

**PIPELINED FFT IMPLEMENTATION**

**PROJECT REPORT**

**CHRISTODOULOS ZERDALIS 03531**

**ATHANASIOS GERAMPINIS 03466**

**ECE494 MICROPROCESSOR DESIGN**

**PROFESSOR: G. KARAKONSTANTIS**



**June 2025**

**Department of Electrical and Computer Engineering**

**University of Thessaly**

## ABSTRACT

This report presents the design and evaluation of three FFT architectures-Radix-2, Radix-4, and Radix-2 Single-Path Delay Feedback implemented on an FPGA. The goal is to compare their performance in terms of area, speed, and scalability for real-time signal processing. The unrolled Radix-2 and Radix-4 designs exploit full parallelism but suffer from high resource usage, making them unsuitable for larger input sizes. To overcome I/O limitations, a memory controller was introduced to serialize data transfers, trading speed for feasibility on the ZedBoard FPGA. In contrast, the Radix-2 SDF uses a single compute unit per stage, significantly reducing resource requirements and enabling larger FFT sizes, though with increased latency. Results highlight the trade-offs between parallelism and scalability across the three designs.

## RESEARCH GOAL

The primary objective of our research is to design, implement, and evaluate various hardware architectures for the Fast Fourier Transform (FFT) algorithm on an FPGA platform. Specifically, we focus on three distinct FFT implementations: Radix-2, Radix-4, and Radix-2 Single-Path Delay Feedback (SDF). Each of these architectures presents unique trade-offs in terms of resource utilization, power consumption, computational speed, and implementation complexity. Through systematic testing and comparison, we aim to analyze how these designs perform under different constraints and workloads. Metrics such as area utilization (LUTs, FFs, BRAMs), latency, throughput, and dynamic power will be measured using FPGA synthesis and implementation tools. Additionally, the designs will be evaluated for scalability and suitability in real-time and resource-constrained signal processing applications. This comparative study will provide valuable insights into the strengths and limitations of each FFT architecture and guide future optimization efforts for high-performance, low-power digital signal processing on reconfigurable hardware.

# CHAPTER 1

## INTRODUCTION TO THE ALGORITHM

### 1.1 FFT in general

The Fast Fourier Transform (FFT) is an efficient algorithm used to compute the Discrete Fourier Transform (DFT) of a sequence, which transforms a signal from the time domain into the frequency domain. This means it breaks down a complex signal into its constituent sinusoidal components-frequencies and their amplitudes-making it easier to analyze patterns such as pitch in audio, frequencies in communications, or periodicities in data. The FFT drastically reduces the computational complexity of DFT from  $O(N^2)$  to  $O(N \log(N))$  by cleverly reusing calculations through a divide-and-conquer approach. This algorithm is ideal for real-life applications like audio processing, image analysis, wireless communication and other biomedical and industrial applications.

The most common FFT algorithm is the **Cooley-Tukey** algorithm, which is widely used due to its simplicity and efficiency. It works best when the input size  $N$  is a power of 2. The most common versions of the algorithm are:

- **Radix-2:** The most basic and common version. Splits the DFT into two halves (even and odd indices) at each stage. Requires  $N$  to be a power of 2.
- **Radix-4:** More efficient than radix-2 when  $N$  is a power of 4. Reduces the number of multiplications compared to radix-2 by grouping four elements at a time.
- **Mixed Radix versions:** They combine multiple radix versions and they're suitable for input sizes that are not powers of 2, but they are more complex.

In our case we focused on the radix-2 and radix-4 versions.

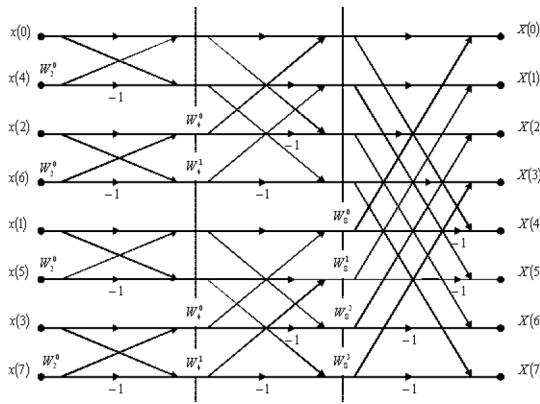


Figure 1.1: Radix-2 Dataflow

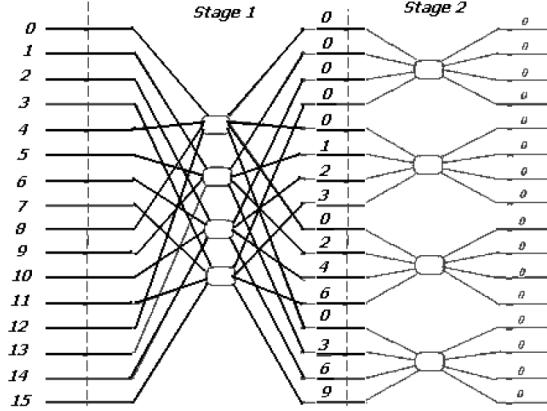


Figure 1.2: Radix-4 Dataflow

## 1.2 More about Radix-2

For a sequence  $x[n]$ , the DFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{kn}, \quad \text{where } W_N = e^{-j\frac{2\pi}{N}}$$

$W_N$  is called the **twiddle factor**. The FFT algorithm reorganizes this sum to reduce computation.

Let's split the input  $x[n]$  into even-indexed ( $e[n] = x[2n]$ ) and odd-indexed ( $o[n] = x[2n + 1]$ ) samples. The DFT becomes:

$$X[k] = \sum_{n=0}^{N/2-1} x[2n] \cdot W_N^{k \cdot 2n} + \sum_{n=0}^{N/2-1} x[2n + 1] \cdot W_N^{k \cdot (2n+1)}$$

Note that  $W_N^{2kn} = W_{N/2}^{kn}$ . So we rewrite:

$$X[k] = \underbrace{\sum_{n=0}^{N/2-1} e[n] \cdot W_{N/2}^{kn}}_{\text{Even terms (E}[k]\text{)}} + W_N^k \cdot \underbrace{\sum_{n=0}^{N/2-1} o[n] \cdot W_{N/2}^{kn}}_{\text{Odd terms (O}[k]\text{)}}$$

Let  $E[k]$  be the DFT of even-indexed and  $O[k]$  the DFT of odd-indexed inputs. Then:

$$X[k] = E[k] + W_N^k \cdot O[k]$$

Because the DFT is periodic with period  $N$ , for  $k = 0, 1, \dots, N/2-1$ , we can compute:

$$X[k] = E[k] + W_N^k \cdot O[k]$$

$$X\left[k + \frac{N}{2}\right] = E[k] - W_N^k \cdot O[k]$$

This operation is called a **butterfly**

For an input sequence of length  $N$ , where  $N = 2^m$ , the FFT has  $m = \log_2(N)$  stages. Each stage performs butterfly operations to progressively combine and transform the input data.

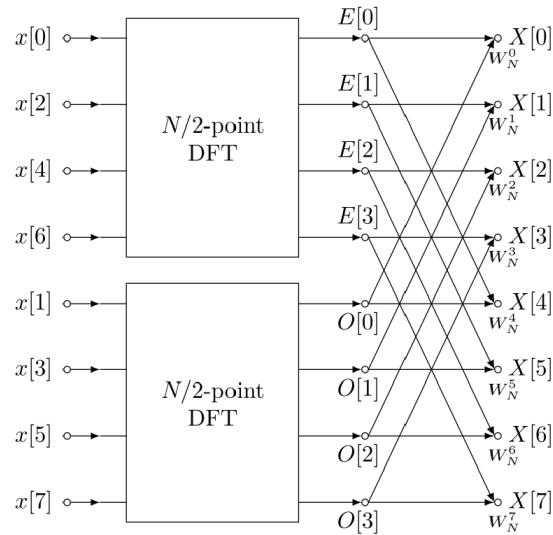


Figure 2: The Radix-2 FFT follows a recursive logic. It combines the results of the previous stages, which complete the FFT on the even and odd sequences.

### 1.3 More about Radix-4

The Radix-4 FFT is an extension of the Radix-2 algorithm that divides the DFT into **4 smaller DFTs** instead of 2. It requires the input size  $N$  to be a power of 4.

If we group  $x[n]$  into 4 interleaved subsequences:

$$x_0[n] = x[4n], \quad x_1[n] = x[4n+1], \quad x_2[n] = x[4n+2], \quad x_3[n] = x[4n+3]$$

Then the DFT becomes:

$$X[k] = \sum_{n=0}^{N/4-1} [x[4n] + x[4n+1]W_N^k + x[4n+2]W_N^{2k} + x[4n+3]W_N^{3k}] \cdot W_{N/4}^{kn}$$

Then:

$$X[k] = X_0[k] + W_N^k X_1[k] + W_N^{2k} X_2[k] + W_N^{3k} X_3[k]$$

If we let  $A = X_0[k]$ ,  $B = W_N^k X_1[k]$ ,  $C = W_N^{2k} X_2[k]$  and  $D = W_N^{3k} X_3[k]$  then the butterfly operation for 4 points becomes:

$$\begin{aligned} X[k] &= A + B + C + D \\ X\left[k + \frac{N}{4}\right] &= A - jB - C + jD \\ X\left[k + \frac{N}{2}\right] &= A - B + C - D \\ X\left[k + \frac{3N}{4}\right] &= A + jB - C - jD \end{aligned}$$

Radix-4 reuses more symmetries and reduces the number of total operations. It also cuts the number of required stages to  $\log_4 N$ , compared to  $\log_2 N$  for radix-2. And, as you'll see next, it is more efficient on hardware, when  $N$  is a power of 4.

## CHAPTER 2

### HARDWARE IMPLEMENTATION

#### 2.1 Twiddle constants

Since our hardware implementations are designed for FFTs of fixed sizes, there is no need for the algorithm to compute the twiddle factors at runtime. Calculating these factors involves complex arithmetic operations, which are computationally expensive and consume valuable FPGA resources. To optimize efficiency, all twiddle factors are pre-computed during compilation and stored in on-chip memory. Because the FFT algorithm operates using multiple computational units in parallel, the choice of memory architecture is critical. The memory must provide sufficient bandwidth to avoid becoming a bottleneck in the data path. To meet these performance requirements, twiddle factors are stored using distributed memory such as LUTRAM or flip-flops, enabling constant-time access and ensuring that each computational unit can retrieve the required constants without delay.

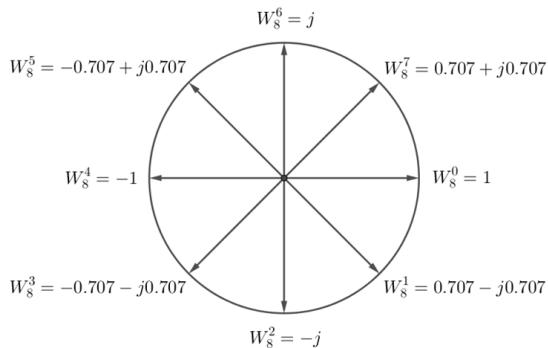


Figure 2.1: Twiddle factors for N=8

Index	W_Real	W_Img	Complex number
0	32768	0	1 + 0j
1	23170	-23170	0.7071 - 0.7071j
2	0	-32768	0 - 1j
3	-23170	-23170	-0.7071 - 0.7071j
4	-32768	0	-1 + 0j
5	-23170	23170	-0.7071 + 0.7071j
6	0	32768	0 + 1j
7	23170	23170	0.7071 + 0.7071j

Figure 2.2: Twiddle values for N=8

#### 2.2 Datatype

The samples are represented using a 32-bit fixed-point format, where the value 1.0 corresponds to the integer 32767 (i.e.,  $2^{15} - 1$ ). Although the data is 32 bits wide, the effective scaling is based on the Q1.15 format. This approach combines the benefits of wider data paths—such as reduced overflow risk and better intermediate precision—with

the simplicity and hardware efficiency of fixed-point arithmetic. It provides sufficient accuracy and dynamic range for most FFT applications.

### 2.3 Unrolled Radix-2/4

This implementation of the Radix-2/4 algorithm fully exploits its inherent parallelism by unrolling the entire computation. Each butterfly operation is assigned a dedicated hardware unit, resulting in a fully unrolled design where all butterfly computations are executed concurrently.

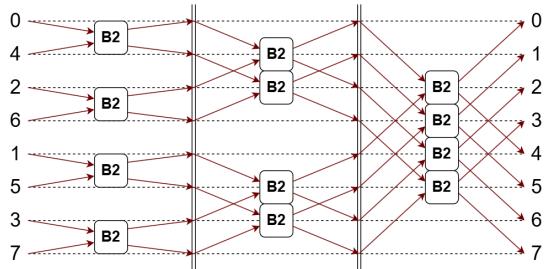


Figure 2.3: This image shows the structure/dataflow of the radix-2 unrolled algorithm with B2 being it's dedicated butterfly

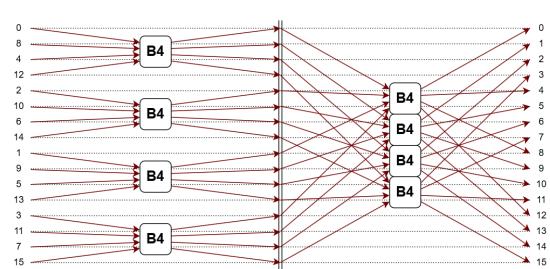


Figure 2.4: This image shows the structure/dataflow of the radix-4 unrolled algorithm with B4 being it's dedicated butterfly

#### 2.3.1 Data Bandwidth Limitation

The fully parallel implementations of the Radix-2 and Radix-4 FFT architectures require all input data to be available before computation begins. This becomes increasingly demanding as the FFT size  $N$  grows. For example, for  $N = 16$ , with 32-bit complex data and four parallel data ports, the input bandwidth requirement reaches  $32 \times 16 \times 4 = 2048$  bits. This demand far exceeds the I/O bandwidth supported by our target FPGA.

To address this limitation, we implemented a custom memory controller that streams input data serially, storing it internally until all required values are available. Once sufficient data is buffered, the FFT computation begins, and the results are also output sequentially. While this approach significantly reduces I/O requirements and makes the design feasible on resource-constrained hardware, it introduces latency due to the serialized input/output process.

Originally, the computation time for a parallel FFT was determined solely by the

number of stages:

$$T = \log_2(N)$$

For instance, a 16-point FFT required just 4 clock cycles. However, with the new memory controller and serialized data transfer, the total computation time increases to:

$$T_{\text{new}} = \log_2(N) + 2N$$

where the additional  $2N$  accounts for input buffering and sequential output.

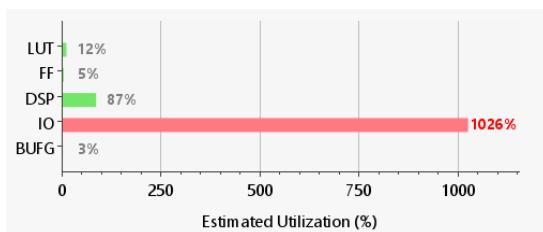


Figure 2.5: Without memory controller

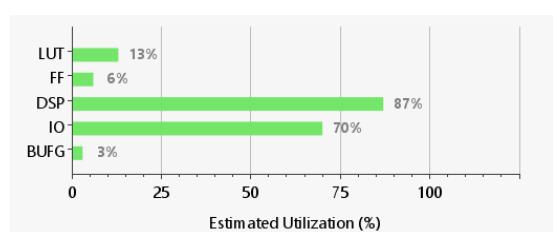


Figure 2.6: With memory controller

### 2.3.2 Scalability Limitations

The unrolled versions of the FFT scale their resource usage roughly with a factor of  $N \log N$ , making them difficult to scale on FPGAs with limited resources. As the input size increases, both logic utilization and area requirements grow significantly. This presents a major constraint—even with a modest input size of 64 points, the design exceeds the available resources on our FPGA and cannot be synthesized or implemented successfully.

Such fully unrolled architectures would be better suited for high-end System-on-Chip (SoC) platforms, where resource limitations are less restrictive.

## 2.4 Radix-2 Single Path Delay Feedback

The Radix-2 Single-Path Delay Feedback (SDF) architecture eliminates the scalability problem by reusing a single butterfly compute unit per stage. Unlike the fully unrolled designs, which instantiate a separate unit for every butterfly operation, the SDF approach processes data sequentially and recycles the hardware across clock cycles. This dramat-

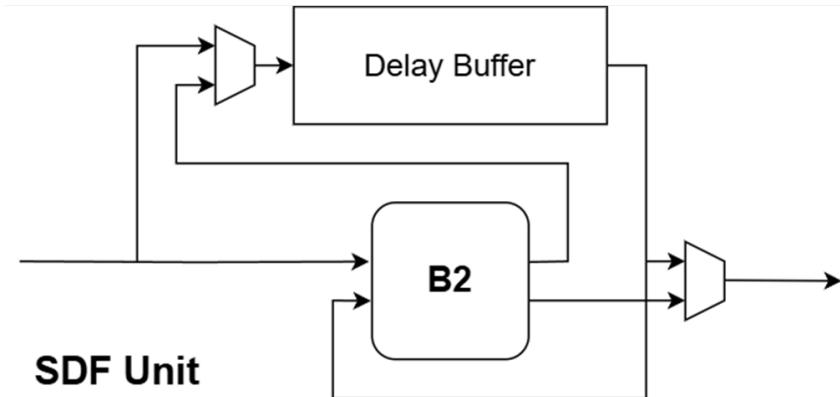


Figure 2.7: Top diagram of an SDF stage unit

ically reduces resource usage, making it well-suited for large input sizes and resource-constrained FPGAs.

Additionally, the SDF architecture naturally supports streaming input and output, enabling continuous data processing without requiring all inputs to be available beforehand. This makes it ideal for real-time signal processing applications, where low area and sustained throughput are more critical than minimal latency.

## CHAPTER 3

### RESULTS

#### 3.1 Resource usage on FPGA

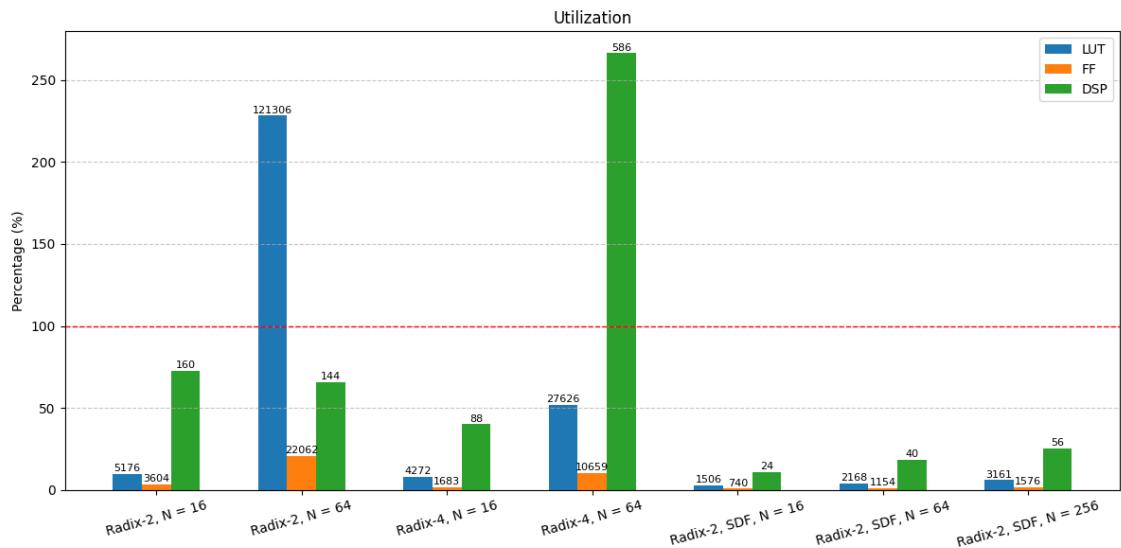


Figure 3.1: FPGA resources for each implementation

We observe that as the number of samples increases, the resource usage for the unrolled versions grows dramatically. This makes the method unsustainable for small FPGAs due to limited hardware resources. On the other hand, the SDF architecture scales linearly with the input size, making it a more suitable choice for low-area, medium-latency applications.

### 3.2 Timing and Energy usage

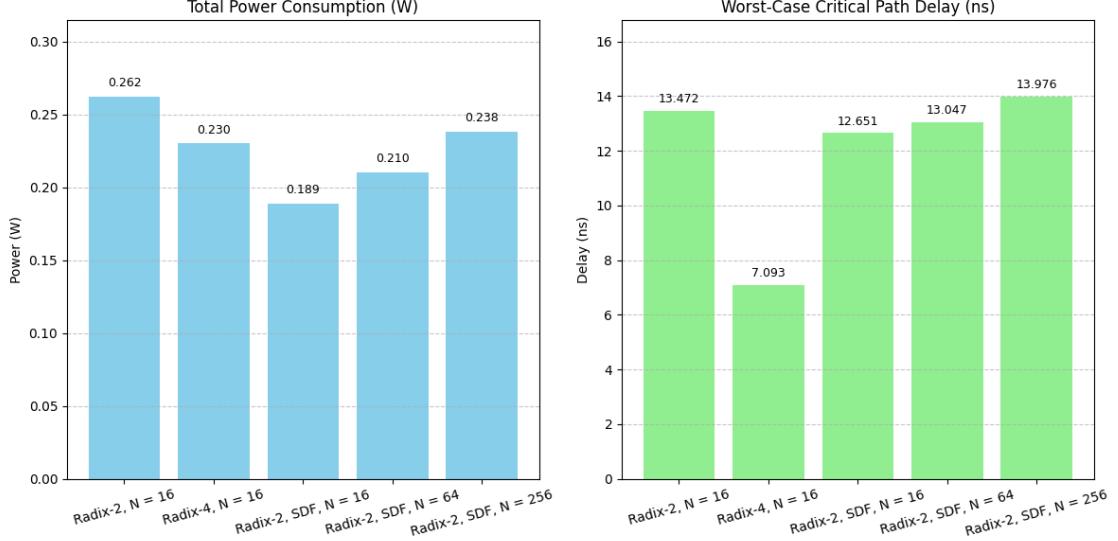


Figure 3.2: Energy and timing metrics

We observe that the worst-case critical path gradually increases—from approximately 12.5 ns to 14 ns—for implementations that use the B2 unit, regardless of whether they are unrolled or SDF-based. In contrast, the Radix-4 implementation demonstrates better performance, with a critical path of around 7 ns. This improvement is attributed to the fact that Radix-4 B4 unit more effectively utilizes the FPGA’s DSP blocks, allowing heavy computational logic to complete more quickly.

### 3.3 Execution times

All three FFT architectures exhibit different execution times due to their structural characteristics. The Radix-2 and Radix-4 unrolled implementations have an execution time of approximately  $\log N$ , as the computation proceeds through  $\log N$  fully parallel stages.

In contrast, the Radix-2 and Radix-4 versions with a memory controller experience increased execution time due to the serialized data input. In this case, the total latency becomes:

$$T = 2N + \log N,$$

where the additional  $2N$  accounts for the sequential loading and unloading of input and output data.

For the Radix-2 SDF architecture, we carefully analyzed the behavioral simulation to understand the propagation delays between stages. Each stage contains a delay buffer, and as the data progresses through the pipeline, these buffers introduce increasing latency.

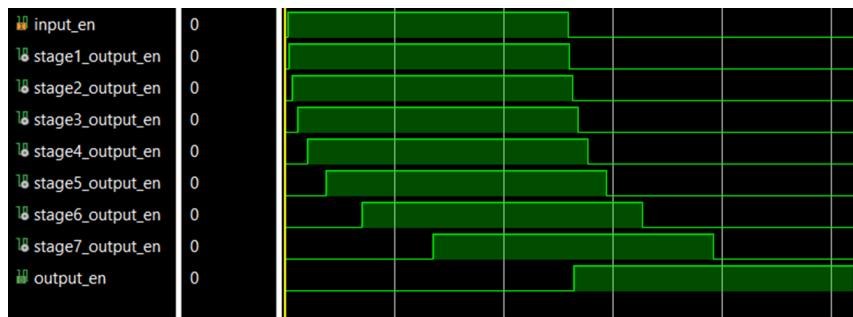


Figure 3.3: Here we can see the progression of the stages and their intermediate delays.

As observed in the simulation for  $N = 256$ , the delay offsets between consecutive output enables form the pattern:

$$1 + 3 + 5 + 9 + 17 + 31 + \dots + 256 \text{ clock cycles.}$$

This can be generalized by the following formula:

$$T_{\text{SDF}} = N + 1 + \sum_{i=1}^{\log_2 N - 1} (2^i + 1) = 2N - 2 + \log N.$$

This expression captures both the input streaming overhead and the cumulative stage delays inherent in the SDF structure.

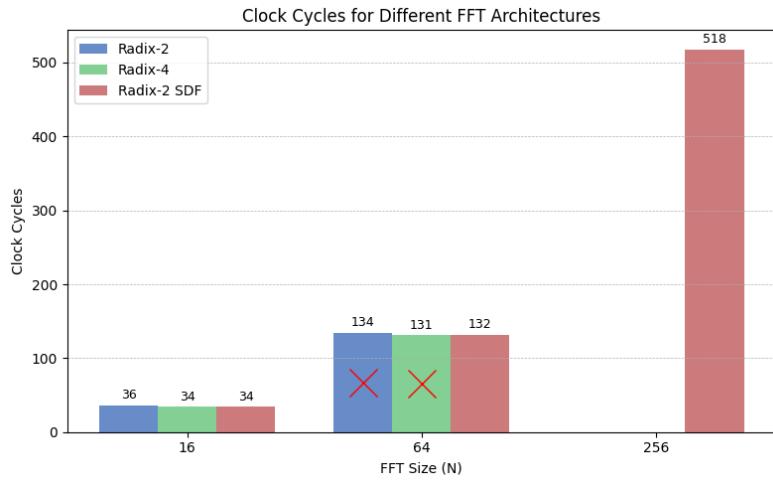


Figure 3.4: Performance of all implementations in clock cycles.

### 3.4 Test Signals

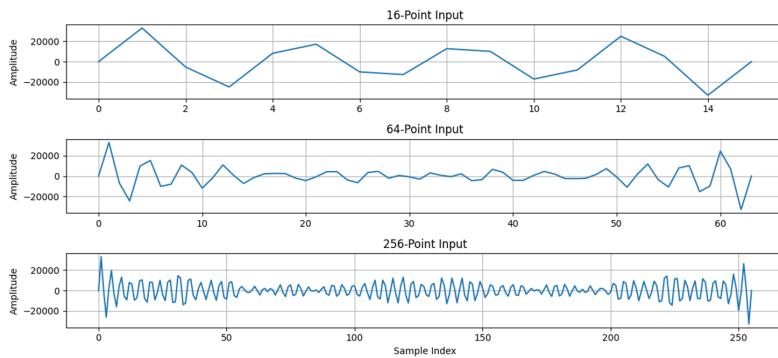


Figure 3.5: Real input signal

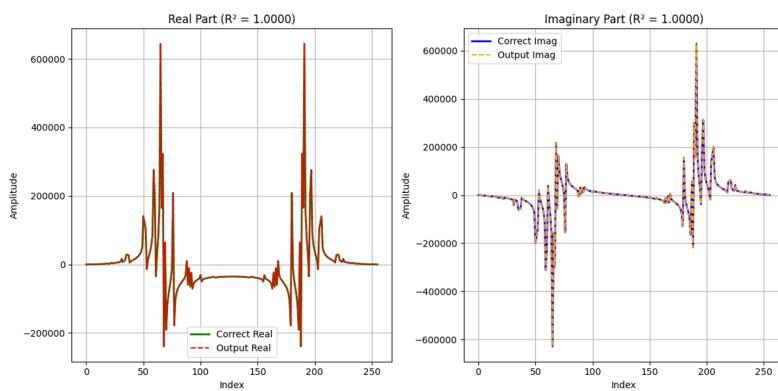


Figure 3.6: In this graph, we can see that the outputs of our FFT implementation match exactly with the expected results.

## CHAPTER 4

### PROBLEMS AND TOOLS

#### 4.1 Problems faced

The main problem we faced was the memory bandwidth bottlenecks in the unrolled versions that forced us to implement a memory controller to feed the data. This decision changed the algorithm's complexity from  $O(\log N)$  to  $O(N)$ . Theoretically, using an AXI master interface could improve the data rate, but even then, the unrolled algorithm remains not scalable for small FPGAs. We also attempted to implement the Radix-4 SDF architecture, but encountered difficulties in understanding its structure and managing the data flow, which made the design too complex to complete within our timeframe.

This chapter provides a summarization of our empirical results and their implications.

#### 4.2 Tools used

To complete our research, we used Vivado for both behavioral and post-implementation simulations to verify the correct operation of our designs. The implementation was targeted to the ZedBoard FPGA, with all hardware modules written in Verilog. Additionally, the FFT algorithms were tested in software using C++, and all synthetic input/output data, including twiddle factors and result validation, were generated and analyzed using Python.