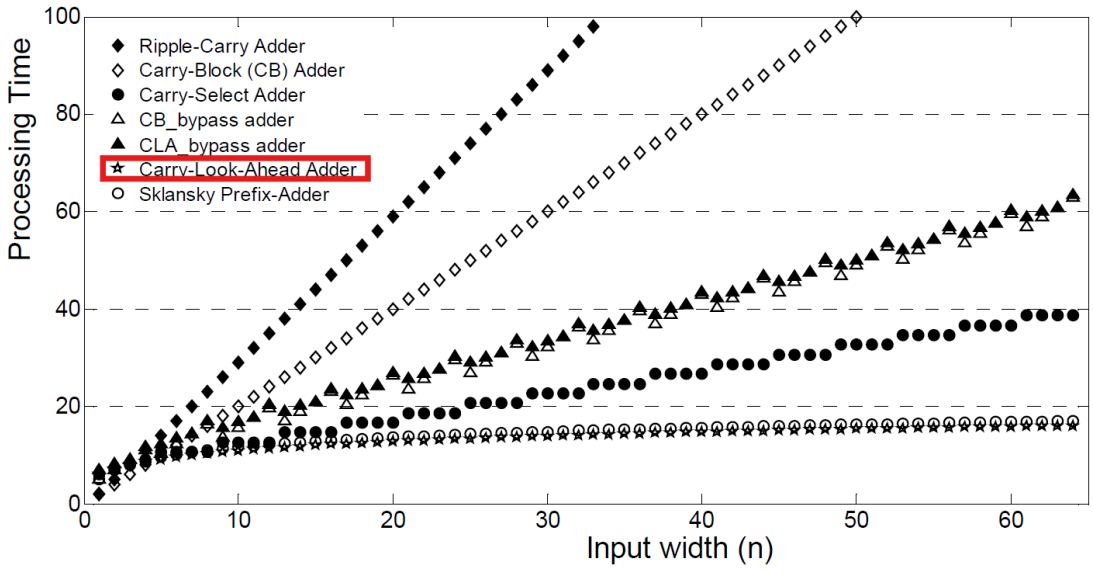

Figure 3.15 : Comparison of processing time of adders [32]

Therefore, the computational load was redistributed—some shifted from Stage 2 to Stage 3, and from Stage 4 to Stage 3—to take advantage of this timing benefit. The goal was to adopt a

faster adder architecture and update the original design (model-1) to improve the speed of binary operations in Stages 2 through 6, while keeping area usage close to that of model-1.

To address RCA's delay limitation, various advanced adder architectures were reviewed in the literature. The selection criteria focused on identifying an adder with processing time under 3 ns to balance speed and area efficiency. Reducing propagation delay alone would not guarantee better overall performance, especially if a complex adder increased the area to the extent that other stages became the new performance bottleneck. According to the analysis in [32], various smallest to the other adder architecture. Additionally, Table 3-3 from [33] shows that, while several adder architectures—excluding RCA and Sklansky—achieve lower path delays than the conventional CLA (highlighted in the red rectangle), they come at the cost of significantly higher area and power consumption. Therefore, the conventional CLA remains the most balanced choice, offering an optimal trade-off among speed, area, power efficiency, and architectural complexity.

Table 3-3 : Performance table of 32-bit CLA variant for 32-28 nm technology (adopted from [33])

Adder Name	Area (μm^2)			Critical Path Delay (ns)	Total Power (μW)
	Cells	Interconnect	Total		
RCA	155.03	10.98	166.01	3.40	42.13
Conventional CLA	350.97	37.44	388.41	2.53	41.69
Ling adder (CLA variant)	392.40	75.21	467.61	2.39	67.48
Proposed CLA	475.50	51.94	**527.44**	**1.13**	**51.33**
Conditional sum adder (CSA)	412.48	77.65	490.13	1.71	69.43
CSLA without BEC (8-7-6-4-3-2-2)	834.35	117.82	952.17	1.20	87.54
CSLA with BEC (8-7-6-4-3-2-2)	683.65	96.22	779.87	1.33	73.43
CSLA without BEC (8-8-8-8)	745.15	95.86	841.01	1.16	79.45
CSLA with BEC (8-8-8-8)	621.89	81.90	703.79	1.28	65.46
Brent–Kung adder (BKA)	419.85	64.40	484.25	2.42	56.65
Sklansky adder	387.06	62.75	449.81	2.74	57.10
Kogge–Stone adder (KSA)	1014.29	174.43	1188.72	0.73	84.99

As a result, the architecture was redesigned by replacing behavioural adders and subtractors with **9-bit and 25-bit CLA**-based units, which reduce propagation delays through parallel carry computation. The two's complement unit was also re-implemented using CLA logic to further enhance performance.

CLA adder design (CLAn):

The Carry Look-Ahead (CLA) adder enhances performance by minimizing carry propagation delay through parallel generation and propagation of carry signals. For two input bits, *A* and *B*, the circuit determines:

Carry Generation: $G_i = A_i . B_i$

Carry Propagation: $P_i = A_i \oplus B_i$

The final sum and carry are

Sum Output: $S_i = P_i \oplus C_i$

Carry Output: $C_{i+1} = G_i + P_i . C_i$

Here C_0 denotes the initial input carry. **An n-bit CLA adder, referred to as CLAn**, consists of parallel half-adder units, a look-ahead carry generator, and an *XOR* block for sum computation—illustrated in Figure 3.16.

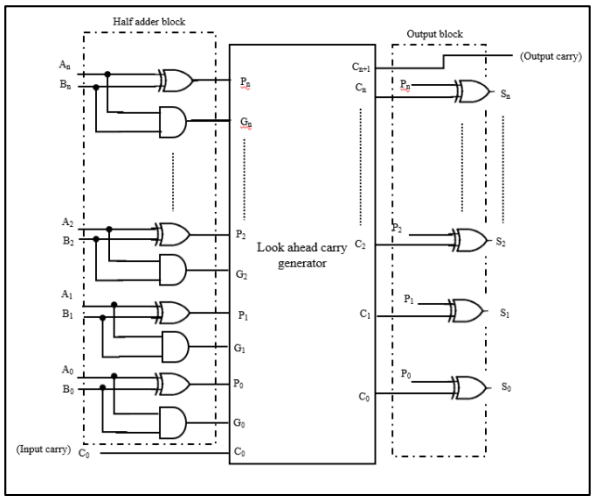

Figure 3.16: n-bit carry looks ahead adder (CLAn)

Conditional Twos complement unit implementation (2Cn):

An **n-bit two's complement unit, denoted as 2Cn**, is implemented using an n-bit Carry Look-Ahead Adder (CLAn) along with n XOR gates. The corresponding figure 3.17 illustrates this configuration. This structure is designed to perform conditional two's complement operations based on a control signal. The two's complement operation requires inverting each bit of a number and then adding 1. *XOR* gates are leveraged here because they can selectively invert bits: when one input of an *XOR* gate is '1', it inverts the other input; when it's '0', the input passes unchanged. In this architecture, input `B` is connected to one side of the XOR gates, while the control signal `C` is fed to the other side of all XOR gates and to the carry-in (C_0) of the CLA adder. The second input of the *CLA* adder (input `A`) is fixed at 0. When the control signal C is 0, the *XOR* gates pass `B` unchanged, and the ***CLA*** performs the addition `B + 0 + 0`, yielding B as the output. When C is 1, the *XOR* gates invert `B (= B')`, and the ***CLA*** performs the

operation $B' + 0 + 1$, which results in the two's complement of B. Thus, the output T equals B when $C = 0$ and equals $-B$ (i.e., two's complement of B) when $C = 1$.

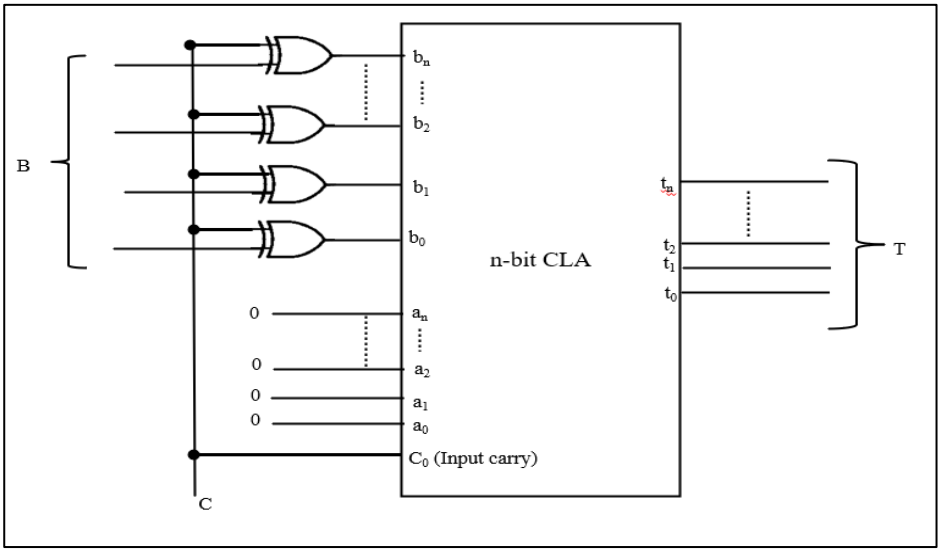

Figure 3.17: n-bit 2's complement unit realization (2Cn)

Re-structure the pipelined stages:

In the updated architecture, significant modifications have been made to optimize the design by integrating the **CLAn and 2Cn** units for that a package called ADDPACK created which contain all the code of architectural **CLA9 (9-bit CLA adder)**, *CLA25 (25-bit CLA adder)* and **CLA25C (CLA25 excluding its input half adder block)**. While the **normalization stage** remains unchanged from **Model-1** due to its already efficient design, other stages have undergone restructuring to redistribute the computational workload more effectively. Specifically, the alignment stage (previously Stage-2) has been simplified by offloading the shifting operation to the Stage-3. Additionally, the half adder section from the ***CLA25*** in the mantissa addition stage (formerly Stage-4) has been moved to Stage-3. Reflecting these functional changes, the stage names have been updated: the **alignment unit** is now referred to as the **Compare Unit**, and the **Twos Complement of Mantissa unit** has been renamed the **Shift Complement Unit**. Importantly, the Deformalize and Normalize stages remain dedicated and isolated; their workloads are not redistributed or shared with any other stages. This is because these stages are not only critical to the floating-point adder but are also foundational in any floating-point arithmetic implementation. In fact, as will be shown in the next chapter, these two stages are used as a common component across the entire CORDIC architecture.

In the **Compare Unit (Stage-2),** a *9-bit CLA*-based subtractor is introduced, replacing the traditional *RCA* subtractor and comparator. In this design, the exponent of input `A` is directly fed into the *CLA9* unit, while the exponent of input `B` is first inverted using *NOT* gates and then provided as the second input to *CLA9*. Besides input carry sated to 1. The *CLA9* outputs a 9-bit result, where the most significant bit **(MSB)** represents the **sign** of the difference. If the *MSB* is 1, the result is negative, indicating that the exponent of `B` is greater than that of `A` (i.e. `EB > EA)`, and thus a two's complement operation is required. The remaining least significant 8 bits of the *CLA9* output are fed into an **8-bit two's complement unit (2C8)**, whose control input is driven by the MSB of *CLA9*. This output is referred to as `diffEX`. Additionally, the MSB of *CLA9*, denoted as `EXCOM`, not only governs the two's complement operation but also functions as a comparator output to identify the larger exponent. Therefore, this stage produces the following outputs: `bigEX` (8-bit larger exponent), `diffEX` (8-bit exponent difference), `EXCOM` (1-bit comparison result), `AMAN` and `BMAN` (24-bit mantissas of inputs `A` and `B`), and `ASIGN` and `BSIGN` (1-bit sign bits of `A` and `B`, respectively). The designed compare unit has been illustrated in Figure 3.18

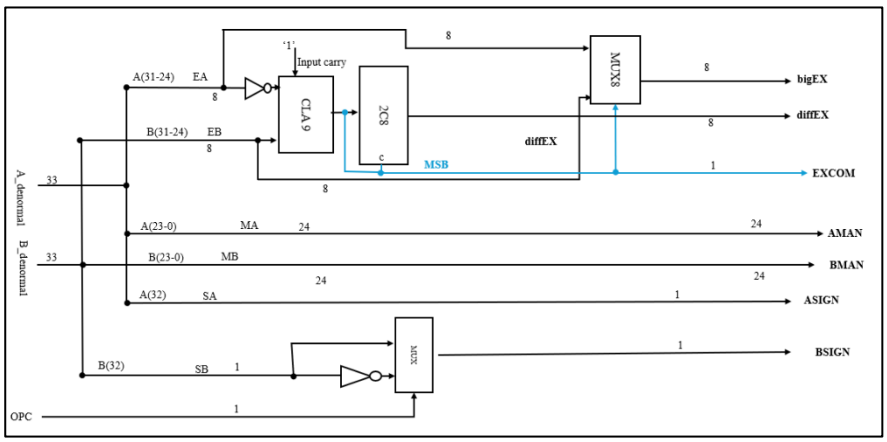

Figure 3.18: Compare unit (after reducing workload of alignment unit)

The **Shift Complement Unit (Stage-3)** consists of three key components: a barrel shifter, a complement logic block, and the parallel half-adder portion of the *CLA25* adder as depicted in Figure 3.19. This stage processes the inputs `AMAN, BMAN, ASIGN, BSIGN, EXCOM, and diffEX`. First, the signs of `ASIGN` and `BSIGN` are compared using an XOR gate, producing a signal named `SIGNCOM`. If the signs are the same, `SIGNCOM` is 0; if they differ, it is 1. For the shifting process, a barrel right shifter (like that in model-01) is used. The `EXCOM` signal determines the select line of a 24-bit multiplexer (`MUX24`), ensuring the appropriate mantissa (with the smaller exponent) is shifted. After this operation, the shifted mantissas are

labelled *AMAN_S* and *BMAN_S*. The complement logic follows. If *SIGNCOM* is 1 (indicating differing signs), the circuit checks the sign of *A* (*ASIGN*). This is done using an AND gate. If *ASIGN* is '1', *AMAN_S* is inverted to form *AMAN_COMPLI*. Otherwise, *BMAN_S* is inverted to form *BMAN_COMPLI*. If *SIGNCOM* is 0 (indicating equal signs), no inversion is required, so *AMAN_COMPLI* and *BMAN_COMPLI* remain the same as *AMAN_S* and *BMAN_S*, respectively. Next, depending on the value of *SIGNCOM*, a sign bit is concatenated: if *SIGNCOM* is '0', a '0' is prepended; if it's '1', the corresponding sign bit (*ASIGN* or *BSIGN*) is prepended. These adjusted values are then passed into the half-adder block of the *CLA25* unit. The outputs from this stage are the carry propagate *(P)* and carry generate *(G)* signals, along with *SIGNCOM*. By generating and forwarding *SIGNCOM* instead of rechecking the sign in later stages, redundant comparisons in subsequent stages are eliminated, optimizing the design.

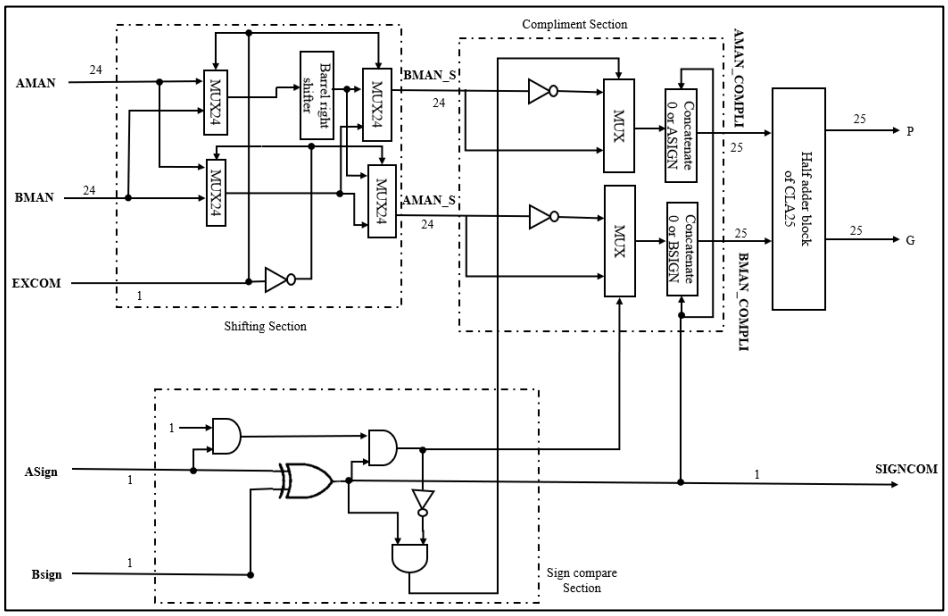

Figure 3.19: Shift-Compliment unit (after shifting some workload from stage-2 and 4 to stage-3)

The **Add Mantissa Unit (stage-4)** consists of the look-ahead carry generator and the output stage of a **25-bit carry look-ahead adder (CLA25)**. This stage takes as input the *P* (propagate) and *G* (generate) signals produced in the previous stage (Shift Complement Unit), along with *SIGNCOM*, which is connected to the initial carry-in input (C_0) of the adder. These inputs allow the *CLA25* to perform fast addition by computing the carries in parallel, significantly reducing delay. The result of the addition is a 25-bit output called *SUM_MAN*, representing the sum of the aligned and processed mantissas.

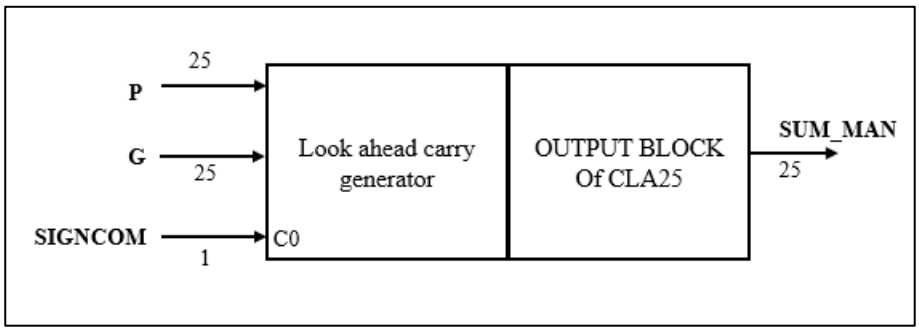

Figure 3.20: Add mantissa unit (contain last two block of CLA)

The **Sum Complement Unit (stage-5)** is implemented (as shown in Figure 3.21) using a **CLA-based two's complement block (2C25)**. The control input of the **2C25** is driven by the logical *AND* of the MSB of `SUM_MAN` and the `SIGNCOM` signal. This configuration ensures that the two's complement operation is applied only when the input signs differ (`SIGNCOM` = 1) and the MSB of `SUM_MAN` = 1, indicating a negative result that needs to be corrected. Otherwise, the output remains unchanged as `SUM_MAN`. Additionally, if the MSB of `SUM_MAN` is '1' while `SIGNCOM` = '0', it indicates an overflow, which triggers a one-bit right shift of the sum. This condition also activates a *CLA9* adder to increment the `BIGEX` value by '1', compensating for the shift in the exponent. The `SUMSIGN` (final sign of the result) is determined similarly to the approach used in model-1, based on whether the signs were originally the same or different. It should be noted that concatenating the outputs of this stage- `SUMSIGN`, `BIGEX`, and `SUMSIGN_COMPLI` (i.e., `SUMSIGN` & `BIGEX` & `SUMSIGN_COMPLI`) forms the 33-bit pre-normalized result of the sum.

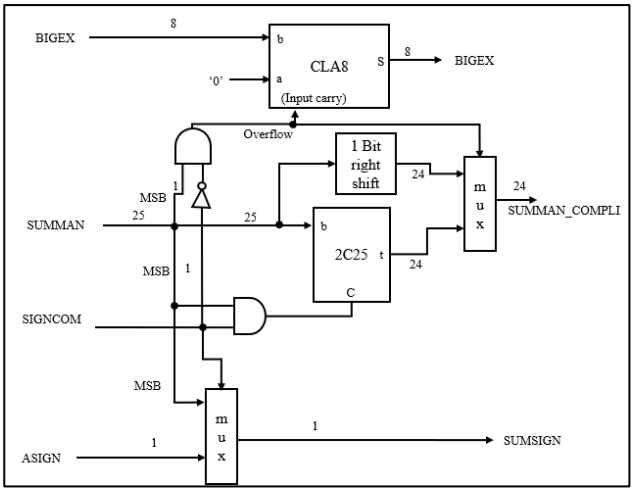

Figure 3.21: CLA25 based twos complement unit

The **Normalization Unit** retains the overall architecture of model-1 but introduces two key optimizations. First, the `first_1` block, responsible for detecting the leading '1' in the mantissa, is implemented as a 24-bit to 5-bit priority encoder using a `when...else` coding style replacing loop based leading zero counter. This structure allows synthesis tools to infer the priority encoder using a hierarchical approach while **eliminating unwanted latches**. Second, the 8-bit subtractor previously used to adjust the exponent is now replaced with a combination of a ***CLA9*** adder and a ***2C9*** two's complement unit, maintaining functional accuracy while aligning with the updated *CLA*-based architecture. Despite these internal improvements, the unit's overall role and behaviour remain consistent with model-1. The Figure 3.22 depicted the chang of normalization unit.

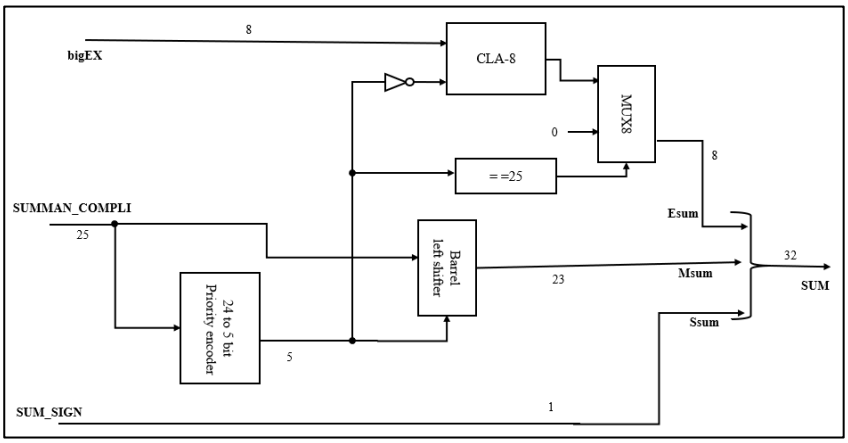

Figure 3.22: Redesigned normalization unit

Register placement and top Hierarchy:

Like model-1, **model-2** also incorporates six sets of registers, named `REG1` through `REG6`, positioned after each corresponding processing stage (*steps 1 to 6*) (as demonstrated in Figure 3.23). These registers serve as pipeline boundaries, allowing each stage of the floating-point adder to operate independently and concurrently. This pipelined architecture enhances throughput by enabling continuous data processing across stages without waiting for the entire computation to complete. The inclusion of these register sets ensures synchronization and data stability between stages, making this design the final optimized architecture of the 32-bit floating-point adder in model-2.

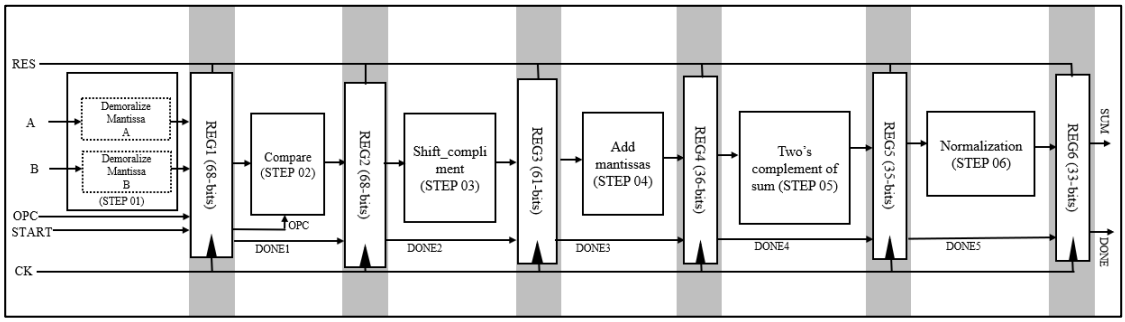

Figure 3.23: Pipelined architecture of floating-point adder (model-2)

Table 3-4 outlines all the registers used in the design along with their respective bit allocations. Each register block—`REG1` through `REG6`—retains the same number of flip-flops as in model-1, maintaining consistency in resource usage. The only exception is `REG2`, which requires 9 additional bits to accommodate two extra signals: the 8-bit *`DiffEX`* (exponent difference) and the 1-bit *`SIGNCOM`* (sign comparison output). This adjustment ensures accurate data propagation and control logic for the modified architecture without significantly increasing complexity.

Table 3-4: Allocated registers for corresponding Data and their length (i.e. flipflop) (model-2)

Register block	Data stored and its length	Total length (bits)
REG1	*A*: 33-bits *B*: 33-bits *OPC*: 1-bit *DONE1*: 1-bit	68
REG2	*BigEX*: 8-bits *DiffEX*: 8-bits *EXCOM*: 1-bit *AMAN_S*: 24-bits *AMAN_S*: 24-bits *ASIGN*: 1-bit *BSIGN*: 1-bit *DONE2*: 1-bit	68
REG3	*BigEX*: 8-bits *AMAN_COMPLI*: 25-bits *AMAN_COMPLI*: 25-bits *ASIGN*: 1-bit *SIGNCOM*: 1-bit	61

	DONE3: 1-bit	
REG4	*BigEX*: 8-bits *SUM_MAN*: 25-bits *ASIGN*: 1-bit *SIGNCOM*: 1-bit *DONE4*: 1-bit	36
REG5	*BigEX*: 8-bits *SUMMAN_COMPLI*: 24-bits *SIGNCOM*: 1-bit *SUM_SIGN*: 1-bit *DONE5*: 1-bit	35
REG6	*SUM*: 32-bits *DONE*: 1-bit	33

3.2.4 Gate level synthesizes of the model-2 and compare with model-1:

Time and area analysis:

For the **UMC13** technology, the minimum clock period required is **3.5 ns**, ensuring reliable operation under timing constraints. This value provides a **positive slack** of **1.0 ns (28.6%)** for **setup time** analysis (i.e., maximum propagation delay) and **0.07 ns (2%) for hold time** analysis (i.e., minimum propagation delay), indicating that both setup and hold requirements are met comfortably.

In comparison, the **GF22** library allows for a significantly faster design, with a minimum clock period of just **0.6 ns**. Under this condition, the **setup time slack is +0.137 ns (23%)**, and the **hold time slack is +0.0099 ns (1.6%)**, again confirming successful timing closure.

The comprehensive evaluation of timing performance, area, and cell usage for both technology nodes are collectively presented in Figure 3.24, highlighting the architectural efficiency and scalability across process technologies.

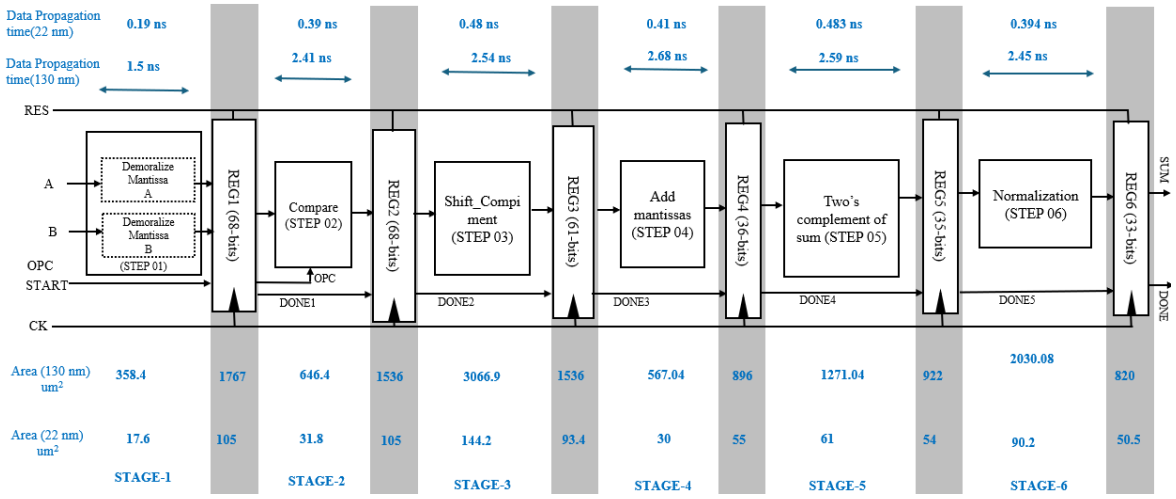

Figure 3.24 : Propagation time and occupied area of each stage (model-2)

Analysis of Mapping:

The **Compare Unit**, which incorporates *CLA9* and *2C8*, is efficiently optimized in the design. As illustrated in Figure 3.25, the *CLA9* used for exponent comparison is closely tied to the *2C8* used for calculating the exponent difference. Importantly, the *CLA9* is not synthesized as two separate components—one for subtraction and one for comparison—but rather inferred as a unified circuit by the synthesizer. This combined implementation results in a more area-efficient and timing-optimized design, leveraging shared logic for both operations and minimizing redundant hardware.

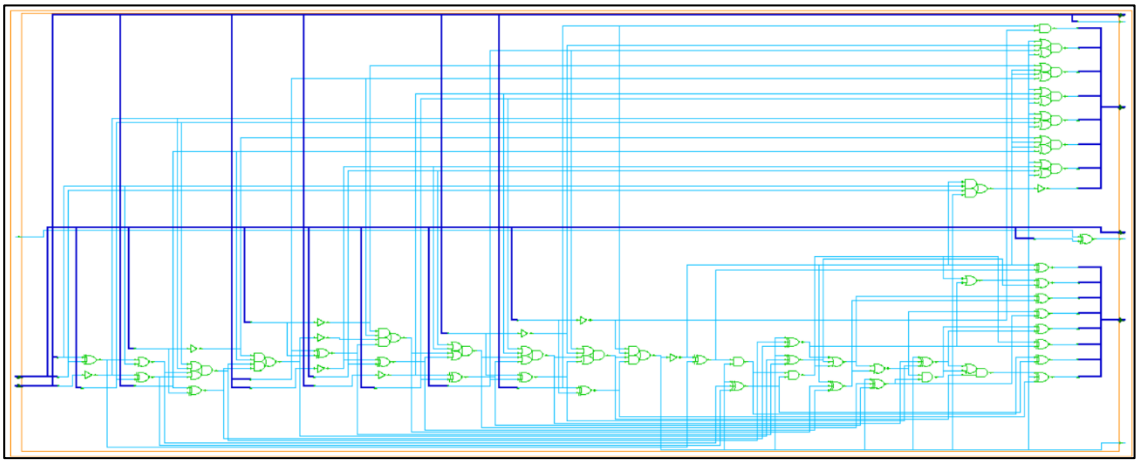

Figure 3.25: Optimized Compare unit

In the **Shift_Complement unit**, the behaviour of the right shifter differs based on the technology used. For **UMC13** technology, the right shifter is not inferred in a hierarchical manner; instead, it is synthesized as a flat structure that is combined with other logic components within the same stage. This results in improved performance and lower propagation

delay due to tight integration. On the other hand, in **GF22** technology, the synthesizer infers the right shifter in a hierarchical form, which slightly increases its propagation delay compared to other stages. However, this delay is still less than that observed in Model-1, indicating overall improvement despite the architectural differences in synthesis.

The look-ahead carry generator and *CLA* output logic are inferred as shown in Figure 3.26. While the implementation is not a **pure flat carry look-ahead adder**, the synthesis tool generates a structure, which effectively optimizes parallelism for speed. This inference indicates that the synthesis tool has successfully parallelized the operations to achieve better timing performance. Although it's technically possible to force the synthesis of a true *CLA* using the `don't_touch` attribute, doing so is not advisable, especially for carry look-ahead adders. According to Synopsys recommendations, this constraint prevents the tool from performing essential optimizations, resulting in increased cell area, higher fanout, and ultimately worse propagation delay. Overall, the sum mantissa unit has been inferred efficiently, leveraging synthesis-driven optimizations rather than rigid structural constraints.

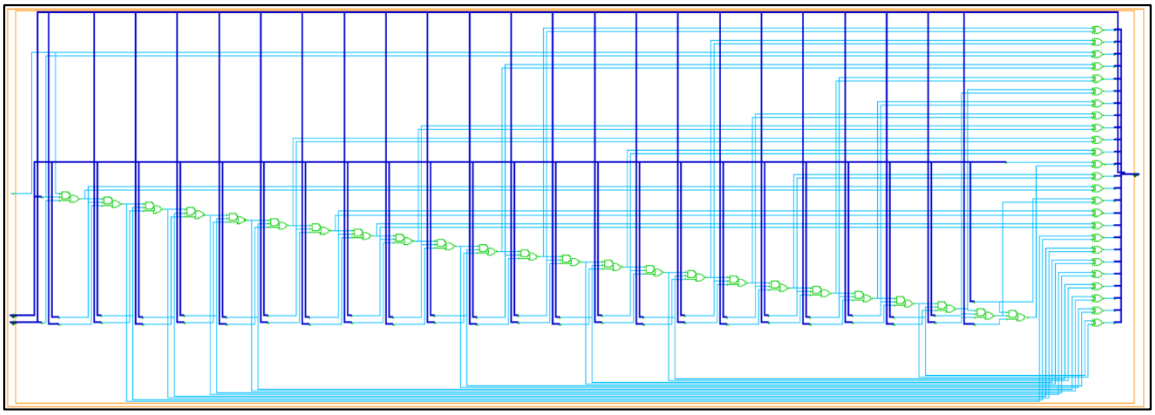

Figure 3.26: 25-bit CLA optimized as tree structure

The **twos complement of sum** unit demonstrates the highest level of optimization in the architecture. This is largely due to its implementation using a 24-bit carry look-ahead adder based 2's compliment **(2C24)**, which inherently involves a high cell count. The synthesizer effectively streamlined the design, achieving an efficient balance between performance and resource usage. Additionally, the overflow control section, which includes a fixed 1-bit right shifter and a 9-bit subtractor, has been logically optimized by realizing flat logic circuit (i.e. no individual shifter and 1 adder). Rather than implementing these elements as standalone blocks,

the synthesizer leveraged their logical interrelations, reducing redundancy and improving overall efficiency.

In the **normalization unit**, key components such as the left shifter, priority encoder, and *CLA9* are efficiently inferred and optimized by the synthesizer as depicted in Figure 3.27. Importantly, no unwanted latches are generated during synthesis, which significantly contributes to reduced propagation delay, area consumption, and power usage.

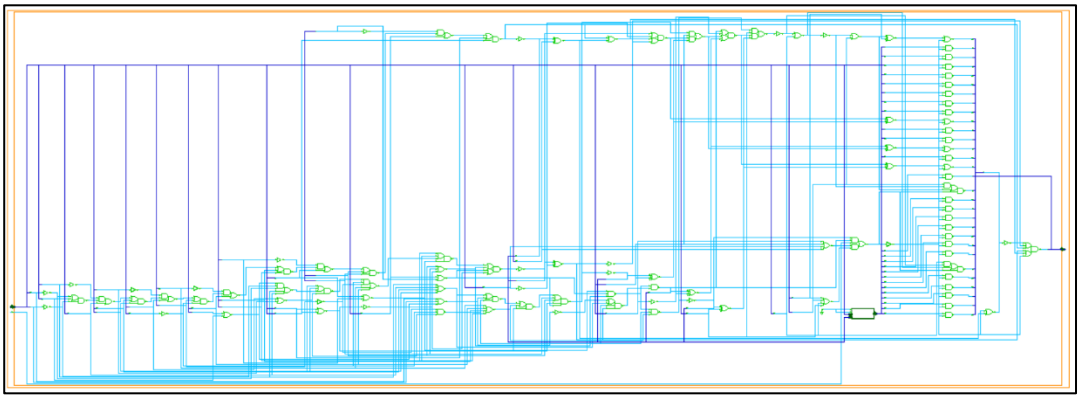

Figure 3.27: Inference of normalization unit without latch

Report timing:

According to the Table 3-5, Model-2 demonstrates a significant performance improvement over Model-1, even when evaluated with a more generous timing margin. Model-2 achieves nearly **1.5×** the throughput compared to Model-1. Additionally, when implemented using 22 nm technology, the design becomes approximately **six times faster** than its 130 nm counterpart. This reflects the principle that throughput is nearly inversely proportional, and minimum clock

Table 3-5: Time analysis report (Pipeline FP-adder)

Techn-ology	Clock period (Frequency)		Setup time analysis (Worst case) Slack (% of clock period)		Hold time analysis (Best case) Slack (% of clock period)	
	Model-1	Model-2	Model-1	Model-2	Model-1	Model-2
UMC13	6 ns (166.7 MHz)	3.5 ns (285.7 MHz)	+0.95 ns (16%)	+1.0 ns (28.6%)	+0.07 ns (1.17%)	+0.07 ns (2%)

GF22	1.1 ns (0.909 GHz)	0.6 ns (1.66 GHz)	+0.18 ns (16.4%)	0.137 ns (23%)	+0.0098 ns (0.9%)	0.0099 ns (1.6%)

period is directly proportional to the technology node length, as evident from the ratio: (1.66 GHz / 285.7 MHz) ≈ 3.5 ns / 0.6 ns ≈ 130 nm / 22 nm. **It is important to note that, since the pipelined architecture produces one output per clock cycle, the throughput in GSamples/s is directly equal to the clock frequency in GHz**.

Furthermore, as illustrated in Figure 3.28 (a) shows the percentage contribution of each stage to the total propagation time, while the Figure 3.28 (b) illustrates the absolute propagation time in units. Model-2 achieves more uniform workload distribution across its pipeline stages compared to Model-1. In Model-1, the alignment unit (stage-2) consumed the highest propagation time (26%–29%), creating a bottleneck that limited the overall speed. In contrast,

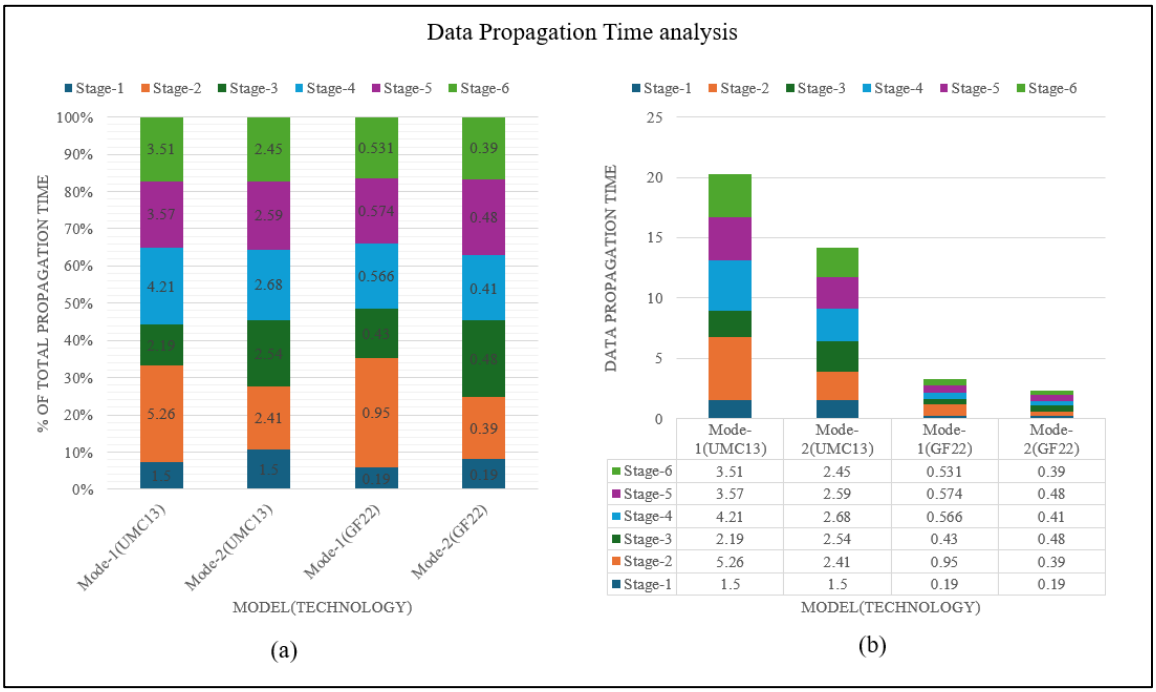

Figure 3.28: Propagation time of FP-pipelined adder: (a) percentage of total propagation time required for each stage, (b) Total propagation time for each model and technology

Model-2 redistributes the workload more evenly across stages 2, 3, and 4, significantly balancing the timing load among pipeline stages and, the highest propagation time required in stage 3 and 5. This balanced architecture contributes directly to the increased throughput of Model-2. Along with, Model-1 exhibits the almost **25% higher total delay** (figure-3.2(b)), with

significant contributions from Stage-2 and Stage-4, whereas Model-2 shows a considerable reduction in these stages, leading to a total delay of approximately **14.16 ns** compared to **20.24 ns** in Model-1. Similar trend also seen for **GF22** technology.

Report Area:

Table 3-6 presents the area report following post gate-level synthesis, illustrating that Model-2 occupies slightly less area and uses a smaller number of cells compared to Model-1, despite achieving higher speed. This clearly indicates that Model-2 is more synthesis-optimized and better suited for efficient hardware implementation. Across both UMC13 and GF22 technologies, Model-2 consistently requires about **5% less area** than Model-1. Additionally,

Table 3-6: Report area and comparison of FP-pipelined adder

	UMC13		**GF22**	
	Model-1	**Model-2**	**Model-1**	**Model-2**
No. of Cells	1537	1431	1607	1473
No. of combinational cell	1220	1122	1290	1163
No. of Sequential cell	301	301	301	301
No. of Buffer/Inverter	249	82	316	126
No. of reference	12	11	11	10
BUF/INV area	963.8	318.72	42	16.8
Combinational Area	10841.6	9840.6	516.24	472.24
Non-combinational area	7980	7705.6	458.12	460.8
Total cell Area	18521.6	17546	974.31	933

Model-2 does require three more sequential cells, which can be attributed to the 8-bit storage needed for the exponent difference, while eliminating the 5-bit latch from the normalization unit. Moreover, a significant observation is that the UMC13 technology node demands nearly 20 times more area than the GF22 node, highlighting the advantages of smaller, more advanced fabrication technologies. This area relationship aligns closely with the square of the technology node ratio, i.e., ~ **17546/933 ≈ (130/22)** ^{2} (approx.)

Table 3-7 highlights the combinational area of each stage, showing a notable reduction in area for Stage-2 (646.4 vs. 2785.3) and Stage-4 (567 vs. 700.2) in Model-2 compared to Model-1, while Stage-3 experiences an increase almost doubled (3066.9 vs. 1779.2), reflecting a deliberate redistribution of workload between stages. A similar trend is observed in the GF22 technology node, where Stage-2 and Stage-4 area requirements significantly decrease (31.7 vs.

Table 3-7: Area occupied by different stage of pipelined adder

STAGE	UMC13		GF22	
	Model-1 (um^2)	Model-2 (um^2)	Model-1 (um^2)	Model-2 (um^2)
STAGE-1	358.4	358.4	17.3	17.3
STAGE-2	2785.3	646.4	122.87	31.7
STAGE-3	1779.2	3066.9	85.53	144.17
STAGE-4	700.2	567	39.2	29.35
STAGE-5	1258.24	1271	60.57	60.97
STAGE-6	2172.2	2030	98.5	90.19

122.87 and 29.35 vs. 39.2, respectively), though Stage-3 area increases (144.17 vs. 85.53). These changes are visually supported by the pie charts in Figure 3.29, which illustrate the area distribution across stages for each model and technology. In Model-1, Stage-2 and Stage-3 combinedly consume almost 50% area. In contrast, Model-2 shows a concentration of area in

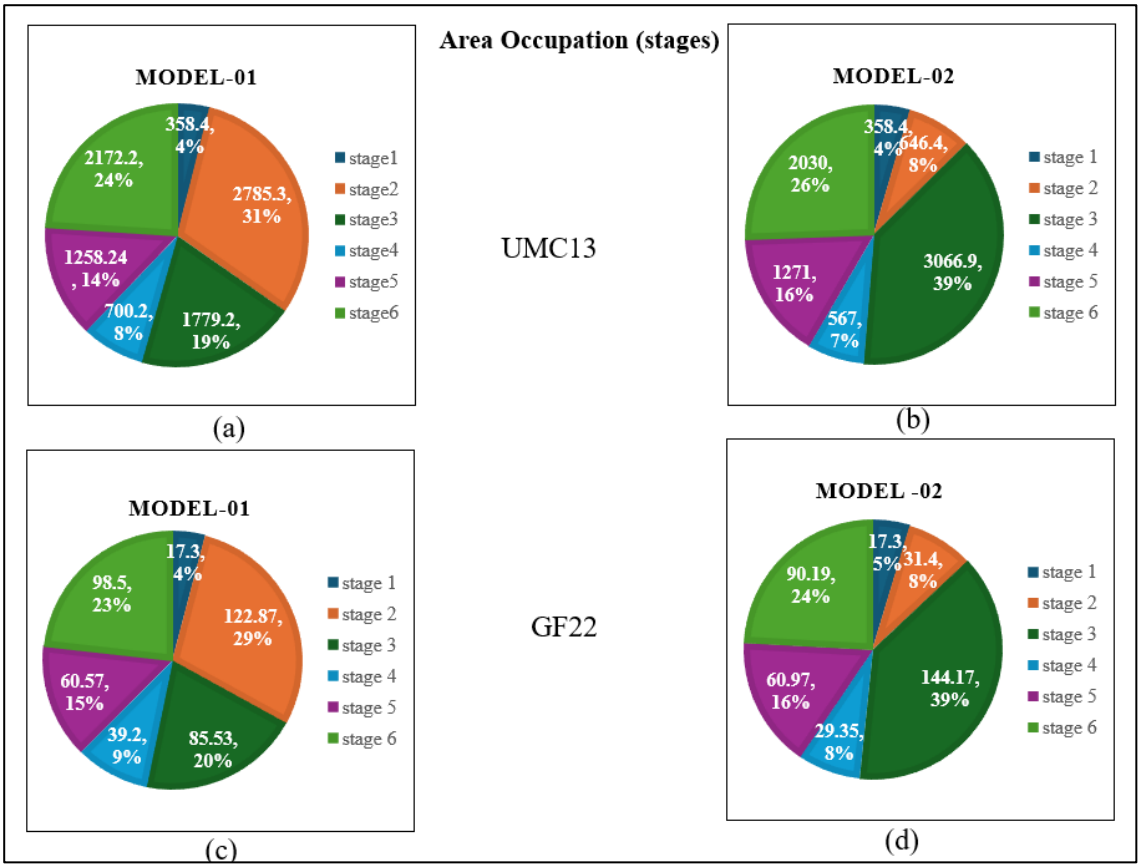

Figure 3.29: Area Occupation by different stage: (a) model-1(UMC13), (b) Model-2 (UMC13), (c) model-1 (GF22), (d) model-2 (GF22)

Stage-3, which accounts for approximately 39% of the total area across both technologies which is the early indication of routing complexity that may arise during physical synthesis. But one of the interesting points is stage-1 and stage-6 i.e. **normalization and denormalization unit combinedly occupy almost 30%** area of the whole adder which is the motivation of Consolidate into a single use in whole CORDIC implementation.

Report Power:

The power analysis, as illustrated in the Table 3-8 and depicted in Figure 3.30, reveals a clear trend across both UMC13 and GF22 technologies. In both cases, Model-2 demonstrates nearly double the dynamic power consumption compared to Model-1. This rise in dynamic power aligns with the increase in clock frequency and throughput, highlighting a fundamental trade-off between speed and power efficiency. Additionally, GF22 technology consistently consumes almost twice the power of UMC13, primarily due to its smaller feature size, which enables

Table 3-8: Report Power of pipelined FP-adder

	UMC13		**GF22**	
	Model-1	Model-2	Model-1	Model-2
Cell internal power (uW)	524.5	936.5	1027.6	1938.4
Net switching power(uW)	86.51	143.1	38.64	70.03
Total Dynamic Power (uW)	611	1079.6	1073.3	2008.5
Cell Leakage power (uW)	3.3	3.18	3.42	3.44
Total Power Consumption (uW)	614.3	1082.8	1069.7	2011.9

higher dynamic activity and clock speeds. Interestingly, even though the 22 nm FD-SOI node exhibits significantly higher dynamic power—dominated by cell internal power rather than switching—the switching component still only accounts for about 4% of total power, indicating a high-power density typical of advanced nodes. On the other hand, leakage power remains nearly constant between Model-1 and Model-2, with only a slight increase in GF22 due to its shorter channel length. In summary, while Model-2 achieves notable improvements in performance and area efficiency, it comes at the cost of increased total power consumption.

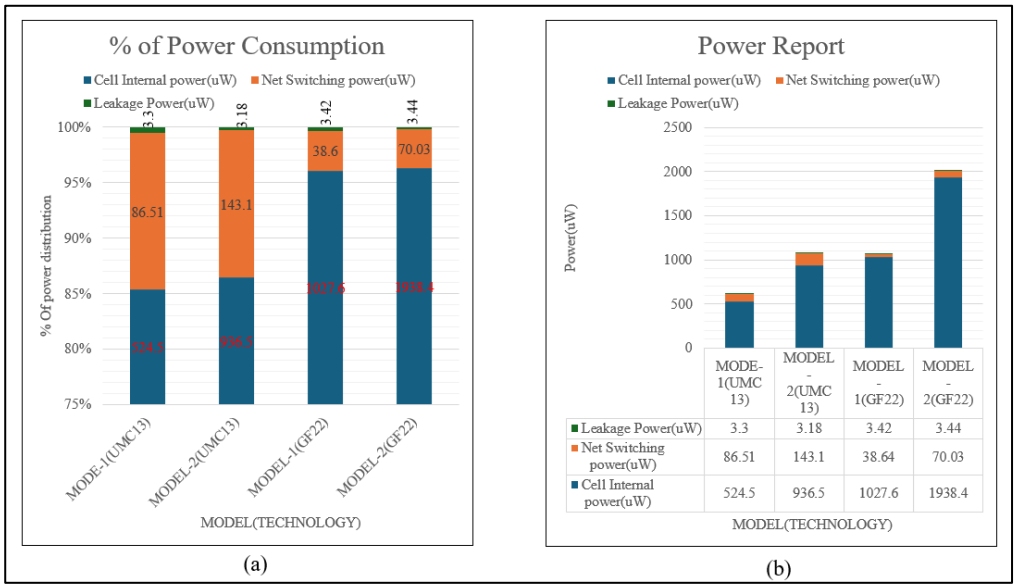

Figure 3.30: Power report visualization of pipelined FP-adder (a) % of Power consumption (b) power report visualization

Trade-off:

The trade-off, driven by increased dynamic behaviour, highlights the critical balance between speed, area, and power consumption—clearly depicted as triangular in Figure 3.31 representing frequency (speed), area, and power. As technology scales down, this triangle shifts leftward, signifying reduced area, increased speed, and a corresponding rise in power demand. Notably, the triangle representing Model-2 consistently lies to the left of that of Model-1 across both technologies. This suggests that Model-2 is more area-efficient and consistently faster than Model-1, though it comes with the drawback of increased power consumption. Specifically, Model-2 achieves a **1.5x** improvement in speed (throughput) over Model-1, but this gain is offset by a **1.8x** increase in power usage i.e. throughput increment proportional to extra power demand. Thus, the visual representation effectively captures the optimization trade-off, where enhanced performance and compact design come at the cost of increased power usage.

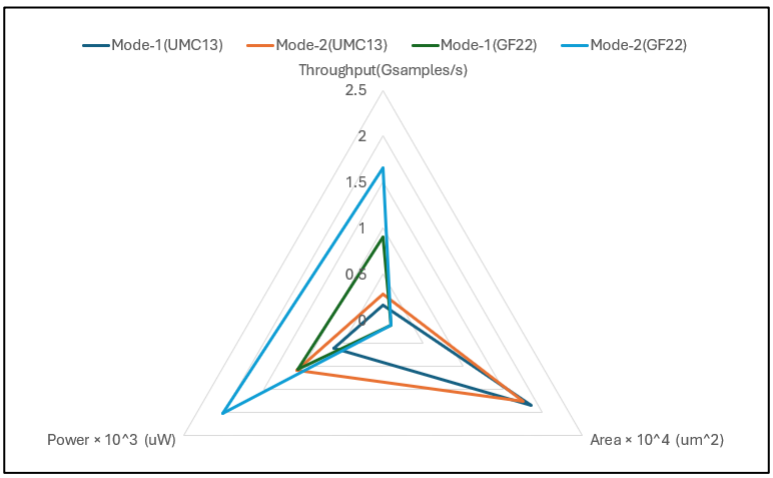

Figure 3.31: Speed, Area and power triangle to show trade-off (Pipelined architecture)

Owing to the superior speed and area efficiency of model-2, it has been selected as the floating-point adder/subtractor for the CORDIC design. Therefore, in all subsequent discussions, the term **"pipelined adder"** or **"addsub"** specifically refers to the model-2 version of the pipelined architecture.

3.2.5 Simulation of Pipelined FP-adder/subtractor:

Functional verification was carried out to ensure that the system operates as expected. Verification was conducted at each stage and for all models. After identifying and fixing any implementation issues and addressing network-related problems, it was confirmed that the verification results remained consistent throughout. The simulation and verification processes involved two key functions for data conversion. The custom **`F_RTOFP32`** function converts real numbers into the IEEE 754 single-precision (32-bit) floating-point format, while **`ieee754_to_real`** performs the reverse, converting the 32-bit binary format back to real values. All data was recorded in a text file using the `std.textio` library. Behavioural verification was categorized into four key areas: Data processing verification, range verification, and control signal activity verification, and structural validation.

Note: In all figures illustrating signal waveform, the output signal appears six clock cycles (latency) after the input is applied. In some cases, the data valid signal remains high (logic 1) consistently due to the pipelined nature of the design, where a valid output is produced in every clock cycle following the input. Besides all the table and figure show represent the behaviour of both model-1 and model-2.

Verification of Data process:

Data processing verification was performed to assess whether the architecture can accurately handle a variety of data that represent the full range of possible input scenarios. Although an infinite number of test cases could be used to validate addition and subtraction operations, a carefully selected set of nine representative samples was used. These test cases include combinations such as both operands being positive real numbers, both negative, one positive and one negative, zero handling, and cases where the mantissa is zero. Table 3-9 presents the test samples along with the corresponding outputs obtained from both the **Verdi** simulator and the behavioural model implemented in **MATLAB**. In the MATLAB model, input and output data conversions were handled using built-in functions for real-to-float and float-to-real transformations. The results from both the **Verdi** simulator and the **MATLAB** model matched exactly, indicating the correct behaviour of the designed floating-point adder/subtractor. This also confirms the accuracy of the custom data conversion functions used in the testbench.

Table 3-9: Addition and subtraction verification for both pipelined (model-1 and model-2) and serial architecture of FP-adder

Sample index	**Remark of the input sample**	**input: (A, B)**	**Operation (OPC=0: ADD, OPC = 1: SUB)**	**OUTPUT (simulator)**	**Output (MATLAB)**
1	Arbitrary FP number	(3.86, 1.069)	ADD	4.929 (409D_BA5E)	4.929 (40D_BA5E)
			SUB	2.7910 (4032_9FBE)	2.7910 (4032_9FBE)
2	real number (0.f format)	(0.2, 0.346)	ADD	0.546 (3F0B_C6A7)	0.546 (3F0B_C6A7)
			SUB	-0.146 (BE15_8106)	-0.146 (BE15_8106)
3	Both zero 0 (zero handling)	(0,0)	ADD	0.0 (0)	0.0 (0)
			SUB	0.0 (0)	0.0 (0)
4	When one input zero (zero handling and maximum exponent difference)	(0, 10.56471)	ADD	10.5647 (4129_090D)	10.5647 (4129_090D)
			SUB	-10.5647 (C129_090D)	-10.5647 (C129_090D)
5	When inputs are non-zero, but output is zero	(3.86, 3.86)	ADD	4.929 (409D_BA5E)	4.929 (40D_BA5E)
			SUB	0.0 (0)	0.0 (0)
6	When all the mantissa is zero	(16, 8)	ADD	24.0 (41C0_000)	24.0 (41C0_000)
			SUB	8.0 (4100_000)	8.0 (4100_000)
7	Input with same sign (+ve)	(105.3, 10.681)	ADD	115.981 (42E7_F646)	115.981 (42E7_F646)
			SUB	94.619 (42BD_3CEE)	94.619 (42BD_3CEE)
8		(-105.3, -10.681)	ADD	-115.981 (C2E7_F646)	-115.981 (C2E7_F646)

	Input with same sign(-ve)		SUB	-94.619 (C2BD_3CEE)	-94.619 (C2BD_3CEE)
9	Input with different sign	(-105.3, 10.681)	ADD	-94.619 (C2BD_3CEE)	-94.619 (C2BD_3CEE)
			SUB	-115.981 (C2E7_F646)	-115.981 (C2E7_F646)

Range verification:

This verification aims to confirm that the floating-point adder/subtractor operates correctly across the full range of the 32-bit IEEE 754 format, including its behaviour near zero, at the upper and lower bounds, and around the **machine epsilon (ε)**. Table 3-10 show all input samples and corresponding output obtained from **VHDL model** and **MATLAB** model with significance remark.

Table 3-10: Table of Range verification report for both pipelined (model-1 and model-2) and serial architecture of FP-adder

Sample index	**Remark of the input sample**	**input: (A, B)**	**Operation**	**OUTPUT (simulator)**	**Output (MATLAB)**
1	Input to set to reach the positive limit of the 32-bit IEEE754	$(2.402823\times10^{38}, 1\times10^{38})$	ADD	3.402823×10^{38} (7F7F_FFFD)	3.402823×10^{38} (7F7F_FFFD)
2	Input to set to reach the negative limit of the 32-bit IEEE754	$(-2.402823\times10^{38}, -1\times10^{38})$	ADD	-3.402823×10^{38} (FF7F_FFFD)	-3.402823×10^{38} (FF7F_FFFD)
3	Input to cross the positive limit of the number range	$(2.402824\times10^{38}, 1\times10^{38})$	ADD	$+\infty$ (7F80_0000)	$+\infty$ (7F80_0000)

4	Input to cross the negative limit of the number range	(-2.402824×10^{38}, -1×10^{38})	ADD	-∞ (FF80_0000)	-∞ (FF80_0000)
5	Closest positive proximity to zero	(4.0×10^{-38}, -2.83×10^{-38})	ADD	1.18×10^{-38} (0080_7D99)	1.18×10^{-38} (0080_7D99)
6	Closest negative proximity to zero	(-4.0×10^{-38}, 2.83×10^{-38})	ADD	-1.18×10^{-38} (8080_7D99)	-1.18×10^{-38} (8080_7D99)
7	closer than the positive proximity limit of zero	(4.0×10^{-38}, -2.83×10^{-38})	ADD	+1.16451×10^{-38} (007F_66D7) (not precise)	+1.16451×10^{-38} (007F_66D7) (not precise)
8	closer than the negative proximity limit of zero	(-4.0×10^{-38}, 2.83×10^{-38})	ADD	-1.16451×10^{-38} (807F_66D7) (not precise)	-1.16451×10^{-38} (807F_66D7) (not precise)
9	interval machine epsilon (ε) checking	(1.18×10^{-38}, 1.18×10^{-38})	ADD	2.36 ×10^{-38} (0100_7D99)	2.36 ×10^{-38} (0100_7D99)
10	out of interval machine epsilon (ε)	(1.18×10^{-38}, 1.17×10^{-38})	ADD	1.18×10^{-38} (80_7D99) (out of resolution)	1.18×10^{-38} (80_7D99) (out of resolution)

To evaluate the range limits, input pairs were chosen such that their sum would approach or exceed the representable limits of the number system. Specifically, the inputs (2.402823 × 10^{38}, 1 × 10^{38}) and (2.402824 × 10^{38}, 1 × 10^{38}) were used. The outputs were 3.402823 × 10^{38} and +∞, respectively. The expected result for the second pair is 3.402824 × 10^{38}, which is beyond the positive range of the IEEE 754 single-precision format. This confirms that the design correctly

handles overflow by producing +∞ and accurately processes values up to approximately 3.402823 × 10^{38}, the maximum representable positive value.

Similarly, to test the negative range, the inputs (-2.402823 × 10^{38}, -1 × 10^{38}) and (-2.402824 × 10^{38}, -1 × 10^{38}) were used. The corresponding outputs were -3.402823 × 10^{38} and -∞, verifying correct handling at the negative edge of the range.

For zero-proximity and precision resolution, the input pairs (4.0 × 10^{-38}, -2.82 × 10^{-38}) and (4.0 × 10^{-38}, -2.83 × 10^{-38}) produced outputs of 1.18 × 10^{-38} and 1.16451 × 10^{-38}, respectively. While the second result shows a slight deviation, it reflects the limits of precision near zero and confirms that the architecture achieves a minimum resolvable magnitude around ±1.18 × 10^{-38}, consistent with the zero-proximity range of the IEEE 754 format. This also aligns with the (machine epsilon) resolution between -1 and 1, validated using test samples 9 and 10 in Table 3-9.

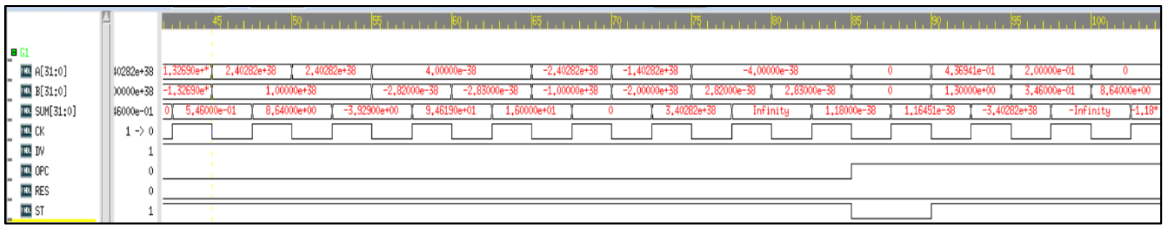

Figure 3.32: Simulation wave of Range verification

As with the data processing verification, results from both the Verdi simulator and the MATLAB model were identical, confirming that the designed adder/subtractor performs correctly across the full normalization range of the IEEE 754 single-precision floating-point format.

Control signal Activity verification:

In the final stage of behavioural verification, signal-level testing was conducted to ensure the pipeline architecture functions correctly across the full normalized range of IEEE 754 single-precision floating-point numbers. While the previous verifications focused on processing accuracy, this phase concentrated on validating critical control signals—*START*, *RES*, *DONE*, and *OPC*—which govern the initiation, reset, completion, and operation type of the system.

Figure 3.33 demonstrates that the *START* signal behaves correctly. When asserted high, it triggers the beginning of a computation. Due to the pipelined architecture, the system can accept a new pair of inputs in every clock cycle, with the corresponding output produced after a fixed latency (6 clock cycle), confirming proper pipeline functionality.

The *DV (DONE)* signal indicates the completion of computation. It goes high exactly six clock cycles after the *START* signal is asserted, which matches the expected pipeline latency and confirms accurate timing behaviour.

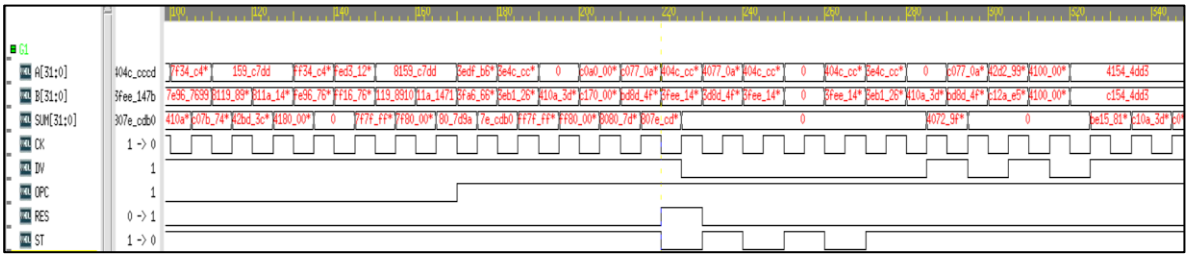

Figure 3.33: Control signal activity verification

The *RES* signal, representing synchronous active-high reset, also functions as intended. It is asserted during the fourth clock cycle in the test scenario, and the figure shows that all the register becomes zero stage at next rising clock edge. This reset approach ensures that all pipeline registers are reset to zero at the same time. As a result, throughout the next 6 clock cycle the *DV* remain low.

Lastly, the *OPC* signal, which selects the operation type (addition or subtraction), was verified to function correctly. When *OPC* = '0', the architecture performs addition. On the contrary, when *OPC* = '1', the system performs subtraction. The correct operation based on *OPC* confirms the proper handling of control logic.

Structure validation:

Structural verification confirms that the VHDL model is accurately synthesized into a valid Register Transfer Level (RTL) representation. Basically, this visual support of Verdi used to debug the architecture during design, and it confirms that all interconnections (nets) between components are correctly established. An incorrect or missing connection can lead to faulty functionality—even if the design appears error-free during initial analysis. This is because synthesis tools primarily check for syntax errors, signal length mismatches, and hierarchical block issues, but may not detect logical connection errors at a structural level.

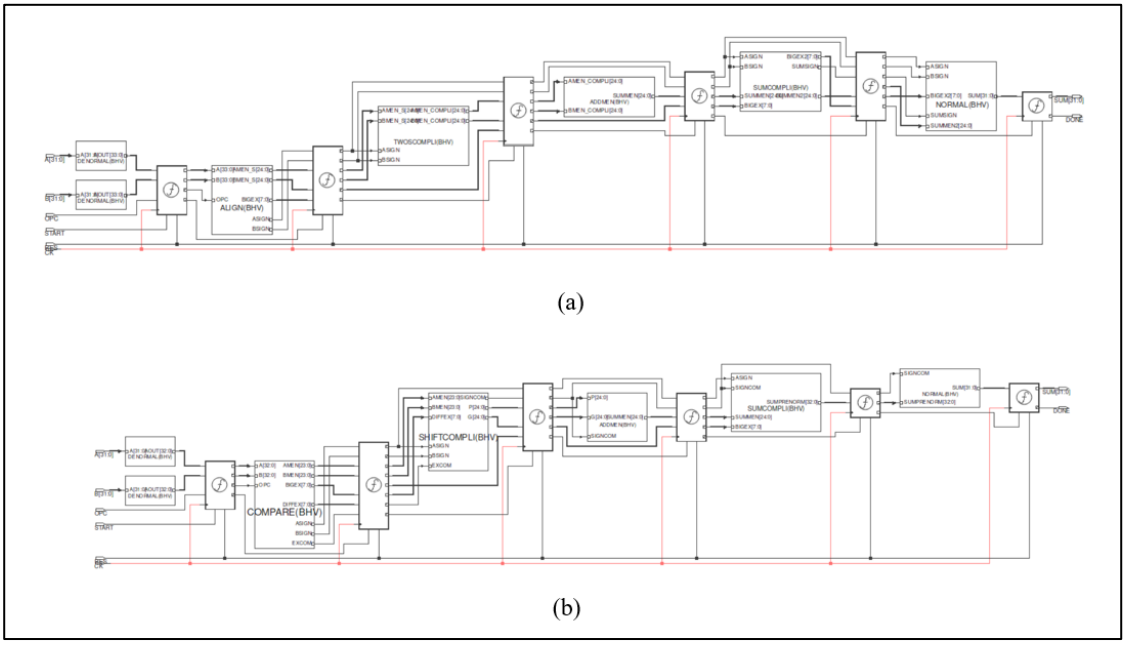

Figure 3.34: Verdi generated Pipelined FP-adder:(a) model-1, (b) model-2

Figure 3.34(a) illustrates the RTL view of Model 1, where all pipeline stages and registers are shown to be correctly placed and connected. Similarly, Figure 3.34(b) presents the RTL of Model 2, confirming that it also follows proper structural design and adheres to pipeline architecture requirements.

Note: All verifications were performed both before and after logic synthesis. Post-synthesis verification was conducted using a netlist generated by the Design Vision tool. The synthesized netlist, in VHDL format, was modified only to update the top-level component name. Using the same testbench, functional verification was repeated, and the results matched those of the pre-synthesis stage. This confirms that the synthesized design preserves the functional correctness of the original RTL model.

3.3 Serial Architecture of Floating-point Adder

The serial architecture operates iteratively, processing one instruction at a time. The primary objective of designing this architecture is to achieve area efficiency and low power consumption, making it suitable for resource-constrained applications.

3.3.1 RTL Design

As it's already realized that **carry look ahead adder (CLA)** adder is beneficial in term of speed and area, so the serial architecture is designed using the *CLA* adder and tried to iteration reduction so that overall input to output delay reduce. The entire process of floating-point addition and subtraction involves several key operations, including a 23-bit barrel **right shift**,

a 23-bit barrel **left shift**, two 9-bit binary **subtractions**, one 8-bit binary **addition**, one 25-bit binary **addition**, two 25-bit **two's complement** operations, one 8-bit **comparison**, and two **denormalization units**. In the

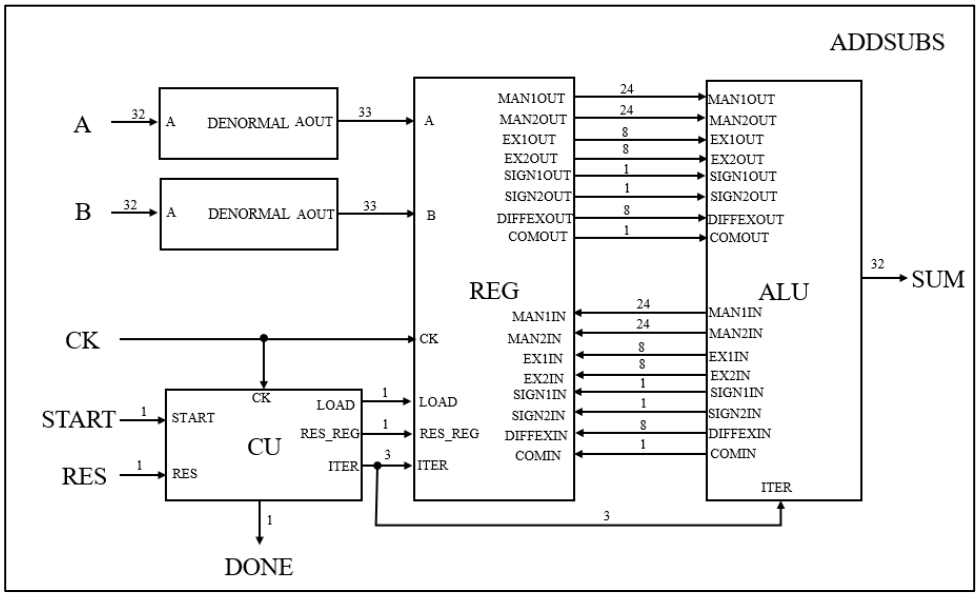

Figure 3.35: Serial architecture of floating-point adder

serial implementation, the architecture is meticulously designed to minimize hardware cell usage while maintaining high computational speed. This optimization is achieved by reusing core components, such as a single 23-bit barrel right shifter, one 23-bit barrel left shifter, one 9-bit carry look-ahead adder (*CLA*), and one 25-bit *CLA*, throughout the design. At the top level, the architecture, named **ADDSUBS**, is organized into four principal blocks: the Arithmetic Logic Unit (**ALU**), Register (**REG**) unit, Control Unit (**CU**) and Denormalization Unit (Denormal). Top hierarchy of serial architecture of floating-point adder/subtractor depicted in Figure-3.34.

Register unit (REG):

The register unit is composed of eight registers, each with varying bit lengths, designed to support different stages of the floating-point operation. This unit receives input from the denormalization block when the control signal `LOAD` is high, allowing initial values to be loaded into the registers. Conversely, when the `LOAD` signal is low, the registers instead capture input from the Arithmetic Logic Unit (ALU), enabling iterative processing during computation. Each register has clearly defined input and output ports, labelled as in and out, respectively, to manage data flow. Throughout different iterations of the computation, these registers hold various intermediate values crucial for executing the floating-point algorithm. The function and

data flow for each register are detailed in Table 3-11, which outlines their roles and how they interact within the system.

Table 3-11: Register activity and corresponding stored data

Register	Length (bits)	Output port	STORED VALUES		
			RES = 1 LOAD = 0	**RES=0 LOAD =1**	**RES = 0 LOAD = 0**
MAN1	24	MAN1OUT	0	A (23 - 0)	MAN1IN
MAN2	24	MAN2OUT	0	B (23 - 0)	MAN2IN
EX1	8	EX1OUT	0	A (31 - 24)	EX1IN
EX2	8	EX2OUT	0	B (31 - 24)	EX2IN
DIFFEX	8	DIFFEXOUT	0	0	DIFFEXIN
SIGN1	1	SIGN1OUT	0	A (32)	SIGN1IN
SIGN2	1	SIGN2OUT	0	B (32)	SIGN2IN
COM	1	COMOUT	0	OPC	COMIN

Control Unit (CU):

The control unit is implemented as a **finite state machine (FSM)**, with a state register holding the current operational state. The process is initiated when the *START* signal is asserted high. Once the process begins, the system ignores any new input until completion. The FSM transitions through a total of five iterative states, labelled *IT1* to *IT5*, and includes a standby state named

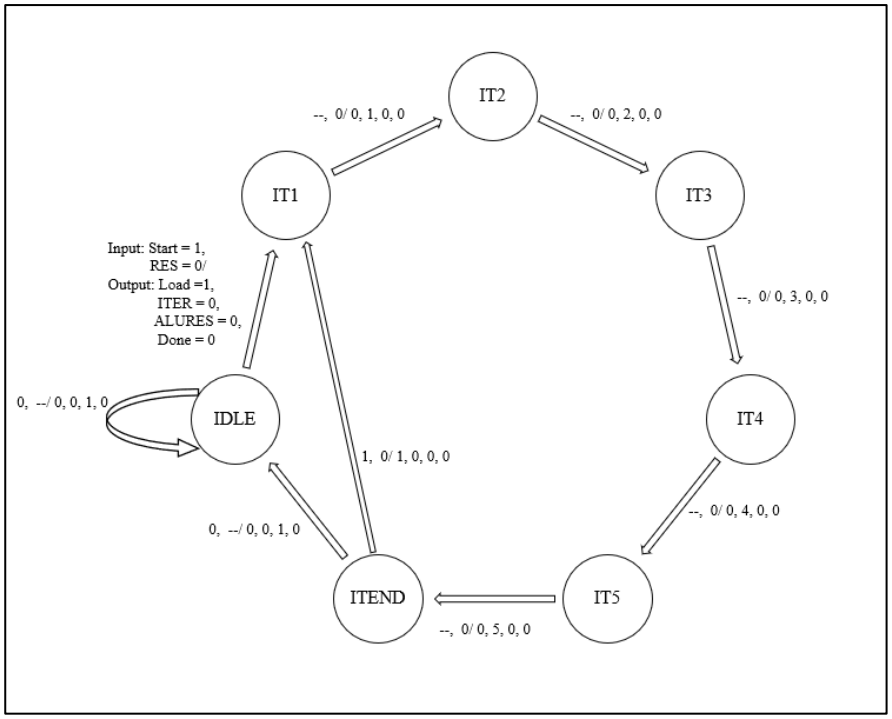

Figure 3.36: State diagram of Control unit

IDLE and end state *ITEND*. The state diagram shown in Figure 3.36 where the state transition and corresponding output shown clearly. The architecture employs a synchronous and active-high reset mechanism: when the *RES* signal is set high, the state register resets to the *IDLE* state. *REG_RES* signal becomes high once at the *IDLE* state, triggering the reset of all registers within the *REG* block. The State transition due to active reset during different state has not shown in the state diagram. During operation, the control unit generates key control signals such as *ITER*, *REG_RES*, and *DONE*. The *ITER* signal reflects the current iteration number, guiding the ALU to execute the appropriate operation for that stage. The *DONE* signal is asserted high once the computation process is complete i.e. *ITEND* state, indicating that the output data is valid and the system is ready to accept new inputs.

ALU: The ALU includes a 9-bit Carry Lookahead Adder (CLA) called ADDER9, a 25-bit CLA adder, a shifting unit, multiplexers (*MUX*), and other combinational logic components that handle operations such as changing the sign of input *B*, calculating large exponents, determining output signs, and detecting overflow.

The Figure 3.37 shows a modular **ADDER9** structure that can perform both addition and subtraction based on control signals. The upper section of the diagram displays two modes of the same 9-bit *CLA* block. On the left (highlighted in orange), it executes regular addition were

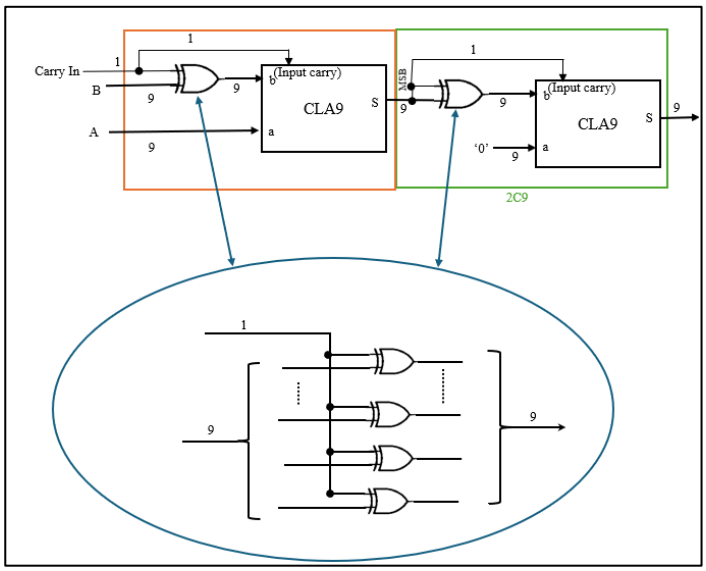

Figure 3.37: 9-bit CLA adder/subtractor with twos compliment unit

two 9-bit inputs, *A* and *B*, are fed directly into the *CLA* along with a carry-in signal. On the right (highlighted in green), input *B* is inverted through *XOR* gates controlled by a single control bit, allowing the block to switch between addition and subtraction using two's complement logic. The *XOR* gate outputs a '1' to invert *B* or a '0' to leave it unchanged, enabling the circuit to

perform *A + B* or *A – B* depending on the control input. The lower portion (inside the blue oval) zooms in on the XOR gate design, illustrating how each bit of *B* is conditionally inverted based on the control signal.

When the carry-in is '0':

XOR output = XOR input = B

S = A + XOR output + 0

= A + B

When the carry-in is '1':

XOR output = not (XOR input) = not *B*

S = *A* + XOR output + 1

= *A* + (NOT *B*) + 1

= *A* + (NOT *B* + 1)

= *A* + twos complement of *B*

= *A* – *B*

This ADDER9 is utilized during iterations 1, 4, and 5. In iteration 1, it subtracts *EX2* from *EX1*. In iteration 4, it acts as an adder to add 1 to the large exponent when an overflow occurs. During iteration 5, it subtracts the *first_1* from the large exponent.

The 25-bit adder/subtractor **(ADDER25)** functions similarly to the orange block shown in Figure 3.38. During iteration 3, it takes inputs *IN25_1* and *IN25_2*, which are assigned based on the value of COM, indicating whether the input signs are the same or different.

- If *COM* = '0' (meaning both inputs have the same sign):
  IN25_1 = MAN1OUT (mantissa of input *A*)
  IN25_2 = MAN2OUT (mantissa of input *B*)
- If *COM* = 1 (meaning the input signs differ):
  IN25_1 is assigned the mantissa with the positive sign
  IN25_2 is assigned the mantissa with the negative sign

The carry-in pin of ADDER25 is connected to *COM*. This means the carry-in is '0' when both

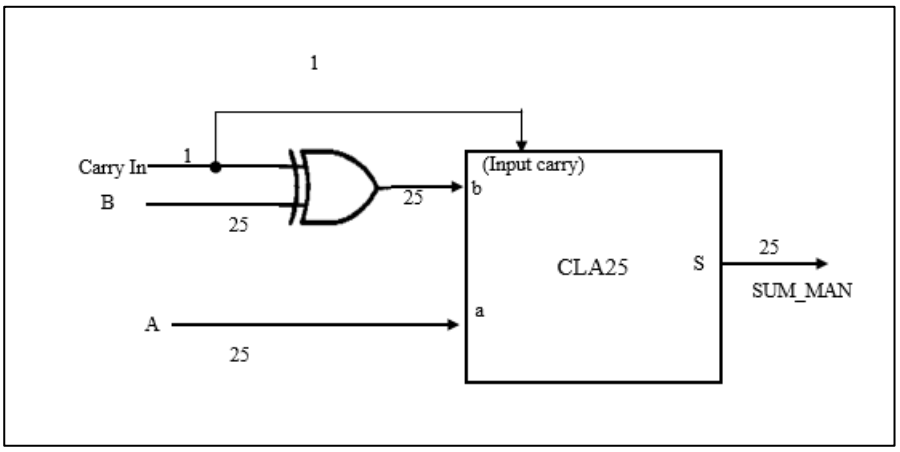

Figure 3.38: 25-bit CLA adder/subtractor

numbers share the same sign, and 1 when their signs differ, allowing the unit to perform addition or subtraction using two's complement arithmetic accordingly. The least significant 24 bit of 25-bit output, called *SUM25*, is sent to *MAN1IN*, so the sum is stored in the *MAN1* register at the next rising clock edge and MSB stored in *SIGN2* register. In iteration 4, the adder computes the two's complement of this stored sum.

More specifically:

- When *COM* = 0,

  IN25_1 = 0

  IN25_2 = *MAN1OUT* (the previously stored sum)

  INPUT CARRY25 = *COM* = 0

  OUTPUT25 = *MAN1OUT*

- When *COM* = '1',

  IN25_1 = '0'

  IN25_2 = *MAN1OUT*

  INPUT CARRY25 = *COM AND* (MSB of *MAN1OUT*), which is '0' if the sum is positive, or '1' if negative

  OUTPUT25 = *MAN1OUT* if MSB is 0

  OUTPUT25 = two's complement of *MAN1OUT* if MSB is 1

Thus, *ADDER25* is used to add two mantissas during iteration 3 and to perform the two's complement operation during iteration 4.

The shifter block consists of two components: a barrel right shifter and a barrel left shifter. It receives three inputs — *SELECTED_MANTISSA, SHIFTBITS*, and *SHIFT_DIRECTION*. The *SELECTED_MANTISSA* is the mantissa that needs to be shifted. *SHIFTBITS* specifies

the number of bits to shift, while *SHIFT_DIRECTION* is a single-bit input that determines the direction of the shift: 0 for right shift and 1 for left shift. This unit is used in iteration 2 to right-shift the mantissa corresponding to the smaller exponent, and in iteration 5 to left-shift the sum mantissa.

Figure 3.39 depicts the architecture of the Arithmetic Logic Unit (ALU), with different signal paths highlighted using colours for better support. The overall operation of the ALU is controlled by the *ITER* signal, which manages the processing stages throughout the various clock cycles.

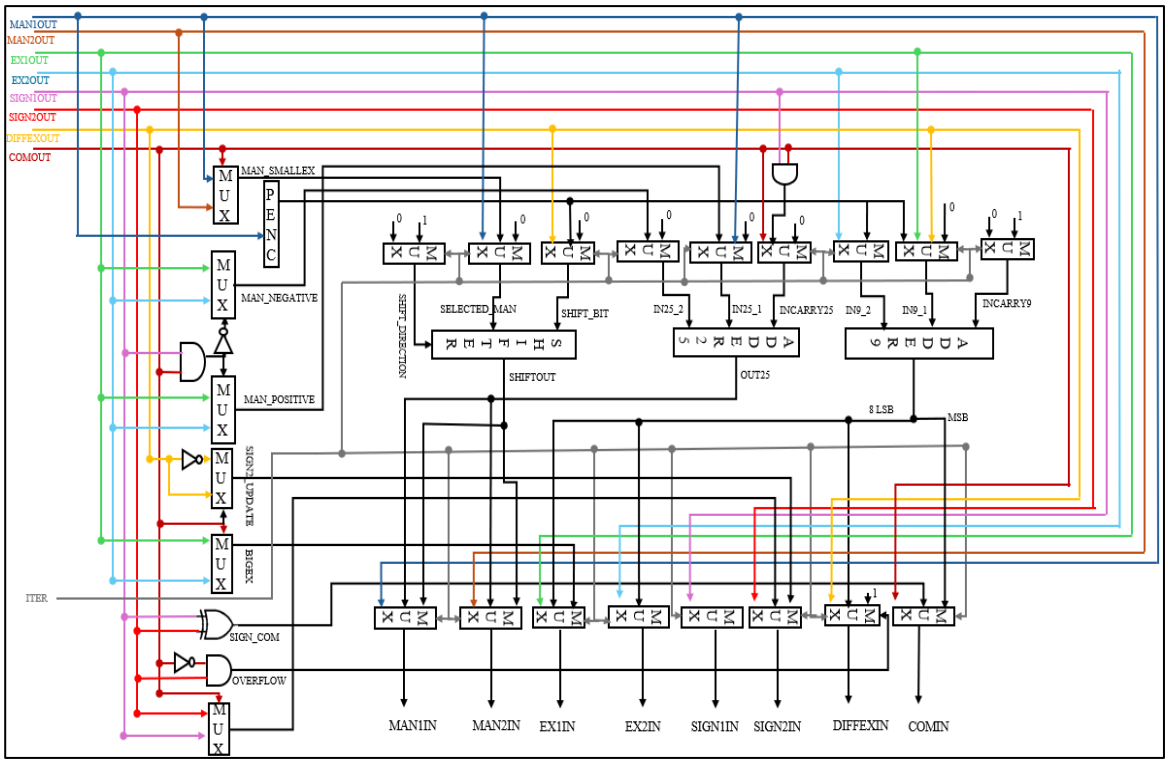

Figure 3.39: ALU of floating-point serial adder

ALU iterative operation:

***ITER* = 1: Exponent Comparison and Sign Detection**

At this stage, the *COM* register captures the operation_control (*OPC*). The *SIGN2* register is updated with *SIGN2UPDATE* only if the *COM* signal is high; otherwise, it remains unchanged. The exponents *EX1* and *EX2* are routed to the *ADDER9*s *IN9_1* and *IN9* respectively, with the carry input (*INCARRY9*) set to '1'. The result $SUM9 = EX1 - EX2$ is computed, where the most significant bit (MSB) of the result indicates which operand has the larger exponent. If the MSB is '1', $EX2 > EX1$; otherwise, $EX1 > EX2$. The 8 least significant bits of *SUM9* are stored in the *DIFFEX* register, while the MSB is stored in the *COM* register.

ITER = 2: Mantissa Alignment and Sign Comparison

Here, the *COM* register determines which operand has the smaller exponent. Based on *COMOUT*, *MAN_SMALLEX* (mantissa with small exponent) is selected: if *COMOUT* = '1', it equals *MAN1OUT*; otherwise, it equals *MAN2OUT*. This mantissa is then passed to the *SELECTED_MAN* input of a barrel shifter, with the shift amount (*SHIFT_BIT*) set by *DIFFEXOUT*. A right shift is performed (*SHIFT_DIRECTION* = '0'), and the shifted result (*SHIFTOUT*) is stored in either *MAN1* or *MAN2*, depending on the value of *COMOUT*. The larger exponent (*BIGEX*) is stored in *EX1*. Simultaneously, the signs *SIGN1* and *SIGN2* are compared using an *XOR* gate; the result ('0' for equal signs, '1' for opposite signs) is saved in the *COM* register.

ITER = 3: Mantissa Addition or Subtraction

Now, the *COM* register indicates whether the input signs are equal or opposite. For equal signs (*COMOUT* = '0'), the mantissas *MAN1OUT* and *MAN2OUT* are directly added using the 25-bit adder (*ADDER25*). If the signs differ (*COMOUT* = '1'), the positive and negative mantissas are routed through *MAN_POSITIVE* and *MAN_NEGATIVE*, with their MSBs cleared to indicate unsigned values. The carry input (*INCARRY25*) is driven by *COMOUT*. The 24 least significant bits (**LSB**) of the sum (*SUM25*) are stored in *MAN1*, while the MSB, representing the result's sign, is stored in *SIGN2*.

ITER = 4: Sign Adjustment and Overflow Handling

In this step, the *COM* register continues to indicate sign comparison results. The final sum is available in the *MAN1* register, and the result's sign is in *SIGN2*. If *COMOUT* and *SIGN2OUT* are both 1, indicating the need for a two's complement, the circuit sets *CARRYIN25* high and subtracts *MAN1OUT* from 0 using ADDER25. If *COM* = '0', overflow detection is performed. If overflow occurs, the result in *MAN1OUT* is right shifted by one (by appending the overflow bit) and the exponent is incremented accordingly by adding the overflow to *EX1OUT* using ADDER9. The updated exponent is stored in *EX1*. Finally, the sign of the output is determined: if *COM* = '0', it uses *SIGN1OUT*; otherwise, it uses *SIGN2OUT*. This value is stored in *SIGN2*.

ITER = 5: Normalization and Final Output

The mantissa in *MAN1OUT* is passed to a priority encoder to detect the index of the leading '1' (*first_1*). This value sets the *SHIFT_BIT* for a left shift (*SHIFT_DIRECTION* = '1')

performed by the shifter. The normalized output (*SHIFTOUT*) is stored in *MAN2*. Meanwhile, the exponent is adjusted by subtracting *first_1* from *EX1* using ADDER9 with *CARRYIN9* = '1'. The updated exponent (*SUM9*) is saved in *EX2*.

The final floating-point result is obtained by concatenating the sign bit (*SIGN2OUT*), the adjusted exponent (*EX2OUT*), and the normalized mantissa (*MAN2OUT (22-0)*). This result becomes available after five clock cycles, triggered by setting the *START* signal of the control unit high.

Note: All connections between blocks (such as those to *MAN1IN*, *EX1IN*, etc.) are handled through multiplexers controlled by the *ITER* value. All data routed through input ports are latched into their respective registers at the rising edge of the clock.

3.3.2 Gate level synthesizes of Serial architecture (FP-ADDER)

Gate-level synthesis and timing analysis of the serial architecture were performed using the same constraints applied to the pipelined architecture. The synthesizer inferred the adder and other lower-level hierarchical blocks consistently, as described in the previous section. To avoid redundancy, the detailed analysis of device inference is omitted here.

Report timing:

Table 3-12 highlights that the serial architecture achieves peak operating frequencies of **200 MHz for UMC13** technology and **1.5 GHz for GF22**, with the critical path determined by the 25-bit mantissa addition stage. This performance ceiling stems from the integration of both twos-complement and addition within the same cycle using a combined CLA25 structure. Although this merging introduces a speed bottleneck due to the increased logic complexity, it simultaneously reduces the overall number of computation cycles, resulting in a net reduction

Table 3-12: Report Timing (Serial FP-adder)

Technology	Clock period (Frequency)	Throughput (Mega sample)	Setup time analysis (Worst case) Slack (% of clock period)	Hold time analysis (Best case) Slack (% of clock period)
UMC13	5 ns (200 MHz)	40 Mega sample/s	+ 0.73 ns (15%)	+0.11 ns (2.2%)
GF22	0.9 ns (1.1 GHz)	220 Mega sample/s	+ 0.273 ns (30%)	0.018 ns (2%)

in total delay. Consequently, the design gains improved efficiency despite localized latency. Additionally, the denormalization unit, positioned at the input, contributes to initial propagation delays of 1.5 ns for UMC13 and 0.19 ns for GF22. These delays slightly narrow the input timing margin but are acceptable within the high-frequency operation goals of the design.

Report area:

Table 3-13 demonstrates that the serial architecture occupies significantly less area compared to the pipelined design—nearly half in total. Under UMC13 technology, the serial architecture utilizes 1077 cells, while for GF22, the count slightly increases to 1188. The total cell area is reduced from 11255 um² (UMC13) to just 533.7 um² (GF22), clearly reflecting the advantage of advanced process nodes. Notably, the largest portion of area is consumed by the Arithmetic Logic Unit (ALU), with the shifter being the most area-intensive individual block—occupying

Table 3-13: Report area (Serial FP-adder)

	UMC13	GF22
No. of Cells	1077	1188
No. of combinational cell	989	1099
No. of Sequential cell	79	79
No. of Buffer/Inverter	172	268
No. of reference	5	5
BUF/INV area	669.4	35.7
Combinational Area	9237.8	413.3
Non-combinational area	2017.3	120.4
Total cell Area	11255	533.7

1,377.3 um² in UMC13 and 114.5 um² in GF22 as shown in Table 3-14. Despite the shifter's size, the reuse of registers across computation cycles in the serial design significantly reduces the overall combinational logic area. The area savings come at the cost of increased multiplexer usage within the ALU depict in Figure 3.40, slightly raising its complexity and size. Specifically, the ALU totals 7,907.8 um² (70%) in UMC13 and 352.9 um² (66%) in GF22. A significant difference is observed in the area occupied by the MUX and other combinational logic blocks, which account for 39% in UMC13 and 26% in GF22. This variation arises from the way the shifter modules are mapped in each technology. In UMC13, the left shifter is

Table 3-14: Area occupied by lower hierarchical units

Units		UMC13 (um^2)	GF22 (um^2)
ALU Hierarchy	**ADDER9**	819.2	35.9
	ADDER25	1287.6	65.9
	SHIFTER	1377.3	114.5
ALU TOTAL		7907.8	352.9
REG		2776.3	153.7
DENORMAL		358.4	17.3
CU		212.5	9.8

implemented using standard cell logical left shifters (LLS), while the right shifter is synthesized as generic combinational logic. As a result, the shifter occupies only 12% of the total area. In contrast, in GF22, both the left and right shifters are mapped to dedicated standard cells (LLS and RLS), leading to a higher shifter area usage of 21%. Additionally, the Register Bank (REG) occupies 25% of the area in UMC13 and 29% in GF22.

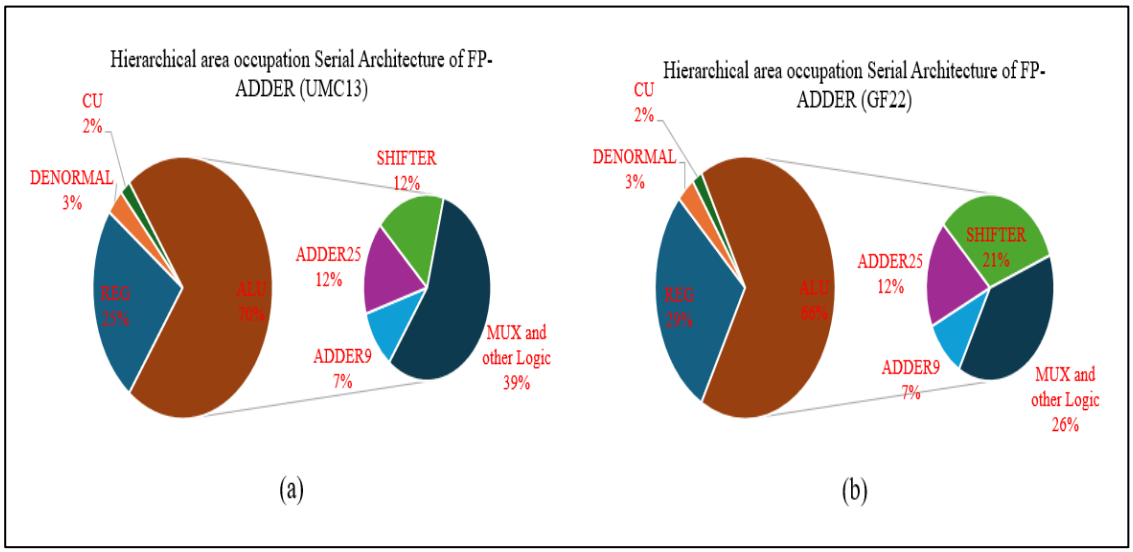

Figure 3.40: Hierarchical Area Occupation: (a) UMC13, (b) GF22

Report Power:

Table 3-15 shows that the serial architecture demonstrates significantly lower power consumption compared to the pipelined Model-2 in both technologies. Specifically, in UMC13, the serial architecture consumes approximately 380.7 uW of total power, while the pipelined Model-2 consumes 1082.8 uW—nearly three times more. Similarly, in GF22, the serial design consumes 473.2 uW, whereas Model-2 reaches 2024.5 uW, more than four times higher. This

significant increase in power for the pipelined version is largely attributed to higher internal power, driven by increased switching activity and clock frequency needed for higher throughput. Despite GF22 technology offering reduced net switching power due to its lower

Table 3-15: Report Power (serial FP-adder)

	UMC13	**GF22**
Cell internal power (uW)	293.2	399.1
Net switching power(uW)	139.7	72.1
Total Dynamic Power (uW)	378.9	471.2
Cell Leakage power (uW)	1.8	2
Total Power Consumption (uW)	380.7	473.2

capacitance and improved efficiency (e.g., 82.65 uW in GF22 vs. 143.1 uW in UMC13 for Model-2), its total dynamic power is still considerably higher due to greater performance demands. Leakage power in both architectures remains nearly the same between technologies, with GF22 showing only a slight increase.

Trade-off:

The triangular chart in Figure 3.41 illustrates the trade-offs between power, area, and frequency for both serial and pipelined floating-point adder architectures implemented in UMC13 and GF22 technologies. As anticipated, serial architectures consistently operate at lower frequencies than their pipelined counterparts. This difference is particularly noticeable in the GF22 process, where the pipelined architecture achieves up to 1.66 GSamples/s, compared to just **0.22 GSamples/s** for the **serial** design—making the pipelined version approximately **7.5** times faster. In terms of throughput, the serial architecture achieves approximatly 1/8 of the throughput of the pipelined counterpart. However, this performance sacrifice results in significantly **reduced power consumption—only 1/4** of that of the pipelined architecture—as shown by the larger footprint along the power axis in pipelined designs. When it comes to area, serial architectures are more efficient, primarily due to a reduced number of registers and arithmetic logics. This trade-off follows the relation: Area Reduction × Power Reduction ≈ Throughput Reduction.

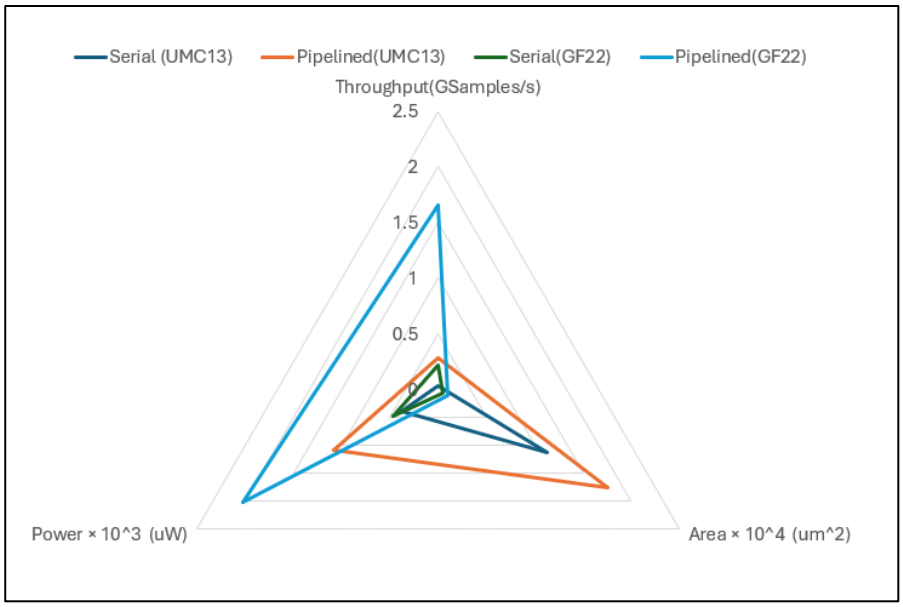

Figure 3.41: Speed, Area and power triangle to show trade-off (serial-pipelined)

In summary, pipelined designs are optimized for speed and throughput, ideal for high-performance systems, whereas serial architectures trade performance for lower power and compact area, aligning well with energy-efficient design goal.

3.3.3 Simulation of serial architecture of FP-adder/subtractor:

Like the pipelined simulation a test-bench has been created and same function used to convert the input and output data. The behavioural verification has divided into four parts.

Verification of Data process:

Data sample was given as input to verify the serial architecture that was given for the pipelined architecture and gate the same output So, the Table 3-8 also represent the serial architecture validation table. The simulated output got from Vardi and behavioural model in MATLAB is identical which validate operation of serial architecture. But the difference for serial architecture is during one process new input don't allow by the system. The system becomes ready to take the new input after completing running process.

Range verification:

The range, precision and resolution also verified giving the same input as the pipeline architecture and get the same result. So, Table 3-9 also represent the range validation table for the serial architecture. Figure 3.42 shows some of the sample and corresponding output.

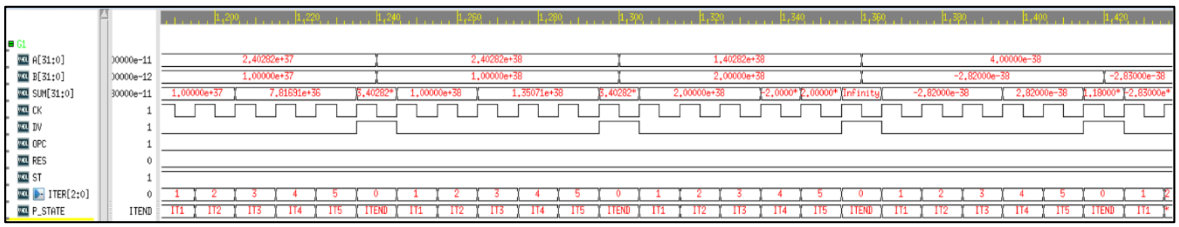

Figure 3.42: Simulation wave of Range verification

Control signal Activity verification: The signals called *START*, *RES*, *DONE* and *OPC* are the crucial part of the system which initiate and validate the data and operation. Those signal work differently than the pipelined architecture. Figure 3.43 illustrate the behaviour of those signal where, the *START* signal initiates the process of controller and the controller control the whole process controlling the signal *LOAD* and *ITER*. Load signal becomes high only at *IDLE* state and all the other state it's become low. During one running process the *START* signal is avoided i.e. it can't start new process. Besides in each iterative state the controller gives the value of iteration index as signal *ITER*. After 5 iteration the *DONE* pin (*DV*) become high which indicate the valid output data. At the *ITEND* state the process become aging *START* signal sensitive. The figure also shows that at *IDLE* state register resat in the absence of high *START* signal. The Reset work as active high and when reset input *RES* is high the system reset to *IDLE* state and reset all register of REG block.

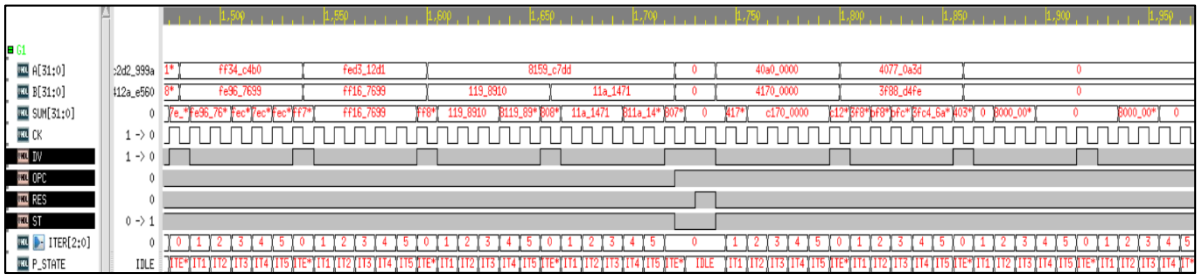

Figure 3.43: Control signal activity verification

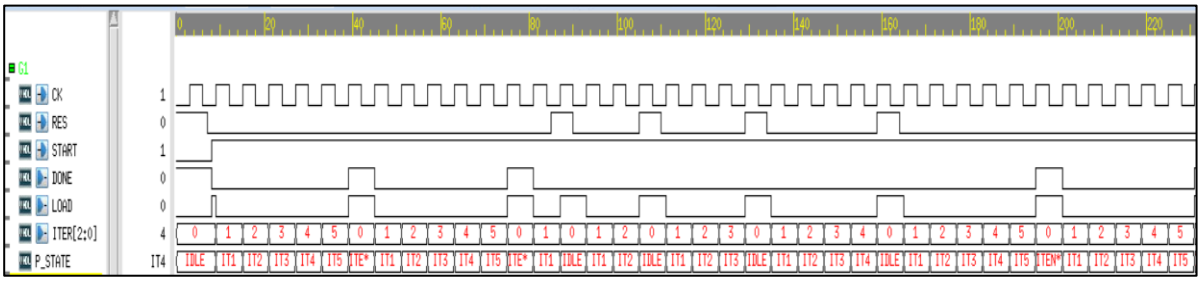

Figure 3.44: Controller verification and validation of state transition due to reset active

Besides another test-bench created to validate the control unit, its state transition and transition due to reset. For that reset pulse activate after reaching each state and found the proper transition

and expected control output signal as shown in Figure 3.44 and *RES* activation successfully change state to *IDLE* from all state.

Structure validation:

The structural verification shows that pin of different hierarchy connected correctly. The Figure 3.45 shows that all net connect as mentioned in the design. This validates the tedious connection of lower hierarchy at top.

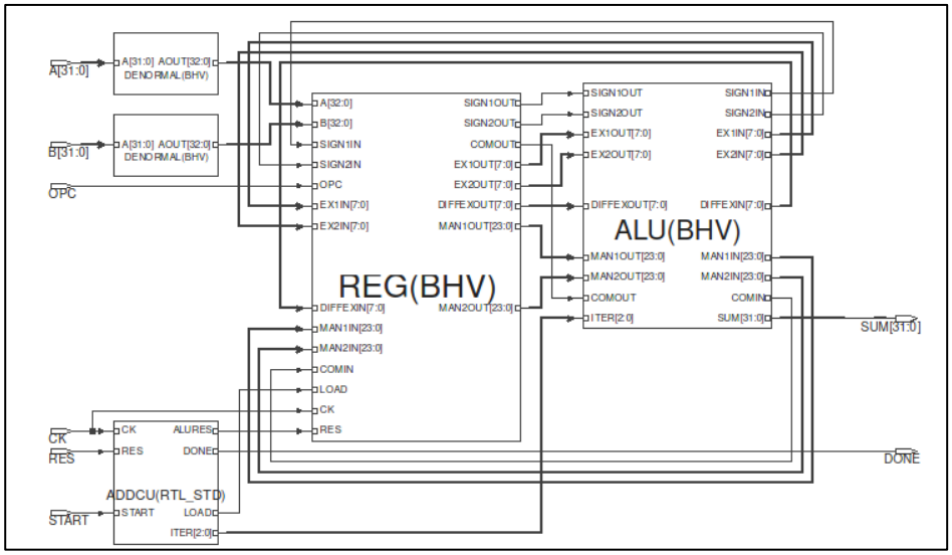

Figure 3.45: Controller verification and validation of state transition due to reset active

3.4 Scaling of IEEE-754 Floating-Point Values by Powers of Two

From Equation 2.1 and Equation 2.2, it is evident that implementing these equations requires a multiplier to multiply a number by 2^{-i}. In a fixed-point system, the scaling by 2^{-i} can be efficiently realized using a right shifter[1], [19], However, rather than performing a full floating-point multiplication, this operation can be efficiently realized by directly modifying the exponent field of the floating-point representation. According to the IEEE-754 format, a 32-bit floating-point number is represented as $(-1)^S \times 2^E \times 1.M$, where S is the sign bit, M is the mantissa, and E is the biased exponent. Multiplying by 2^{-i} is mathematically equivalent to subtracting i from the exponent, leaving the sign and mantissa unchanged (as shown in Figure 3.46). This results in a significant hardware optimization, as it eliminates the need for complex multiplication units and allows the operation to be performed using simple arithmetic on the exponent field. **In CORDIC this 8-bit subtractor implemented using 8-bit carry look ahead adder (CLA8)**.

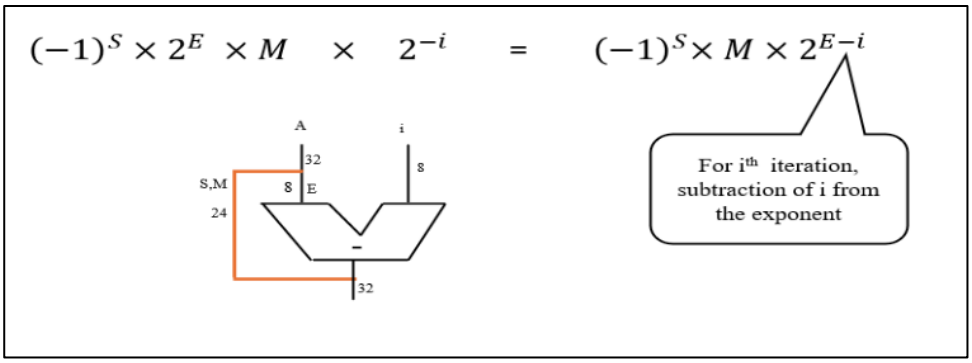

Figure 3.46: multiplier/shifter realization in floating point number

Chapter 4

CORDIC Co-processor Design

This project presents the ASIC implementation of a floating-point CORDIC processor designed to compute sine and cosine values in rotation mode. This chapter details the architecture of the FP-CORDIC processor along with its simulation results, gate-level logic synthesis, and pre-layout analysis in terms of timing, area, and power. The architecture is described using VHDL, and simulations are conducted using the Synopsys Verdi tool. A behavioral model is also developed in MATLAB to validate the design's functionality. Gate-level synthesis is carried out using Synopsys design software called, Design Vision.

The sections in this chapter are organized as follows:

- **Section 4.1** Pipelined architecture of FP-CORDIC
- **Section 4.2** simulation of Pipelined architecture
- **Section 4.3.** Logic synthesis and analysis of Pipelined architecture
- **Section 4.4** Serial architecture of FP-CORDIC
- **Section 4.2** simulation of Serial architecture
- **Section 4.3.** Logic synthesis and analysis of Serial architecture

4.1 Pipelined Architecture of FP-CORDIC

In this design, a dedicated package is created to initialize variables, constants, building blocks, and precomputed micro-angles. All floating-point adders and subtractors used follow a pipelined architecture (Model-2). As a result, the pipelined CORDIC is built using pipelined floating-point adders, enabling a deeply pipelined architecture—extending pipelining from the top level down to lower hierarchical blocks. Based on the placement of normalization and denormalization units, two architectural variants are proposed:

1. **ARC-1:** Local normalization and denormalization
2. **ARC-2:** Global normalization and denormalization

4.1.1 FP-CORDIC pipelined architecture with Local Denormalization and Normalization (ARC-1)

In the ARC-1 architecture, each pipeline stage of the FP-CORDIC processor independently handles denormalization and normalization operations. This means that every stage includes its

own dedicated denormalization and normalization units, which are embedded within the floating-point adder/subtractor modules. As a result, the input to each stage is first denormalized, then processed through arithmetic computation, and finally normalized before passing to the next stage. This localized handling simplifies the timing and data format consistency across the pipeline but introduces redundant hardware due to repeated normalization and denormalization units in each stage.

CORDIC iteration Unit:

The unrolled architecture implements Equations 1.11, 1.12, and 1.13 in parallel, forming the top-level CORDIC data path. For clarity, this data path is referred to as the **X-path**, **Y-path**, and **Z-path**. The floating-point adder/subtractors used in each path are denoted as **addsubX**, **addsubY**, and **addsubZ**, corresponding to the **X, Y**, and **Z** processing paths respectively. Each floating-point adder/subtractor includes integrated denormalization and normalization units. As a result, during every pipeline stage, the input is first denormalized and then normalized at the end of stage. Each iteration stage consists of three floating-point adders and two 8-bit subtractors (implemented as **CLA8**), which are functionally equivalent to fixed-point shifters. This arrangement forms what is referred to as the **CORDIC iteration unit**, illustrated in Figure 4.1. The 8-bit subtractors adjust the exponent of the floating-point values by applying a scaling

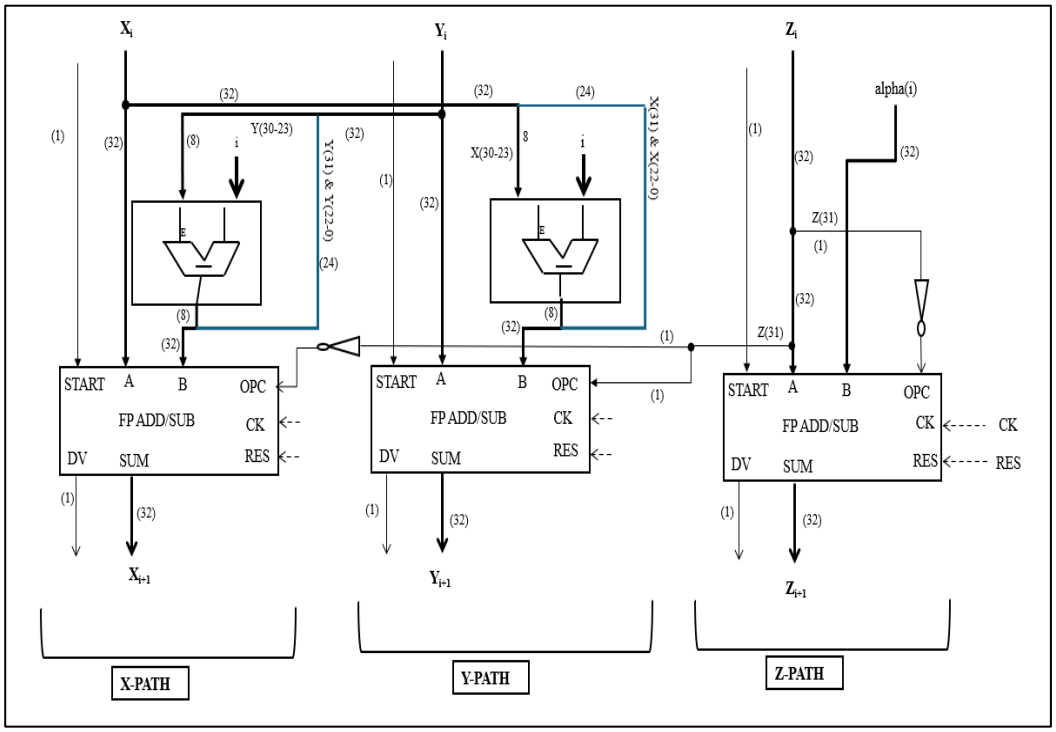

Figure 4.1: Iteration unit of FP-CORDIC

factor of 2^{-i}, effectively emulating a right shift. The internal architecture of the floating-point adder/subtractor was detailed in the last chapter. For generalized discussion, it is assumed that the focus is on the **ith iteration stage**, where three parallel processes operate on the inputs `Xi`, `Yi`, and `Zi`, respectively.

In the case of the **Z-path**, the input `Zi` is directly connected to the *port-A* of **addsubZ**. Notably, the *port-B* of addsubZ is connected to a fixed value, specifically the i^{th} element of the precomputed angle table, denoted as `alpha(i)`. The *operation control signal* (`OPC`) of addsubZ is driven by the most significant bit (**MSB**) of `Zi` (`i.e. Z (31)`) through a **NOT** gate. This setup ensures that when `Zi` is positive (MSB = '0'), *OPC* becomes '1', and the precomputed angle `alpha(i)` is subtracted from `Zi`, indicating a counterclockwise rotation. Conversely, when `Zi` is negative (MSB = '1'), `OPC` becomes '0', resulting in the addition of `alpha(i)` to `Zi`, corresponding to a clockwise rotation. This mechanism directly implements **Equation 1.13** of the CORDIC algorithm.

For the **Y-path**, the input `Yi` is directly connected to the *port-A* of **addsubY**, while the *port-B* receives a scaled version of `Xi`. This scaling is achieved using an **8-bit subtractor**, which subtracts the current iteration number ***i*** from the exponent of `Xi`, effectively performing a multiplication by 2^{-i}. The result of the 8-bit subtractor is then reassembled with the remaining bits of `Xi` to form the properly scaled value—specifically, by combining `Xi(31)`, the 8-bit output of the subtractor, and `Xi(22:0)`. The *operation control signal* `(OPC)` for addsubY is directly connected to the most significant bit (**MSB**) of `Zi`. When `Zi` is positive (MSB = '0'), `OPC` is set to '0', and **addsubY** performs an addition of `Yi` and the scaled `Xi`. Conversely, when `Zi` is negative (MSB = '1'), `OPC` is set to '1', and addsubY performs a subtraction. This logic effectively implements **Equation 1.12** of the CORDIC algorithm.

In the **X-path**, the signal `Xi` is connected to *port-A* of **addsubX**, while *port-B* receives a scaled version of `Yi`, obtained using the same method applied in the **Y-path**. The *operation control signal* `(OPC)` is driven by the inverse of the most significant bit (**MSB**) of `Zi`. When `Zi` is positive (i.e., MSB = '0', so `OPC` = '1'), **addsubX** executes a subtraction, computing "`Xi – scaled Yi`". In contrast, when `Zi` is negative (MSB = '1', so `OPC` = '0'), it performs an addition of `Xi` and the scaled `Yi`. This operation corresponds to the functionality defined in **Equation 2.11** of the CORDIC algorithm.

The outputs of **addsubX**, **addsubY**, and **addsubZ** serve as the inputs X_{i+1}, Y_{i+1}, and Z_{i+1} for the subsequent iteration stage. Each stage begins its operation upon receiving a ***Start*** signal, and completion is indicated by the assertion of the ***Done*** flag. Additionally, it is important to note that the precomputed **micro-angles** are stored in **32-bit IEEE 754** floating-point format, ensuring compliance with standard floating-point representations.

Design Pre-iteration stage:

The input angle range supported by the CORDIC core is from –99.88° to +99.88°. This range can be extended to cover the full angular span of –180° to +180° by introducing a pre-rotation stage called **pre-iteration** stage. In this design, a pre-rotation unit is implemented to enable the CORDIC core to generate a complete sine and cosine waveform over a full period. The pre-rotation logic operates as follows:

If the input angle θ>90˚, then:

- $\theta' = \theta - 90°$
- $Xi' = -Yi$
- $Yi' = Xi$

If the input angle $\theta < -90°$ then,

- $\theta' = \theta + 90$
- $Xi' = Yi$
- $Yi' = -Xi$

For hardware implementation (as illustrated in Figure 4.2), the input angle θ is first compared with 90°, which is represented in IEEE 754 single-precision format as:

fp90 **= 0 10000101 01101000000000000000000**

The exponent of ***fp90*** is 10000101, which corresponds to a biased exponent value of 133 in IEEE 754 format. This bias indicates that the actual exponent falls within the range 2^6 to 2^7-1 (i.e., from 64 to 127 in decimal). Therefore, to accurately determine whether the input angle exceeds |90°|, both the **exponent** and **mantissa** fields must be compared against the corresponding fields of $fp90$. It is also important to note that the input angle θ must be applied to the Zi (denoted as ***ExZI*** for exponent and ***MANZI*** for mantissa of Zi) input of the CORDIC processor to compute the corresponding sine and cosine values. Based on the sign and magnitude of θ, four distinct cases can arise:

Case 1: | θ | < 64°

- ⇨ *ExZI < fp90(30-23)*
- ⇨ *MANZI* = don't care

Case 2: 64° < | θ | < 90°

- ⇨ *ExZI = fp90(30-23)*
- ⇨ *MANZI < fp90(22-0)*

Case 3: 90° < | θ | < 128°

- ⇨ *ExZI = fp90(30-23)*
- ⇨ *MANZI > fp90(22-0)*

Case 4: | θ | > 128°

- ⇨ *ExZI > fp90(30-23)*
- ⇨ *MANZI* = don't care

To implement the detection logic for angles greater than 90°, three comparators are used in parallel. First, an 8-bit comparator compares the exponent of the input `Zi` with the exponent of `fp90` (i.e., `fp90(30:23)`). If the exponent of `Zi` is greater than `fp90(30:23)`, the comparator outputs a logic high ('1'); otherwise, it outputs a logic low ('0'). Second, another 8-bit equality comparator checks whether the exponent of `Zi` is equal to `fp90(30:23)`, outputting '1' if equal, and '0' otherwise. Third, a 23-bit comparator compares the mantissa of `Zi` with `fp90(22:0)`. If the mantissa of `Zi` is greater, a signal named `MANCOM` is set to '1'; otherwise, it is '0'. The outputs of the exponent equality comparator and the mantissa comparator are then fed into an **AND** gate. The result of this **AND** operation, along with the output of the exponent greater-than comparator, is passed through an **OR** gate. The final output of this **OR** gate is a signal called ***greater90,*** which indicates whether the magnitude of the input angle |θ| exceeds 90°. When `greater90` = '1', it implies */Zi/* > 90°; otherwise, */Zi/* < 90. By evaluating the **most significant bit (MSB) of *Zi*** along with the *greater90* signal, the exact angular range of `Zi` can be determined as follows:

- If MSB = '0' and *greater90* = '1', then *Zi*>+90°
- If MSB = '0' and *greater90* = '0', then 0<*Zi*<+90°
- If MSB = '1' and *greater90* = '1', then *Zi*<−90°
- If MSB = '1' and *greater90* = '0', then −90°<*Zi*<0

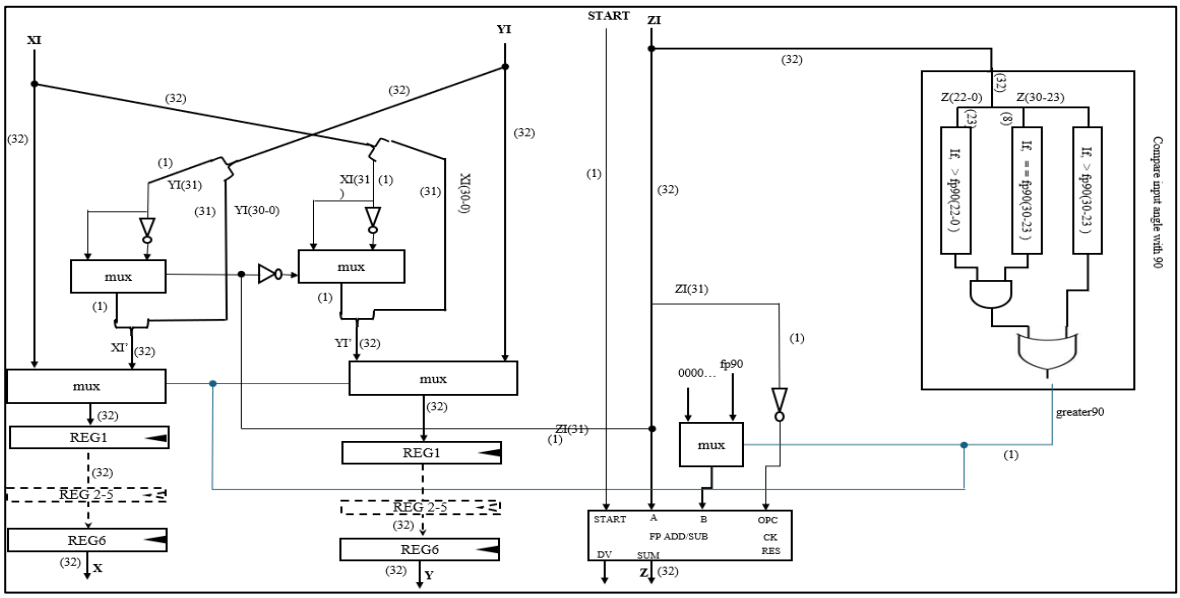

Figure 4.2: Pre-iteration (pre-rotation) stage of ARC-1

In the **Z-path**, a floating-point adder/subtractor is employed to perform angle compensation during the pre-rotation stage. The input angle `Zi` is fed to the *port-A* of the FP-adder/subtractor. The operation control signal (`OPC`) is connected to the inverse of the most significant bit (MSB) of `Zi`, ensuring that a subtraction of 90° is performed for positive input angles and an addition of 90° for negative ones. A multiplexer, controlled by the `greater90` signal, selects between `fp90` and zero. When `greater90` is high ('1'), the multiplexer outputs `fp90`, enabling a 90° adjustment; otherwise, it passes zero, meaning no compensation is applied. The selected output from the multiplexer is connected to the *port-B* of the adder/subtractor. The result of this operation is the adjusted angle, which is then used in the subsequent stages of the processor. The output of this FP-adder/subtractor is as,

Sum = *ZI,* if *greater90* = 0

Sum = *ZI* – *fp90,* if *greater90* = '1' and MSB of *ZI* = '0'

Sum = *ZI* + *fp90* if *greater90* = '1' and MSB of *ZI* = '1'

For the **X-path**, when `greater90` = '0', the value of *X* is simply `Xi`. Otherwise, the value of *X* depends on the sign of `Zi`: if the MSB of `Zi` is '0', then *X* = NOT (`Yi(31)`) & `Yi(30:0)`, effectively negating the sign bit of `Yi`; if the MSB of `Zi` is '1', then *X*=`Yi`. This logic is implemented using two multiplexers. The first multiplexer's select pin is connected to the MSB of `Zi`. When the MSB is '1', the output is −`Yi` (the sign bit of `Yi` inverted), otherwise it passes `Yi` unchanged. The remaining bits `Yi(30:0)` are concatenated with this updated sign bit, forming an intermediate value `Xi'`, which corresponds to

Xi'=−Yi, if the input angle is positive, or *Xi'=Yi* if the input angle is negative. The second multiplexer uses *greater90* as its select signal. If *greater90* = '1', it outputs *Xi'*; otherwise, it outputs *Xi*. This completes the logic as:

If the input angle > 90°, $X=-Yi$

If the input angle < 90°, $X=Yi$

Otherwise, $X=Xi$

The updated *X* value is then stored in a pipeline of **six 32-bit registers** arranged sequentially. This pipeline introduces a six-cycle delay to synchronize with the floating-point adder/subtractor in the **Z-path**, which has six pipelined stages. Each register corresponds to one clock cycle of delay, ensuring proper timing alignment across the processing stages.

The implementation of the **Y-path** is like that of the **X-path**, with one key difference in the condition for updating *Yi'*. Specifically, when the input angle is greater than +90°, *Yi'=Xi;* otherwise, when the input angle is less than -90°, *Yi'=−Xi*. To achieve this, the select pin of the first multiplexer in the Y-path is connected to the inverted MSB of *Zi*. This inversion ensures the correct sign adjustment is applied to *Xi* depending on the sign of the input angle.

CORDIC TOP:

The top level of the pipelined FP-CORDIC architecture (illustrated in Figure 4.3) consists of 23 CORDIC iteration units cascaded sequentially, enabling 23 iterative stages. The pre-iteration stage is positioned at the apex of this pipeline structure. For each X-path, Y-path, and Z-path—the *START* signal of each floating-point adder is connected to the DONE signal of the adder in the previous stage along the same path, ensuring proper synchronization between stages. The output of each adder, provided via the *SUM* port, is fed as the input to the subsequent stage. The pipeline's first stage is the pre-iteration stage, where the input *pins XI , YI* and *ZI* are connected directly to the processor's input ports *XI, YI,* and *ZI,* respectively.

The ***START*** pin of the FP-adder/subtractor in the pre-iteration stage is connected to the ***LOAD*** port of the processor, which triggers the start of the operation. The outputs of the processor are taken from the 23rd (final) stage of the CORDIC pipeline. Specifically, the output ports *XO, YO*, and *ZO* are directly connected to the *SUM* outputs of the final **stage's addsubX, addsubY,** and **addsubZ** units, respectively. Additionally, the processor's final data valid (***DV***) flag is connected to the *DONE* pin of the addsubZ unit in the last stage. (Note: the *DONE* pin of addsubX

or addsubY could also be used, as all three paths—X, Y, and Z—are synchronized.) This *DV* signal indicates the completion of the computation for a given input.

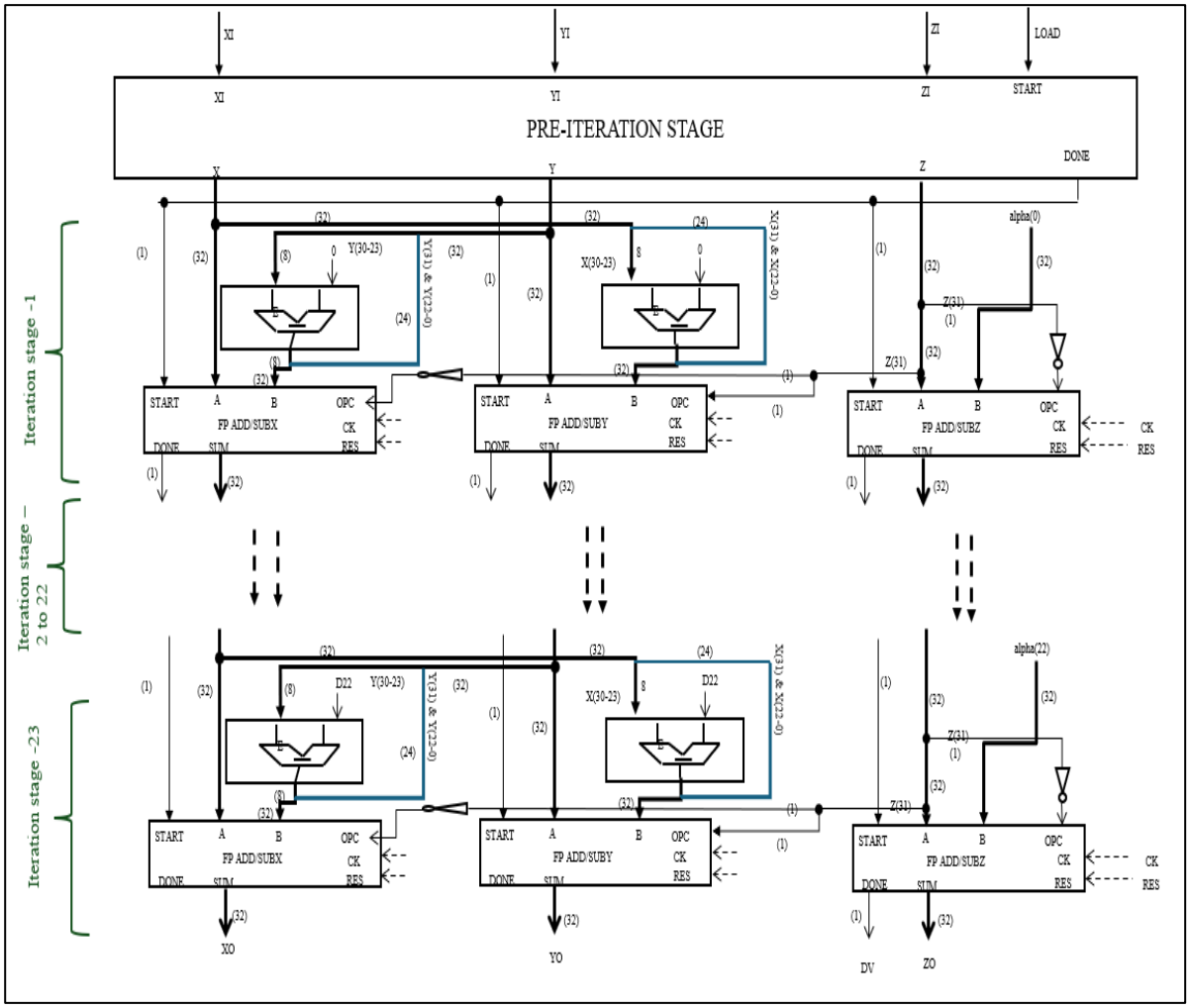

Figure 4.3: Designed Pipeline architecture of FP-CORDIC (Local normalization and denormalization)

It is important to note that, although it may appear there are no registers between the pipeline stages, this is not the case. Due to deep pipelining, each FP-adder inherently includes internal registers. Specifically, the final register of one FP-adder and the initial register of the next effectively serve as inter-stage registers. Therefore, the inter-stage data flow can be summarized as follows:

Normalization Unit → Register → 8-bit Subtractor → Denormalization Unit → Register.

4.1.2 FP-CORDIC architecture with Global Denormalization and Normalization (ARC-2)

In the ARC-2 architecture, the denormalization and normalization processes are handled globally rather than at each pipeline stage. Denormalization is performed once at the input stage, and normalization is applied only at the final output stage. This global handling eliminates the need for redundant normalization and denormalization units within each floating-point adder/subtractor. The motivation behind adopting global normalization and denormalization lies in achieving an area- and power-efficient hardware implementation. As discussed in Chapter 3, the normalization and denormalization units together account for approximately 30% of the total area of the floating-point adder/subtractor. By implementing these units globally, the input is denormalized once at the beginning of the computation, and normalization is performed only once at the end, just before producing the final output. This architectural modification significantly reduces hardware redundancy and optimizes resource usage. However, since intermediate values between stages are not normalized, this approach demands careful management of data format and precision throughout the pipeline. The structural change introduced by this approach is illustrated in Figure 4.4.

From the Figure 4.4, it can be observed that the denormalization unit is placed at the top of the pipeline tree. In this implementation, a register block is added immediately after the denormalization unit to ensure that the pipeline's register-to-register propagation time does not reduce the minimum allowable clock period. The denormalization unit outputs a 33-bit result by explicitly appending the hidden leading '1' in the mantissa. As a result, all subsequent stages in the pipeline operate on 33-bit denormalized data and produce 33-bit outputs. Compared to the ARC-1 architecture, the key difference here is the data format: the 32nd bit represents the sign, bits 24 to 31 represent the exponent, and the remaining 24 bits store the mantissa. All pipeline stages—from the pre-iteration stage to the 23rd CORDIC iteration—use only stages 2 to 5 of the pipelined floating-point adder/subtractor (Model-2). This modified version is referred to as the **unnormalized floating-point adder/subtractor (Uaddsub)**. As shown in Figure 4.5, this adder takes 33-bit denormalized inputs (*AD* and *BD)* and produces a 33-bit unnormalized sum (*USUM*). The *USUM* output is a concatenation of the ***SUM_SIGN***, ***BIG_Exponent***, and the 24-bit unnormalized ***SUM_Mantissa***.

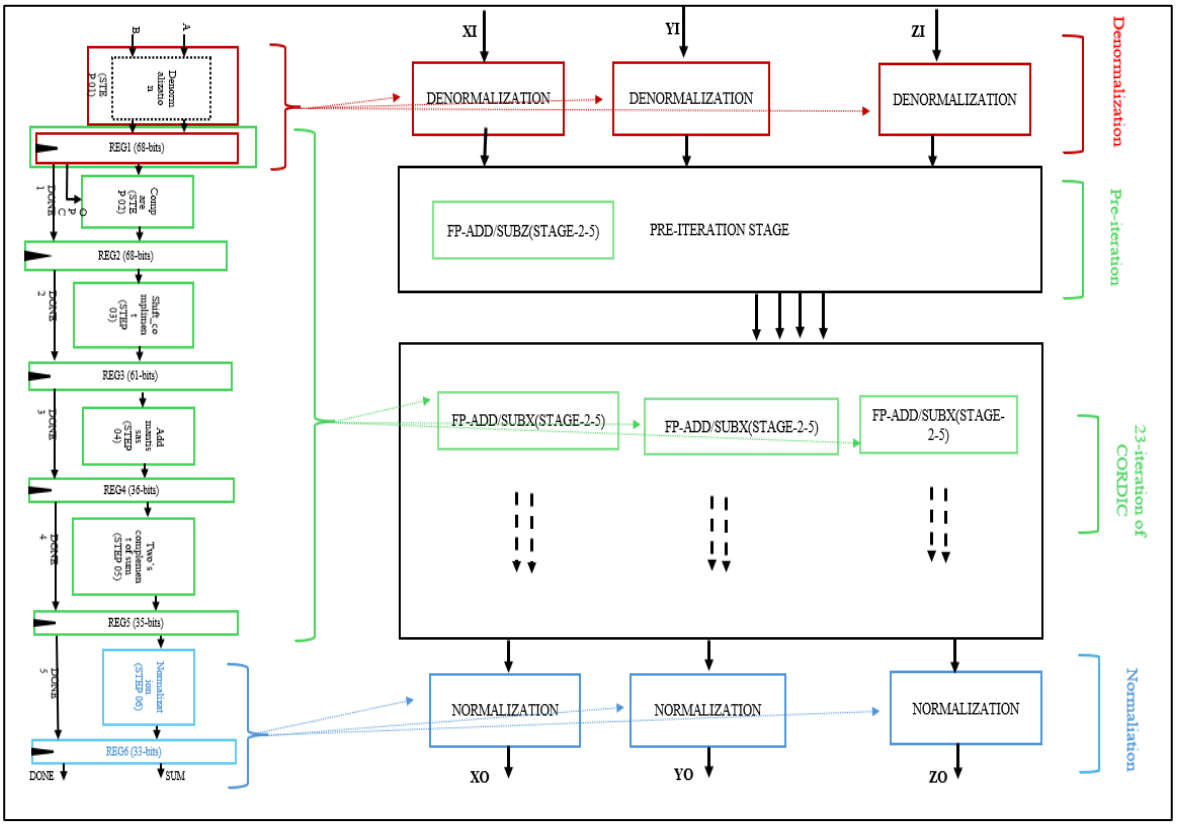

Figure 4.4: Distribution of the operations of floating addition to implement global normalization and denormalization scheme

It is important to note an additional output signal from this unit called ***SIGNCOM***, which indicates whether the input operands have the same sign. *SIGNCOM* is '0' when both inputs have the same sign and '1' when they differ. This signal is essential for the normalization stage and for handling special cases such as zero detection, as discussed in Chapter 3. Although *SIGNCOM* requires an additional flip-flop in register stage 5 (REG5), it is only generated in the final **Uaddsub** unit used in the **23rd iteration stage**—just before the normalization block. All other **Uaddsub** units from the pre-iteration to the 22nd iteration do not produce this output.

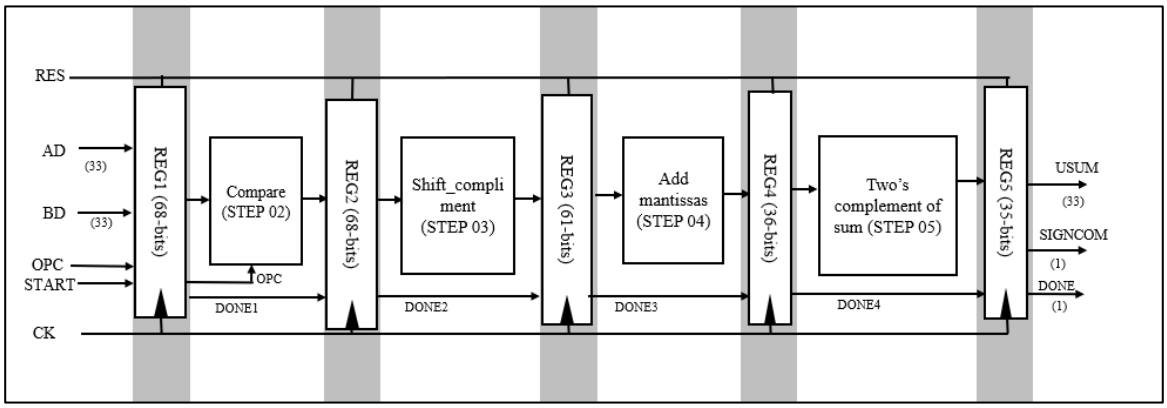

Figure 4.5 Architecture of UADDSUB (i.e. ADDSUB excluding normal and denormal unit)

The **pre-iteration stage** for this version of the CORDIC architecture is illustrated in Figure 4.6. This stage closely resembles that of Architecture-1, with the primary difference being that it operates on **33-bit denormalized data** for each path. Additionally, instead of using the regular floating-point adder/subtractor, this design employs the **unnormalized adder/subtractor (Uaddsub)**. This modification also impacts the pipelined delay registers in the X-path and Y-path. Specifically, **five register blocks** are used instead of six, and each register is **33 bits wide** to accommodate the denormalized format. Figure 4.6 presents the updated pre-iteration stage, where all changes from Architecture-1 are clearly highlighted in red.

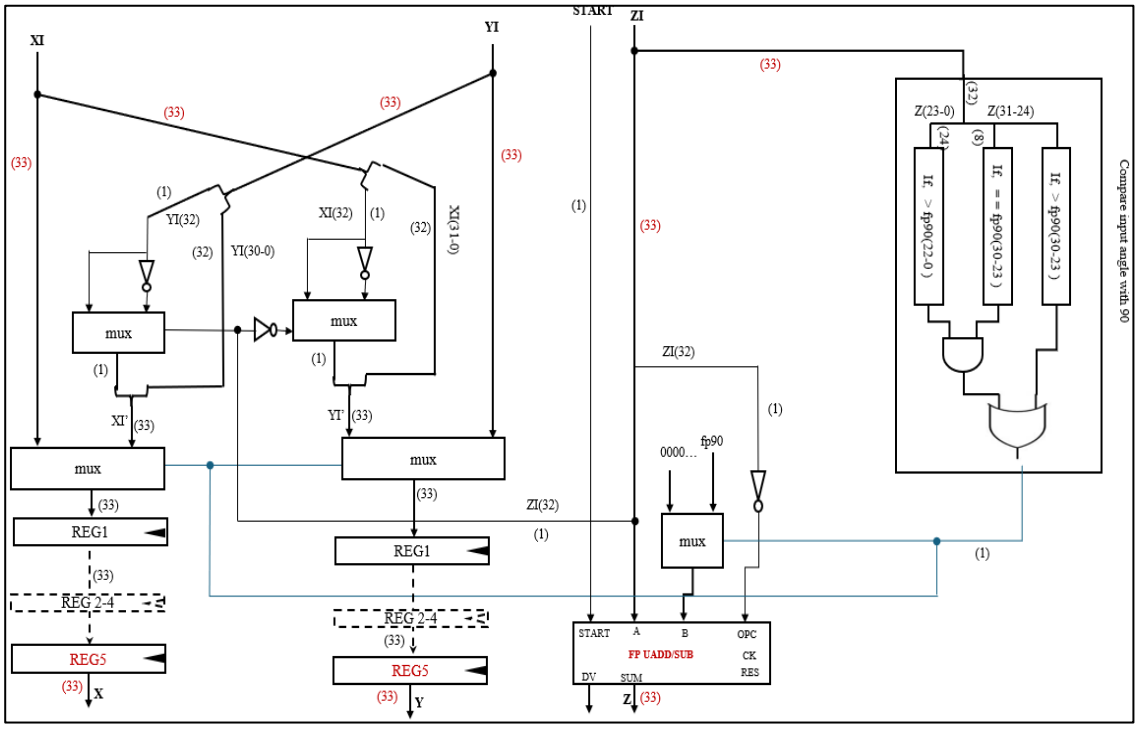

Figure 4.6: Pre-iteration (pre-rotation) stage of ARC-2

The **CORDIC iteration units** and the **top-level structure of the CORDIC processor** in this architecture closely resemble those of the previous pipelined stages. However, a key difference is that this version uses the **unnormalized adder/subtractor (Uaddsub)**, operating on **33-bit denormalized data** for both input and output during each clock cycle.

It is important to note that the **precomputed micro-angles** in this design are also **33 bits long**, representing denormalized values. These are structurally the same as in the previous architecture, except that a leading '1' is explicitly concatenated before the mantissa. This adjustment is essential because, in this architecture, denormalization is performed only once—

at the start—rather than at every iteration stage. As a result, the micro-angles must be stored in their denormalized form.

Overall, the **denormalization unit** is placed at the **top of the pipeline**, while the **normalization unit** is positioned at the **bottom**, finalizing the data before output. Within each CORDIC iteration stage, instead of using the full pipelined floating-point adder, only **stages 2 to 5** of the FP-adder are employed (i.e., the **Uaddsub),** ensuring efficient floating-point addition or subtraction. The top-level structure of this pipelined CORDIC architecture is depicted in Figure 4.7, where the updated elements are clearly shown.

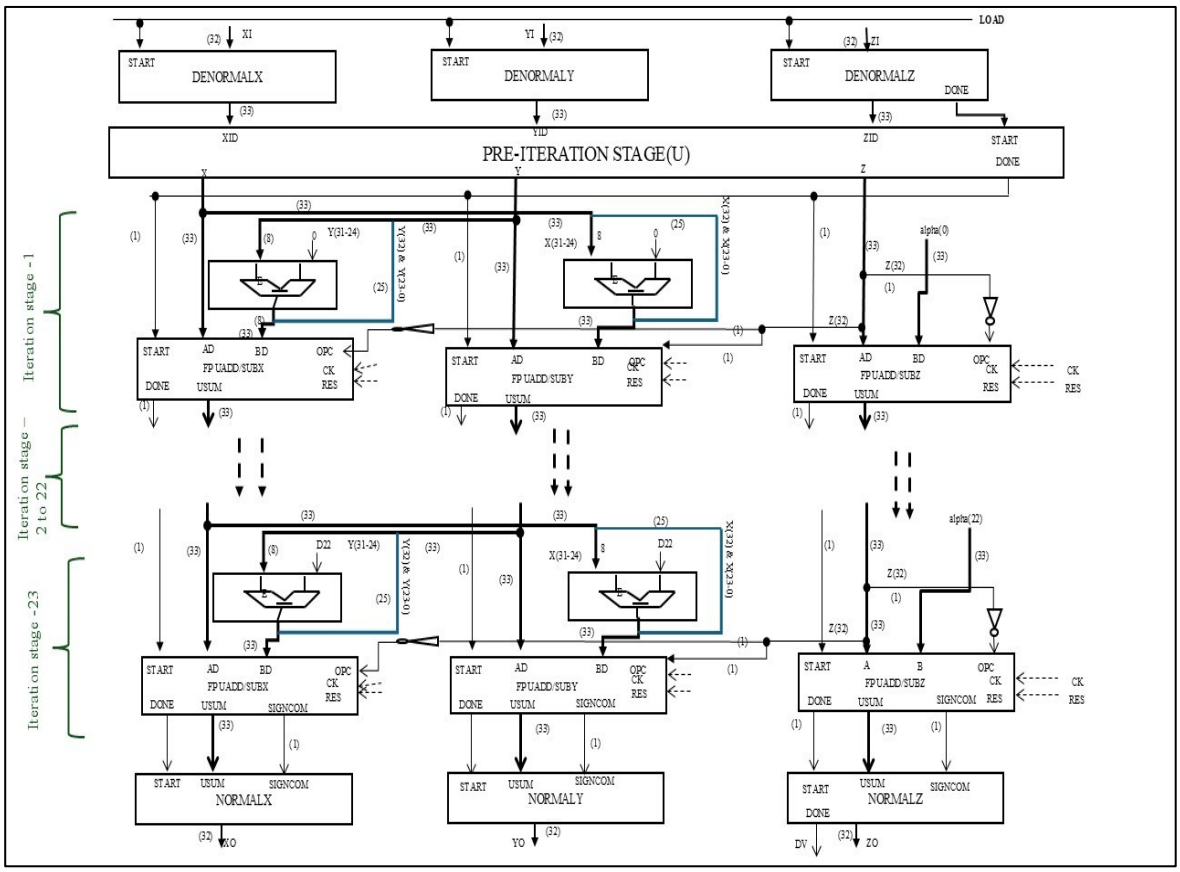

Figure 4.7: Designed Pipeline architecture of FP-CORDIC (Global normalization and denormalization)

4.1.3 Structural comparison of Architecture-1 and Architecture-2 of Pipelined FP-CORDIC implementation

The structural comparison between Architecture-1 and Architecture-2 is presented in Table 4-1. It is evident that Architecture-2 occupies significantly less combinational area, primarily due to the substantial reduction in the number of normalization and denormalization units. Furthermore, Architecture-2 achieves improved overall latency.

Table 4-1 : Comparison of pipelined FP-CORDIC architecture ARC-1 and ARC-2

	ARC-1	ARC-2
FP-adder used	addsub (6-stage)	usddsub (4-stage)
Total number of Denormalization unit	2×3×23 + 2 = 140	3
Total number of normalization unit	3×23+1 = 70	3
Number of clock cycle required to complete each pipeline stage	6 cycles	5 cycles (except global normalization and denormalization stage which require 1 clock cycle)
Total latency	6 × 24 = 144 clock cycle	5 × 24 + 2 = 122 clock cycle
pre-computed micro angle length (bits)	32 (single precision format)	33 (denormalized)

4.2 Simulation of Pipelined FP-CORDIC

For simulation and behavioural verification, a VHDL testbench and a MATLAB model of the FP-CORDIC were developed. To compute sine and cosine values, the inputs are configured as follows: the `XI` port receives 1/K, where K≈0.607252935008881256169 4, the `YI` input is set to 0, and the `ZI` input receives the target angle at each clock cycle. In both Architecture-1 and Architecture-2, valid outputs are obtained after 144 and 122 clock cycles, respectively, for a given input. The **sine** value is produced at the ***XO*** output, and the **cosine** value is obtained from the ***YO*** output. The ***ZO*** port provides the **residual angle** remaining after the full 23 iterations. Synopsys Verdi was used for behavioural verification during simulation.

4.2.1 Behavioural validation

For dynamic verification, 1449 test samples were generated in the range of −181° to +181° with a step size of 0.25°. These test samples were calculated using MATLAB and saved in a text file. The text file was then read in the VHDL testbench using the `std.textio` library. The simulation outputs were converted to IEEE 754 format and stored as a text file for further analysis. The same set of test samples was also fed into the MATLAB model of the FP-CORDIC. Figure 4.8 presents the plot of **cos(θ)** obtained from the ***XO*** outputs of both Architecture-1 and Architecture-2. Additionally, the output of the **custom MATLAB floating-**

point CORDIC model is plotted alongside the output of MATLAB's **built-in cos function** for comparison. As shown in the figure, all curves closely overlap, forming a proper cosine wave. This confirms the correctness and consistency of both FP-CORDIC architectures.

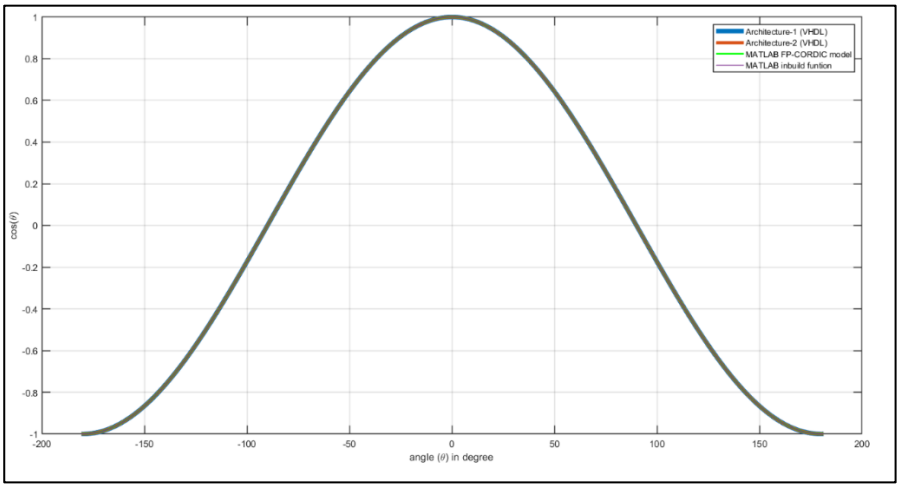

Figure 4.8: Cos curve plotted from data get from behavioural test for pipelined and serial architecture

Similarly, using the data obtained from the **`YO`** output of both **architectures**, **the custom MATLAB model**, and **MATLAB's built-in sin function**, the **sine** curves were plotted (as shown in Figure 4.9). The resulting curves form a proper sinusoidal waveform and are nearly identical, with all curves closely overlapping. This indicates the correct and consistent behavior of both FP-CORDIC architectures.

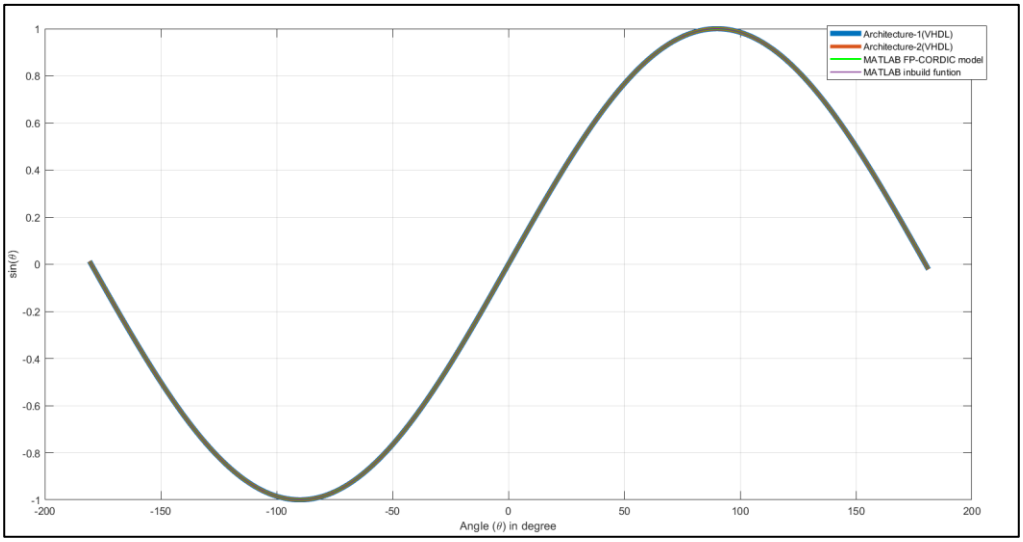

Figure 4.9: Sin curve plotted from data get from behavioural test for pipelined and serial architecture

4.2.2 Behavioural accuracy of the Architecture-1:

The behavioural accuracy of Architecture-1 has been validated through comparisons with both a custom MATLAB CORDIC model and MATLAB's built-in sin and cos functions. While the output plots from Architecture-1 closely align with those from the MATLAB models, they are not perfectly overlapped. These slight deviations arise from several key factors. Firstly, the limited number of CORDIC iterations restricts the precision with which the algorithm can converge to the desired angle. Secondly, bit loss occurs during alignment and normalization stages, particularly affecting the mantissa in floating-point operations. Lastly, the inherent precision limitations of the IEEE 754 single-precision (32-bit) number format, along with architecture-specific implementation details, contribute to the observed discrepancies.

Figure 4.10(a) illustrates the difference in cosine values generated by Architecture-1 compared to those obtained from the MATLAB CORDIC model. This plot highlights the deviation between the results from the VHDL simulation and the MATLAB model. For most inputs, the difference remains below 2.5×10^{-7}, with the maximum observed deviation being 5.96046×10^{-7}. Additionally, the inherent precision constraints of the IEEE 754 single-precision format contribute to minor inaccuracies, particularly near values close to 0 and 1.

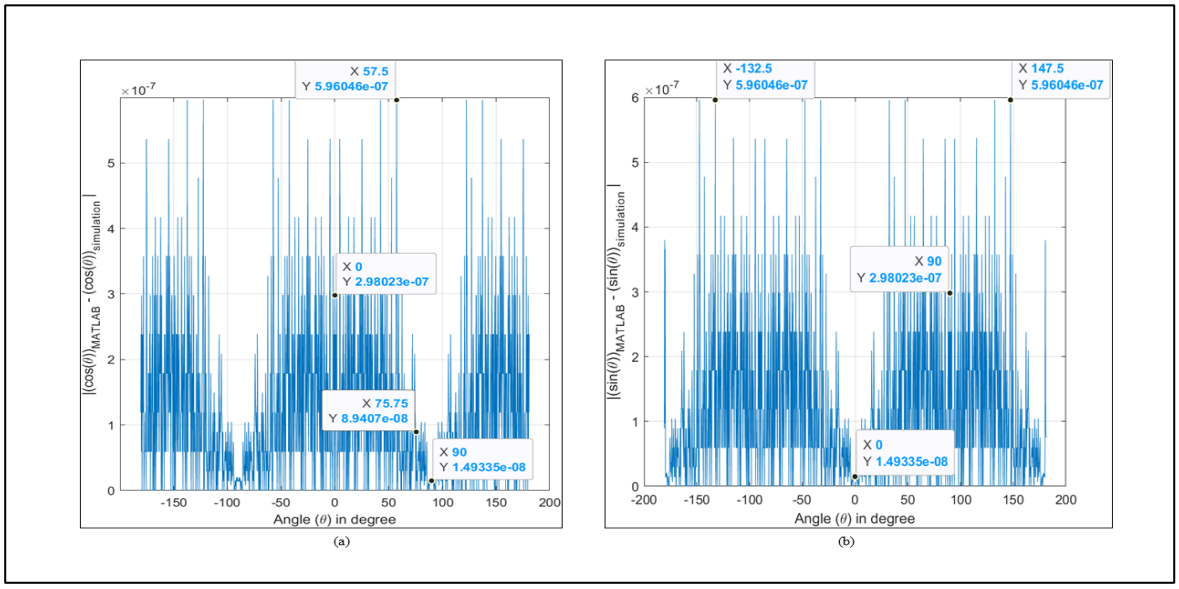

Figure 4.10: Difference between the value get from MATLAB model and VHDL Architecture-1: (a) cos (b) sin

To better illustrate the differences in cosine values due to precision and architectural limitations, Figure 4.11 present zoomed-in plots of the COS(θ) function at specific angles: 0° and 90°, which correspond to values 1 and 0, respectively. At 0° (Figure 4.11(a)), the expected output is 1, but the VHDL simulation of Architecture-1 produces a result of approximately 0.9999997.

Similarly, at 90° (as shown in Figure 4.11(b)), where the expected cosine value is 0, the output deviates slightly to -1.7533×10⁻⁷. This discrepancy arises primarily from the 23-bit precision limit imposed by the 23 CORDIC iterations and the 7-digit decimal resolution of the IEEE 754 single-precision format.

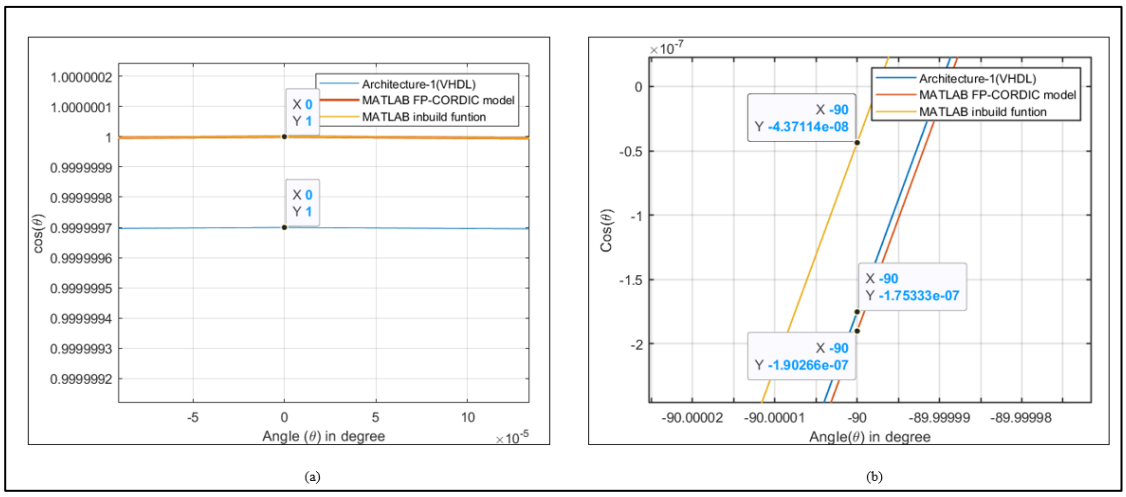

Figure 4.11: Cos value comparison of Architecture-1 for angles (a) 0° (b) 90°

For the sine values, the deviations for different input angles are shown in Figure 4.10(b), where the maximum deviation is 5.69046×10⁻⁷—identical to the maximum deviation observed in the cosine values. This clearly indicates that the deviation primarily stems from the limited precision of 32-bit floating-point representation and bit-loss during alignment. Additionally, Figure 4.12 provides zoomed-in views around output values near 0 and 1, which confirm the same level of deviation as seen in the cosine results. Overall, Architecture-1 demonstrates correct behaviour with a maximum error of 5.69046×10⁻⁷, a minimum value near zero of -1.7533×10⁻⁷, and a value near one of approximately 0.9999997.

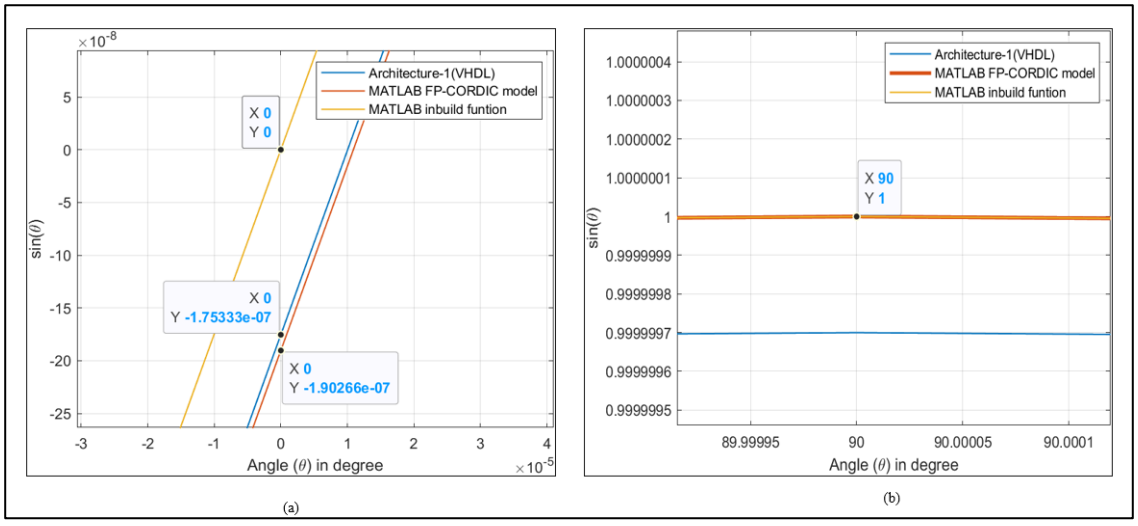

Figure 4.12: SIN value comparison for Architecture-1 angles (a) 0° (b) 90°

4.2.3 Behavioural accuracy of the Architecture-2:

In Architecture-2, the precision is expected to vary due to the absence of normalization from the pre-iteration stage through the 22nd CORDIC iteration stage. Figure 4.13 shows the differences in the output values for cosine and sine calculations. The maximum deviation for the cosine value is 1.07288×10^{-6}, while for the sine value it is 1.35228×10^{-6}, as shown in Figure 4.13. These values indicate that the maximum deviation has increased compared to Architecture-1. However, for most input angles, the deviation remains less than 4×10^{-7}.

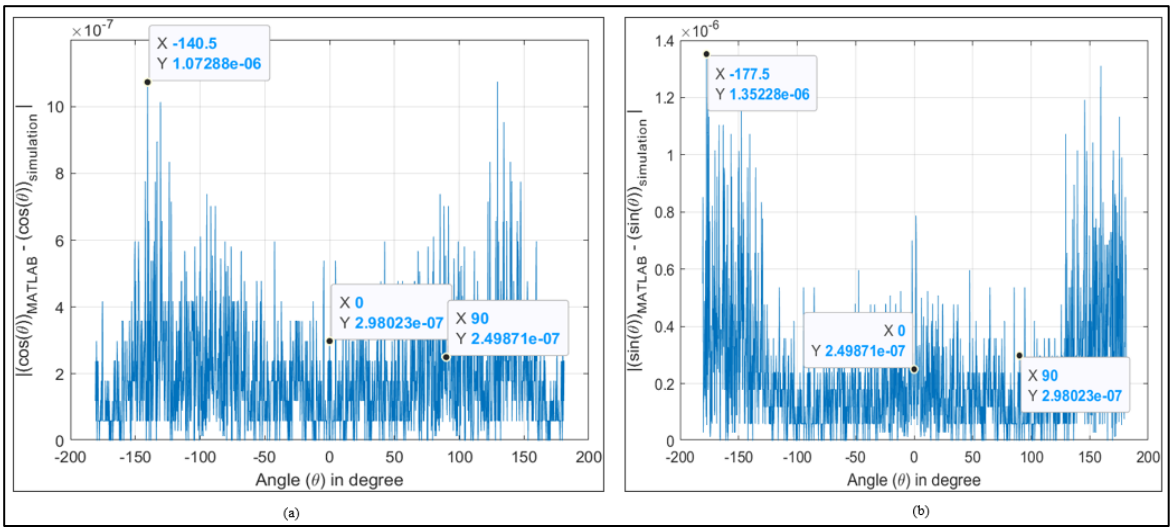

Figure 4.13: Difference between the value get from MATLAB model and VHDL pipelined Architecture-2 and serial architecture: (a) cos (b) sin

Near the zero output points for cos(90°) (Figure 4.14(a)) and sin(0°) (Figure 4.14(b)), the curves are zoomed in to highlight the increased deviation. While Architecture-2 shows a larger deviation compared to Architecture-1, it remains closer to zero and aligns more closely with the behaviour of MATLAB's built-in functions. If consider MATLAB's built-in function as a reference, the Global Normalization and Denormalization approach of Architecture-2 is demonstrated to be more accurate than the Local Normalization and Denormalization approach of Architecture-1.

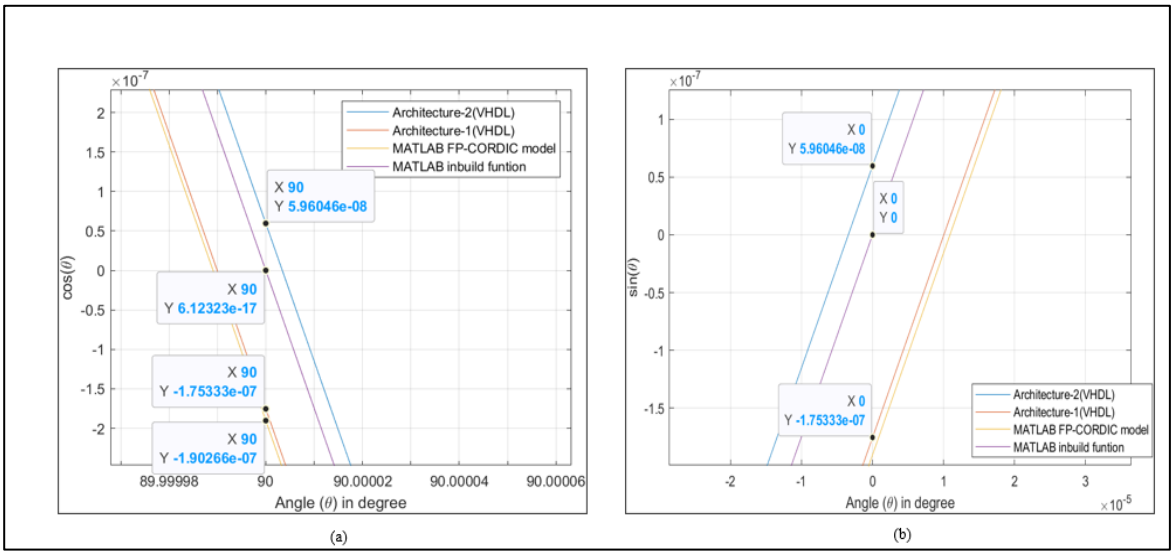

Figure 4.14: Precision at the vicinity of '0' for pipelined architecture and serial architecture (a) cos(90) (b) sin(0)

Similarly, in the vicinity of the output value 1 for $\cos(0°)$ (Figure 4.15(a)) and $\sin(90°)$ (Figure 4.15(b)), the curves were zoomed in to examine any deviations. However, no noticeable difference in behaviour was observed between Architecture-1 and Architecture-2 in this region.

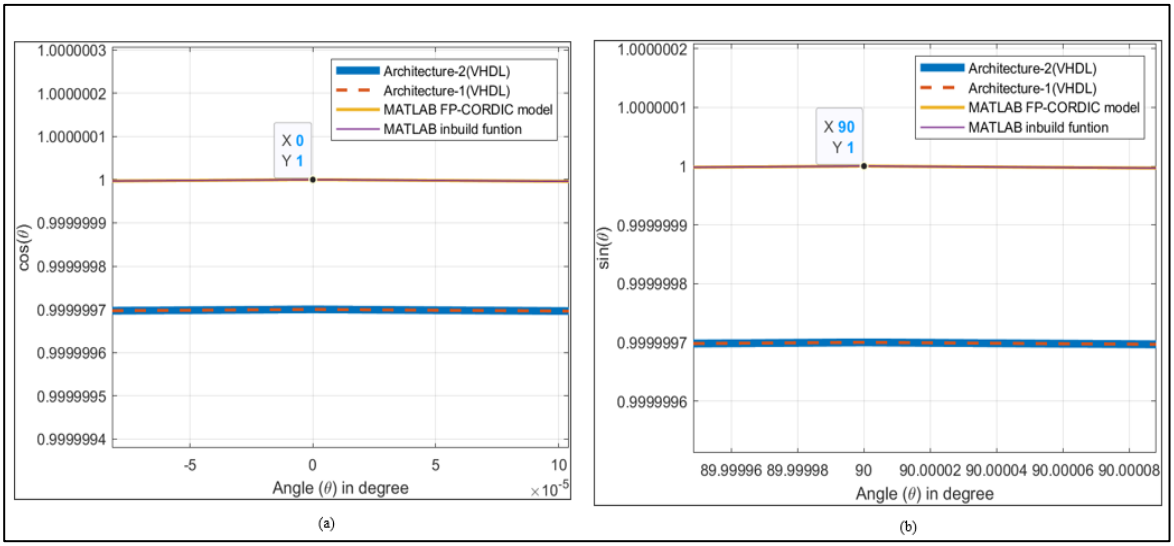

Figure 4.15:Precision at the vicinity of '1' for pipelined architecture and serial architecture (a) cos(0) (b) sin(90)

4.3 Gate-level synthesis of Pipelined FP-CORDIC

For gate-level synthesis, Design Vision software was used, with all constraints kept the same as those specified for the synthesis of the floating-point adder/subtractor in the previous chapter (see Table 3.1). The throughput target and expected to match the high throughput of the floating-point adder.

In the synthesis trials for the UMC13 process, different strategies such as flattening, Compile Ultra, and incremental mapping were explored. Although `Compile_Ultra` was expected to provide the best optimization results, in the case of UMC13, it sometimes resulted in poorer optimization. Most of the time, the best-optimized solutions were obtained using the traditional `'compile'` command with the following settings:

- `exact_map` enabled
- `map_effort` set to medium
- `area_effort` set to medium
- `power_effort` set to medium
- `fixed design rules and optimization` enabled

For the GF22 process, maintaining high throughput at the lower hierarchy levels with an adequate slack margin (≥15% of clock period) is particularly critical. This is due to the deep pipeline and the long hierarchy (up to level 4), which demand the use of the highest-performance cells available in the library. To achieve a time-efficient solution, several synthesis strategies were explored. These included using the traditional compile command with varying `map_effort` settings (e.g., medium and high), performing `incremental mapping` to improve quality of results **(QoR)**, and ungrouping specific hierarchical blocks such as the level-2 hierarchy (floating-point adder), the denormal and normalization units, and all 8-bit subtractors.

However, none of these strategies yielded satisfactory optimization results. The most efficient solution for Architecture-2 achieved a minimum clock period of 0.8 ns with a 13.5% positive slack margin. Ultimately, the best timing-optimized result was obtained using `compile_ultra`, which applied very high `map_effort` settings to meet the timing constraints and utilized the highest-performing cells from the target library. Despite its longer compilation and optimization time, `compile_ultra` proved to be the most effective approach.

The synthesis setup for `compile_ultra` included:

- `Using a timing` script to prioritize timing constraints
- Enabling exact mapping `(exact_map)`
- Disabling automatic ungrouping (`No auto_ungroup` selected) to preserve hierarchy and avoid flattening, which can complicate placement and routing
- Enabling `fixed_design_rule and optimization`

All synthesis trials were evaluated under both worst-case (setup-time) and best-case (hold-time) timing conditions.

4.3.1 Report time:

Table 4-2 presents the timing report of the pipelined FP-CORDIC architecture for both UMC13 and GF22 technologies. Owing to the deeply pipelined design, the clock frequency directly reflects the processor's throughput. For the UMC13 process, Architecture-1 operates with a clock period **of 4.5 ns (200 MHz**), whereas Architecture-2 operates at **4.0 ns (250 MHz),** resulting in a **1.125× increase in throughput**. The setup time analysis shows a positive slack of +0.82 ns (18%) for Architecture-1 and +0.96 ns (24%) for Architecture-2. In terms of hold time (best-case analysis), both architectures exhibit a slack of approximately +0.07 ns, corresponding to 1.6% and 1.75% of the clock period, respectively.

For the GF22 process, Architecture-1 operates at a clock period of **0.7 ns (1.48 GHz),** while Architecture-2 achieves a shorter period of **0.6 ns (1.66 GHz),** yielding a **1.166× throughput improvement**. Architecture-2 also maintains a higher positive slack of +0.132 ns (22%) compared to +0.0967 ns (13.82%) for Architecture-1 in the setup time analysis. The hold time slack is +0.0166 ns (2.38%) for Architecture-1 and +0.0144 ns (2.4%) for Architecture-2.
Additionally, as shown in Table 4.0, Architecture-2 exhibits a latency advantage, requiring 22 fewer clock cycles than Architecture-1. This reduction, combined with the higher clock frequency and improved slack margins, demonstrates the superior performance and timing robustness of Architecture-2 across both technologies.

Table 4-2: Report timing (pipelined architecture)

Technology	**Clock period (Frequency)**		**Setup time analysis (Worst case) Slack (% of clock period)**		**Hold time analysis (Best case) Slack (% of clock period)**	
	ARC-1	**ARC-2**	**ARC-1**	**ARC-2**	**ARC-1**	**ARC-2**
UMC13	4.5 ns (200 MHz)	4.0 ns (250 MHz)	+0.82 ns (18%)	+0.96 ns (24%)	+0.07 ns (1.6%)	+0.07 ns (1.75%)
GF22	0.7 ns (1.48 GHz)	0.6 ns (1.66 GHz)	+0.0967 ns (13.82%)	+0.132 ns (22%)	+0.0166 ns (2.38%)	0.0144 ns (2.4%)

4.3.2 Report Area:

Due to its unrolled structure and corresponding pipelined architecture, the CORDIC processor occupies a relatively large area, as expected. Table 4-3 presents the area report after gate-level synthesis. However, Architecture-2 demonstrates a notable reduction in **area**—approximately **20% less** than Architecture-1—for both technologies. This improvement is primarily attributed to the elimination of separate normalization and denormalization units, as well as a reduction in the number of registers (flip-flops). The reduction in flip-flops is further reflected in a nearly **12%** decrease in the area occupied by sequential cells.

Table 4-3: Report area (pipelined architecture)

	UMC13		**GF22**	
	ARC-1	**ARC-2**	**ARC-1**	**ARC-2**
No. of Cells	103782	78832	104722	82375
No. of combinational cell	81651	59202	84187	64227
No. of Sequential cell	21454	19224	19951	17715
No. of Buffer/Inverter	6254	3840	11240	8554
No. of reference	120	127	120	127
BUF/INV area	24200.9	14845.44	1498.4	1140.6
Combinational Area	719210.2	543344.64	26584.6	20194.1
Non-combinational area	549222.4	492134.41	30542.6	27119.5
Total cell Area	1268432.7	1035479	57127.2	47313.6

Table 4-4 also provides a detailed breakdown of area distribution across different hierarchical blocks. In addition, Figure 4.16 visualizes the percentage of area consumed by each building block: subfigures (a) and (b) correspond to UMC13, while (c) and (d) represent GF22. For Architecture-1, which employs local normalization and denormalization, the floating-point adder/subtractor (**addsub**) dominates the area usage, occupying nearly **96%** of the total FP-CORDIC processor area. Within this, the **normalization and denormalization** units account for approximately **13%** of the overall area in UMC13 and **10%** in GF22.

Table 4-4: Area occupied by different lower hierarchical blocks

	UMC13		GF22	
	ARC-1	ARC-2	ARC-1	ARC-2
Total FP-ADDSUB/UADDSUB	1212721.9 (ADDSUB)	972844.81 (UADDSUB)	55370.53 (ADDSUB)	45257.14 (UADDSUB)

Normal + denormalization unit	164805.0 (part of addsub)	11976.96 (not the part of uaddsub)	5743.8 (part of addsub)	544.1 (not the part of uaddsub)
8-bit CLA subtractor	25082.9	25141.76	292.33	292.33
Pre-iteration unit	30526.7	25332.48	1460.4	1210.1

Following architectural optimization—specifically, the adoption of global normalization and denormalization—the floating-point adder (uaddsub) occupies approximately **93%** of the total area of the FP-CORDIC processor. In this optimized architecture, the floating-point adder no longer includes internal normalization or denormalization units. Instead, three normalization and three denormalization units are placed externally at the input and output stages, respectively. Together, these units account for only about 1% of the total CORDIC area.

The pre-iteration unit occupies approximately 2% to 3% of the total area. It is important to note that the area reported for the floating-point adder in Table 4.2 excludes the internal adder present within the pre-iteration unit, whereas the area attributed to the pre-iteration unit includes its internal floating-point adder. Additionally, the 8-bit subtractor (**CLA8**) contributes around 1% to 2% of the total area.

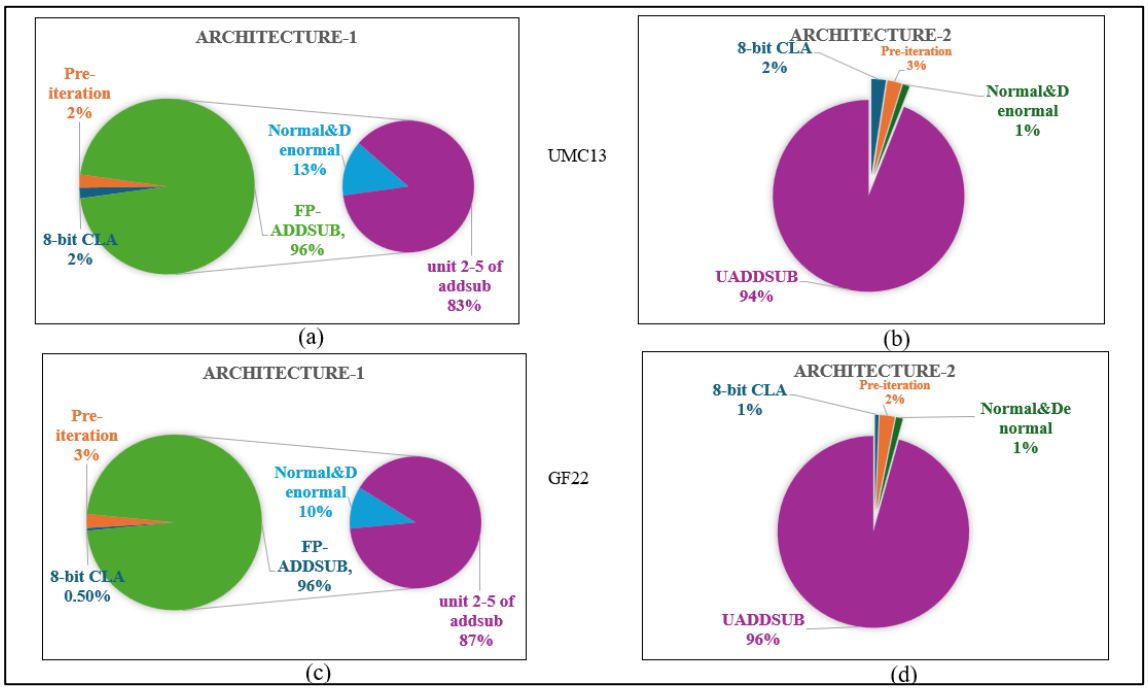

Figure 4.16: Area consumed by low hierarchy (a) ARC-1(UMC13), (b) ARC-2 (UMC13), (c) ARC-1(GF22), (d) ARC-2(GF22)

4.3.3 Report Power:

Table 4-5 presents the power report of both architectures. It is observed that, in the case of UMC13, Architecture-2 consumes only about 1% more power than Architecture-1. For GF22,

this difference increases slightly, with Architecture-2 consuming approximately 3% more power.

When comparing the two technologies, GF22 requires nearly twice the power of UMC13. This increase is primarily due to GF22's ability to support significantly higher operating frequencies. For both technologies and architectures, leakage power constitutes only about 0.2% of the total power consumption. Notably, GF22 exhibits exceptionally low leakage power, a result of its SOI (Silicon-On-Insulator) technology.

Table 4-5: Power report (pipelined architecture)

	UMC13		**GF22**	
	ARC-1	ARC-2	ARC-1	ARC-2
Cell internal power (mW)	40.77	41.145	95.51	98.9
Net switching power(mW)	0.56	0.581	0.230	0.228
Total Dynamic Power (mW)	41.332	41.73	95.74	99.03
Cell Leakage power (mW)	0.20	0.178	0.1707	0.193
Total Power Consumption (mW)	41.55	41.91	95.91	99.3

Architecture-2 also shows slightly higher switching power, likely due to its increased activity at higher frequencies. Figure 4.17(a) illustrates the percentage distribution of power components, while Figure 4.17(b) provides an overall power consumption comparison between the architectures and technologies.

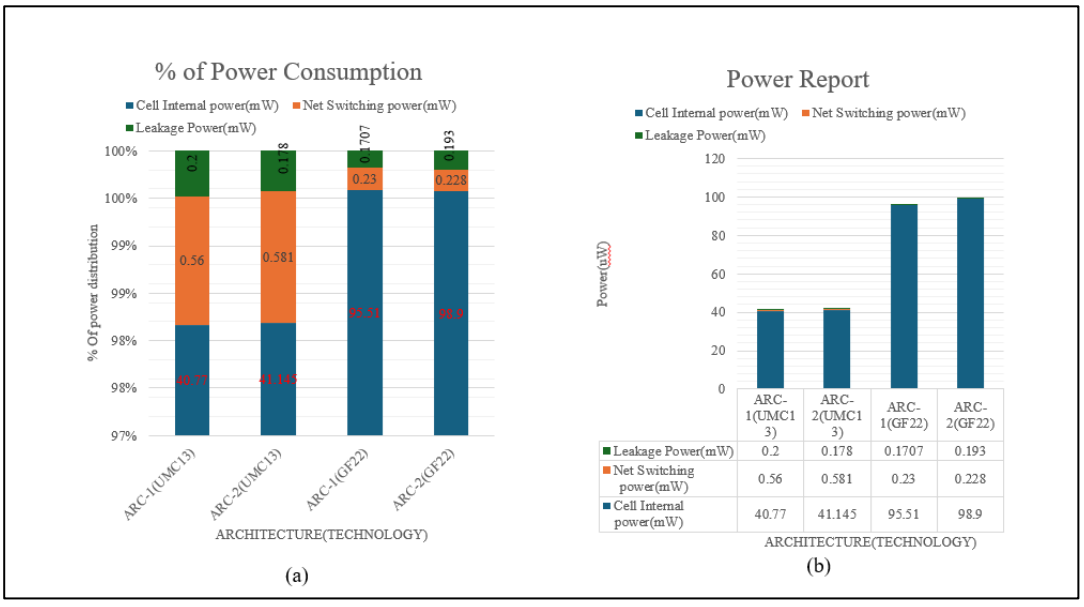

Figure 4.17: Power report visualization of pipelined FP-CORDIC (a) % of Power consumption (b) power comparison

Overall, Architecture-2 (with global normalization and denormalization) supports higher operating speed and demonstrates better area efficiency, while maintaining total power consumption comparable to Architecture-1 (with local normalization and denormalization). However, the behavioural accuracy comparison is more nuanced. When using **MATLAB's built-in high-precision** `sin` and `cos` functions as a reference, Architecture-2 exhibits behaviour that is more precise than that of Architecture-1. Conversely, when using the **custom MATLAB model** developed as part of this project as the reference, Architecture-1 shows a closer match in behaviour compared to Architecture-2.

4.4 Serial Architecture of FP-CORDIC

The serial architecture of the Floating-Point CORDIC (FP-CORDIC) has been implemented with the objective of reducing area while still attempting to improve throughput. In this design, a single CORDIC iteration unit is employed, and all iterations are performed sequentially—one after another. Consequently, the selection of an appropriate FP-adder architecture is crucial.

4.4.1 Selecting FP-adder/subtractor for serial FP-CORDIC

A **pipelined floating-point adder** is not well-suited for this context, as the serial nature of the CORDIC architecture produces a valid output only after completing all iterations. Likewise, a **serial floating-point adder** is also not ideal, since it typically requires five clock cycles to complete a single addition. Additionally, not all CORDIC iterations require the same computation time, and the iteration with the longest delay dictates the minimum achievable clock period—thus limiting the maximum operating frequency and overall throughput.

To address these challenges, a fully combinational (flat) version of the floating-point adder/subtractor, without internal registers, is adopted. This configuration—referred to as *flat-uaddsub*—is shown in Figure 4.18. The **flat-uaddsub** design offers reduced latency per addition and better aligns with the timing requirements of the serial CORDIC architecture, enabling more efficient operation without introducing pipeline-related delays. Although this architecture operates at a relatively low frequency (i.e., has a longer clock period) due to its fully combinational design, it offers superior performance in terms of throughput (speed of generating each output), power efficiency, and area. For testing, logic synthesis and timing analysis of the **flat-uaddsub** were performed under the same constraints and environment setup as described in Chapter 3, with one register placed at the input and another at the output.

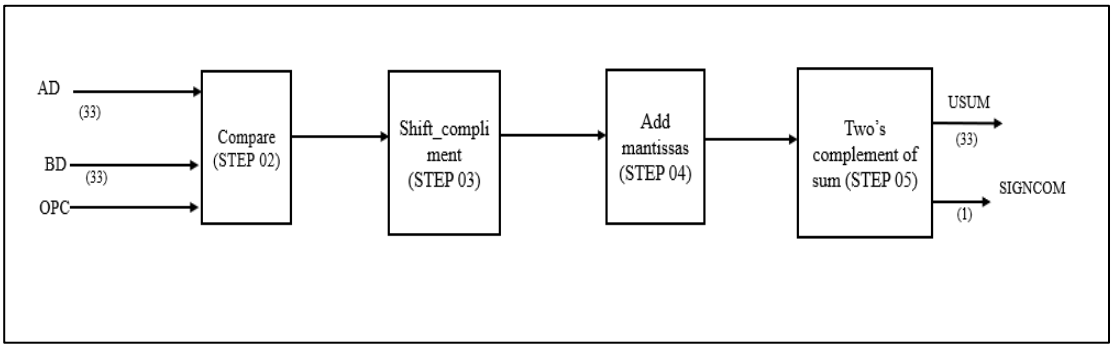

Figure 4.18: Full combinational implementation of Floating-point unnormalized adder/subtractor (FLAT-UADDSUB)

The minimum clock period for the *flat-uaddsub* was found to be 11 ns, in addition to an estimated 2 ns delay for the 8-bit subtractor. Therefore, completing one CORDIC iteration using the flat-uaddsub requires approximately 13 ns. Table 4-6 presents an early estimation of the throughput of the FP-CORDIC processor using different FP-adder architectures. This estimation is based on data obtained from the UMC13 technology node, as similar trends were observed for GF22. The total processing time to generate each output is calculated as:

Input to output delay=Clock period×Number of clock cycles to complete one addition (i.e., on CORDIC iteration)×(Number of CORDIC iterations+1)+2×Number of external normalizatio n/denormalization unit

Here, the term "number of CORDIC iterations + 1" means that one additional iteration is added to the total number of CORDIC iterations to account for the pre-iteration stage. It is evident from the table that the *flat-uaddsub* architecture achieves the lowest overall processing time while expected to maintaining a smaller area footprint compared to other adder architectures.

Table 4-6: Estimated performance of serial architecture of FP-CORDIC for different adder integration

	ADDSUB	**UADDSUB**	**Flat-uaddsub**	**Serial FP-adder/subtractor (Local denormalization and normalization)**	**Serial FP-adder/subtractor (Global denormalization and normalization**
Clock period	3.5 ns	3.5 ns	11+2 ns	5 ns	5 ns
clock cycle to complete addition	6	5	1	5	4
No of. External denormal + normal in a path	0	2	2	0	2

total input to output delay	(3.5×6×24) =504 ns	(3.5×5×24) + 3.5×2 =427 ns	(13×1×24) + (13×2) = 338 ns	(5×5×24) =600 ns	(5×4×24) + 5×2 =490 ns

4.4.2 Serial architecture design of FP-CORDIC

The serial architecture of the CORDIC processor comprises an arithmetic logic unit, a control unit, a ROM, a denormalization unit, and a normalization unit shown in Figure 4.19.

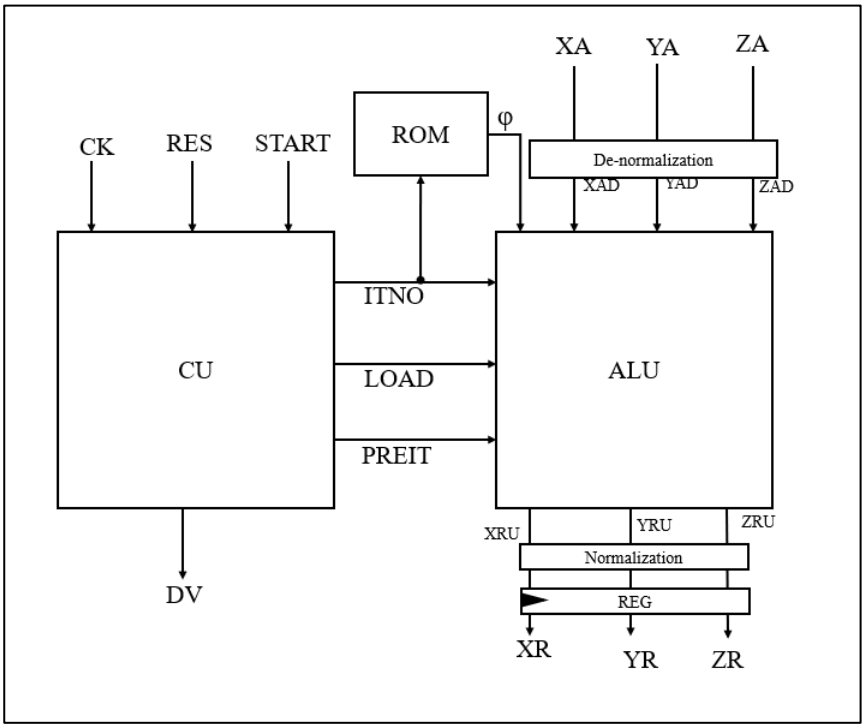

Figure 4.19: Designed serial architecture of FP-CORDIC

Denormalization unit and normalization unit:

The normalization and denormalization units are the same as those described in Chapter 3. In this architecture, each unit is utilized only once and placed at the data input and output stages, outside the CORDIC iteration circuitry. Consequently, normalization and denormalization are not performed during each iteration. This approach implements global normalization and denormalization in the serial FP-CORDIC architecture, thereby ensuring higher speed and reduced processing overhead per iteration.

ROM:

The ROM is used to store the precomputed micro-angles. The signal called `ITNO` (iteration number), generated by the control unit, serves as the address to the ROM, and the corresponding

micro-angle is provided as output signal called *alpha*. Like Architecture-2 of the pipeline design, the micro-angles are stored in denormalized form—that is, the leading 1 of the mantissas is retained. Consequently, each micro-angle is represented using 33 bits.

Control unit (CU):

The control unit is implemented as a finite state machine (FSM) with a single state register storing the current state. The process begins when the input signal ***START*** is set to '1', and it continues until the entire operation is complete. Upon completion, the flag signal ***DV*** (Data Valid) is asserted to '1'. The state diagram of the control unit is shown in Figure 4.1, with the initial state labelled as **IDLE**. The **LOAD_XYZ** state is responsible for accepting new input data, indicated by the ***LOAD*** signal. The subsequent state, **IT_PRE**, is dedicated to the pre-iteration phase and activates the signal ***PREIT***. Following this, 23 CORDIC iterations are processed sequentially through states named IT0 to IT22. The state diagram of this control unit illustrated in Figure 4.20.

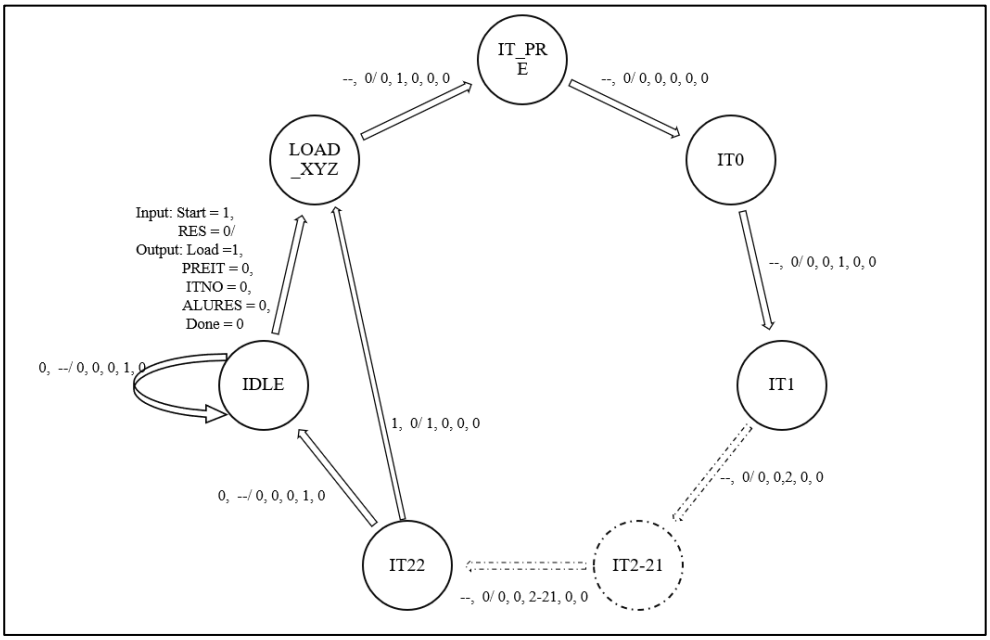

Figure 4.20: State diagram of Control unit (CU) (serial FP-CORDIC)

Arithmetic logic unit (ALU):

The ALU performs all the CORDIC logic and arithmetic operations (addition, subtraction, comparison etc.). The *LOAD* signal initializes the process by accepting new inputs—*XAD, YAD,* and *ZAD*—which are stored in three separate registers. It should be noted that, all inputs provided to the ALU are already denormalized. After each iteration, the outputs from the **X-path, Y-path**, and **Z-path** are stored in registers *X, Y,* and *Z*, respectively. The second state

(*IT_PRE*) of the process is dedicated to the pre-iteration phase, which operates as described in the previous section. Figure 4.21 depict the ALU where all signal paths related to the pre-iteration are highlighted in red.

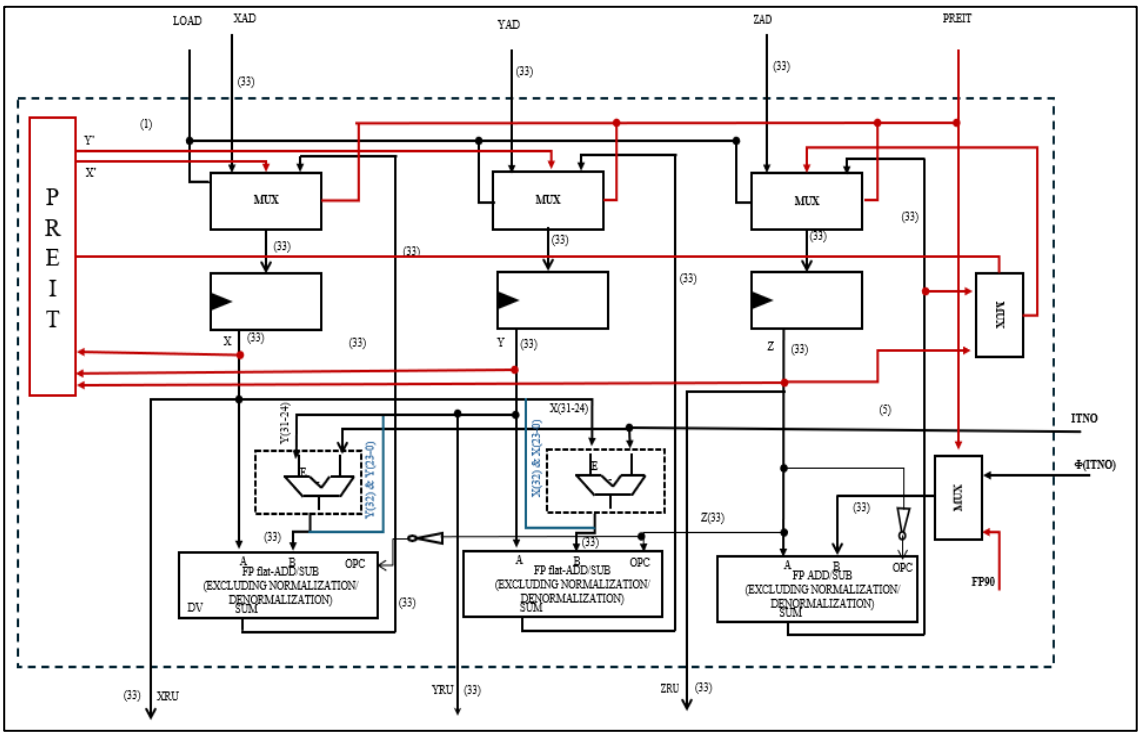

Figure 4.21: Arithmetic logic unit (ALU) (serial FP-CORDIC)

Figure 4.22 illustrates the PREIT unit. The main difference between the pre-iteration units of the two architectures is that the pipeline architecture uses a separate dedicated FP-adder for the pre-iteration stage, whereas the serial architecture does not. Instead, the serial architecture uses the floating-point adder in the Z-path itself for the pre-iteration. When the signal *PREIT* is asserted ('1'), the *input-B* of the floating-point adder/subtractor receives a constant value of 90 degrees. Consequently, the output of this adder represents θ ± 90° (i.e. *ZAD± 90°*). A control signal named *greater90* is set to '1' when the input angle exceeds ±90°, and '0' otherwise. This signal serves as the select input for a multiplexer: if *greater90* is '1', the register stores the output of the subtractor; otherwise, it stores the original angle *ZAD* value. This mechanism ensures the correct scaling of the input angle.

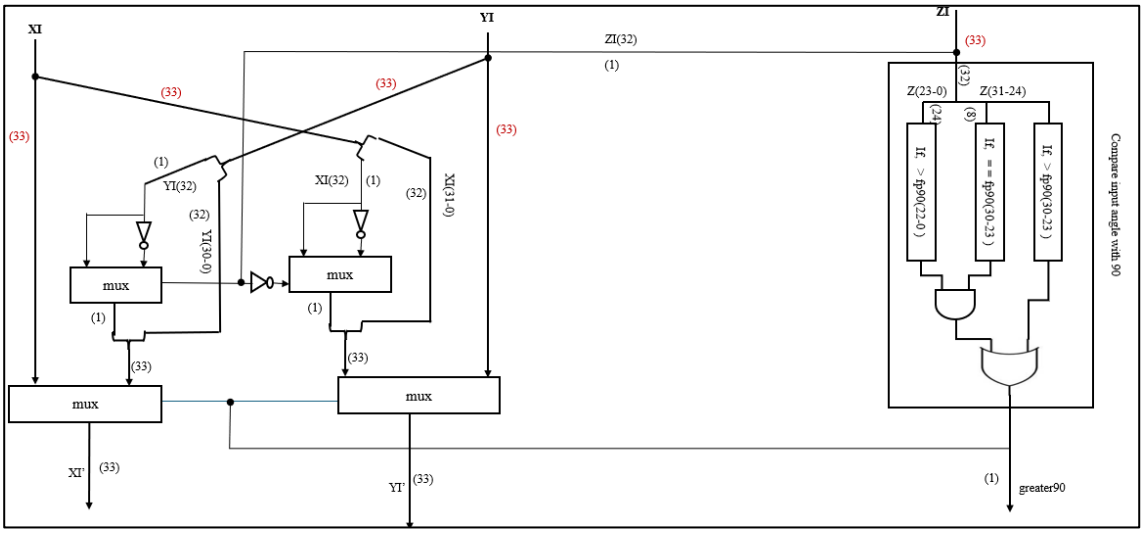

Figure 4.22: Logic circuit of Pre-iteration (integrated part of ALU show as the block PREIT in Figure 4.21)

Following the pre-iteration, the 23 CORDIC iterations are performed over the next 23 clock cycles. During these iterations, the `PREIT` signal is reset to '0' by the controller, and the micro-angle corresponding to each iteration `(alpha (ITNO))` is used. It is important to note that the pre-iteration logic does not add any additional hierarchy; rather, it is integrated as a purely combinational part of the ALU.

4.5 Simulations of serial architecture of FP-CORDIC

A testbench was created to simulate and verify the serial architecture, providing one input only after the previous computation completes (i.e. `DV` = '1'). The same test samples used for the pipeline architecture were applied, yielding identical results. Therefore, the plots shown in Figures 4.8, Figure 4.9 Figure 4.13, Figure 4.14 and Figure 4.15 also represent the behavioural performance of the serial architecture. In terms of accuracy, it matches that of the pipeline architecture with global normalization and denormaization (ARC-2). To avoid redundancy, all behaviour-related plots and discussions are omitted.

The primary characteristic of the serial architecture is that it processes one operation at a time. It accepts input when the `START` signal is asserted ('1') and produces output only after completing all iteration stages. This behaviour was validated through waveform traces generated using Verdi. Figure 4.23 displays a waveform segment that illustrates the iterative process, including state transitions and control signals such as `START` and `DV` ***(Data Valid)***, all functioning as defined in the architecture. The output data are considered valid when the `DV`

signal is asserted (‘1’), which occurs after the final iteration state, **`IT22` (the 23rd CORDIC iteration)**.

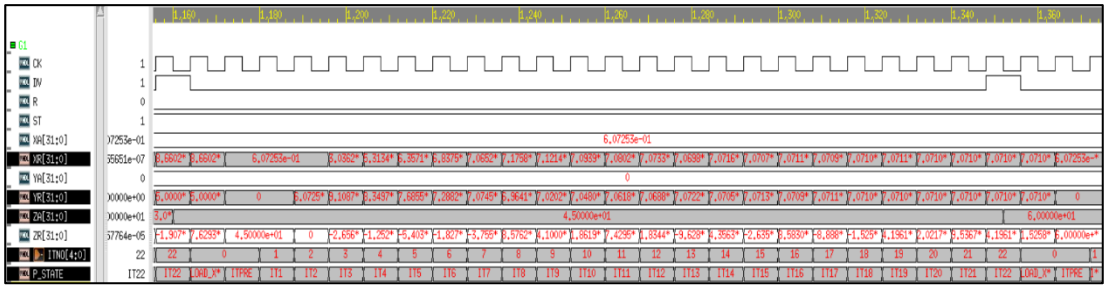

Figure 4.23: Behaviour of serial architecture of FP-CORDIC

4.6 Gate level Synthesis of the serial architecture of FP-CORDIC

The gate-level synthesis of the serial architecture of the FP-CORDIC was carried out using the same constraints specified in Table 3.5. Since the serial architecture employs a single CORDIC iteration unit and performs all iterations sequentially, a low area footprint is inherently expected. However, this comes at the cost of higher latency. Therefore, the synthesis process was configured to maximize operational speed. The compilation settings included:

- `Exact_map` enabled
- `Map_effort` medium
- `Area_effort` medium
- `Power_effort` medium
- `fixed design rules and optimization` enabled

4.6.1 Report Timing:

Table 4-7 presents the timing report of the serial FP-CORDIC architecture. The minimum achievable clock period is **12 ns for UMC13** technology and **6 ns for GF22**. The primary bottlenecks limiting the speed are combinedly the FP-UADDSUB unit and the 8-bit subtractor (serves as the functional counterpart to the shifter uses in fixed-point implementations). The lower frequency is mainly due to the fully combinational design of the floating-point adder. Although both pipeline and serial implementations of the FP-adder/subtractor are capable of operating at higher speeds individually, the overall performance of the serial CORDIC processor is significantly constrained—this has been previously explained and summarized in Table 4-6.

The maximum achievable throughput of the serial architecture is **3.47 MSamples/s** for UMC13 and **17.36 MSamples/s** for GF22, which is substantially lower than that of the pipelined

counterpart. Additionally, the fully serial nature of the architecture restricts it to processing only one operation at a time, and it requires 24 clock cycles to produce a valid output after receiving an input.

Table 4-7: Report timing (serial FP-CORDIC)

	Clock period (Frequency)	**Throughput Mega Samples/s**	**Setup time analysis (Worst case)** **Slack (% of clock period)**	**Hold time analysis (Best case)** **Slack (% of clock period)**
UMC13	12 ns (83.33 MHz)	3.47 M Sample/s	+1.8 ns (15%)	+0.11 ns (0.92%)
GF22	2.4 ns (416.67 MHz)	17.36 M Sample/s	+0.398 ns (16.6%)	+0.0107 ns (0.44%)

4.6.2 Report Area:

As the serial architecture uses only a single CORDIC iteration unit, it occupies a significantly smaller area compared to its pipelined counterpart. For the **UMC13** technology, it requires only **32,966.4 µm²**, and for **GF22**, it requires just **1,409.7 µm²**—both of which are approximately **1/30th** the area of the pipelined architecture. This drastic reduction is primarily due to the reuse of a single functional unit across all iterations, rather than duplicating hardware for parallelism as in the pipelined design. The area report summary shown in Table 4-8

Table 4-8: Report Area (serial FP-CORDIC)

	UMC13	**GF22**
No. of Cells	3031	3386
No. of combinational cell	2799	3157
No. of Sequential cell	203	203
No. of Buffer/Inverter	274	437
No. of reference	11	11
BUF/INV area (um2)	1068.8	58.17
Combinational Area (um2)	27769.6	1098.91
Non-combinational area (um2)	5196.8	310.8
Total cell Area (um2)	32966.4	1409.7

Table 4-9 illustrates the area distribution of each unit within the **serial FP-CORDIC architecture**, which is visually represented in Figure 4.24. The **logic unit (ALU)** is the most area-consuming component in the serial FP-CORDIC architecture, occupying approximately **68% to 69%** of the total cell area. Within the ALU, the floating-point adder/subtractor (UADDSUB) alone accounts for a substantial portion, consuming around **51%** of the total area in **UMC13** technology and **46%** in **GF22** technology. The remaining portion of the ALU—comprising multiplexers (MUXs), registers, the pre-iteration related logic, and other supporting logic components—collectively occupies about **15%** of the area in **UMC13** and **19%** in **GF22**.

Table 4-9: Area occupied by different unit of serial FP-CORDIC

	UMC13 (um^2)	**GF22 (um^2)**
Total FP-ADDSUB/UADDSUB	16617 (5539 each) (Flat UADDSUB)	653 (217.7 each) (flat UADDSUB)
8-bit Subtractor	1090.6 (545.3 each)	41.53 (20.77 each)
Mux, REG, pre-it and others logic	4984.24	267.79
ALU	22691.84	962.32
Normal + denormalization unit	9024	394.57
CU	455.68	23.09
ROM	777	29.42

The **global normalization and denormalization units** combinedly consume approximately **28%** of the total area in the serial FP-CORDIC architecture. It is important to note that, unlike in the pipelined design, this global approach does not lead to area savings in the serial architecture. However, it significantly **improves speed** by allowing the CORDIC iterations to proceed without repeatedly performing normalization and denormalization at each stage. This architectural choice reduces processing overhead within the iterative data path, enabling a more efficient and streamlined computation process despite the additional area cost.

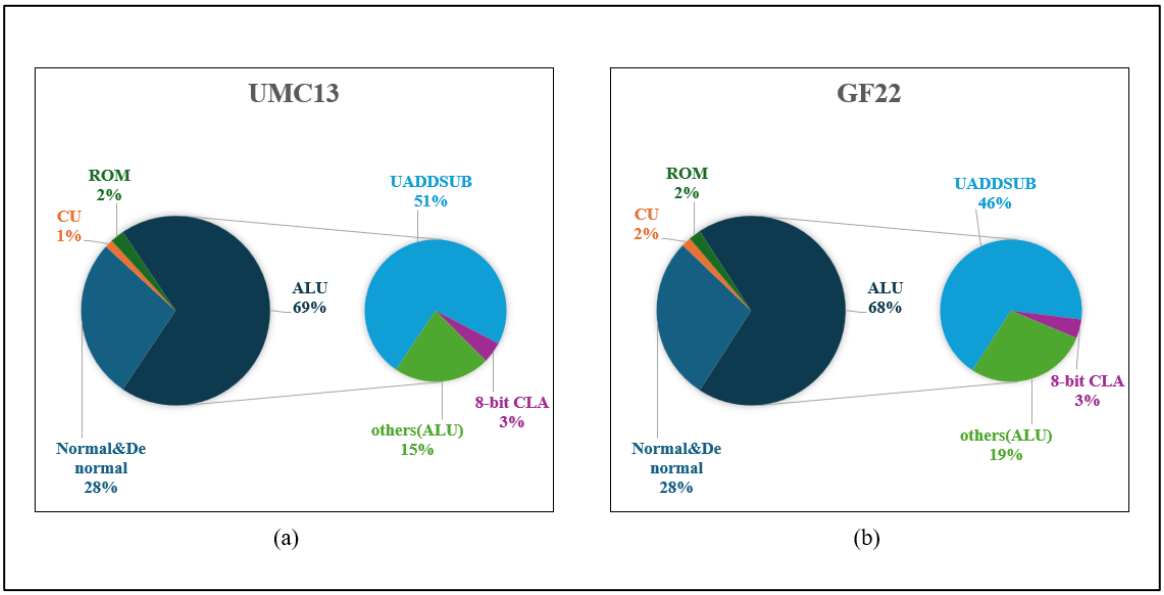

Figure 4.24: Visualization of % of area allocated for different unit of serial FP-CORDIC (a) UMC13 (b) GF22

The **ROM** occupies approximately **2%** of the total area in the serial FP-CORDIC architecture. It is important to note that no dedicated memory library (such as NAND-based memory macros) was used to implement the ROM. Instead, it was synthesized as a fully combinational logic block. Therefore, it is more appropriate to consider the ROM as a programmed lookup table, where the input address (denoted as signal `ITNO`) and the corresponding output `alpha(ITNO)` (pre-computed micro-angle) are realized using Boolean logic expressions. The realization of the ROM structure for this architecture is illustrated in Figure 4.25.

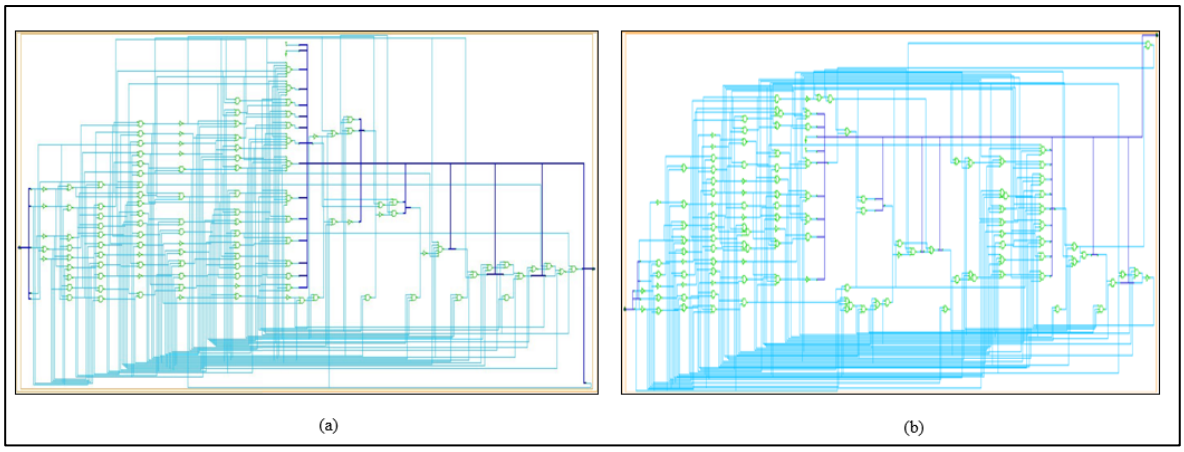

Figure 4.25: ROM realization to store the micro-angle (serial architecture): (a) UMC13 (b) GF22

4.6.3 Report Power:

The power report presented in Table 4-10 shows that the GF22 implementation consumes less power than UMC13, primarily due to its lower switching power requirements. Interestingly,

although the throughput of the GF22 implementation is nearly four times higher, it still exhibits lower overall power consumption. This efficiency is largely attributed to the shorter channel length which significantly reduce dynamic (switching) power.

Table 4-10: Report power (Serial FP-CORDIC)

	UMC13	**GF22**
Cell internal power (uW)	361.6	438.83
Net switching power(uW)	244.6	99.14
Total Dynamic Power (uW)	606.2	537.97
Cell Leakage power (uW)	6	4.8
Total Power Consumption (uW)	612.2	542.8

Although the serial architecture offers significantly lower throughput than its pipelined counterparts, its superior area and power efficiency make it an ideal choice for low-power, cost-sensitive applications. Its compact footprint is especially advantageous for area-constrained environments such as portable, embedded, and IoT devices, where minimizing chip size and cost is crucial.

Chapter 5

Physical Design

The final stage of the digital VLSI design flow is physical design, where the abstract gate-level netlist is translated into a detailed geometric representation of the circuit suitable for silicon fabrication. This chapter details the physical implementation of the proposed Floating-Point CORDIC (FP-CORDIC) architecture, marking the critical transition from synthesized logic to a layout-ready design. The physical design phase bridges the gap between logical functionality and physical manufacturability.

The process involves several essential steps, including floorplanning, power planning, placement, clock tree synthesis (CTS), signal routing, design rule checking (DRC), timing analysis, and post-layout verification. Each stage contributes to ensuring that the final layout satisfies the required timing, area, and power constraints, while also adhering strictly to the design rules imposed by the target semiconductor technology.

The primary goal of this phase is to achieve a functionally correct, timing-closed, and manufacturable layout that mirrors the behaviour of the synthesized RTL. In this project, the physical design was implemented using **Cadence Innovus**, a state-of-the-art physical design and implementation tool widely used in industry.

This chapter describes the complete physical design flow adopted for both pipelined and serial versions of the FP-CORDIC architecture. It outlines the design methodologies, tools, and optimization strategies employed, and presents the final layout views for each implementation. Furthermore, key metrics such as core area utilization, routing congestion, clock quality, power distribution, and DRC/LVS results are analysed and discussed in detail.

The sections in this chapter are organized as follows:

- **Section 4.1** Physical Design steps
- **Section 4.2** Layout Pipelined architecture of FP-CORDIC
- **Section 4.3.** Layout Serial architecture of FP-CORDIC

5.1 Physical design Steps

5.1.1 Design file Preparation

The physical design process commenced with the preparation of all essential input files to establish a valid and functional design environment. The gate-level netlist, generated from the logic synthesis stage using tools *Design_Compiler*, was saved in Verilog format. In

parallel, timing and design constraints were captured in an SDF (Standard Delay Format) file to ensure accurate delay modeling during layout and timing analysis. To facilitate the physical implementation flow, a script file was created to define the paths to necessary resources, including the technology LEF (Library Exchange Format) files, standard cell library LEF files, and other setup configurations. Additionally, a post-synthesis netlist was refined prior to being saved: the netlist was uniquified to eliminate any redundant modules, and unconnected internal nets were removed, resulting in a cleaner and more manageable input for place and route tools.

A global configuration file was employed to organize references to critical design inputs such as technology LEF files, cell library LEFs, and the MMMC (Multi-Mode Multi-Corner) setup. The MMMC configuration specified the locations of all required timing libraries, including .lib files (for functional and timing characterization) and their corresponding .db (database) files, categorized under worst-case (maximum delay), best-case (minimum delay), and typical (nominal) conditions. It also defined the operating corners and modes for comprehensive timing analysis under various voltage, process, and temperature conditions.
Finally, the layout process was initiated by loading the Verilog netlist, SDF file, and the global setup file, which included the MMMC configuration. Global power nets (VDD) and ground nets (VSS) were also defined, completing the environment setup and allowing the physical implementation to proceed.

5.1.2 Floor Planning

In this stage of the design flow, the chip area was meticulously planned and partitioned to allocate sufficient space for key components, including standard cells, power grids, and routing channels. Special attention was given to the definition of the core and I/O regions, ensuring that physical resources were efficiently utilized. The core utilization factor—which represents the ratio of area occupied by standard cells to the total core area—was carefully chosen to strike a balance between area efficiency and routing feasibility. A higher utilization increases density but can lead to routing congestion, while a lower value improves routability at the cost of area. By optimizing this factor, the design achieved efficient placement of logic cells while maintaining sufficient whitespace for routing and clock distribution, contributing to better timing closure and overall design performance.

5.1.3 Power Planning

The power distribution network (PDN) was architected to ensure uniform and reliable power delivery across the entire chip, minimizing IR drop and maintaining signal integrity. Initially, the global power and ground nets were connected to the respective VDD and VSS pins defined in the technology library. To facilitate robust power distribution, power rings were created around the perimeter of the core, followed by the insertion of vertical stripes within the core region. These stripes were aligned with metal layers to reduce resistance and connect the power grid effectively to all standard cells. Additionally, horizontal power lines were introduced to bridge the stripes and complete the grid, forming a well-connected mesh. This hierarchical PDN structure ensured a stable and sufficient power supply to all functional blocks and minimized the risk of localized voltage drops, thus enhancing the overall performance and reliability of the design.

5.1.4 Placement

The primary objectives of the placement phase in VLSI design are to minimize the total interconnection (wire) lengths and to alleviate interconnect congestion, thereby optimizing both performance and area utilization. Placement is typically executed using two broad categories of algorithms: constructive placement and iterative improvement techniques. The process generally begins with a constructive placement approach, which generates an initial layout based on predefined rules or heuristics—common examples include the Min-Cut algorithm and the Spectral (Eigenvalue) method. These methods provide a coarse initial solution that serves as a baseline. Subsequently, iterative placement improvement algorithms are applied to refine the initial layout by incrementally adjusting the positions of logic cells to improve design metrics. These adjustments are guided by cost functions that evaluate parameters such as wirelength, cell overlap, timing delay, and routability. The output of the placement process is a detailed mapping of cells and functional blocks to specific physical locations on the chip, forming the foundation for the subsequent routing phase in the physical design flow.

5.1.5 Clock Tree Synthesis (CTS)

A clock distribution network was carefully designed to deliver the clock signal to all sequential elements with minimal skew and balanced timing, ensuring synchronized operation throughout the circuit. To handle the high fanout nature of the clock signal, additional buffers were inserted strategically, helping to maintain signal integrity and timing consistency. During each

optimization phase, setup time violations were addressed using cell grouping techniques, while hold time violations were mitigated by inserting extra buffers in critical paths. The clock tree synthesis (CTS) process was optimized for both timing accuracy and power efficiency, achieving a robust and energy-aware clock distribution framework. Following CTS, post-CTS optimization was performed, focusing on resolving remaining setup and hold violations, and meeting design rule constraints (DRC) such as maximum capacitance, maximum fanout, and transition limits, to ensure the design met all physical and electrical requirements.

5.1.6 Routing

The **routing stage** in the physical design flow involves creating the actual interconnections between standard cells using metal wires, with the goal of achieving **100% net connectivity** while minimizing **total wirelength**, **crosstalk**, and **signal delay**. Routing is typically divided into two key phases: **Global Routing** and **Detailed Routing**.

- **Global Routing** is a coarse-level routing phase that generates an approximate routing path for each net without defining exact geometries. Its primary goals are to estimate routing resources and allocate nets to appropriate routing regions. This phase is generally carried out in three main steps:
    - Region Definition: The chip layout is partitioned into distinct routing regions or bins.
    - Region Assignment: Each net is assigned a series of routing regions it will traverse, based on congestion and net priority.
    - Pin Assignment: Specific pins or access points are selected for each net within its assigned regions.
- **Detailed Routing**, on the other hand, defines the precise geometric layout of the interconnects. This stage converts the approximate global routes into actual metal layers and vias, adhering to the design rules of the manufacturing process. Routing is performed incrementally, region by region, with a routing order determined by net criticality (timing sensitivity) and regional routing density. During this phase, efforts are also made to minimize design rule violations, crosstalk, and electromigration effects, ensuring the interconnects meet both electrical and physical constraints.

5.1.7 Static Timing Analysis (STA)

Following each major step in the physical design flow, timing constraints were carefully evaluated and optimized to ensure the design consistently met all functional and performance

specifications. Static Timing Analysis (STA) was conducted using industry-standard tools to validate that the circuit operated reliably within the defined clock period, under all specified process, voltage, and temperature (PVT) corners. STA provided a comprehensive assessment of both setup and hold timing across all paths in the design, helping identify and resolve any timing violations that could impact functionality. By iteratively refining the design and re-analyzing timing at each stage—such as placement, clock tree synthesis (CTS), and routing—timing closure was achieved. This rigorous validation process ensured that the design was functionally robust, timing clean, and ready for tape-out and manufacturing.

5.2 Layout Pipelined architecture of FP-CORDIC

5.2.1 UMC13 implementation

In the UMC 13µm technology implementation, the floorplan was carefully designed with a 10-micron margin around the core. These intentional spacing plays a crucial role in accommodating the power ring, IO pad connections, and signal routing channels required for seamless integration with the chip's periphery. By reserving this boundary, routing congestion near the IO boundary is minimized. Figure 5.1(a) presents the final layout of the FP-CORDIC unit, demonstrating the complete physical integration of the design using the UMC13 process. A magnified view of the red-circled region in Figure 5.1(b) reveals the detailed floorplan at that location, highlighting the use of Metal 7 and Metal 8 for implementing the power ring. These upper metal layers are specifically chosen due to their greater width and lower sheet resistance, which are essential for carrying the relatively large currents required for stable operation across the entire chip. Metal 8 is additionally leveraged to form power stripes that extend into the core region, ensuring uniform power distribution and reducing IR drop.

For the initial stages of power routing, Metal 1 is utilized to connect local standard cell rails to the higher-level metal structures. This hierarchical power routing strategy helps balance routing density across layers and ensures design rule compliance. The layered approach also simplifies power grid verification and facilitates easier debug in post-layout analysis. Figure 5.1(c) offers a further zoomed-in physical view of the same highlighted region, showing the offset between the core and the outer boundary. This offset is optimally used to place IO ports and route global signals, providing sufficient space for signal integrity buffers.

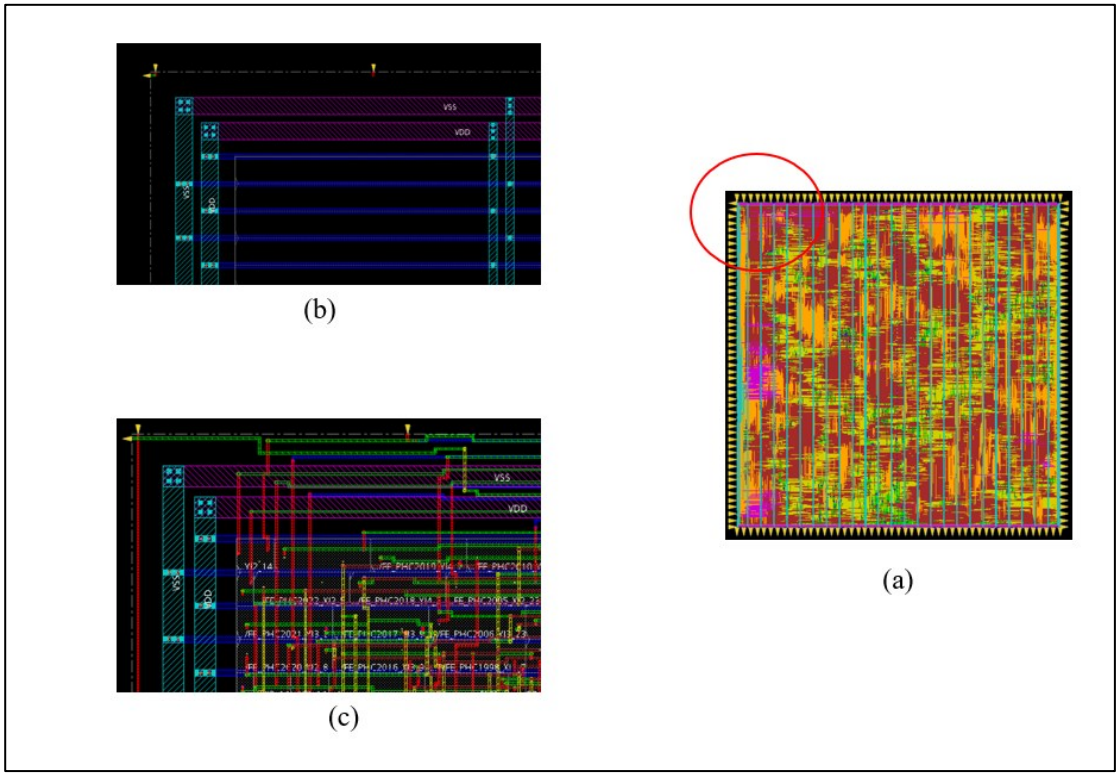

Figure 5.1: UMC13: (a) layout pipeline arcitecture (b) zoomed(red circle region) floor plan view (c) zoomed(red circle region) view show IO port and cell route

Figure 5.2 illustrates the final cell placement after the insertion of filler cells, showing a nearly complete utilization of the core area. The placement is highly optimized, with approximately 80% area utilization, which typically raises concerns regarding routing congestion and potential design rule violations. Such a high utilization can often lead to routing difficulties, increased parasitic, and timing degradation. However, in this design, no DRC (Design Rule Check) violations, metal fill issues, or routing blockages were observed after the final routing stage. This successful outcome can be attributed to several strategic measures taken during physical design. First, an incremental placement approach with cell spreading was adopted to balance density across the layout. This method allowed the design tool to distribute standard cells more evenly, thus reducing local congestion hotspots. Additionally, a high-effort congestion optimization strategy was employed during placement, specifically targeting critical areas with increased routing demand. This approach helped ensure that even with tight cell packing, enough routing resources remained available.

Another key factor contributing to clean routing was the deliberate choice to maintain adequate core-to-boundary offset. This buffer zone facilitated the routing of global signals and IO connections without adding stress to the core routing tracks. Furthermore, higher metal layers

(such as Metal 7 and Metal 8) were strategically utilized for global and power routing, which alleviated pressure on the lower layers that are typically used for signal interconnects.

Collectively, these optimization techniques enabled a compact yet routable layout, achieving both area efficiency and design rule compliance. The result demonstrates that with careful planning and tool-driven optimization strategies, high utilization designs can still meet all physical constraints.

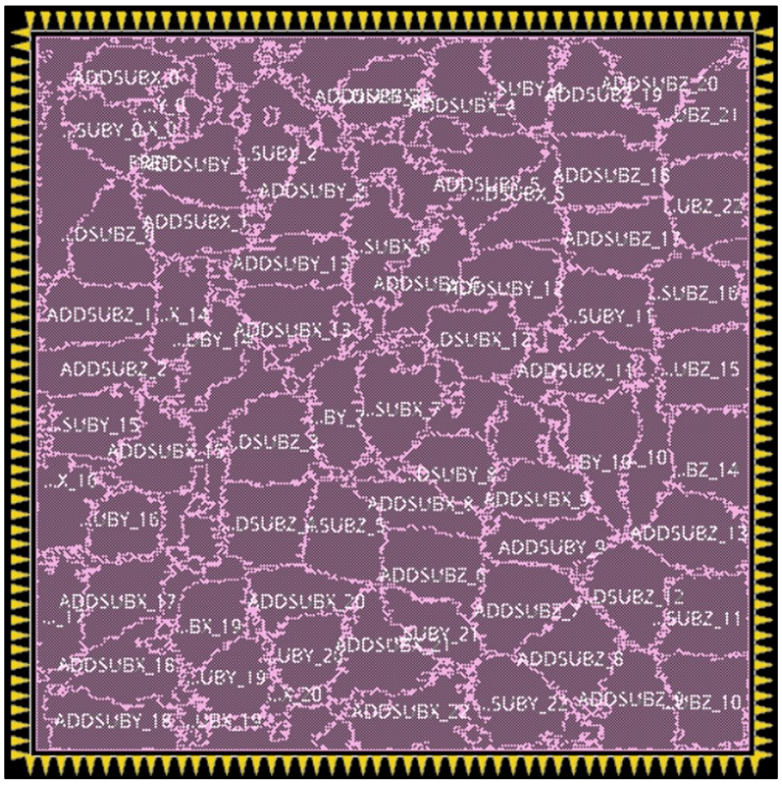

Figure 5.2: Placement of standard cell (Pipeline architecture) using UMC13

After final routing the DRC, connectivity, power via and metal density verified without any violation. Final sign off hold (slack +0.05 ns) and setup (slack +0.5 ns) report don't show any negative slack.

5.2.2 GF22 implementation

The placement and routing process using GF22 (GlobalFoundries 22nm technology) posed several challenges and required multiple iterative refinement cycles to achieve a clean and functional final layout. Due to the increased complexity and tighter design rules at this technology node, special care was taken during floor planning and track configuration to ensure DRC-clean results and efficient routing.

During the **floorplan** phase, the tracks for standard cell placement and routing were customized. A deliberate 3-micron gap was introduced between the standard cell placement track and the routing track grid. This adjustment provided the necessary spacing to avoid routing pitch violations and helped reduce local congestion. By separating placement and routing resources, the design tool was able to utilize routing layers more effectively without encountering track alignment issues or design rule conflicts.

In the **power planning** stage, additional care was taken to meet the specific requirements of the GF22 process. The global power net (VDD) was connected not only to the standard library power pins (VDD) but also to the buried N-well contact pin (VNW_P). Similarly, the ground net (VSS) was routed to both the standard ground pins (VSS) and the buried P-well contact pin (VPW_N). These connections are crucial for ensuring substrate biasing and bulk connection integrity, which are particularly important in advanced process nodes like GF22. These connections and their placement strategy are illustrated in Figure 5.3 (adopted from [34]).

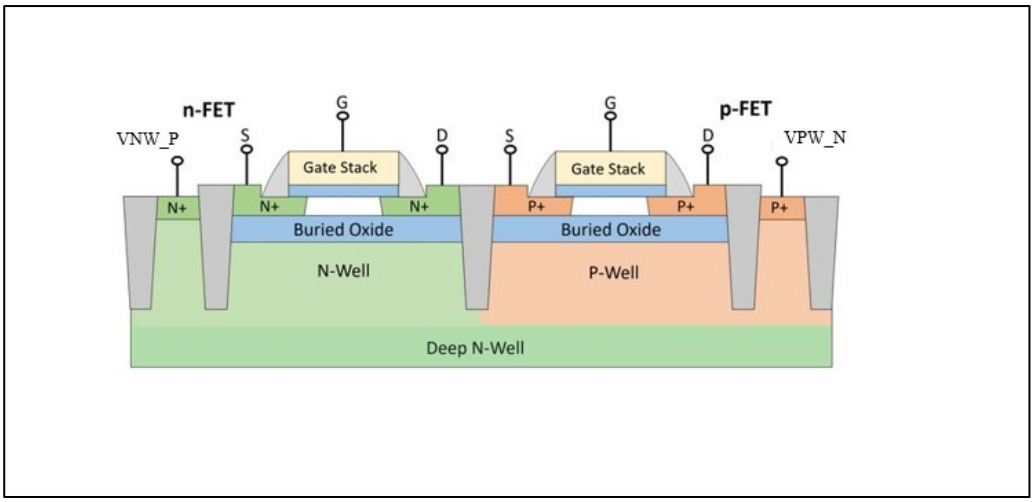

Figure 5.3: GF22 device architecture (adopted from [34])

Although GF22 technology supports up to 11 metal layers, only Metal 7 and Metal 8 were utilized for constructing the power ring and power stripes in this design. These upper metal layers were selected due to their greater width, lower resistance, and higher current-carrying capability, making them ideal for robust power delivery across the entire chip. The remaining metal layers were reserved for signal routing to maintain a clean separation between power and signal networks, which is critical in advanced process nodes. Figure 5.4 illustrates the final placement of standard cells within the core area. The layout reveals a cell density exceeding 85%, indicating highly optimized area utilization.

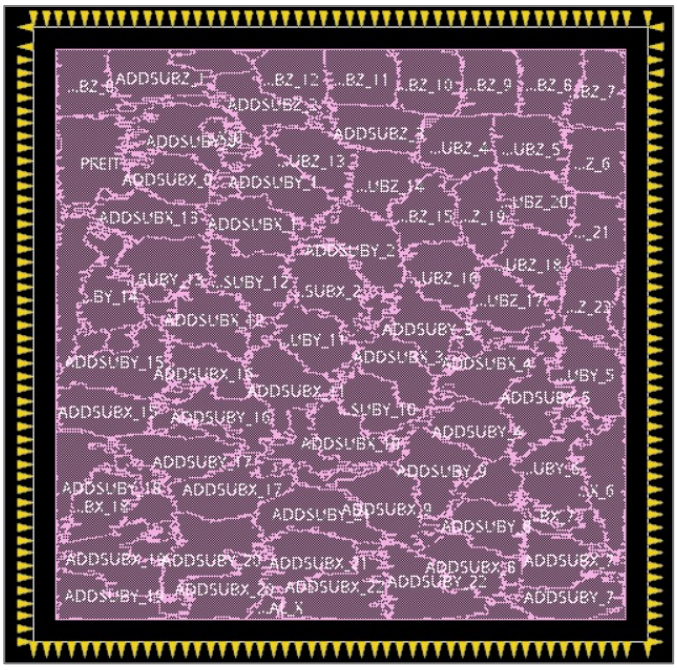

Figure 5.4: Placement of standard cell (Pipeline architecture) using GF22

While this contributes to a more compact chip footprint, it also introduces significant routing challenges due to the limited free space available for interconnects. Compared to the UMC13 implementation, the routing congestion in GF22 was noticeably higher, primarily due to the tighter design rules, reduced routing pitch, and denser cell packing characteristic of more advanced nodes.

Despite these challenges, careful planning—such as customized track spacing, congestion-aware placement strategies, and effective layer utilization—allowed the routing to be completed without any DRC violations or timing failures. Figure 5.5(a) shows the final layout of the pipelined FP-CORDIC design implemented in GF22 technology. The layout demonstrates a compact and efficient physical implementation that adheres to the advanced design rules of the 22nm node. To provide more insight into the critical layout features, Figure 5.5(b) presents a zoomed-in view of the top-left corner of the layout. In this magnified section, key elements such as the metal power ring, signal routing, and IO port connections are clearly visible.

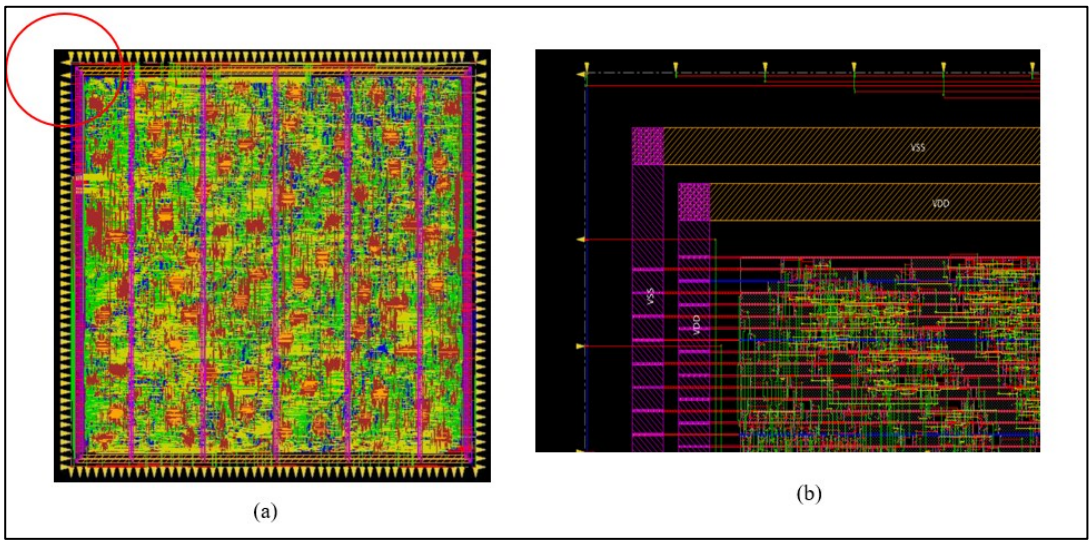

Figure 5.5: (a) Layout Pipeline FP-CORDIC using GF22 (b) zoomed view show power ring and IO-port connection

5.3 Layout Serial architecture of FP-CORDIC

5.3.1 UMC13 implementation

For the serial architecture implementation using UMC13 technology, the layout follows a similar approach to that of the pipelined version in terms of metal layer selection and floor planning strategy. The same design rules, metal stack (primarily using Metal 7 and Metal 8 for power distribution), and placement considerations were applied to maintain consistency and ensure optimal power and signal routing.

Figure 5.6 shows the annotated view of the placed standard cells for the serial FP-CORDIC architecture. As evident in the layout, the Arithmetic Logic Unit (ALU) occupies the majority of the core area. This is expected, as the ALU handles the core iterative computation in the serial design. In contrast, the ROM block, which provides precomputed angle values for the CORDIC iterations, is implemented using combinational Boolean logic rather than a traditional memory structure. As a result, it has been heavily optimized by logic synthesis tools and consumes a relatively small portion of the layout area.

The Normalization Units occupy the second largest area and are strategically placed at the bottom-right side of the layout, close to the output ports. This placement reflects the timing-driven optimization during physical design, as the normalization stage forms the final step in the global normalization and denormalization pipeline of the FP-CORDIC architecture. Positioning this unit near the output ensures reduced interconnect delay and aligns with the natural dataflow, contributing to better overall timing closure.

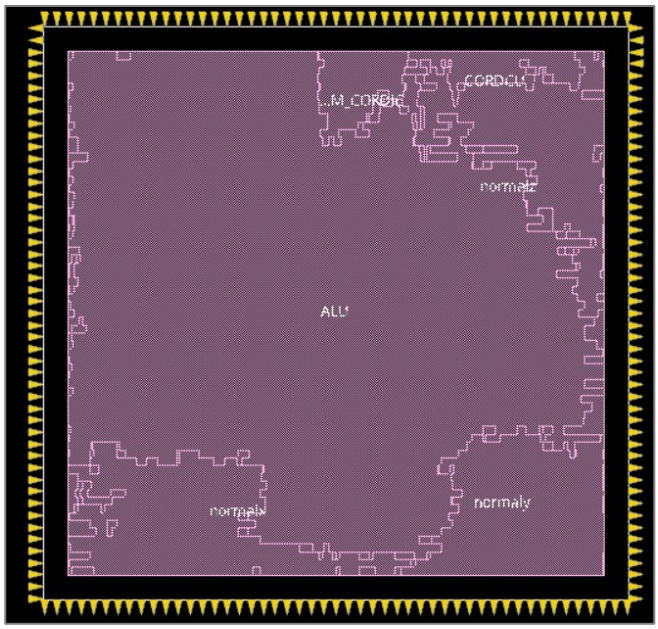

Figure 5.6: Standard cell placement of serial FP-CORDIC using UMC13

The post route layout shown Figure 5.7(a) and Figure 5.7(b) zoomed view of the place highlighted by red circle. It shown that the routing perfectly occupy the offset and pin-power ring gap. Final signoff DRC, connectivity, metal density verified and now violation found Finally, the setup and hold timing report show a positive slack although the value of hold time slack was very small (0.008 ns).

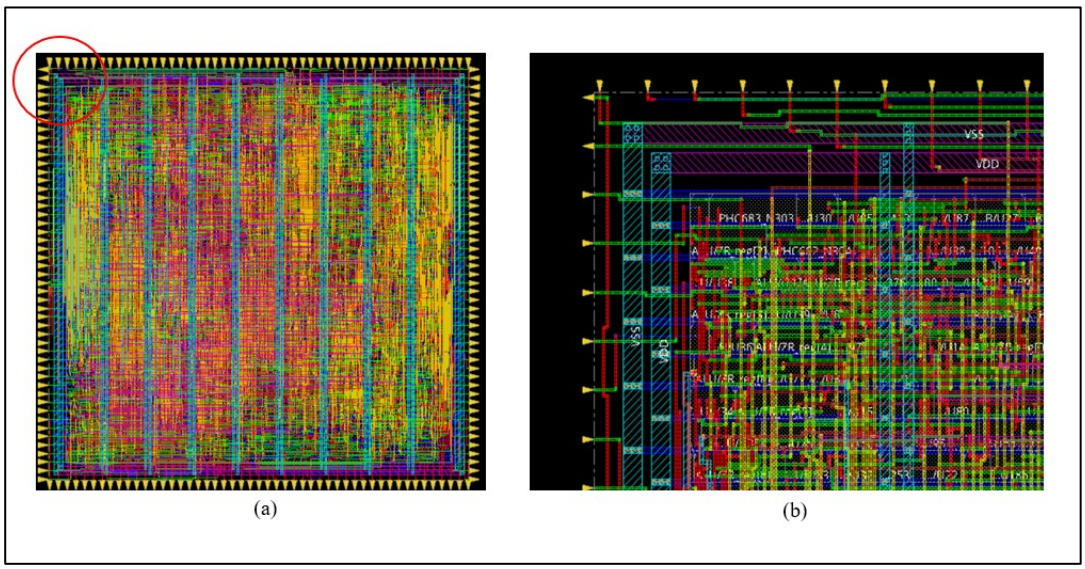

Figure 5.7: UMC13: (a)Physical design of serial architecture (b) Power ring and IO-port route

5.3.2 GF22 implementation

For the GF22 implementation of the serial FP-CORDIC architecture, the same physical design techniques were applied as described for the pipelined version. This includes similar strategies for metal layer utilization, power planning, track configuration, and floorplanning. However, due to the inherently lower complexity and reduced parallelism in the serial architecture, the placement and routing process was comparatively less critical than that of its pipelined counterpart.

Figure 5.8(a) presents the final layout of the serial FP-CORDIC design in GF22 technology. A clear observation from this figure is the lower routing density, which serves as a visual indication of the reduced routing complexity compared to the pipelined design. The serial architecture, having fewer concurrently active modules and simpler dataflow, results in fewer interconnects and relaxed routing constraints.

Figure 5.8(b) illustrates the placed standard cells within the design. Similar to the pipelined architecture, the layout reflects timing-driven placement optimization, with critical modules positioned strategically to minimize data path delays. Specifically, the denormalization unit is placed near the output port, indicating that the place-and-route tool successfully aligned module locations with the dataflow and timing requirements of the design. This careful alignment helps to ensure minimal critical path delay, supports easier timing closure, and reduces routing congestion near timing-sensitive regions.

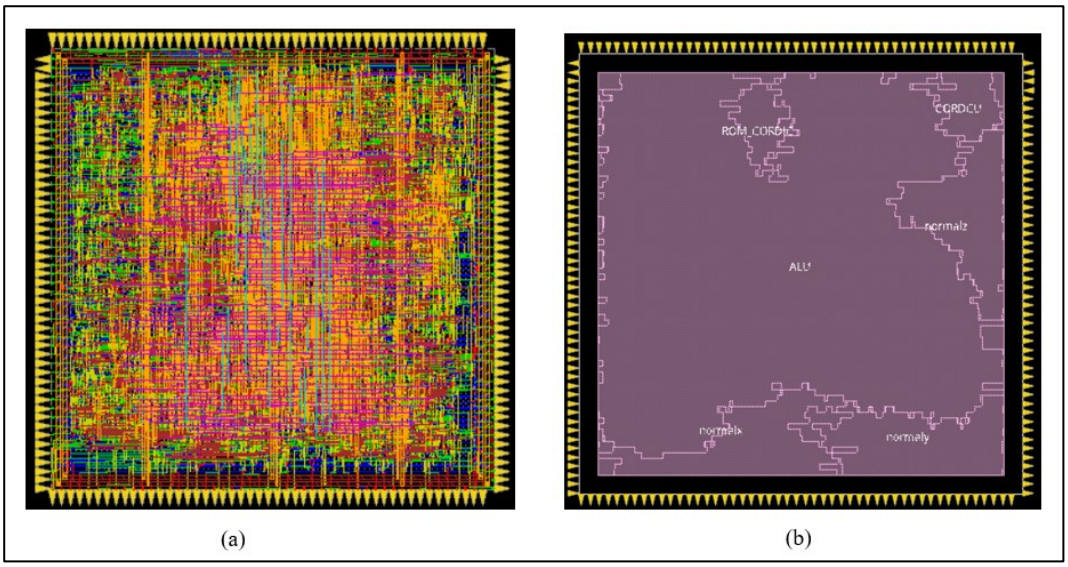

Figure 5.8: GF22: (a) Layout serial FP-CORDIC (b) Standard cell placement

Chapter 6

Conclusions

This thesis presented the design and implementation of a floating-point CORDIC (Coordinate Rotation Digital Computer) architecture optimized for computing sine and cosine functions in rotation mode. Fully **floating-point CORDIC** implementations often face challenges related to latency, speed, and area efficiency—particularly in high-performance, precision-critical applications.

To address these issues, a **pipelined floating-point CORDIC (FP-CORDIC)** architecture was developed with a focus on performance and area optimization. The design incorporates a deeply pipelined floating-point adder and a **global normalization and denormalization scheme**, which together enhance throughput and reduce hardware overhead. Additionally, **carry look-ahead adders (CLA)** were selectively used in key arithmetic stages to reduce critical path delay and improve overall throughput.

The pipelined architecture achieves maximum throughputs of **250 MSamples/s** on **UMC13** and **1.66 GSamples/s** on **GF22** technologies. These high data rates, combined with reduced area (down by **20%** compared to traditional designs) and manageable power usage, make this architecture particularly well-suited **for computationally intensive, real-time applications** such as direct digital synthesizers (DDS) [7], [8], FFT processor [35] and radar systems [36], where both speed and precision are paramount. The total pipeline latency is **122 clock cycles**, with area footprints of **1,035,479 um²** (UMC13) and **47,313.6 um²** (GF22).

For low-power and area-constrained applications such as portable, embedded, and IoT systems, a **serial version** of the FP-CORDIC was also proposed. This architecture uses a **fully combinational floating-point adder** and the same global normalization strategy to reduce latency and improve efficiency. It achieves **24 clock cycles** of latency and maintains an ultra-compact footprint—**32,966.4 um²** (UMC13) and **1,409.7 um²** (GF22)—with very low power consumption: **612.2 uW** and **542.8 uW**, respectively. Despite its simpler structure, the serial architecture delivers improved performance over traditional low-power designs, reaching throughputs of **3.47 MSamples/s** (UMC13) and **17.36 MSamples/s** (GF22). Its minimal size and energy efficiency make it ideal for use in **resource-constrained systems**, including **low-cost digital signal processors and compact IoT devices**.

References

[1] J. E. Volder, “The CORDIC Trigonometric Computing Technique,” *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, 1959, doi: 10.1109/TEC.1959.5222693.

[2] P. Choudhary and A. Karmakar, “CORDIC based implementation of Fast Fourier Transform,” in *2011 2nd International Conference on Computer and Communication Technology, ICCCT-2011*, Mar. 2011, pp. 550–555. doi: 10.1109/ICCCT.2011.6075128.

[3] C. Zhang *et al.*, “The Ultrasound Signal Processing Based on High-Performance CORDIC Algorithm and Radial Artery Imaging Implementation,” *Applied Sciences*, vol. 13, no. 9, 2023, doi: 10.3390/app13095664.

[4] Y.-S. Juang, T.-Y. Sung, L.-T. Ko, and C.-I. Li, “FPGA implementation of a CORDIC-based joint angle processor for a climbing robot,” *Int J Adv Robot Syst*, vol. 10, p. 1, Mar. 2013, doi: 10.5772/53377.

[5] A. Agarwal and B. Lakshmi, “FPGA implementation of digital down converter using CORDIC algorithm,” in *International Conference on Communication and Electronics System Design*, V. Janyani, M. Salim, and K. K. Sharma, Eds., SPIE, 2013, p. 87601K. doi: 10.1117/12.2012307.

[6] A. Paliwal, B. Mohindroo, and K. Suneja, “Hardware Design of Image Encryption and Decryption Using CORDIC Based Chaotic Generator,” in *2020 5th IEEE International Conference on Recent Advances and Innovations in Engineering (ICRAIE)*, Dec. 2020, pp. 1–5. doi: 10.1109/ICRAIE51050.2020.9358354.

[7] P. J. Katkar and Y. S. Angal, “Realization of cordic algorithm in DDS: Novel Approch towards Digital Modulators in MATLAB and VHDL,” in *2015 International Conference on Information Processing (ICIP)*, Dec. 2015, pp. 355–359. doi: 10.1109/INFOP.2015.7489407.

[8] S. Ma, X. Wang, Y. Li, K. Long, B. Zhu, and X. Lei, “A Low Complexity DDS Based On Optimized CORDIC Algorithm,” in *2019 IEEE 13th International Conference on ASIC (ASICON)*, Oct. 2019, pp. 1–5. doi: 10.1109/ASICON47005.2019.8983676.

[9] “IEEE Standard for Binary Floating-Point Arithmetic,” *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985, doi: 10.1109/IEEESTD.1985.82928.

[10] “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019, doi: 10.1109/IEEESTD.2019.8766229.

[11] N. Prashar, “FPGA Implementation of Pipelined CORDIC Sine Cosine Digital Wave Generator,” in *Computer Science & Information Technology*, Mar. 2012, pp. 435–440. doi: 10.5121/csit.2012.2243.

[12] G. J. Hekstra and E. F. A. Deprettere, “Floating point Cordic,” in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, 1993, pp. 130–137. doi: 10.1109/ARITH.1993.378100.

[13] R. Andraka, “A survey of CORDIC algorithms for FPGA based computers,” in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, in FPGA ’98. New York, NY, USA: Association for Computing Machinery, 1998, pp. 191–200. doi: 10.1145/275107.275139.

[14] V. V M, A. K. A, and B. S, “Implementation of Floating Point CORDIC Algorithm Using 45 nm Technology For COS SINE Generation,” in *2023 International Conference on Next Generation Electronics (NEleX)*, 2023, pp. 1–5. doi: 10.1109/NEleX59773.2023.10420961.

[15] K. Sharma, K. Apoorva, U. Sharrma, and P. K S, “A Review on Design and Application of CORDIC algorithm,” *IOSR Journal of VLSI and Signal processing*, vol. 10, pp. 1–8, Mar. 2020, doi: 10.9790/4200-10010108.

[16] “A Review Paper on CORDIC Algorithm and Its Applications for Current Technology,” *International Journal of Science and Research (IJSR)*, vol. 5, p. 770, Mar. 2016, doi: 10.21275/v5i6.NOV164208.

[17] K. Raj, R. Tomar, and P. Singh, “A Review of CORDIC Algorithms and Architectures with Applications for Efficient Designing,” *Int J Sci Eng Res*, vol. 4, pp. 236–246, Mar. 2013.

[18] N. Prashar, “FPGA Implementation of Pipelined CORDIC Sine Cosine Digital Wave Generator,” in *Computer Science & Information Technology*, Mar. 2012, pp. 435–440. doi: 10.5121/csit.2012.2243.

[19] A. Puli, “FPGA implementation of the trigonometric function using the CORDIC Algorithm,” Mar. 2019. doi: 10.1109/ICACCS.2019.8728315.

[20] J. Sudha, M. C. Hanumantharaju, V. Venkateswarulu, and J. H, “A Novel Method for Computing Exponential Function Using CORDIC Algorithm,” *Procedia Eng*, vol. 30, pp. 519–528, 2012, doi: https://doi.org/10.1016/j.proeng.2012.01.893.

[21] H. Chen, K. Cheng, Z. Lu, Y. Fu, and L. Li, “Hyperbolic CORDIC-Based Architecture for Computing Logarithm and Its Implementation,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 11, pp. 2652–2656, Nov. 2020, doi: 10.1109/TCSII.2020.2971974.

[22] R. G. Kumar and K. Padmakumar, “A VLSI Implementation of Floating Point CORDIC Coprocessor,” *International journal of engineering research and technology*, vol. 3, 2014, [Online]. Available: https://api.semanticscholar.org/CorpusID:60774306

[23] H. Dawid and H. Meyr, “The differential CORDIC algorithm: Constant scale factor redundant implementation without correcting iterations,” *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 307–318, Mar. 1996, doi: 10.1109/12.485569.

[24] H. Dawid and H. Meyr, “VLSI implementation of the CORDIC algorithm using redundant arithmetic,” in *[Proceedings] 1992 IEEE International Symposium on Circuits and Systems*, May 1992, pp. 1089–1092 vol.3. doi: 10.1109/ISCAS.1992.230290.

[25] B. Zhu, Y. Lei, Y. Peng, and T. He, “Low Latency and Low Error Floating-Point Sine/Cosine Function Based TCORDIC Algorithm,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 4, pp. 892–905, Apr. 2017, doi: 10.1109/TCSI.2016.2631588.

[26] J. Rudagi and S. Subbaraman, “Comparative analysis of radix-2, radix-4, radix-8 CORDIC processors,” in *2017 International Conference on Inventive Computing and Informatics (ICICI)*, Nov. 2017, pp. 378–382. doi: 10.1109/ICICI.2017.8365377.

[27] A. K. Singh, M. K. Singh, and K. C. Ray, “Design and Implementation of Quadruple Floating-Point CORDIC,” in *2015 IEEE International Symposium on Nanoelectronic and Information Systems*, Dec. 2015, pp. 286–290. doi: 10.1109/iNIS.2015.23.

[28] M. Nachtigal, H. Thapliyal, and N. Ranganathan, “Design of a reversible floating-point adder architecture,” in *2011 11th IEEE International Conference on Nanotechnology*, Aug. 2011, pp. 451–456. doi: 10.1109/NANO.2011.6144358.

[29] A. Nannarelli, “Tunable Floating-Point Adder,” *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1553–1560, Oct. 2019, doi: 10.1109/TC.2019.2906907.

[30] Louca, Cook, and Johnson, “Implementation of IEEE single precision floating point addition and multiplication on FPGAs,” in *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1996, pp. 107–116. doi: 10.1109/FPGA.1996.564761.

[31] K. Saranya, B. Saravanan, V. Senthil, S. Kumar, and P. Kumar, “An energy efficient approximate multiplier for low power applications,” in *AIP Conference Proceedings*, Mar. 2023, p. 40001. doi: 10.1063/5.0125144.

[32] B. Jovanovic and M. Jevtić, “Optimization of the Binary Adder Architectures Implemented in ASICs and FPGAs,” vol. 195, pp. 295–308, Jun. 2013, doi: 10.1007/978-3-642-33941-7-27.

[33] P. Balasubramanian and N. Mastorakis, “High-Speed and Energy-Efficient Carry Look-Ahead Adder,” *Journal of Low Power Electronics and Applications*, vol. 12, p. 46, Jun. 2022, doi: 10.3390/jlpea12030046.

[34] Q. H. Le *et al.*, “W-Band Noise Characterization with Back-Gate Effects for Advanced 22nm FDSOI mm-Wave MOSFETs,” Jul. 2020. doi: 10.1109/RFIC49505.2020.9218369.

[35] A. Mukil, N. Ashokkumar, N. S. Lai, and R. Lakshmanan, “Cordic-based, high-performance, and resource-efficient FFT processor: Speech processing application,” *AIP Conf Proc*, vol. 3161, no. 1, p. 20147, Jul. 2024, doi: 10.1063/5.0229435.

[36] P. Sivaprasad, V. Anandi, and S. Murthy, “Highly refined phase estimation and design model for next-generation RADAR applications,” *AIP Conf Proc*, vol. 3162, no. 1, p. 20087, Jul. 2025, doi: 10.1063/5.0243342.