Full Custom design of DIT and DFT.

BECE303L VLSI System Design Project

Submitted by

Hariharan S - 22BEC1028

Nikhil Rout - 22BEC1020

Jayant S - 22BEC1053

Rohan G - 22BEC1129

Surya T - 22BEC1098

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

VIT CHENNAI

SENSE

B. Tech (Electronics and Communication Engineering)

Nov 2024

ABSTRACT

The project focuses on the full custom design and development of Decimation-In-Time (DIT) and Decimation-In-Frequency (DIF) Fast Fourier Transform (FFT) architectures, tailored for high-performance VLSI applications. FFT is a cornerstone of signal processing, and efficient implementations are crucial for minimizing computational complexity while maintaining accuracy. This work adopts a modular design methodology to construct critical components, including complex address, subtractors, and multipliers, to build the FFT butterfly structures central to both DIT and DIF architectures.

The design incorporates Vedic multipliers of various dimensions (4x4, 8x8, and 16x16) alongside ripple carry adders (16-bit and 24-bit) to achieve high precision and operational efficiency. Additionally, buffer elements (4-bit and 8-bit) are employed to manage data flow seamlessly across the architecture. The DIT and DIF FFT designs are validated through custom-built testbenches for 8-point FFT configurations. These testbenches simulate the functional behavior of the architectures and are augmented with a MATLAB-based FFT verification framework to ensure numerical accuracy.

A Python script is utilized to convert complex 32-bit binary inputs into analog values, aligning the digital designs with real-world signal processing applications. The project also integrates hexadecimal representations to simplify internal data representation and processing. Simulation results showcase the successful implementation of the complete DIT and DIF FFT architectures, demonstrating accurate computation of FFT twiddle factors and output values (Xk0 to Xk7).

This study underscores the practical realization of FFT architectures in a full-custom design flow, offering significant contributions toward efficient computation in digital signal processing and paving the way for further optimizations in area, power, and speed for large-scale FFT implementations.

Contents

I. Introduction and Literature Review

The **Fast Fourier Transform (FFT)** is one of the most important algorithms in digital signal processing (DSP). It efficiently computes the Discrete Fourier Transform (DFT) and its inverse, transforming signals from the time domain to the frequency domain. This process is fundamental in modern technologies, including telecommunications, multimedia systems, radar, biomedical signal processing, and audio and image compression. Unlike the direct computation of DFT, which requires *O(N^2)* operations, FFT significantly reduces the computational complexity to *O(Nlog₂N)*, making it a cornerstone of efficient signal processing.

FFT algorithms, particularly **Decimation-In-Time (DIT)** and **Decimation-In-Frequency (DIF)**, are well-suited for **modular VLSI architectures** due to their recursive nature. These architectures enable scalability and parallelism, which are essential for handling high data throughput and reducing latency in real-time systems. While general-purpose processors and digital signal processors (DSPs) are widely used for FFT implementations, they often face challenges such as high power consumption and large area requirements for custom applications. This drives the need for **full custom VLSI design** approaches, which optimize FFT architectures at the transistor and layout levels, ensuring high performance and low power consumption.

The project focuses on the **design and implementation of DIT and DIF FFT architectures** through a full custom approach. It emphasizes the development of critical components such as **complex adders, subtractors, and multipliers**, which form the backbone of FFT butterfly structures. These designs incorporate **Vedic multipliers** for fast and efficient computation and **ripple carry adders** for enhanced precision. The use of a **fixed-point [8.8] representation** ensures numerical accuracy while minimizing hardware complexity, allowing for seamless integration into VLSI systems.

Several key techniques and methodologies are explored to optimize FFT architectures for VLSI implementation. **Modular design** allows for the reuse of components, reducing design complexity and enabling scalability for higher-order FFTs. The incorporation of **buffer elements** ensures efficient data flow and synchronization across the architecture. Additionally, this project employs **hexadecimal representation** to streamline internal data processing and improve computational efficiency.

The FFT algorithm has been extensively studied and implemented in various domains, leading to numerous advancements in its architecture and optimization techniques.

Cooley and Tukey's groundbreaking paper in 1965 introduced the divide-and-conquer approach for computing FFT, laying the foundation for modern signal processing [2]. Since then, research has focused on improving the efficiency and scalability of FFT implementations, particularly in the context of hardware design.

Custom FFT Architectures

Custom FFT architectures are critical for applications requiring high-speed and low-power processing. Recent studies have explored **pipeline architectures**, which leverage parallelism to achieve high throughput. For instance, Xu and Chen [9] proposed a radix-4 FFT processor that uses pipelined stages to enhance processing speed while minimizing area overhead. Similarly, Ansari [10] investigated low-power FFT designs for embedded systems, demonstrating the importance of resource sharing and optimized multiplier designs.

Multipliers and Adders in FFT

The performance of FFT architectures is heavily influenced by the efficiency of its arithmetic components. Vedic multipliers, known for their fast computation and low power consumption, have gained significant attention in recent years. Lakshmi and Shruthi [6] highlighted the use of Vedic multipliers in FFT butterfly units, demonstrating a reduction in computational delay. Ripple carry adders, although slower than other adder architectures, provide high precision and are suitable for applications where accuracy is paramount.

Data Representation

The choice of data representation significantly impacts the accuracy and hardware efficiency of FFT architectures. Fixed-point formats, such as the [8.8] representation used in this project, are widely adopted due to their simplicity and suitability for hardware implementation. Rabaey et al. [7] discussed the trade-offs between fixed-point and floating-point representations in digital systems, emphasizing the benefits of fixed-point formats for power-efficient designs.

Testing and Validation

Testing and validation are critical in the development of FFT architectures. MATLAB has long been a preferred tool for verifying the functional correctness of FFT implementations due to its robust computational capabilities and extensive library support [11]. Python, with its versatility and growing adoption in DSP applications, offers an efficient platform for preprocessing and converting data formats. This project employs a Python script to convert complex binary inputs into analog values, bridging the gap between digital designs and real-world applications.

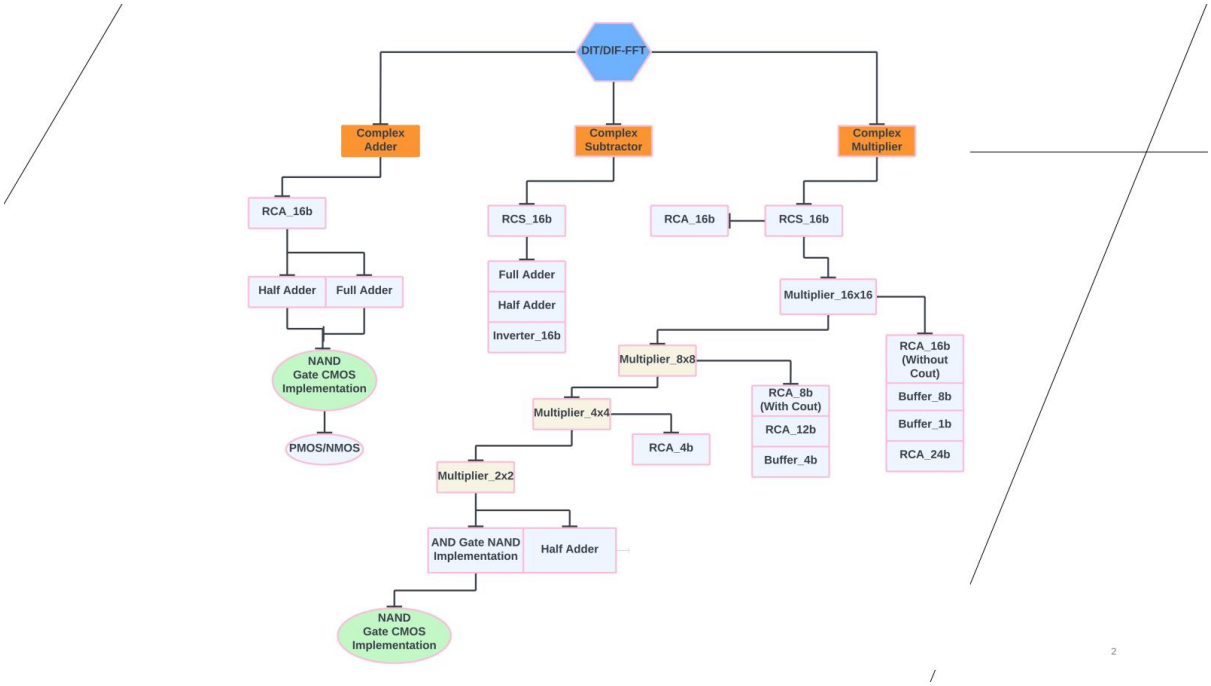
Scalability and Future Directions

While this project focuses on 8-point FFT architectures, scalability remains a key area of research. Techniques such as hybrid radix algorithms and hierarchical designs have been explored to address the challenges of implementing large-scale FFTs. Oppenheim and Schafer [8] discussed the theoretical foundations of hybrid radix designs, providing insights into their potential for improving computational efficiency. As VLSI technology continues to advance, integrating FFT architectures with emerging technologies such as quantum computing and machine learning is expected to open new avenues for research.

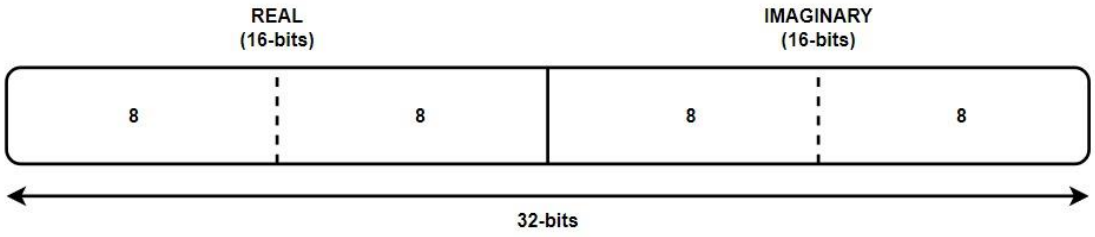
By building on these advancements, this project aims to demonstrate the practical realization of DIT and DIF FFT architectures in a full custom design flow. The integration of advanced multiplier and adder designs, combined with modular and hierarchical testing methodologies, highlights its potential impact on high-performance VLSI applications.

II. Methodology for Design of FFT Architectures

Implementing the FFT architecture in hardware requires the design of multiple hierarchies of arithmetic modules. These modules are eventually instantiated to perform the complete FFT computation.



2.1 Custom Complex Number Format Representation



32-bit Complex Number Representation (Fixed-Point Format):

- The complex number consists of **two 16-bit parts**: one for the **real part** and one for the **imaginary part**.
- Each part (real and imaginary) is represented in an **[8.8] fixed-point format**:
  - **8 bits** for the **whole number (integer)** part.

- **8 bits** for the **fractional** part.

Structure of Each Part:

Each 16-bit part (either real or imaginary) is divided as follows:

Real Part (16 bits):

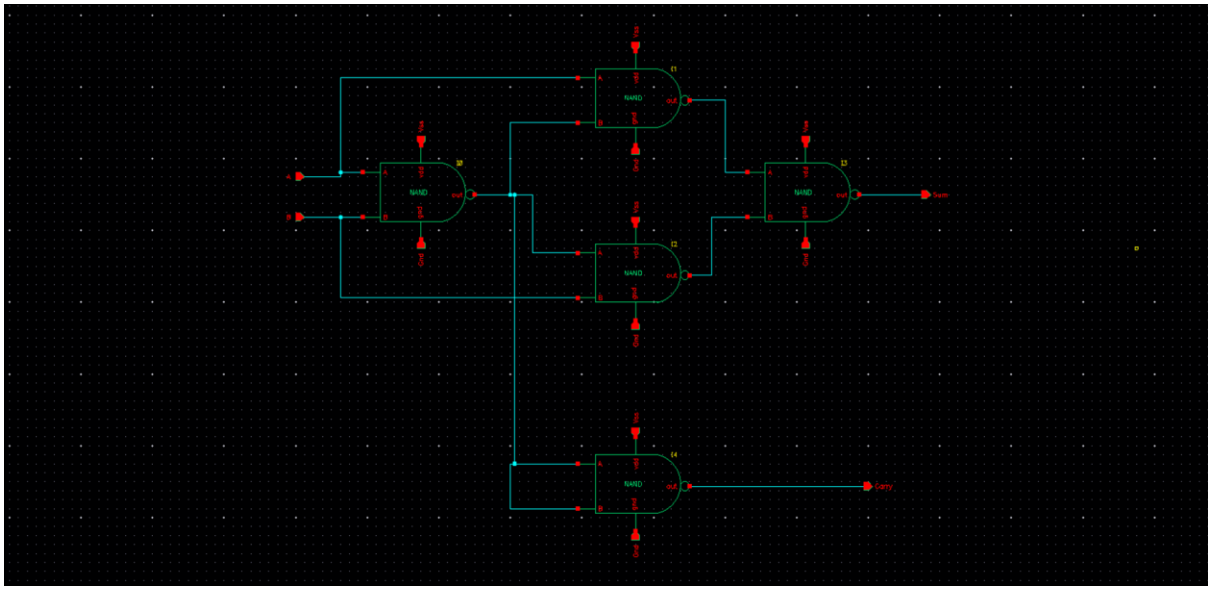
- **Bits 0-7 (Whole Number)**: The first 8 bits represent the integer part of the real number.
  - The **most significant bit (MSB)** of this 8-bit section (Bit 7) is the **signed bit**, indicating whether the real number is positive or negative.
  - The remaining 7 bits (Bits 0-6) represent the magnitude of the integer part.
- **Bits 8-15 (Fractional Part)**: The next 8 bits represent the fractional part of the real number.
  - These bits represent values between -1 and 1, with **Bit 15** being the signed bit for the fractional part.

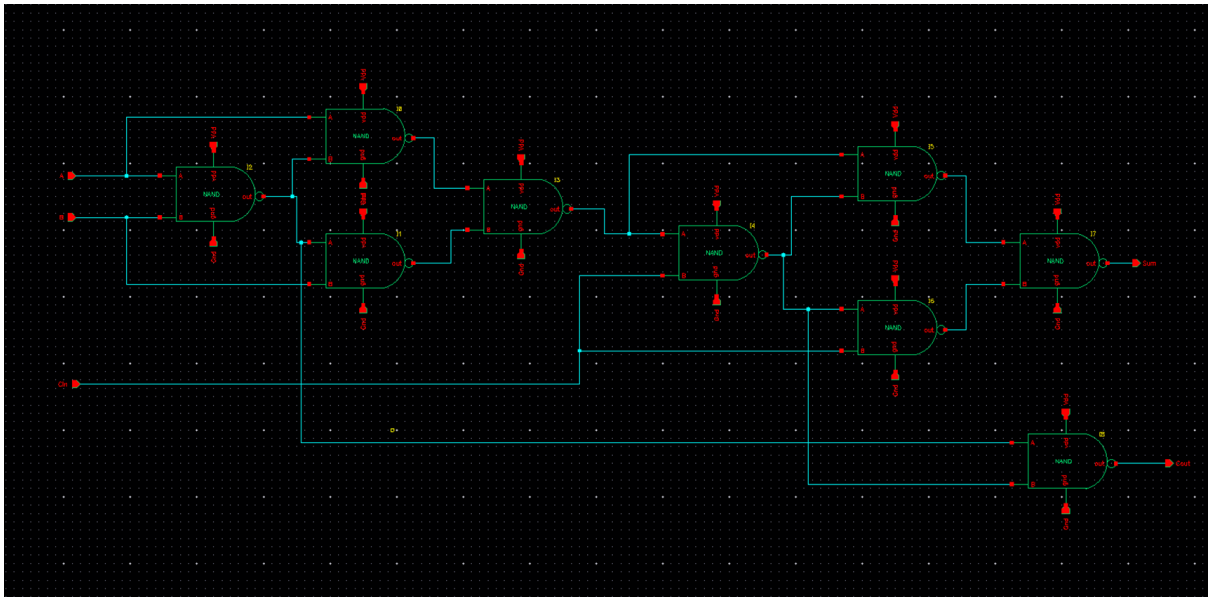
Imaginary Part (16 bits):

- **Bits 16-23 (Whole Number)**: The first 8 bits represent the integer part of the imaginary number.
  - Similar to the real part, the **most significant bit (MSB)** (Bit 23) is the **signed bit**, indicating the sign of the integer part.
  - The remaining 7 bits (Bits 16-22) represent the magnitude of the integer part.
- **Bits 24-31 (Fractional Part)**: The next 8 bits represent the fractional part of the imaginary number.
  - These bits represent fractional values between -1 and 1, with **Bit 31** being the signed bit for the fractional part.

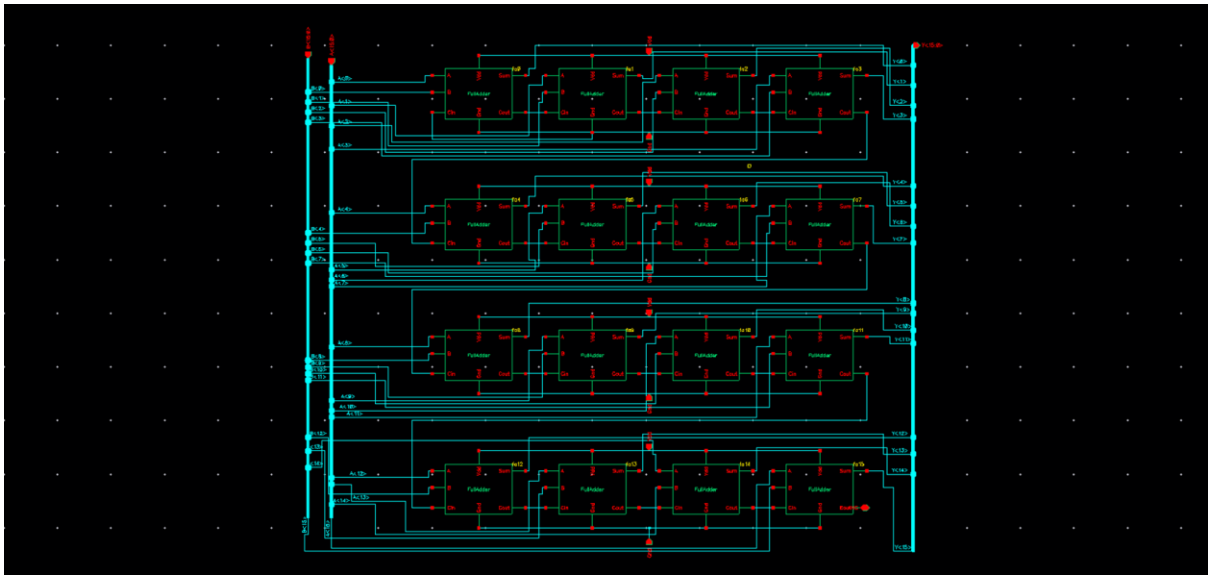
2.2 Complex Adder

Half and Full Adder: The design of half and full adders using only NAND gates exploits the NAND gate's versatility as a universal logic gate. A half adder, which computes the sum and carry of two binary inputs, is implemented by deriving the XOR function for the sum and an AND function for the carry using combinations of NAND gates. Similarly, the full adder extends this principle by combining two half adders with additional logic to handle the carry-in. The XOR, AND, and OR operations required in a full adder are synthesized using optimized arrangements of NAND gates. This approach highlights the NAND gate's ability to replicate any logic function, making it foundational in custom digital circuit design.

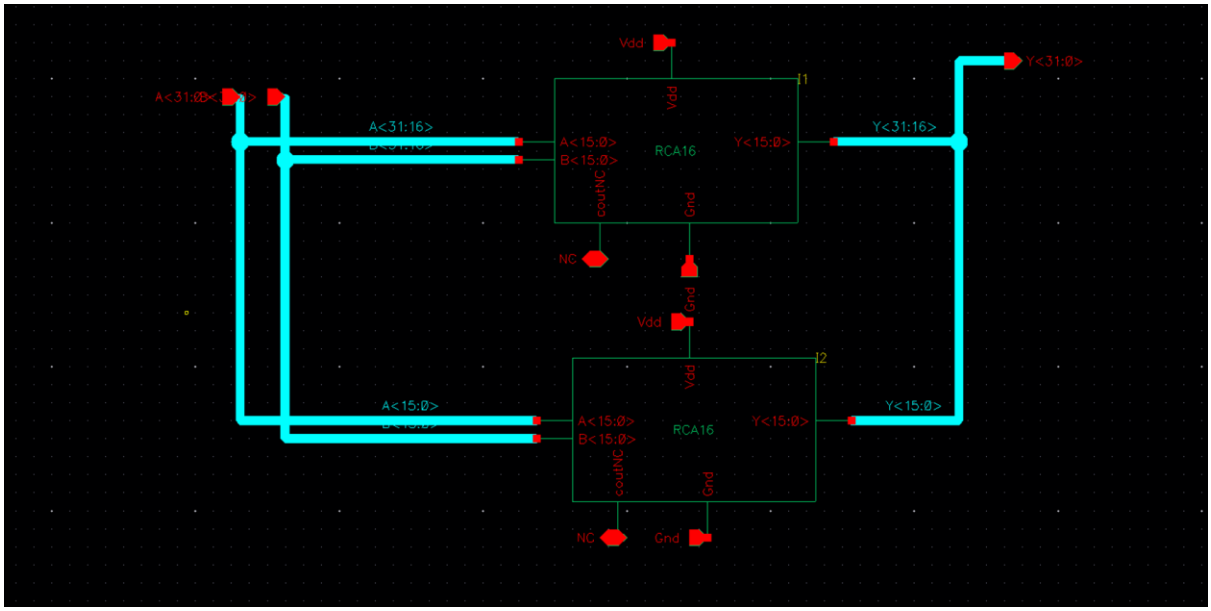




16-bit Ripple Carry Adder: A 16-bit ripple carry adder (RCA) can be constructed by cascading 16 single-bit adders, starting with a half adder for the least significant bit (LSB) and full adders for the remaining bits. The half adder processes the two least significant input bits (A_0 and B_0), generating the initial sum (S_0) and carry-out (C_0). For the subsequent bits, each full adder takes three inputs: the corresponding bits from the operands (A_i and B_i) and the carry-out (C_{i-1}) from the previous stage. Each full adder generates a sum bit (S_i) and propagates a carry-out (C_i) to the next stage. The design operates sequentially, with the carry-out rippling through the chain, which introduces a delay proportional to the number of bits due to the linear dependency of the carry propagation path. By leveraging previously designed half and full adders, the RCA architecture emphasizes modularity and scalability, albeit with performance limited by the cumulative propagation delay.

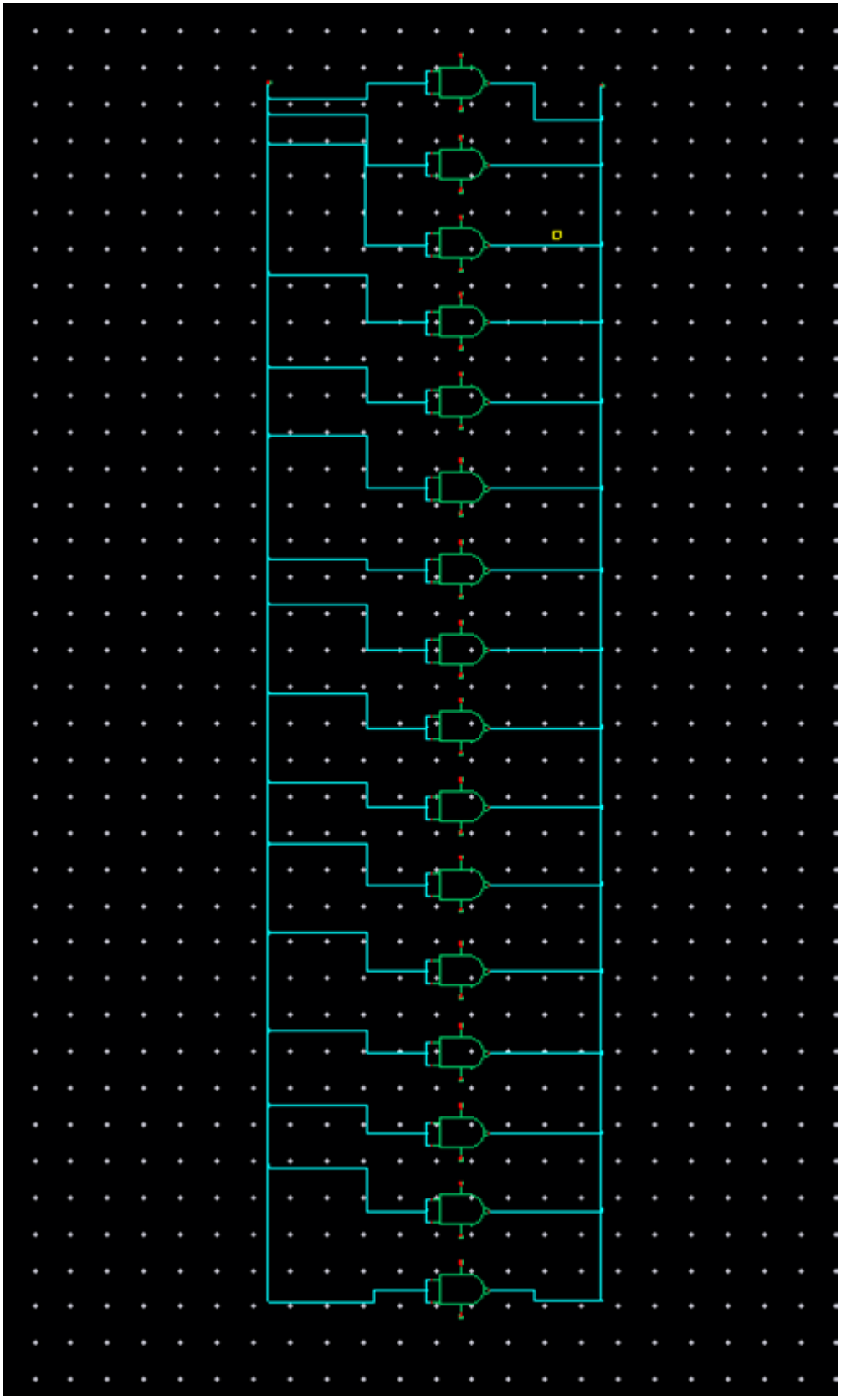


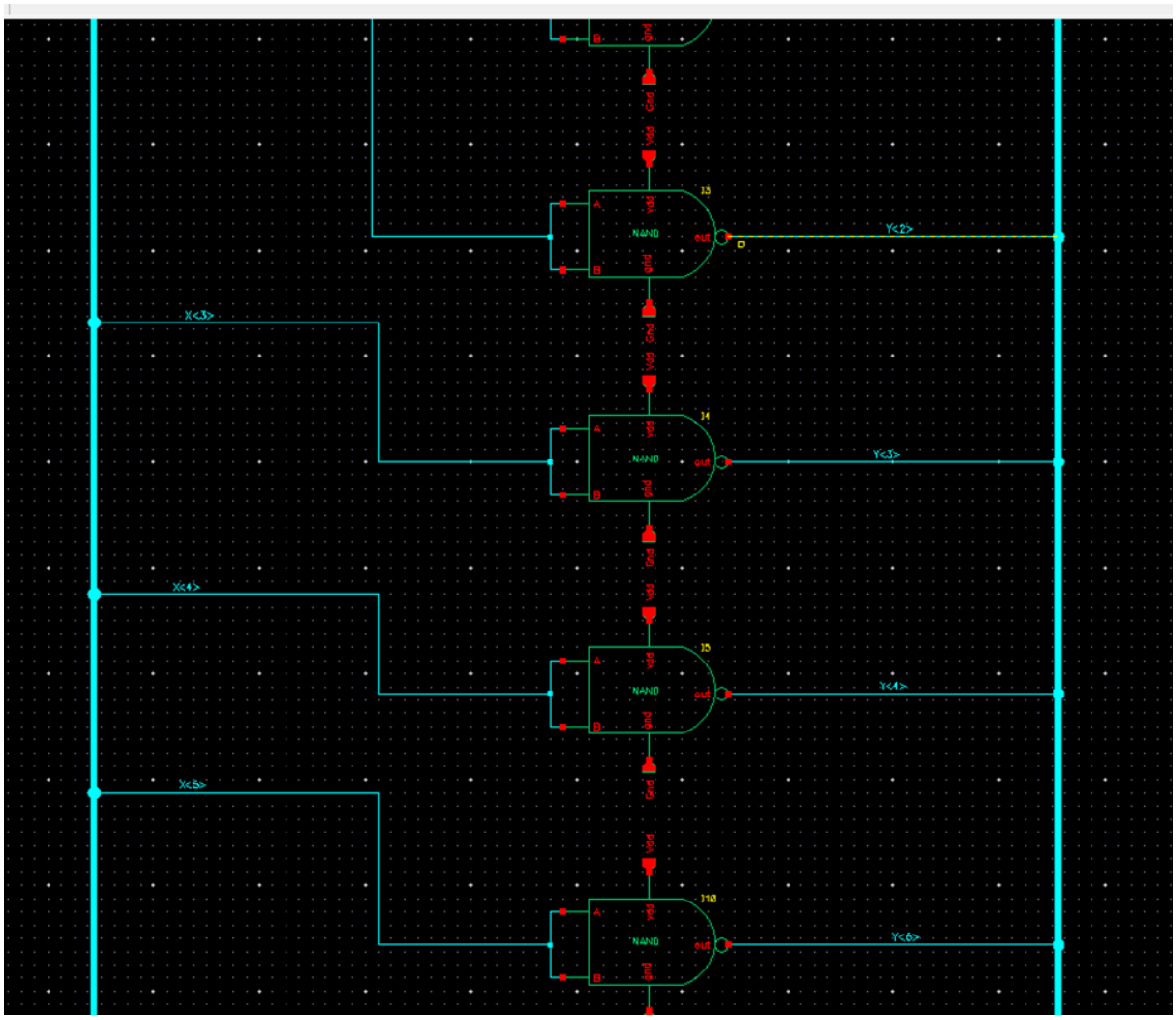
Complex Adder: The Complex Adder block in the FFT architecture leverages the custom-designed 16-bit Ripple Carry Adder (RCA) to efficiently compute the addition of two complex numbers. Each 32-bit input is split into its real and imaginary parts: the most significant 16 bits (31:16) represent the real component, while the least significant 16 bits (15:0) represent the imaginary component. For addition, the real and imaginary parts are processed separately using two independent instances of the 16-bit RCA. The real components of the two inputs are fed into one RCA, while the imaginary components are input to another, ensuring accurate computation of both real and imaginary sums. The outputs of these RCAs are then concatenated to form the final 32-bit result. This modular design, rooted in the custom 16-bit RCA, ensures scalability and minimizes propagation delay for each complex addition operation.



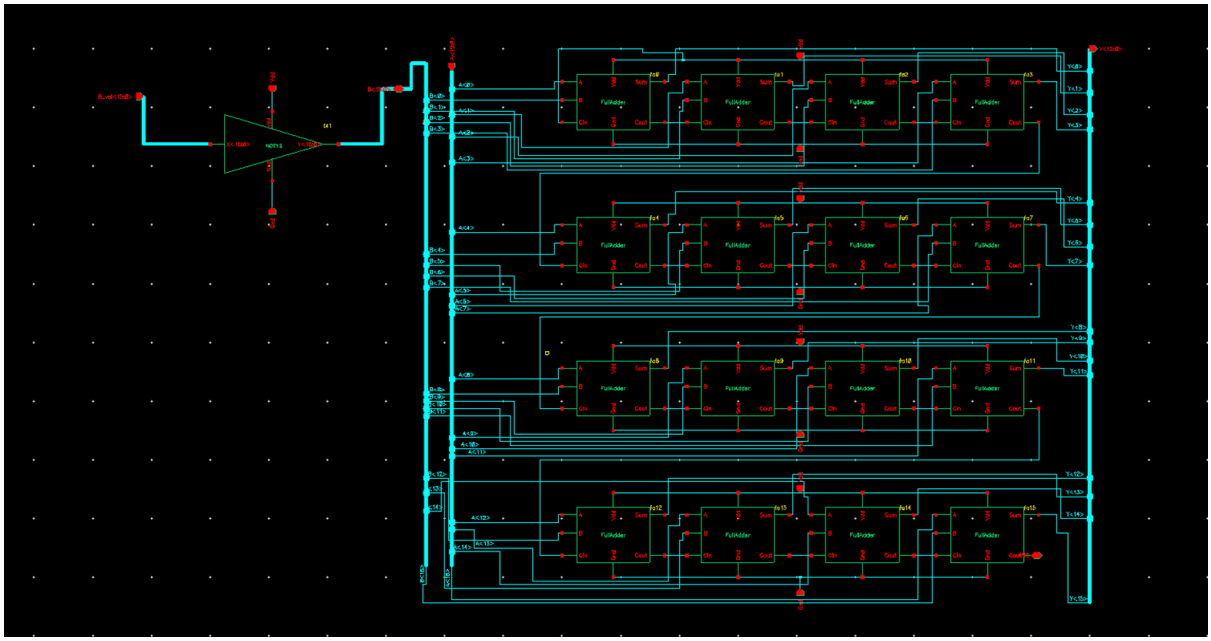
2.3 Complex Subtractor

16-bit NOT: The design of a 16-bit NOT operation involves the bitwise inversion of each individual bit within a 16-bit input bus. The 16-bit bus is logically decomposed into 16 single-bit signals, each routed to a dedicated NOT gate (inverter). The NOT gates invert the logic levels of their respective inputs, transforming a logic ‘1’ to ‘0’ and vice versa. Once each bit has been inverted, the 16 resulting single-bit outputs are concatenated back into a 16-bit output bus to form the final result. This parallelized design ensures that all bits are inverted simultaneously, leveraging the inherent parallelism of digital logic to minimize the propagation delay. The circuit is compact and efficient, as it relies solely on 16 standard inverters, making it ideal for high-speed and low-area applications in digital systems.

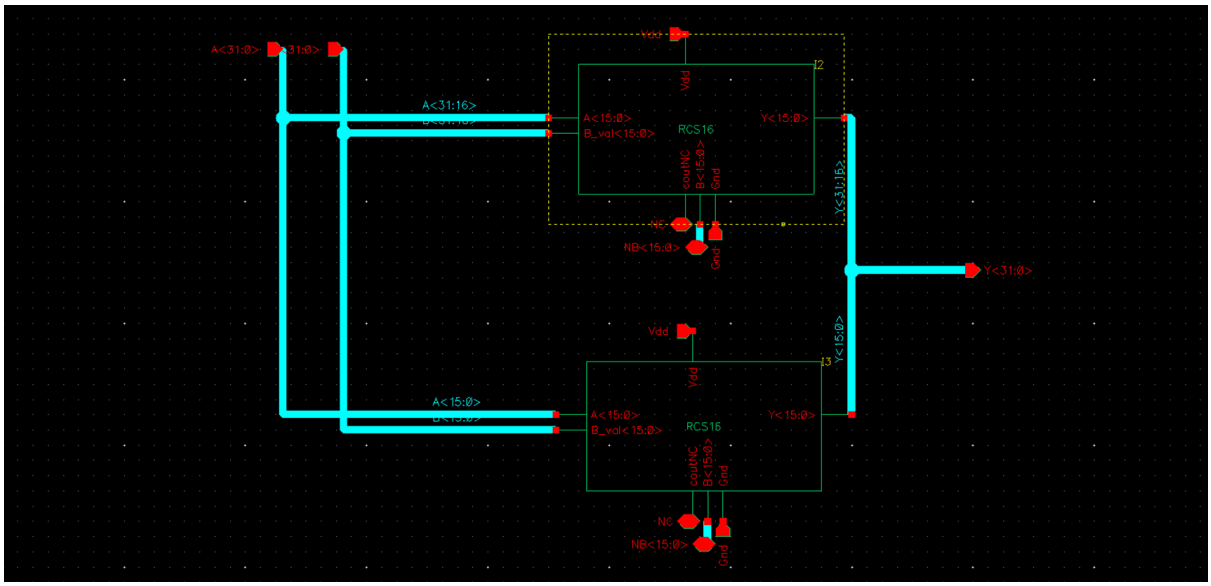




16-bit Subtractor: The design of a 16-bit ripple carry subtractor utilizes the 16-bit NOT gate and the previously designed 16-bit Ripple Carry Adder (RCA) to compute the subtraction of two operands A and B as A - B. In this design, the 16-bit input bus B is first passed through the 16-bit NOT gate, which performs a bitwise inversion of B to produce ~B. This result is equivalent to the 1's complement of B. To complete the subtraction, ~B is then fed as one operand to the 16-bit RCA along with the other operand A, and a carry-in of 1 is provided to perform the 2's complement addition, effectively implementing A + (~B + 1). The 16-bit RCA processes the addition and propagates the carry sequentially across all bits, generating the final 16-bit result. This design achieves efficient subtraction using modular components, leveraging the NOT gate for operand inversion and the RCA for addition, while ensuring scalability and reusability in larger arithmetic systems.



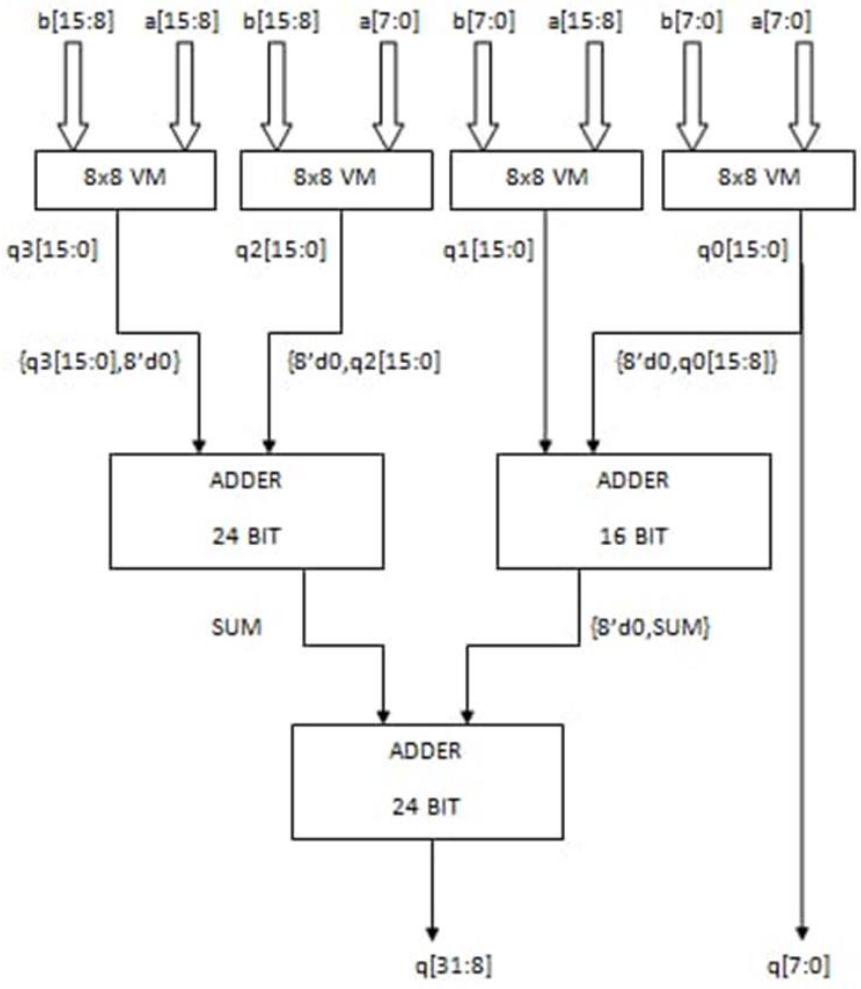
Complex Subtractor: This block is designed to compute the subtraction of two complex numbers by independently processing their real and imaginary components. Each 32-bit input is split into its real part (bits 31:16) and imaginary part (bits 15:0). The real components of the two inputs are fed into one 16-bit ripple-carry subtractor, while the imaginary components are fed into another. The subtractor internally uses a 16-bit NOT gate to invert one operand and a 16-bit Ripple Carry Adder (RCA) with a carry-in of 1 to perform the 2's complement subtraction. The outputs from the subtractors represent the real and imaginary differences, which are concatenated to form the final 32-bit complex difference. This design modularizes the operation, ensuring efficient and scalable subtraction for complex-valued computations.



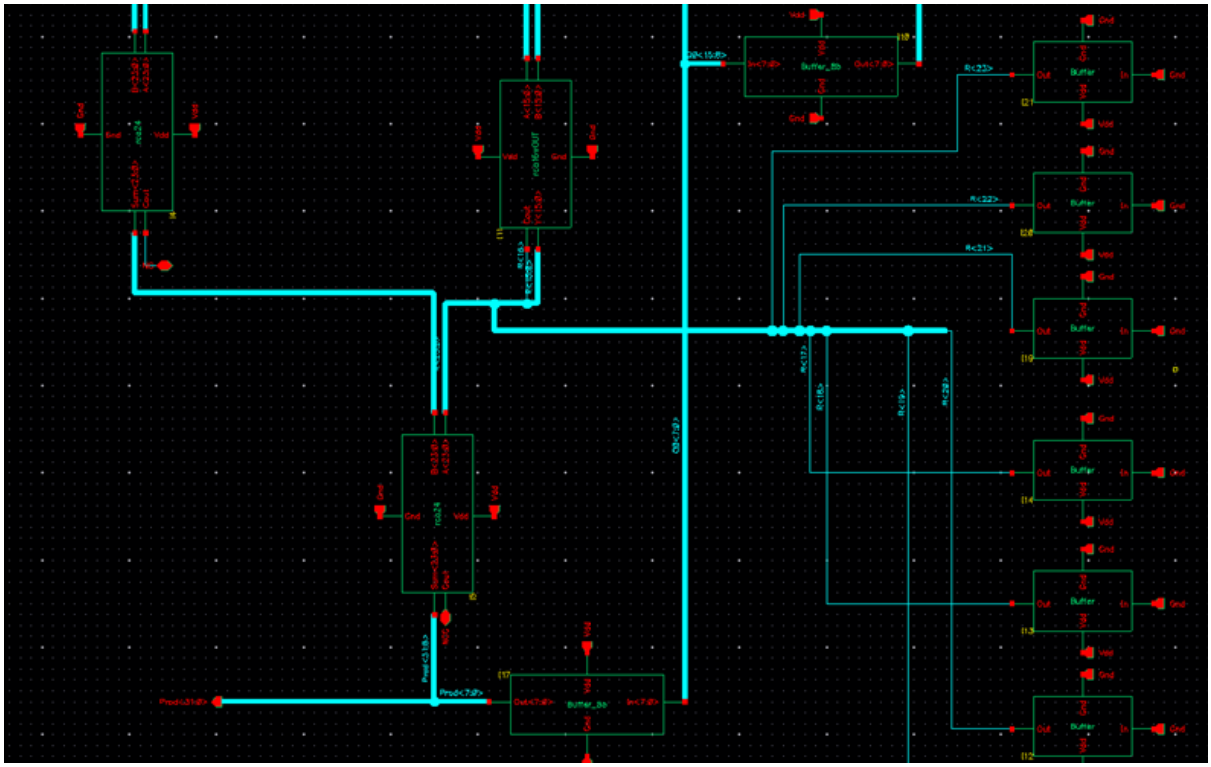
2.4 Complex Multiplier

The 16-bit Vedic multiplier: This is designed using a hierarchical approach, building upon smaller multiplier blocks through recursive composition. The implementation follows the Urdhva Tiryakbhyam (vertical and crosswise) Vedic multiplication sutra, which enables efficient multiplication through column-wise and cross-wise calculation. Specifically, the 16-bit multiplier is constructed by combining four 8x8 bit multipliers with additional 12-bit and 24-bit adders to aggregate the partial products.

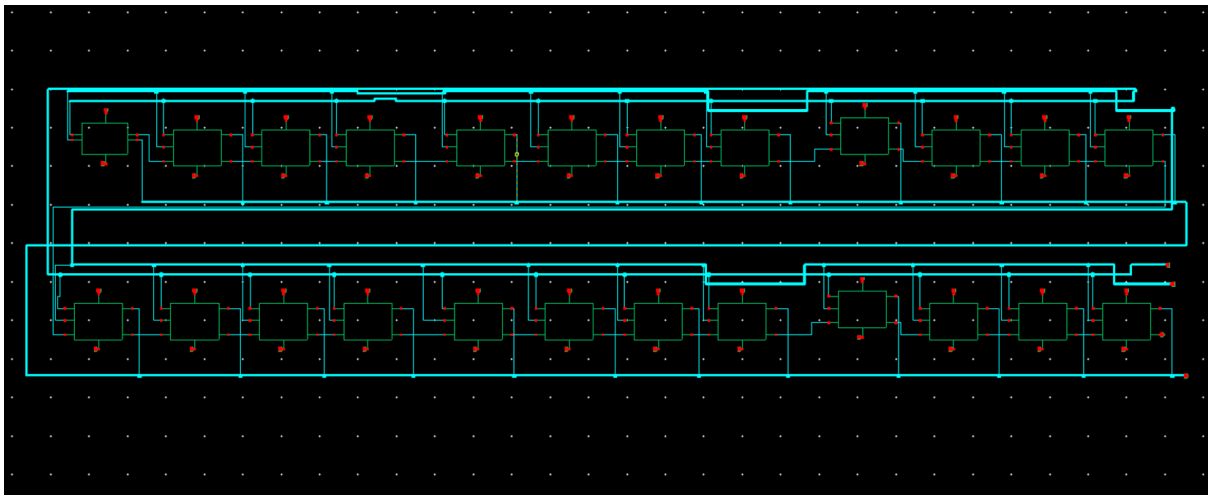
The design progression involves first creating a 2x2 Vedic multiplier using four NAND gates and two half adders, then scaling up to a 4x4 multiplier by employing four 2x2 multipliers and three adders. The 8x8 multiplier further extends this approach, utilizing four 4x4 multipliers along with 8-bit and 12-bit adders. The final 16x16 multiplier follows the same architectural pattern, integrating four 8x8 multipliers with 12-bit and 24-bit adders to generate the complete 32-bit product. This modular approach allows for systematic multiplication of larger bit-width numbers while maintaining the computational efficiency inherent in the Vedic multiplication technique.



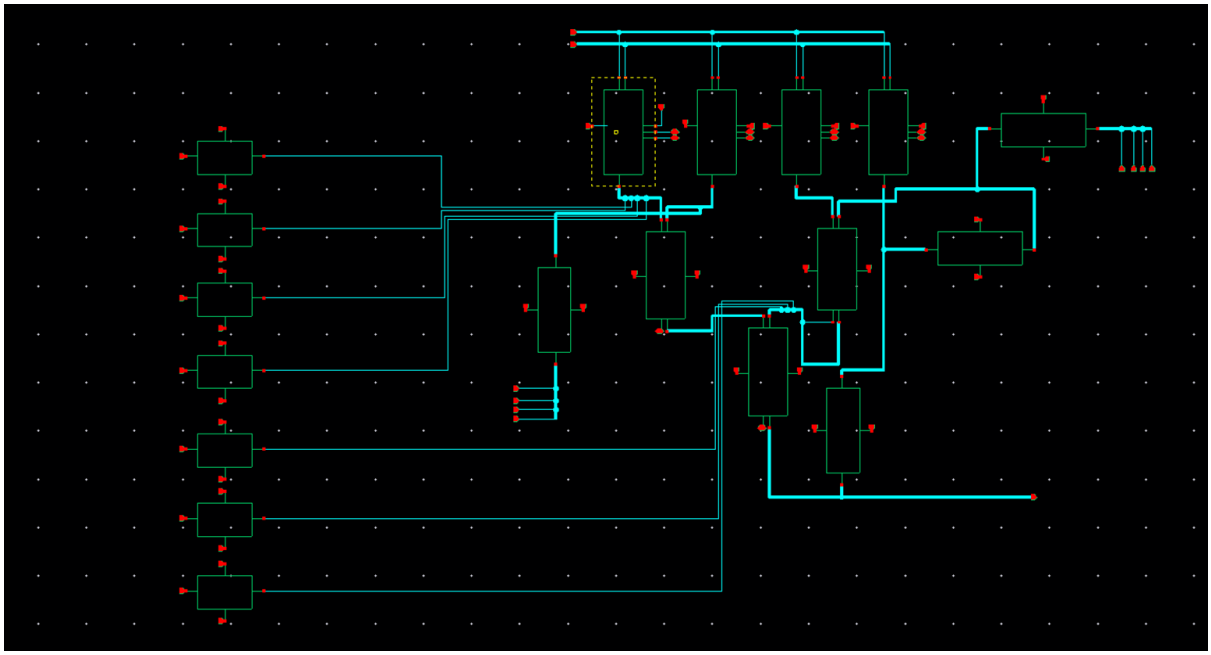
16-bit Vedic Multiplier



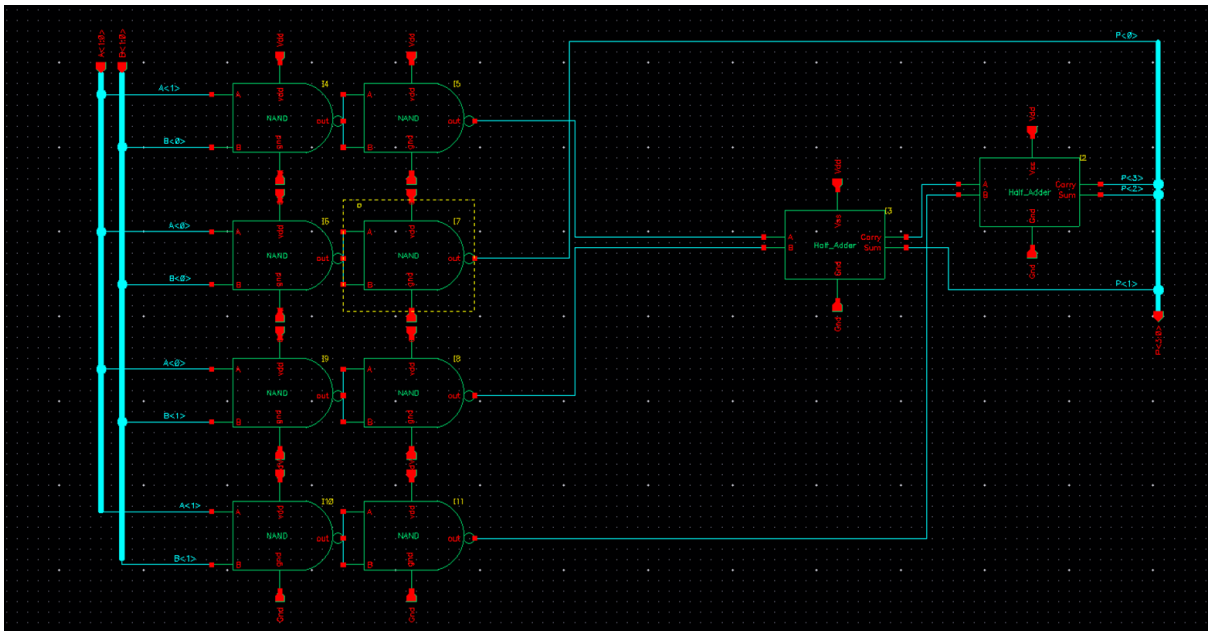
24-bit RCA



8-bit Vedic Multiplier



4-bit Vedic Multiplier



Complex Multiplier: This circuit computes the product of two complex numbers using the formula:

$(a+ib)\times(c+id)=(ac-bd)+i(ad+bc)(a + ib) \times (c + id)$

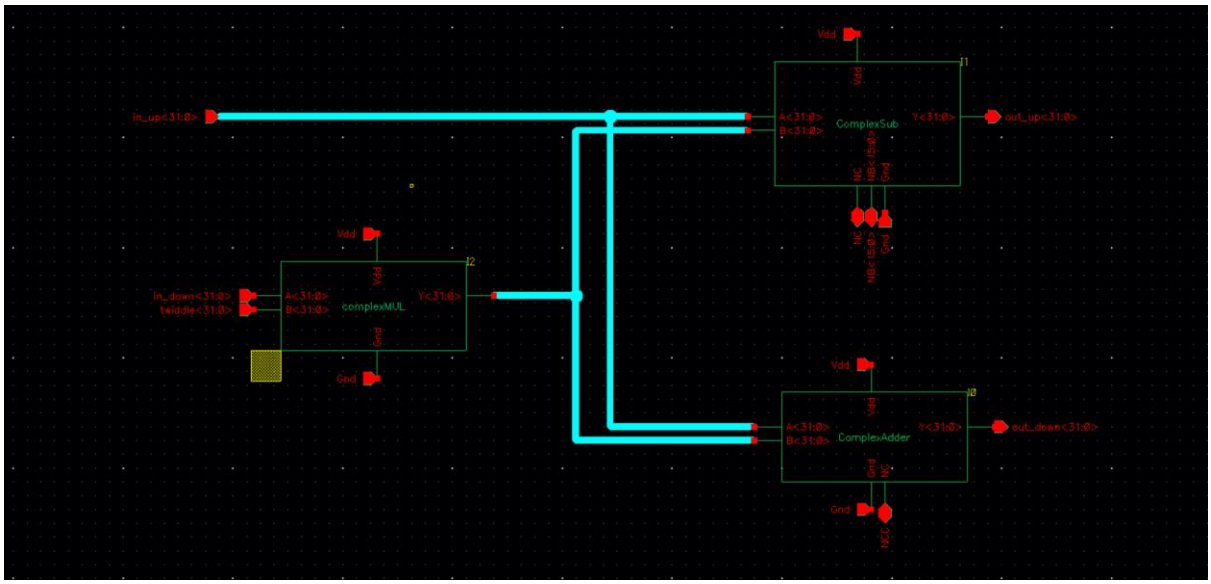
$= (ac - bd) + i(ad + bc)(a+ib)\times(c+id)$

=(ac−bd)+i(ad+bc)

This module first calculates the partial products: the product of the real parts (ac), the product of the imaginary parts (bd), and the cross products (ad and bc). These results are stored in separate wires. Afterward, the real part of the product is computed as ac−bd, and the imaginary part is calculated as ad+bc. These two values are then combined into a 32-bit output. The comp_add and comp_sub modules are used to add and subtract the corresponding real and imaginary components of the input complex numbers to produce the final product.

2.5 DIT Block

In the design of a single DIT butterfly block, the complex adder, subtractor, and multiplier that I have previously custom-designed are instantiated to perform the necessary operations for computing the butterfly's outputs. The butterfly operation involves two inputs, typically denoted as X_{even} and X_{odd}, and a twiddle factor Wk. The first step is to multiply the X_{odd} input by the twiddle factor using the complex multiplier. This results in the intermediate complex product. After that, the complex adder and subtractor modules are used to combine the X_{even} input with the twiddle-modified X_{odd}. The adder computes the sum, and the subtractor computes the difference, producing the two outputs of the butterfly block: one for the upper half of the DFT and one for the lower half. By using these custom-designed components, the butterfly block efficiently computes the DIT-FFT, ensuring proper real and imaginary component handling through addition and subtraction, along with accurate complex multiplication.

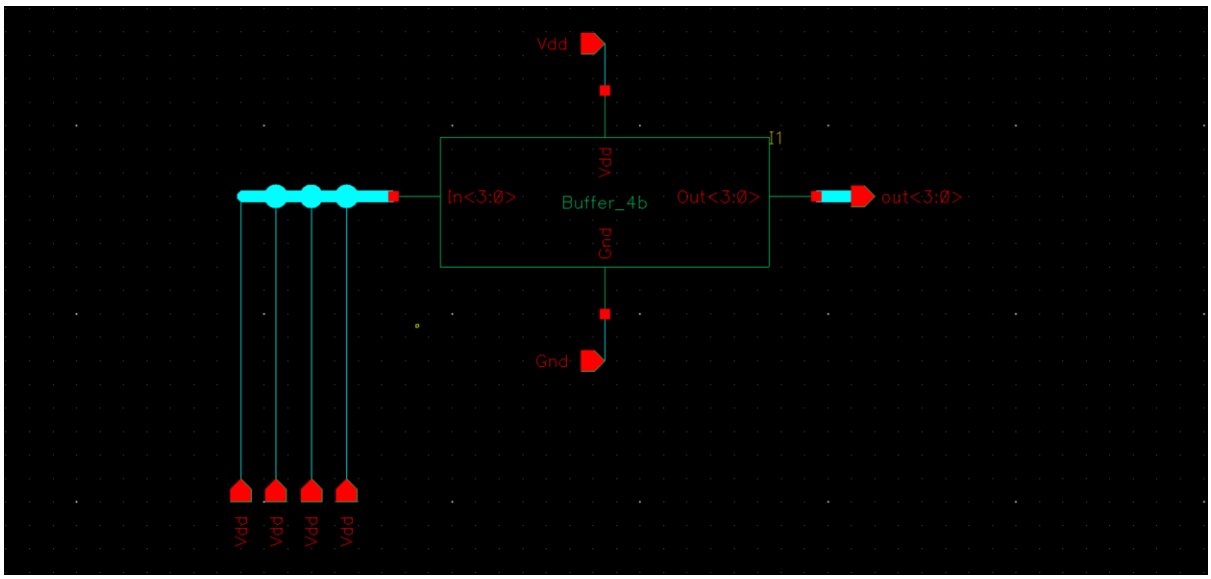


Twiddle Factors: To design twiddle factors in Full-Custom Design, we concatenated single wires either supplied with Vdd or shorted to Gnd to represent 1 and 0 respectively. The hexadecimal equivalent to the required twiddle factors are as follows:

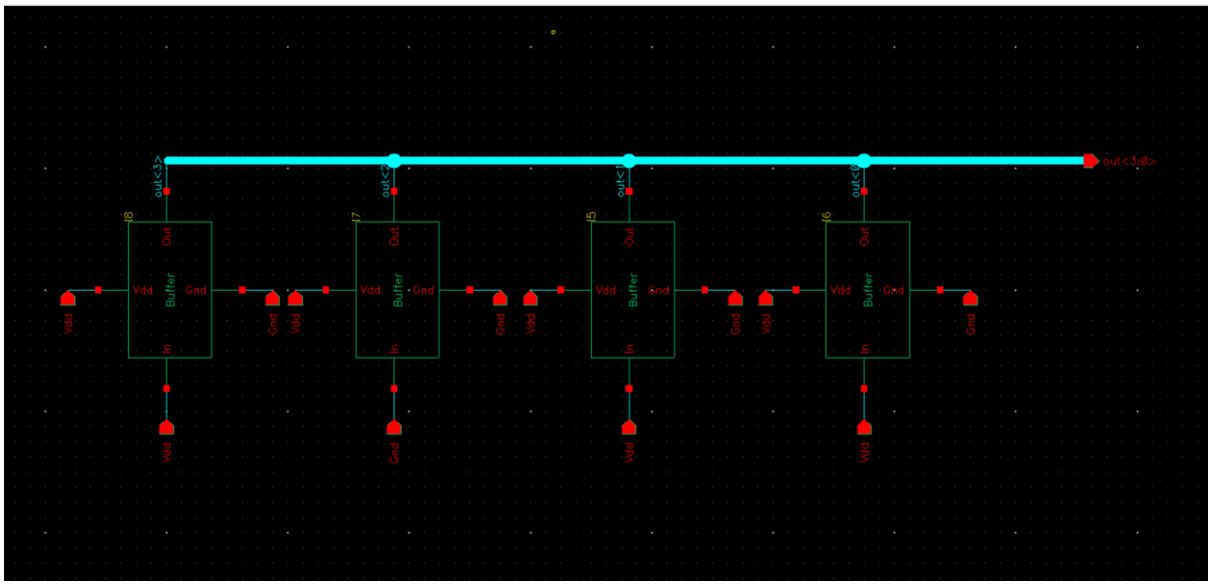
```
integer W_0 = 32'h0100_0000; //1

integer W_1 = 32'h00b5_ff4b; //0.707-0.707j

integer W_2 = 32'h0000_ff00; //-j
```

integer W_3 = 32'hff4b_ff4b; //-0.707-0.707j

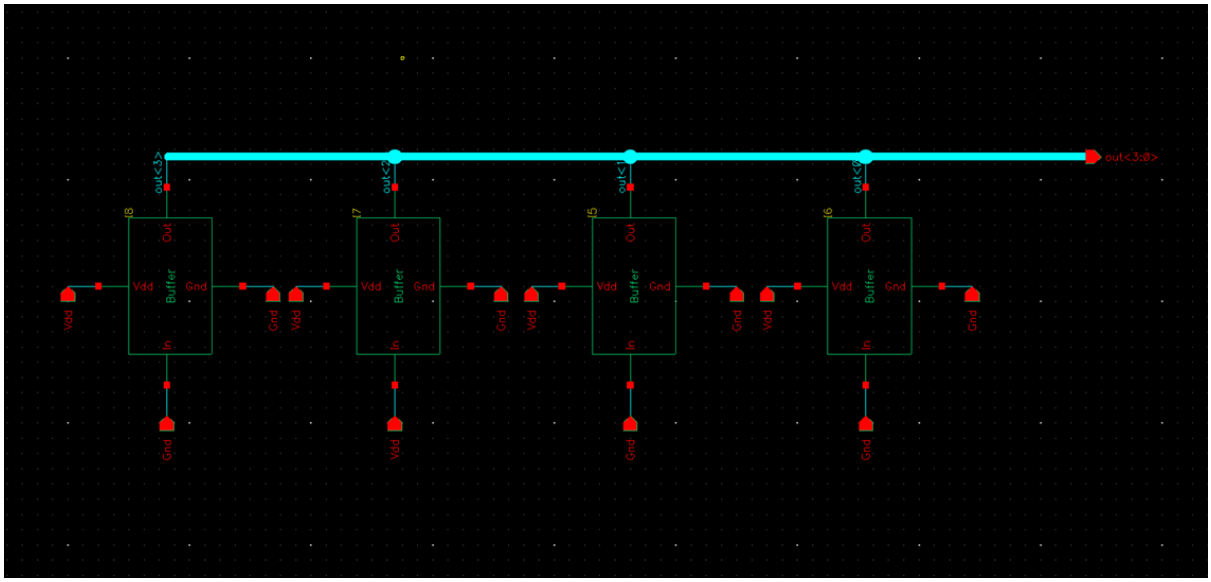
HEXADECIMAL F - REPRESENTATION



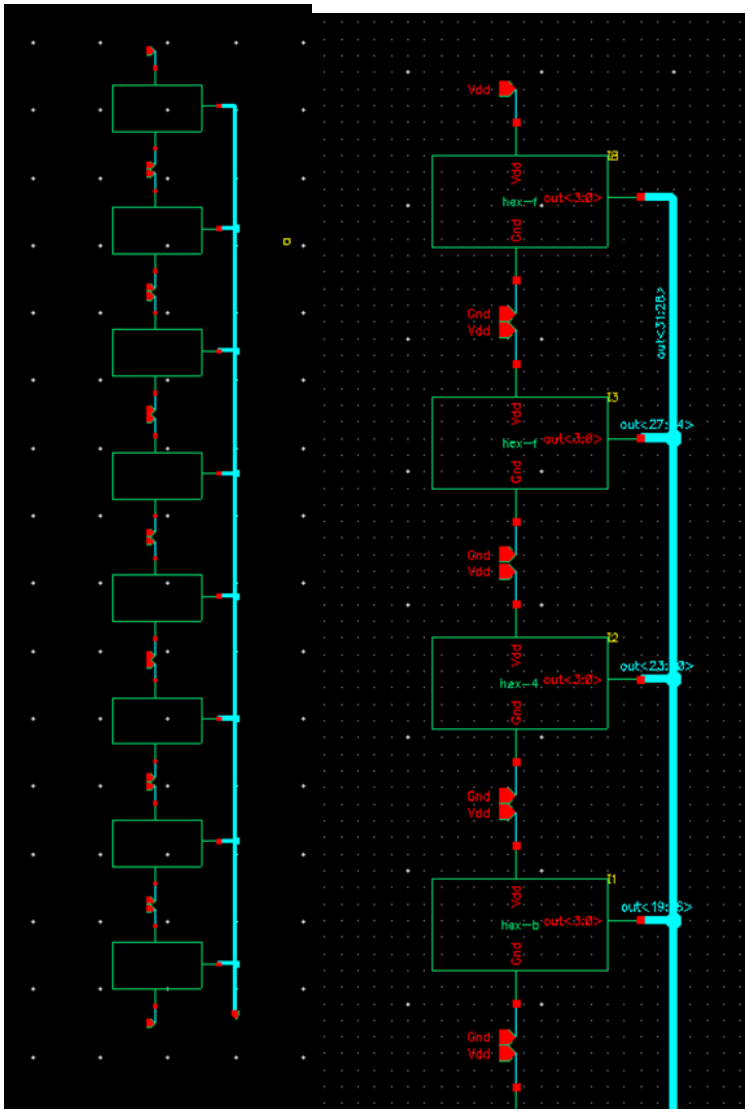
HEXADECIMAL B - REPRESENTATION



HEXADECIMAL 4 - REPRESENTATION

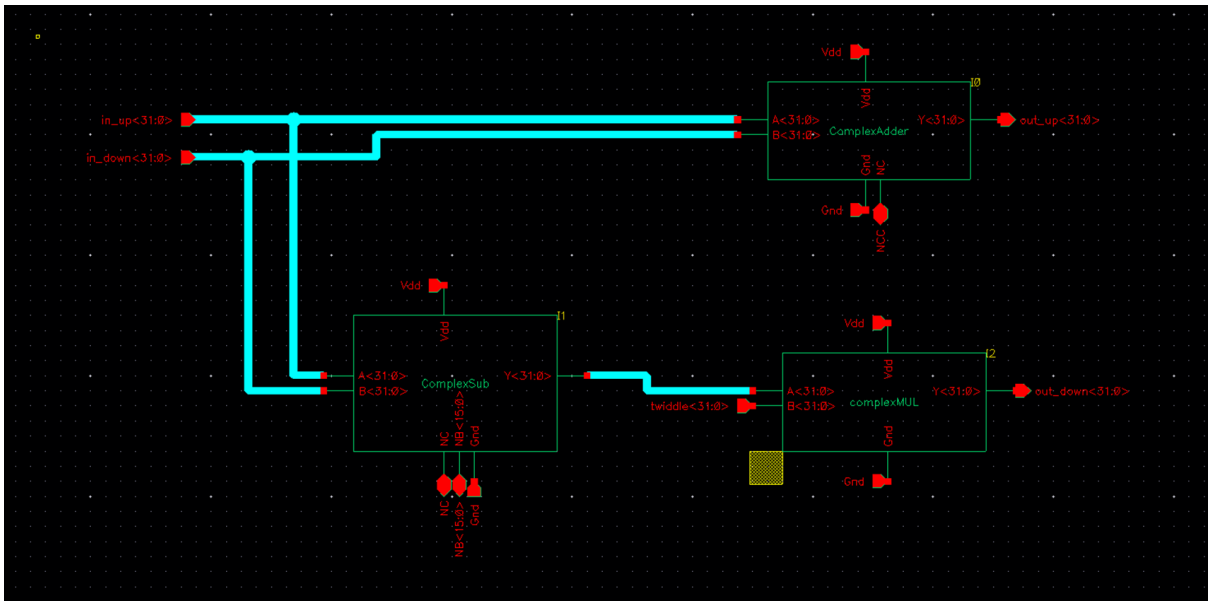


TWIDDLE FACTOR - 32'hFF4BFF4B



2.6 DIF Block

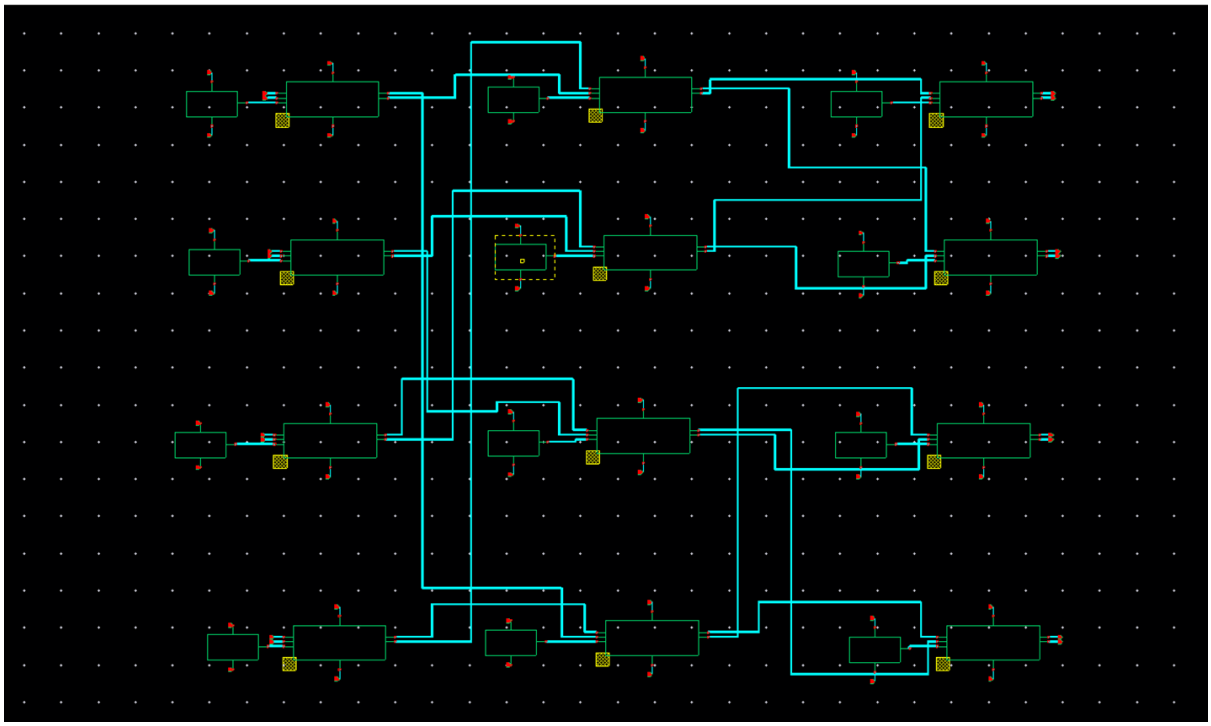
In the design of a DIF (Decimation in Frequency) butterfly block, the custom-designed complex adder, subtractor, and multiplier are instantiated to efficiently compute the butterfly's operations. The DIF butterfly block involves two main inputs, typically denoted as X_even and X_odd, and uses a twiddle factor W. The first operation in the butterfly is the addition of the X_even and X_odd inputs, performed by the custom-designed complex adder module. The resulting sum forms the upper output of the butterfly. The difference between X_even and X_odd is then computed by the complex subtractor, and this difference is subsequently multiplied by the twiddle factor using the custom complex multiplier. The multiplier handles the real and imaginary parts separately, applying the formula for complex multiplication. The final output of the DIF butterfly is obtained by combining the sum and the twiddle-modified difference. This process ensures that both the real and imaginary components are handled correctly, allowing the DIF-FFT algorithm to be efficiently computed using the custom butterfly block.



2.7 Complete FFT Architectures

To implement the complete FFT architecture, both DIT (Decimation in Time) and DIF (Decimation in Frequency) butterfly blocks, along with the necessary twiddle factor modules, are instantiated and connected in a stage-wise manner. The FFT algorithm involves recursively breaking down the input signal into smaller parts, which are then combined using the butterfly operations. The twiddle factors, which are complex exponentials, are used to modulate the input signals at each stage of the computation. For the DIT-FFT, each stage involves the application of the DIT butterfly block, where pairs of inputs are multiplied by the corresponding twiddle factors and then processed through the complex adder and subtractor to generate the outputs. These outputs are further processed in subsequent stages, applying the appropriate twiddle factors at each stage. Similarly, for the DIF-FFT, the DIF butterfly block is used, where the inputs are added and subtracted before multiplication by the twiddle factors, which are again applied at each stage of the process. The stages of both the DIT and DIF implementations follow a similar structure, with each stage halving the number of points being processed, ensuring efficient computation of the

FFT. By carefully managing the connections between the butterfly blocks and controlling the application of the twiddle factors, the complete FFT computation is achieved, resulting in the desired frequency-domain representation of the input signal.

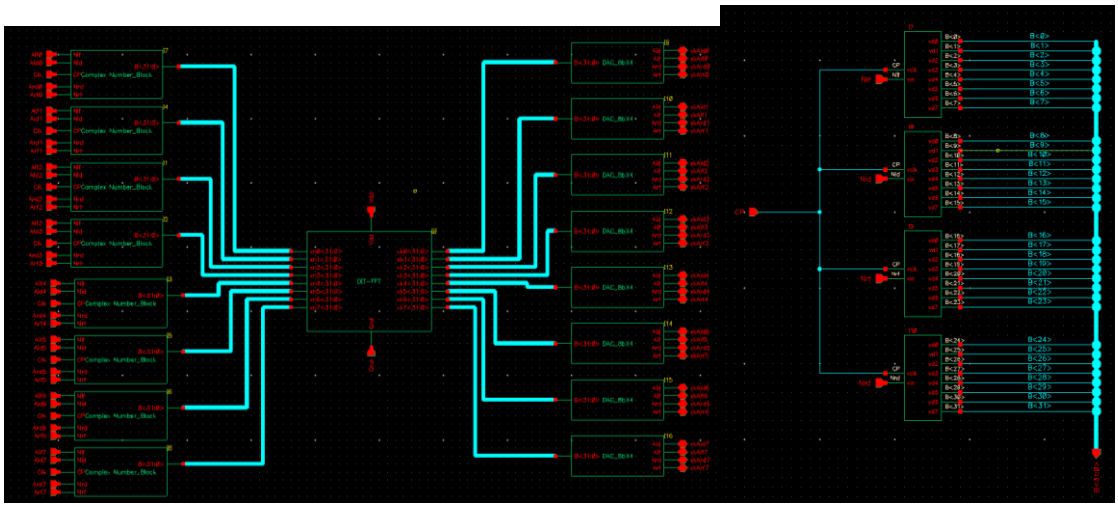


III. Methodology For Testing

3.1 ADC Block

The ADC (Analog-to-Digital Converter) is a crucial component of the testing framework for the FFT block. It serves to convert the analog input signals into digital representations suitable for processing in the FFT architecture. Here's how the ADC block was designed and utilized:

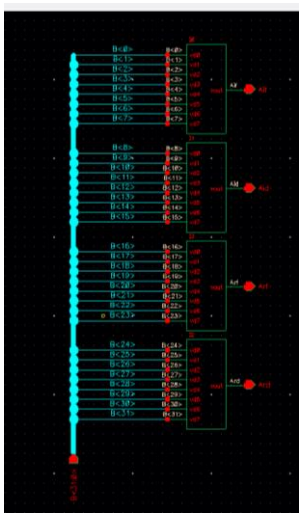
- **Input Representation**:
  Each input to the FFT block was represented as a 32-bit complex number, split into a 16-bit real and 16-bit imaginary part. Each of these parts was further divided into 8 bits for the integer portion and 8 bits for the fractional portion, following the fixed-point [8.8] format.
- **Operation**:
  The ADC converted the analog input voltage, which ranged from **0 to 1.8 V**, into an 8-bit binary representation for each segment. This process was repeated for all eight input points, generating 32-bit digital representations for the FFT.
- **Design**:
  Four 8-bit ADCs were employed to encode each 32-bit input complex number. These ADCs worked in parallel to handle the real and imaginary parts, ensuring efficient and accurate conversion for all eight input points.
- **Purpose**:
  The ADCs simplified the process of injecting test values into the FFT block, enabling a straightforward interface for testing real-world signal scenarios. By converting the analog input into a digital format, the ADC block ensured compatibility with the digital FFT design.

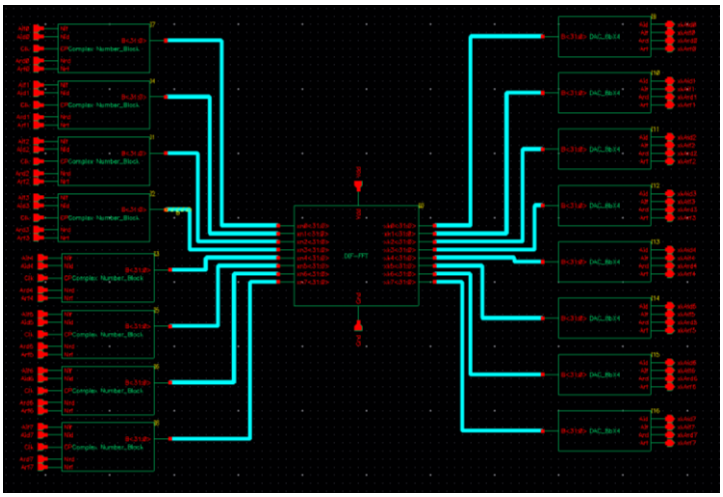


3.2 DAC Block

The DAC (Digital-to-Analog Converter) block played an essential role in converting the digital outputs of the FFT block back into analog signals for verification purposes. Here's an overview:

- **Output Representation**:
  The FFT block's output consisted of eight 32-bit complex numbers, each encoded in the same [8.8] fixed-point format as the input.
- **Operation**:
  Each 32-bit output was divided into four 8-bit segments, which were then passed through four 8-bit DACs. These DACs converted the digital values back into analog voltages within the range of **0 to 1.8 V**, proportional to the digital values.
- **Design**:
  A total of **32 DACs** were used—four DACs for each of the eight output points. This parallel setup ensured that all output data could be converted simultaneously, maintaining the real-time integrity of the test environment.
- **Purpose**:
  The DAC block facilitated the observation and analysis of FFT output in analog form, allowing direct comparison with the expected results from MATLAB simulations. This conversion provided an intuitive way to verify the accuracy of the FFT design.





3.3 MATLAB Verification

MATLAB was utilized to validate the FFT implementation by providing a reference for expected output values. The verification process included the following steps:

- **Input Generation**:
  The actual input complex numbers (in floating-point format) were directly entered into MATLAB. The `fft` function was used to compute the FFT of these numbers, serving as the ground truth for comparison.
- **FFT Analysis**:
  The MATLAB-computed FFT outputs were analyzed to ensure they matched the expected theoretical results. This included verifying the magnitude and phase of each frequency component.
- **Comparison with DAC Outputs**:
  The analog outputs obtained from the DACs were digitized and compared against MATLAB results. The correspondence between the two confirmed the correctness of the FFT block's implementation.

- **Significance**:
  MATLAB verification provided a robust means to ensure functional accuracy. Its ability to handle floating-point calculations allowed precise validation of the fixed-point implementation in the VLSI design.
- **Code**:

```
N = 8;
input_sequence = [1+1j, 2+0.5j, 0.8-1j, -1+2j, -0.5-0.5j, 1-1.5j, 0.3+0.8j, -1.2-0.3j];
fft_output = fft(input_sequence, N);
disp(fft_output);
```

- **Expected Output:**

```
Columns 1 through 6

 2.4000 + 1.0000i   3.3062 - 0.0607i  -3.3000 - 4.5000i   5.7749 + 1.3636i   0.8000 - 0.4000i  -3.9062 + 2.0607i

Columns 7 through 8

 2.1000 + 5.9000i   0.8251 + 2.6364i
```

3.4 Custom Complex Number Format Representation

The below code complements this structure by focusing on the [8.8] fixed-point format, which is used for representing either the real or imaginary parts in 16 bits.This makes giving the inputs significantly easier by decreasing the complexity , Here's how the code ties in:

1. **Conversion to Fixed-Point [8.8] Format**:
   - The `float_to_fixed_point()` function converts a floating-point value (real or imaginary) into its [8.8] binary representation.
   - The 8-bit integer and 8-bit fractional parts correspond to the sections shown in the diagram.
2. **Analog Conversion**:
   - The `convert_to_analog()` function calculates the equivalent analog voltage for the binary values. This could represent real-world scaling (e.g., outputting a real or imaginary signal).
3. **Purpose of the Code**:
   - The code processes input values (real or imaginary) for digital signal systems, where such fixed-point representations are used.
   - For example:
     - The real part could be processed first to obtain its binary and analog values.
     - The imaginary part could be processed similarly, resulting in a full 32-bit representation like in the diagram.

By combining the integer and fractional parts of both the real and imaginary components, the full 32-bit representation of a complex number is achieved.

```
def float_to_fixed_point(value):
```

```
    # Check if the value is within the range for [8.8]
    if value < -128 or value >= 128:
        raise ValueError("Value must be in the range [-128,
127.99609375]")

    # Split value into integer and fractional parts
    integer_part = int(value)
    fractional_part = value - integer_part

    # Convert integer part to binary (8 bits)
    int_bin = format(integer_part & 0xFF, '08b')  # Masking with 0xFF
ensures it fits in 8 bits

    # Convert fractional part to binary (8 bits)
    frac_bin = ''
    for _ in range(8):
        fractional_part *= 2
        bit = int(fractional_part)
        frac_bin += str(bit)
        fractional_part -= bit

    # Combine both parts
    fixed_point_binary = int_bin + frac_bin
    return fixed_point_binary, int_bin, frac_bin

def convert_to_analog(binary_string, v_ref):
    """Convert an 8-bit binary string to an analog voltage."""
    decimal_value = int(binary_string, 2)  # Convert binary string to
decimal
    analog_value = (decimal_value / 255) * v_ref  # Scale to Vref
```

```
    return abs(analog_value)  # Return absolute value


# Main function to get user input and display result
def main():
    try:
        user_input = float(input("Enter a decimal number (-128 to 127.99609375): "))
        fixed_point_binary, int_bin, frac_bin = float_to_fixed_point(user_input)

        print(f"Fixed-point [8.8] binary representation: {fixed_point_binary}")

        # Convert upper and lower bits to analog values
        v_ref = 1.8
        upper_analog_value = convert_to_analog(int_bin, v_ref)
        lower_analog_value = convert_to_analog(frac_bin, v_ref)

        print(f"Upper 8 bits analog value: {upper_analog_value:.8f} V")
        print(f"Lower 8 bits analog value: {lower_analog_value:.8f} V")

    except ValueError as e:
        print(e)

if __name__ == "__main__":
    main()
```

Analog Conversion:

- After splitting the real and imaginary parts, we separately convert:
    - The **integer** and **fractional parts of the real number** into analog signals.

    - The **integer** and **fractional parts of the imaginary number** into analog signals.
- This results in **4 analog values per tap**:
    - Real integer part → Analog Value 1
    - Real fractional part → Analog Value 2
    - Imaginary integer part → Analog Value 3
    - Imaginary fractional part → Analog Value 4

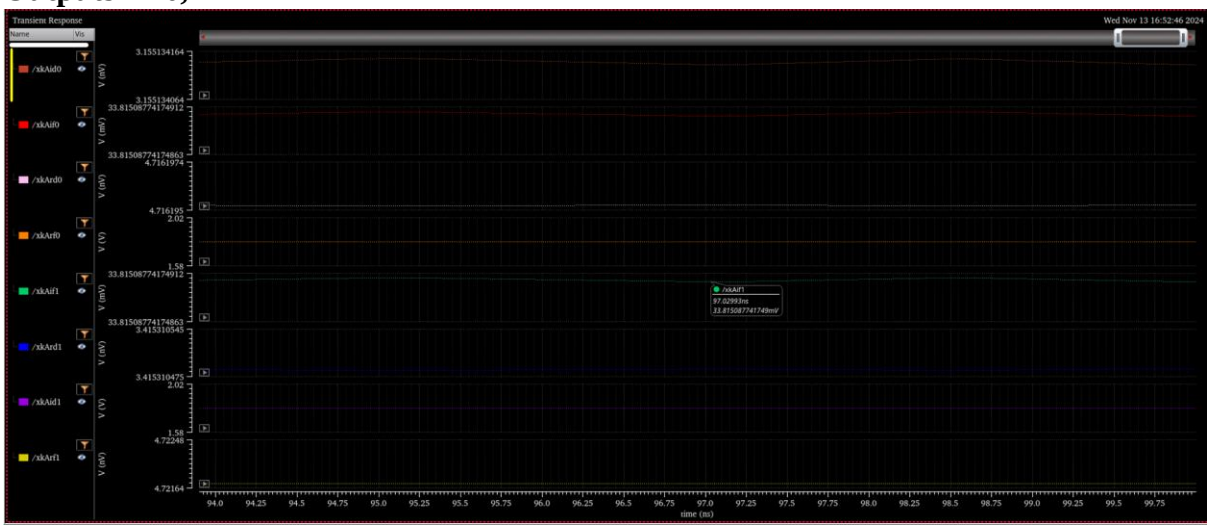
8-Tap FFT:

- In an **8-tap FFT**, there are 8 complex inputs, where each input represents a signal sample in the time domain.
- For each of the 8 taps, this method allows us to map **4 analog values per tap**, providing a highly precise representation of the signals as they undergo FFT processing.

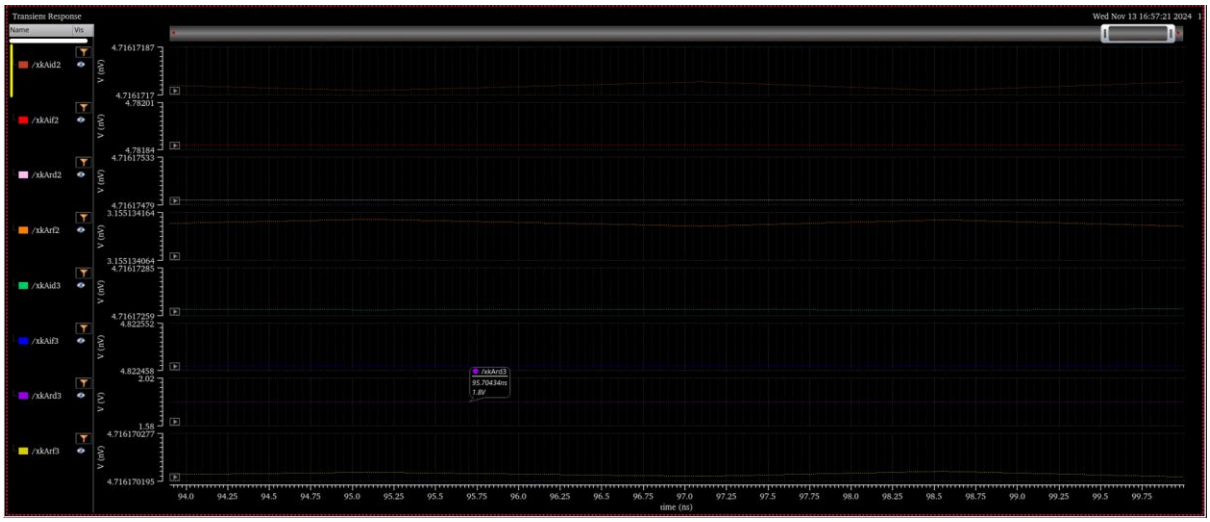
IV. Results

Now that we have designed blocks for easing the process of simulating our designed FFT block, we simulate it in Cadence Virtuoso by applying inputs via the ADE L Stimuli and observing the produced graphs.

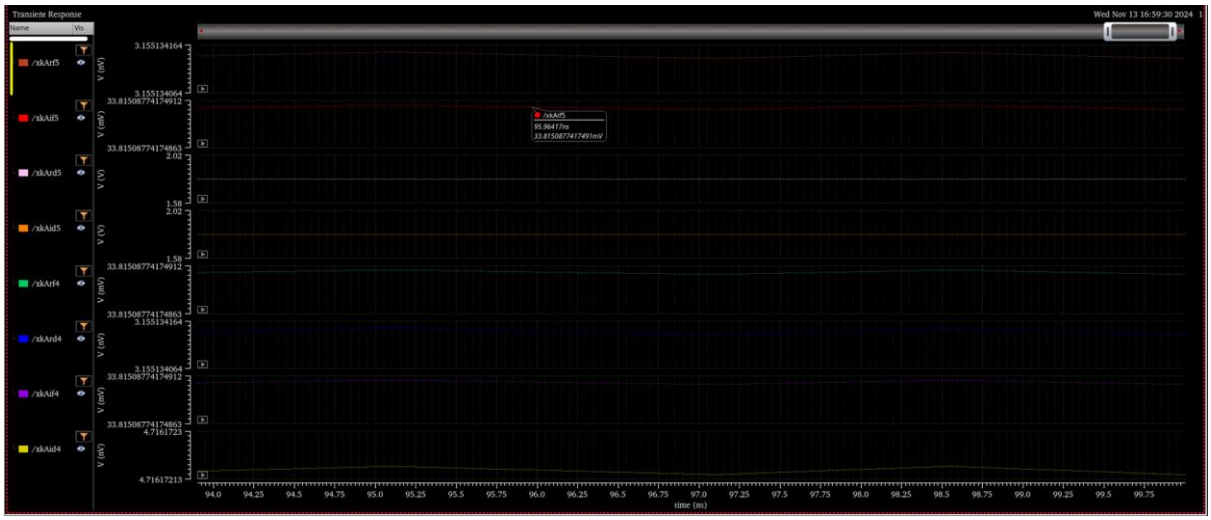
Outputs Xk0, Xk1



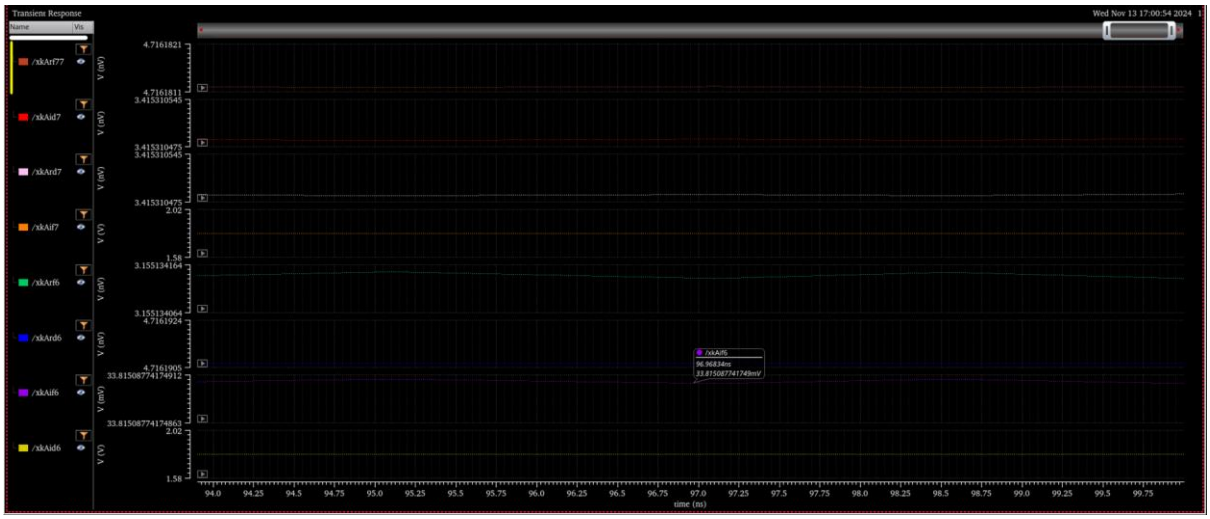
Outputs Xk2, Xk3



Outputs Xk4, Xk5



Outputs Xk6, Xk7



Observing the obtained voltage levels for each output port and cross checking them with the obtained voltage equivalents for the expected outputs via MATLAB prove that the designed modules perform a highly accurate FFT Computation.

V. Conclusion

The full-custom design of Decimation-In-Time (DIT) and Decimation-In-Frequency (DIF) Fast Fourier Transform (FFT) architectures demonstrates the critical potential of modular, hierarchical design in high-performance digital signal processing systems. By leveraging custom-designed arithmetic modules such as complex adders, subtractors, and Vedic multipliers, the implemented 8-point FFT architecture achieves efficient computation with minimal hardware complexity. The [8.8] fixed-point representation provides a judicious balance between numerical precision and resource utilization, enabling accurate signal transformation while maintaining computational efficiency. The comprehensive testing methodology, integrating ADC and DAC blocks with MATLAB verification, validates the design's functional correctness and highlights the systematic approach to hardware implementation of digital signal processing algorithms.

Future research directions for this work include exploring scalability of the proposed architecture to higher-point FFT implementations, investigating adaptive bit-width techniques to further optimize resource utilization, and integrating machine learning-driven optimization strategies for arithmetic module design. Potential avenues for enhancement involve exploring alternative multiplication algorithms, implementing dynamic range adaptation mechanisms, and investigating low-power design techniques such as power gating and clock-frequency scaling. Moreover, the modular design framework established in this project provides a robust foundation for exploring emerging signal processing paradigms, including neuromorphic computing architectures and edge computing applications that demand high-performance, energy-efficient signal transformation techniques.

VI. References

[1] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd ed. California Technical Publishing, 1999.

[2] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965, doi: 10.1090/S0025-5718-1965-0178586-1.

[3] V. K. Madisetti and D. B. Williams, *The Digital Signal Processing Handbook*, 2nd ed. Boca Raton, FL: CRC Press, 1997.

[4] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. Hoboken, NJ: Wiley, 1999.

[5] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, 4th ed. Upper Saddle River, NJ: Pearson Education, 2007.

[6] C. Lakshmi and P. Shruthi, "Efficient Design of FFT Butterfly Unit Using Vedic Multiplier," *Int. J. Adv. Res. Electron. Commun. Eng.*, vol. 4, no. 8, pp. 2200–2205, 2015.

[7]J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, 2nd ed. Upper Saddle River, NJ: Pearson, 2003.

[8]A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd ed. Upper Saddle River, NJ: Pearson, 2010.

[9]X. Xu and Y. Chen, "Radix-4 FFT Processor Design for High-Performance Applications," *IEEE Trans. Circuits Syst. II*, vol. 68, no. 5, pp. 1203–1213, May 2021, doi: 10.1109/TCSII.2021.3053032.

[10]R. Ansari, "Design of Low-Power FFT Architectures for Embedded Systems," *IEEE Embedded Syst. Mag.*, vol. 15, no. 3, pp. 30–36, Jul. 2020.

[11]MathWorks, MATLAB Documentation, MathWorks Inc. [Online]. Available: https://www.mathworks.com/help/.

[12]Python Software Foundation, *Python Programming Documentation*. [Online]. Available: https://www.python.org/doc/.

[13]C. S. Burrus and T. W. Parks, *DFT/FFT and Convolution Algorithms: Theory and Implementation*. New York, NY: Wiley, 1985.

[14]E. H. Wold and A. M. Despain, "Pipeline and Parallel Pipeline FFT Processors for VLSI Implementation," *IEEE Trans. Comput.*, vol. C-33, no. 5, pp. 414–426, May 1984, doi: 10.1109/TC.1984.1676478.

[15]H. Johansson, "High-Performance FFT Architectures for Real-Time Signal Processing," *IEEE Trans. Signal Process.*, vol. 58, no. 7, pp. 3585–3592, Jul. 2010, doi: 10.1109/TSP.2010.2046574.