METHODS FOR SCALABLE LEVELS OF PARALLELISM IN RADIX-2 FFTS FOR FPGA SYNTHESIS

By

Felipe Minotta Zapata

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS

May, 2014

Approved by:

Manuel Jiménez, Ph.D. Chairman, Graduate Committee	Date
Gladys O. Ducoudray, Ph.D. Member, Graduate Committee	Date
Rogelio Palomera, Ph.D. Member, Graduate Committee	Date
Domingo Rodríguez, Ph.D. Member, Graduate Committee	Date
Ana C. Gonzalez, M.S. Graduate Studies Representative	Date
Pedro I. Rivera-Vega, Ph.D. Department Chairperson	Date

Abstract of Thesis Presented to the Graduate School
of the University of Puerto Rico in Partial Fulfillment of the
Requirements for the Degree of Master of Science

METHODS FOR SCALABLE LEVELS OF PARALLELISM IN RADIX-2 FFTS FOR FPGA SYNTHESIS

By

Felipe Minotta Zapata

May 2014

Chair: Dr. Manuel Jiménez
Department: Electrical and Computer Engineering Department

The Fast Fourier Transform (FFT) is the main block in many communication systems and signal processing applications, as it allows the fast computation of the discrete Fouier transform (DFT). The DFT, in turn, is used to obtain the spectrum of any finite discrete signal. Hardware implementations of this operation are highly regarded as they provide improved performance with respect to software-based implementations. The purpose of this work was developing a consistent and scalable procedure of generating the address patterns of permutation for any power-of-2 transform size and any folding factor in FFT cores with addressing schemes. Our approach was, mainly, based in 2β memory blocks, an address generator, and β radix-2 butterflies. The number β of butterflies determines the level of parallelism. The expected high performance of this FFT core lies in the fact it does not need dedicated permutation hardware between stages. Instead, the data flow is controlled by an address generator. Using this scheme, the impact on consumed resources is significantly mitigated when the number of points of the core is increased. As a result, we obtained a fully scalable FFT core including parallelism level, number of points, and numeric format using this approach.

Resumen de tesis presentado a la Escuela Graduada
de la Universidad de Puerto Rico como requisito parcial de los
requerimientos para el grado de Maestría en Ciencias

MÉTODOS PARA ESCALAR LOS NIVELES DE PARALELISMO EN FFTS DE BASE 2 PARA SÍNTESIS EN FPGA

Por

Felipe Minotta Zapata

Mayo 2014

Consejero: Dr. Manuel Jiménez
Departamento: Ingeniería Eléctrica y Computadoras

La Transformada Rápida de Fourier (FFT por sus siglas en inglés) es el bloque principal en muchos sistemas de comunicación y aplicaciones de procesamiento de señales, ya que permite la rapida computación de la Transformada Discreta de Fourier (DFT por sus siglas en inglés). Por su parte, la DFT es usada para obtener el espectro de cualquier señal discreta finita. Las implementaciones en hardware de esta operación son altamente apreciadas debido a que proveen mayor rendimiento con respecto a las implementaciones basadas en software. El propósito de este trabajo fue el desarrollar un procedimiento consistente y escalable para generar los patrones de direccionamiento de las permutaciones para cualquier tamaño de transformada potencia de 2 y cualquier factor de plegado en núcleos FFT con esquemas de direccionamiento. Nuestro diseño se basó, principalmente, en 2β bloques de memoria, un generador de direcciones y β mariposas base 2. El número β de mariposas determina el nivel de paralelismo. El alto rendimiento del núcleo radica en el hecho de que el flujo de datos es controlado por un generador de direcciones, el cual mitiga el consumo de recursos cuando se incrementa el número de puntos de la FFT. Como resultado, se obtuvo una implementación de FFT enteramente escalable incluyendo el nivel de paralelismo, número de puntos y formato numérico usando este enfoque.

To my family, specially to my mother Patricia, my father Francisco, and little sis Maria José, who have always given me their love, affection and support to keep going.

Acknowledgements

I would like to express my thanks to my advisor Professor Dr. Manuel Jiménez, thank you for encouraging my research and for allowing me to grow as a student, researcher, and professional. Also, thanks to my committee members, professor Dr. Domingo Rodríguez, professor Dr. Rogelio Palomera, professor Dr. Gladys O. Ducodray, for serving as my committee and for taking part in the review of my work. I would especially like to thank Sandy, the graduate academic counselor and friend, who guided and helped me through this experience in a new university and country. Last but not least, I want to thanks all my Puerto Rican and Colombian friends, who have encourage and supported me to move forward during good and bad moments.

Table of Contents

Abstract in English . . . . . **ii**

Abstract in Spanish . . . . . **iii**

Dedicated to... . . . . . **iv**

Acknowledgements . . . . . **v**

List of Tables . . . . . **viii**

List of Figures . . . . . **ix**

1 INTRODUCTION **1**

2 THEORETICAL BACKGROUND **2**

2.1 Discrete Fourier Transform (DFT) 2

2.2 The Fast Fourier Transform (FFT) 3

2.3 Pease FFT Factorization 5

3 PREVIOUS WORK **9**

3.1 FFTs with address generation 9

3.2 FFTs with dedicated data permutation logic 14

3.3 Summary 15

4 PROBLEM STATEMENT AND HYPOTHESIS **17**

4.1 Problem Statement 17

4.2 Hypothesis 17

5 OBJECTIVES **18**

5.1 General Objective 18

5.2 Specific Objectives 18

6 METHODOLOGY **19**

6.1 System Blocks and FFT Architecture 19

6.2 Number Representation Format 20
6.3 Arithmetic Unit Design 21
6.3.1 Complex Adder/Subtractor 22
6.3.2 Complex Multiplier 23
6.4 Memory Organization 24
6.5 Data Switch Design 26
6.5.1 Data Switch (Read) 26
6.5.2 Data Switch (Write) 28
6.6 Address Generation Schemes 29
6.6.1 Data Address Generator Design 31
6.6.2 Phase Factor Scheduler Design 34
7 RESULTS AND ANALYSIS 40
7.1 Core Validation 40
7.2 Timing Performance 41
7.3 Resource Consumption 44
7.4 Analysis and Comparisons 46
7.5 Limitations 47
8 CONCLUSIONS 49
9 CONTRIBUTIONS AND FUTURE WORK 50
Bibliography 52

List of Tables

3.1 Summary of reviewed works 16

List of Figures

2.1 Graphical representation of the operation in Eqs. 2.10 and 2.11 . . . 5

2.2 Bitreversal and stride-2 permutations example for $N = 8$ 6

2.3 8-point Pease FFT Architecture 7

2.4 8-point Pease FFT with horizontal folding 8

6.1 Basic Architecture 20

6.2 (a) Single Precision Floating Point (b) Double Precision Floating Point (c) Quadruple Precision Floating Point 21

6.3 Radix-2 Butterfly Structure 22

6.4 Complex Adder Architecture 23

6.5 Complex Multiplier Architecture 1 24

6.6 Complex Multiplier Architecture 2 25

6.7 Memory Access Process 25

6.8 (a) Bit Reversal Permutation. (b) Stride-2 Permutation. (c) Modified stride-2 Permutation. 27

6.9 Data Switch (Read) 28

6.10 Bit-reversal Permutation 28

6.11 Stride-2 Permutation 29

6.12 Modified Stride-2 Permutation 29

6.13 (a) First Permutation. (b) Second Permutation. 30

6.14 Data Switch (Write) 30

6.15 Data Addressing Sequence with $N = 16$ and $\beta = 1$ 32

6.16 Data Addressing Sequence with $N = 32$ and $\beta = 2$ 33

6.17 Data Addressing Sequence with $N = 32$ and $\beta = 4$ 34

6.18 How Stride-2 Permutation is calculated throughout the stages . . . 35

6.19 How Modified Stride-2 Permutation is calculated throughout the stages 36
6.20 Phase Factor Scheduling for $N = 32$ and $\beta = 1$ 37
6.21 Phase Factor Scheduling for $N = 32$ and $\beta = 2$ 37
6.22 Phase Factor Scheduling for $N = 32$ and $\beta = 4$ 38
6.23 Principal parameters behaviour of the phase factor scheduling for $N = 32$ 38
7.1 Structure used to validate the simulation of the FFT 41
7.2 Mean Percentage Error of our core compared with MATLAB 42
7.3 Clock Cycles Comparison 43
7.4 Maximum Working Frequency Comparison 43
7.5 Slice LUTs Consumption Comparison 44
7.6 Slice Register Consumption Comparison 45
7.7 DSP48 Consumption Comparison 45
7.8 Memory Usage Comparison 46
7.9 Computation Time 48

Chapter 1
INTRODUCTION

The Fast Fourier Transform (FFT) is a fundamental tool in signal processing and communication systems to obtain the frequency of signals. Hardware implementations of the FFT are highly regarded as they provide improved performance characteristic with respect to software-based sequential implementations. Developing an efficient hardware implementation represents a significant burden for hardware engineers today despite feel that an FFT algorithm can be easily understood.

A typical FFT core is composed of processing elements and a resulting permutation blocks. The processing elements are arithmetic blocks and the number of them is defined by the folding factor ϕ, which determine the level of parallelism in the implementation. The permutation block can be implemented with dedicated logic or with an addressing scheme. This document presents a method to design the permutation block when the folding factor is scaled in FFT cores implemented on FPGA using an address generation scheme. Our effort was centered in the development of a general addressing scheme that could perform the necessary permutations through the stages regardless the number of points and folding factor.

The following Chapter presents the theoretical foundations of this work. Chapter 3 presents a considerable number of hardware implementations of FFTs. Problem Statement and Objectives of this thesis are shown in Chapters 4 and 5 respectively. Chapter 6 shows complete design in hardware of our FFT core, and the outcome of our design including performance and consumed resources are presented in Chapter 7. Finally, we present the contributions and the future work in the remaining Chapter.

Chapter 2
THEORETICAL BACKGROUND

This chapter presents the underlying concepts behind our work. In the first part, we review the original Discrete Fourier Transform (DFT) formulation. Then, we explain the Fast Fourier Transform (FFT) Cooley-Tukey formulation [1] and its advantages. Afterwards, we explain why we chose the Pease factorization of the FFT. And, how we can modify the Pease architecture to obtain a trade-off between latency and hardware consumed resources. Finally, we explain why one would choose addressing generation schemes over dedicated permutation logic.

2.1 Discrete Fourier Transform (DFT)

The DFT is an operator that takes a finite-length sequence, representing a signal in the time domain, and transforms the signal to the frequency domain. The DFT of an arbitrary discrete signal $x[n]$, of length N, is given by

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi \frac{kn}{N}} \quad 0 \leq k \leq N-1, \tag{2.1}$$

where $X[k]$ is the signal in the frequency domain. Commonly, the DFT definition is also expressed by making the substitution

$$W_N = e^{-j2\pi \frac{kn}{N}}, \tag{2.2}$$

where the W_N values are commonly named phase or twiddle factors. This substitution allows us to represent the DFT as a matrix multiplication of the form:

$$\mathbf{X} = \mathbf{W}_N \times \mathbf{x} \tag{2.3}$$

where $\mathbf{x}$ is a column vector representing the signal in the time domain, $\mathbf{X}$ is a column vector representing the signal in the frequency domain, and $\mathbf{W}_N$ is a square matrix whose elements are given by Equation 2.2. Thus, we can represent:

$$\mathbf{X} = \begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ \dots \\ X[N-1] \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \dots \\ x[N-1] \end{bmatrix} \tag{2.4}$$

$$\mathbf{W}_N = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & W_N & W_N^2 & \dots & W_N^{N-1} \\ 1 & W_N^2 & W_N^4 & \dots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & \dots & W^{(N-1)(N-1)} \end{bmatrix} \tag{2.5}$$

2.2 The Fast Fourier Transform (FFT)

The algorithm for the DFT is considered computationally expensive as it requires $\mathrm{O}(N^2)$ operations. The development of the FFT algorithm by Cooley-Tukey in 1965 simplified the practical implementation of the DFT as it reduced the number of operations to $\mathrm{O}(N \log_2 N)$ [1]. The FFT formulations are based on the periodicity and symmetry properties of the Twiddle Factors, i.e. the elements of W_N in Equation 2.5, and the fact that an FFT of N points can be expressed as 2 FFTs of $\frac{N}{2}$ points. The linearity property allows for expressing the DFT as two different summations, one with the indexed even values and the other with indexed odd values, as shown in equation 2.6 and 2.7.

$$X[k] = \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r] W_N^{2rk} + \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r+1] W_N^{(2r+1)k} \tag{2.6}$$

$$X[k] = \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r](W_N^2)^{rk} + W_N^k \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r+1](W_N^2)^{rk} \tag{2.7}$$

Due to the symmetry in W_N, we have that $W_N^2 = exp(-j\frac{4\pi}{N}) = W_{N/2}$. Therefore, both summations correspond to two independent DFT implementations, the first one with the indexed even input values and second one with the indexed odd input values, as shown in Equation 2.8.

$$X[k] = \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r]W_{N/2}^{rk} + W_N^k \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r+1]W_{N/2}^{rk} \tag{2.8}$$

According to the Equation 2.8, it can be observed that the term W_N^k affecting the second DFT corresponds to a shift in the time domain in one sample. Which was expected, since the difference between a odd sample and a even sample is one. Finally, Equation 2.8 can be rewritten as,

$$X[k] = F[k] + W_N^k G[k], \tag{2.9}$$

where F[k] and G[k] are the DFT of the even indexed and odd indexed samples of signal $x[n]$ respectively. Equation 2.9 suggests that the calculation of an N-point FFT can be made making successive 2-points FFT partitions of the original input signal. Each 2-point FFT is simple as it is only composed of a multiplication, an addition, and a subtraction (Equation 2.10 and Equation 2.11) and is called a butterfly due to its graphical representation (Figure 2.1). It only takes to combine the solution of all 2-point FFTs for several stages to obtain the complete FFT

$$X[0] = x[0] + W * x[1] \tag{2.10}$$

$$X[1] = x[0] - W * x[1] \tag{2.11}$$

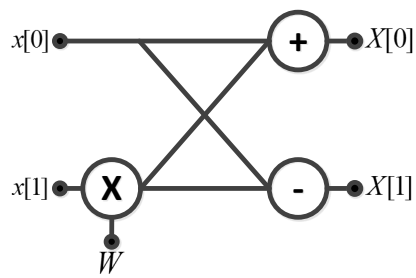

Figure 2.1 : Graphical representation of the operation in Eqs. 2.10 and 2.11

2.3 Pease FFT Factorization

After Cooley-Tukey [1], Pease [2], Korn and Lambiotte [3], and Stockham [4] performed different factorizations to the original formulation of the FFT. The Pease factorization is one of the most suitable for our purposes due to its repeating structure. For the development of the Pease algorithm, only two permutations are needed. Figure 2.2 shows how the permutations are performed for $N = 8$. The first permutation is a bit-reversal and is only performed at the beginning of the algorithm. This permutation can be thought of as doing a mirror operation on the $\log_2(N)$-bit word representing the index of every value of the signal. The second permutation is called a Stride-2 permutation which is performed in the rest of the algorithm. A Stride-2 permutation separates the signal values into two groups one with even indexed values and the other with odd indexed values.

Figure 2.3 shows the structure of an 8-point Pease FFT. The rectangles contain operating elements forming the *Butterflies*. Figure 2.3 also shows the possibility of doing a horizontal and/or vertical foldings. A horizontal folding consists on reducing the number of columns, while a vertical folding applies the same process to the rows. If the original structure were implemented, the latency of the core would be $Tb \log_2(N)$, where Tb is the latency of a single butterfly. If we do a complete horizontal folding, we would have one column of butterflies and we would need a block to perform the permutations, as shown in the Figure 2.4 . The permutation block controls the data

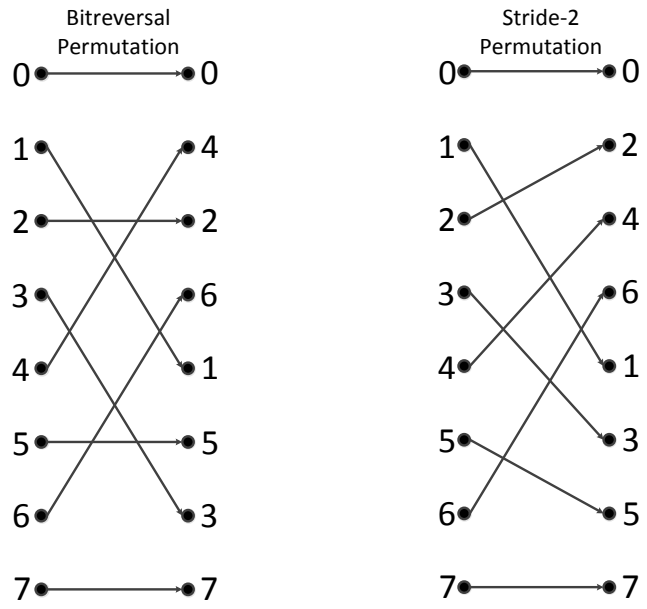

Figure 2.2 : Bitreversal and stride-2 permutations example for $N = 8$

flow between the stages. A structure with these characteristics would have a latency also depending on the number of points N and the vertical folding factor ϕ. This factor consists in using one column of β ($\beta = N/(2\phi)$) butterflies depending of the level of parallelism desired for the design. Because each butterfly accepts two points at the same time, the maximum number of β is $N/2$. This approach leads to an increase in latency but uses less arithmetic hardware resources. For this reason, a typical FFT core is implemented with a full horizontal folding and the vertical folding factor is scaled depending on the desired parallelism level. This is very important because it has a direct impact in the total latency and the consumed resources. For small values of β, the latency is high and the consumed resources are low. On the contrary, for large values of β, the latency is lower and the consumed resources are higher.

To conclude the explanation, the permutation block is implemented using dedicated permutation logic or through emulation with an addressing scheme. The former uses logic blocks to permute the data, which consumes a significant amount

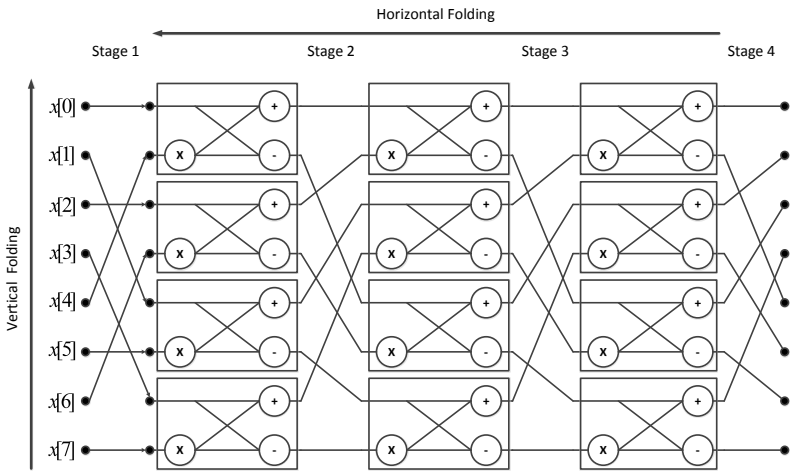

Figure 2.3 : 8-point Pease FFT Architecture

of hardware resources for numerous points, while the latter indirectly performs the permutations by addressing the data from multi-bank memories. This represents an enhancement in design automation and implementation. Previous works using the addressing scheme, which are presented in the Chapter 3, have shown that this is a resource-efficient implementation, which is why our work is based on this approach.

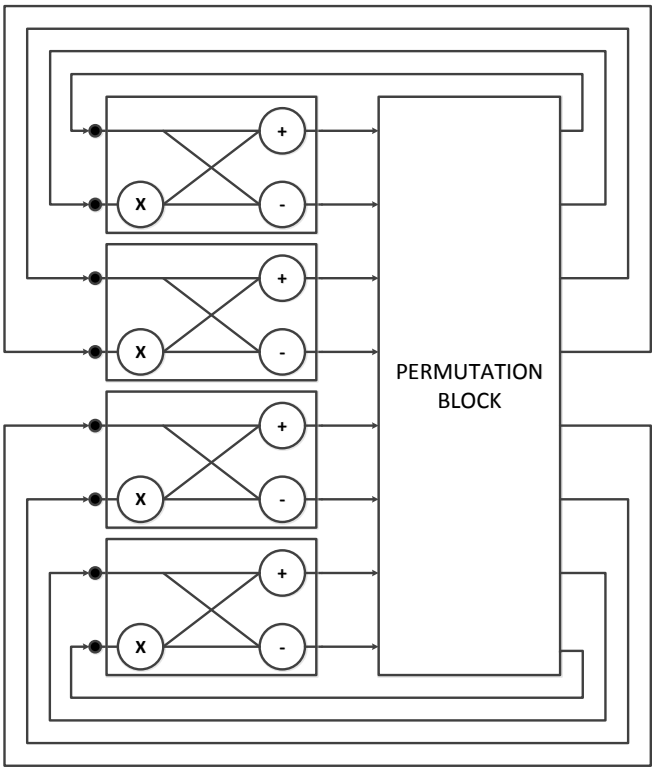

Figure 2.4 : 8-point Pease FFT with horizontal folding

Chapter 3
PREVIOUS WORK

In this Chapter, a relevant hardware implementations of FFT documented in recent literature are discussed. The approaches reviewed here can be classified in two groups: FFT designs based on address generation schemes and dedicated permutation logic. In the first group, data are stored in memories and a block generates the addresses depending the permutation needed. In the second group, there is a dedicated logic to control the data flow. In this Chapter, we make a review of the works considered more relevant in this kind of approaches.

Section 3.1 summarizes the characteristics for address generation FFTs, followed in Section 3.2 by a discussion of different dedicated permutation logic FFTs. The Chapter concludes with a summary of the most important characteristics of the discussed methods.

3.1 FFTs with address generation

Johnson described an address generation scheme for decimation in time and decimation in frequency for radix r FFTs [5]. The author explains how to generate addresses using shifters and adders for both data and phase factors. The method uses multi-bank memories, such that all data needed at a given butterfly could be accessed at once. It uses conflict-free addressing by writing into the same locations being read. The proposed addressing allowed to keep the data storage requirement at N locations. This is a theoretical work, with no hardware implementation. From

the point of view of consumed resources, this design was suitable for large values of N.

Wang, et al, designed and implemented a 64-point FFT to meet the requirements of Wireless Local Area Network (WLAN) [6]. The design was based on the radix-2 decimation in frequency algorithm. The hardware required for complex multiplication at the butterfly was reduced using a convenient factorization that allowed for using only three real multipliers. The author uses a conflict-free memory addressing scheme for minimum memory requirements. The design also took advantage of the property of reusability of the phase factors so it only required storage for 11 of them. The design was not scalable and the complete FFT calculation could be completed after 76 clock cycles at a maximum frequency of 31.69 MHz on an Altera's Cyclone II device.

Polychronakis, et al, presented a parallel addressing technique for radix-2 decimation in frequency FFT architectures [7]. The processor used a single memory with $N/2$ location, which stored two FFT elements at each address. Address generation was specified to access data throughout the stages. However, there is no documentation of how the twiddle factors were generated, and the numeric format used. An example of a 256-point FFT with 12 bits real part and 12 bits imaginary were provided. The entire processor occupied 213 slices and 6 of 48 DSP48Es of a Xilinx Virtex XC5VLX50T-3.

Gautam, et al, designed and implemented a scalable radix-2 FFT core for OFDM applications [8]. The processor used conflict free in-place memory for intermediate data storage. The address generator was scalable to support up to 2^{13} points. The twiddle factors were generated using CORDIC (CO-ordinate Rotational Digital Computer). An 8-point FFT was implemented on a Xilinx Virtex-5 FPGA "xc5vlx100t-3-ff1136". The authors reported a 200 MHz as a maximum allowed frequency for this core.

Hongxia and Shitan presented an addressing scheme for twiddle factors in an FFT with mixed-radix based on multi-bank memory [9]. The memory for twiddles were improved using the quarter of the locations due to reusability. The design was based on a look-up table to stores the twiddle factors by a maximum value of N and the address generator was capable of addressing for values less than or equal of N. However, since the lookup table is the same for any number of points, implementations with a low number of points has a large memory consumption.

Chad, et al, presented a design method for a real-time decimation in frequency radix-4 1024 point FFT processor [10]. The processor worked in fixed point and used an overflow controller which controls dynamically shift of fix point operand according to the result of the butterfly. The author also showed a method of address mapping and generation with in-place memory addressing strategy. The twiddle factors were generated and the complex multiplications for this operation were implemented using a lifting scheme that allowed for using three real multipliers. The system was capable of completing a radix-4 butterfly per clock cycle. The maximum supported frequency was 127 MHz and execution time of 10.1μ.

Szedo, et al, proposed a 16-bit, 1024-point, radix-4 decimation in frequency, radix-4 FFT core [11]. The system used two butterflies, the first one for the first $\log_4(N) - 1$ ranks and a second one multiplication free for the last rank. The core used 4 memory banks of four dual port block memories and took exactly 1024 clock cycles to process data continuously. The maximum frequency was 100 MHz and consumed 2593 logic slices, 12 multipliers and 22 block RAMs.

Ramesh, et al, evaluated the use of address generation algorithms for accelerating the execution of DSP Kernels [12]. In their article they showed the implementation of address generation units for accessing data in bit reversed order for FFTs and in zig-zag order for Discrete Cosine Transform (DCT). To illustrate the algorithm, the authors gave an example in which they implemented an 8-point FFT and the address

generation unit for twiddle factors was also developed. They completed the 8-point FFT in 28 cycles. The address generation scheme produced a single data address at a time. Using the proposed scheme they claimed to complete a N-point FFT in $N \log_2 N$ cycles.

Jiang, et al, designed a radix-2 FFT algorithm to reduce the frequency of memory accessed as well as multiplication operations [13]. This achievement was made by reusing the phase factor during the calculation of the transform. The reduction of the frequency of memory accessed was achieved using temporal registers to store the twiddle factors while they are needed. The reduction of the multiplication operations was accomplished taking advantage of several twiddle factors that are equal to 1. With the proposed approach, the required memory accesses due to phase factors was reduced to $N/2 - 1$.

Tsai, et al, proposed an address generation scheme for multiple processing units [14]. It specified an addressing scheme for data values but it did not account for twiddles addressing. Fixed and mixed radix operations were possible. The method completed calculations in $N/2 \log_8 N$ cycles and required 2N data locations for a size N transform. The work did not describe any implementation on FPGA or ASIC.

Xiao, et al, proposed a method for designing the address generator with reduced logic [15]. The address generator avoids the parity checkers an barrel shifters and is primarily based on inverters, counters, and multiplexors. The system used the minimum memory requirements for in-place operation. As case study they synthesized a 16-point FFT with 32 bit complex number using CMOS $0.18\mu m$ technology. For a 65536-point FFT the maximum frequency supported by the address generator was 629 MHz.

Polo, et al, designed a scalable fixed point FFT core for FPGA synthesis [16]. Their approach exploited the structural regularity from the Kronecker formulation to perform a complete folding of the transform. Algorithms for producing the address

sequences and phase factor scheduling were provided. The core was compared with the one developed by Xilinx and the results show an improvement of nearly 40% in slices, 6% in memory used, and 7% in latency.

Shome, et al, proposed an architectural design for a highly programmable radix-2 Decimation-in-Frequency FFT processor [17]. The design supported FFT sizes from 64 to 1024 points. The system used 5 different dual-port memories. The data were stored in the first memory. The second and third memories were used for the calculation of the FFT, and the fourth and fifth ones were used for the final data reordering. The system described were only an Address Generation Block for the data and phase factor. No results about consumed hardware resources and latency were reported.

Yang, et al, implemented a memory based radix-2 Decimation-in-frequency FFT processor with address generation [18]. The architecture used 4 single port memories instead of two dual-port memories. The system performed an N-point FFT in $\log_2 N+1$ clock cycles. The twiddle factors were stored in a ROM but the addressing scheme were not specified.

Wey, et al, proposed a radix-2 memory-based FFT design suitable for OFDM applications which used an address generator approach to perform data permutation [19]. The architecture used single port instead dual port memories for area saving. The design achieved reduced resource consumption and required storage for N data words using 24 bit width in fixed point. However, no address generation scheme was specified for phase factors and they were stored in a ROM. The design was synthesized to an ASIC and had a maximum operating frequency of 198 MHz and a latency of 55296 cycles for a 8192-point FFT.

Frias, et al, developed a 1024-point decimation in time radix4 FFT VHDL core [20]. The developed core was implemented on a Xilinx Spartan-3 XC3S200 FPGA by taking advantage of the FPGAs low cost. The FFT calculation was reported

to run in 7680 cycles with a 50MHz master clock. The processor worked in fixed point. The computation time was 153.84μs. Although it had lower performance than commercially available cores from Xilinx, it had the advantage of using fewer on-chip resources, making it feasible to be implemented in lower cost FPGAs like Spartan-3.

Ayinala, et al, developed a scalable architecture for in-place Fast Fourier Transform computation for real valued signals [21]. The proposed computation was based on a modified radix-2 algorithm, which removed the redundant operations from the flow graph. The architecture used two radix-2 butterflies to process four inputs in parallel. Address generation was specified to access data throughout the stages but there was no explanation of how the twiddle factors were generated. In the article, they showed the latency for different number of points but they did not specify the numeric format. The address generation was extended to support multiple processing elements and it had 12 different addressing patterns depending on the stage.

3.2 FFTs with dedicated data permutation logic

Babionitakis, et al, presented an implementation of a 4096-point radix-4 complex FFT on a Virtex II FPGA and a VLSI chip [22]. The input values of the architecture were expressed in fixed point. The maximum frequency was 200MHz and the throughput was of 4096/20.48μs for the FPGA. The VLSI implementation achieved a throughput of 4096/3.89μs and a worst case post-route frequency of 604.5MHz using a 0.13μm process.

Yang, et al, proposed an FFT processor suitable for MIMO-OFDM based SDR systems [23]. Synthesized using a 0.13μm standard cell library, the design supported 64, 128, 512, 1024 and 2048 point transform lengths. The processor used mixed radix algorithms which minimized the number of non-trivial multiplications. The core supported computation of the transform on four channels simultaneously. The design improved over a previous 4-channel radix-2 multiple delay commutator based

design both in memory and logic resource consumption, achieving 16.4% and 26.8% reduction respectively.

Montaño, et al, designed an scalable floating point FFT core for Xilinxs FPGAs [24]. The scalable parameters included size, word length and folding factor. The data permutations were made using an array of switches. This permutation hardware did not allow for sizes beyond 64 points when targeting a Xilinx Virtex IV device. Phase factors scheduling had no re-use, thus requiring full tabulation and logic resources for their storage instead of a ROM.

Chen, et al, proposed a permutation network for configurable and scalable FFT processors [25]. It consisted of several independent RAM blocks and two interconnection networks, capable of operating in a pipeline fashion. The authors claimed their method achieved a high level of parallelism, thus high throughput, for sizes ranging from 2 to 8192 points, obtained by implementing various computational stages and permutation networks together. The reported results regarding the performance and logic consumption were not clear since the authors did not specify under which parameters, procedures and technology the numbers were obtained.

3.3 Summary

Table 3.1 summarizes the works reviewed and their relevant aspects. None of the reviewed works address an architecture with scalable folding factor which is the main scope of this proposal. It can be seen that only one work deals with floating point, twelve with fixed point, and five others do not specify number format. Our approach is expected to work with any numeric format.

Table 3.1 : Summary of reviewed works

Author	Number Format	Scalable	Address Generation	Radix
Wang [6]	Fixed Point	No (64p)	□ △	2
Polychronakis [7]	Fixed Point	No (256p)	□	2
Gautam [8]	Fixed Point	Yes	□	2
Hongxia [9]	N/S	Yes	△	Mixed
Chad [10]	Fixed Point	No (1024p)	□	4
Szedo [11]	Fixed Point	No (1024p)	□	4
Ramesh [12]	Fixed Point	Yes	□ △	2
Jiang [13]	N/S	Yes	□	2
Tsai [14]	N/S	Yes	□ △	2^q
Xiao [15]	Fixed Point	Yes	□	2
Polo [16]	Fixed Point	Yes	□ △	2
Shome [17]	N/S	Yes	□ △	2
Yang [18]	N/S	Yes	□ △	2
Wey [19]	Fixed Point	Yes	□	2
Frias [20]	Fixed Point	No (1024p)	□	4
Ayinala [21]	N/S	No	□	2
Babionitakis [22]	Fixed Point	No (4096p)	No	4
Yang [23]	Fixed Point	Yes	No	2
Montaño [24]	Floating Point	Yes	No	2
Chen [25]	N/S	Yes	None	2

Legend: □ :Data Address Generation △ :Twiddle Address Generation

Chapter 4
PROBLEM STATEMENT AND HYPOTHESIS

4.1 Problem Statement

The problem addressed in this thesis is that of generating the address patterns of permutations when the folding factor is scaled in FFT cores. To the best of our knowledge, although there have been implementations for specific folding factors, no generalized approaches have been reported. This generalized addressing mode allows scaling the level of parallelism in a FFT processor, which directly impacts the latency of the synthesized cores.

4.2 Hypothesis

Our work was proposed under the hypothesis that it is possible to develop a general rule of addressing regardless the folding factor of the structure. Furthermore, with this addressing, the latency is expected to decrease up to 2 times for every parallelism level without impacting significantly the consumed hardware resources.

Chapter 5
OBJECTIVES

This section describes the objectives that have been formulated for the proposed work.

5.1 General Objective

To develop a method to design the permutation block when the folding factor is scaled in FFT cores based on an address generation scheme for FPGA implementation.

5.2 Specific Objectives

1. Determining which FFT algorithm offers best regularity for the general addressing for the data flow and the phase factors.
2. Identifying a general rule governing the pattern of the address sequence for data point and phase factor regardless the folding factor.
3. Designing and implementing an HDL model to test the functionality of the addressing sequence.
4. Analysing how architectural changes affect the latency and the hardware consumed in the HDL model when the folding factor is scaled.

Chapter 6
METHODOLOGY

An FFT core might be designed for low latency or low hardware resources consumed depending on the application. The two types of design objectives can be achieved augmenting the level of parallelism for low latency, or reducing the level for low hardware resources consumed. The methodology established in this work is based on developing a scalable addressing scheme for both, data reordering and phase factor scheduling. In order to succeed in this task, it is necessary to develop a general addressing rule that can perform the two permutations through the stages regardless the number of points and folding factor.

In this Chapter, we first give a brief introduction to the general architecture of our core, including the main purpose of each block. In the latter sections, we explain in detail the basic functioning of each block and the respective hardware implementation.

6.1 System Blocks and FFT Architecture

As we explained in Chapter 2, the core is composed, mainly, of an arithmetic unit and and permutation block. The former can be one or more butterflies depending the folding factor. And the latter has a memory bank, a data addresser, and data switches. Other blocks equally important are the phase factor scheduler and the control unit. This approach can be classified as Radix-2 because it is based on a Radix-2 butterflies. Also can be classified as a Burst I/O design as the user must wait for the complete FFT calculation to finish before providing a new input signal.

Figure 6.1 shows the basic architecture of the core. The address generator is in charge of generating the addressing patterns to the memory bank, which is the data to be processed. The twiddle addresser generator produces the addressing patterns to the Twiddle memory in order to provide the correct phase factor to each butterfly. The read and write switches permutate the data while they are being read and written. The processing units perform the arithmetic operations over the data to be processed. Finally, the control unit generates the control signal to every block to ensure a correct functioning.

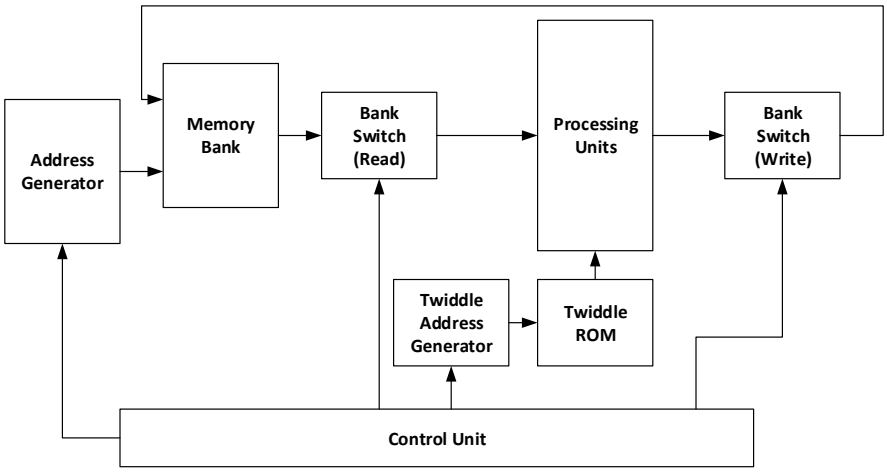

Figure 6.1 : Basic Architecture

6.2 Number Representation Format

Our core is entirely based on Floating Point Arithmetic. The main advantage of Floating Point arithmetic over fixed point is the dynamic range for accommodating extremely large numbers and high precision for very small numbers. This helps to alleviate the underflow and overflow problems often seen in fixed-point formats [26]. Furthermore, this numeric format has constant bit-word size, which helps the implementation. Floating point uses a sign-magnitude representation, where the magnitude is obtained by the multiplication of a fractional and an exponential term.

The IEEE-754 is the most commonly used standard for Floating Point representation. In theory, our core supports any floating point format, only limited by the FPGA resources. Figure 6.2 shows an example of the quantities we can represent in our core. Figure 6.2 a. shows a single precision floating point representation, the standard specify 8 bits for the exponent and 23 bits for the mantissa. Figure 6.2 b. shows a double precision floating point representation, which uses 11 bits for the exponent and 52 bits for the fractional part. Figure 6.2 b. shows a quadruple precision floating point representation, which offers results more reliably and accurately by minimising overflow and round-off errors in intermediate calculations since it uses 15 bits for the exponent and 112 bits for the mantissa.

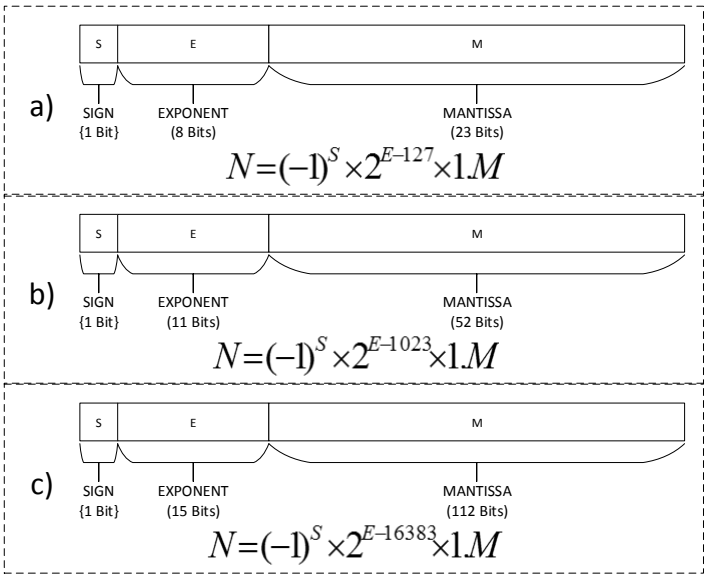

Figure 6.2 : (a) Single Precision Floating Point (b) Double Precision Floating Point (c) Quadruple Precision Floating Point

6.3 Arithmetic Unit Design

As explained earlier, the FFT calculation is performed around a basic arithmetic unit called a Butterfly. This unit consists of a complex adder/subtractor, and a complex multiplier. For the hardware implementation, floating point was used to

represent the numbers. Every unit was optimized to work is this numeric format. Due to the complexity of the involved operations, the most efficient architecture to implement the Butterfly is a pipeline architecture. The resulting butterfly had an initial non-zero latency of 20 cycles, after which it was capable of producing a pair of outputs per clock cycle. Figure 6.3 shows the architecture of the complex radix-2 butterfly.

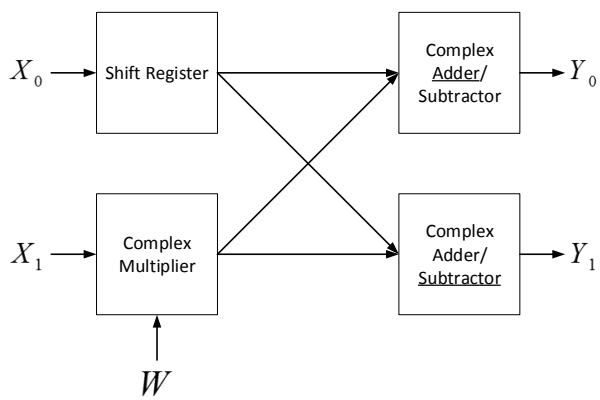

Figure 6.3 : Radix-2 Butterfly Structure

6.3.1 Complex Adder/Subtractor

The addition/substraction of any two complex numbers Z_1, Z_2, where $Z_1 = X_1 + jY_1$ and $Z_2 = X_2 + jY_2$, is performed as indicated by Equation 6.1. Thus a Complex Adder/Subtractor requires two floating point adders, the first to add the real part and the second to operate the imaginary part.

$$Z = Z_1 + Z_2 = (X_1 + X_2) + j(Y_1 + Y_2) \tag{6.1}$$

Figure 6.4 shows the diagram of a complex adder. Also, in the Figure it can be seen that the latency of the complex adder is the same as the floating point adder, in our case to five cycles.

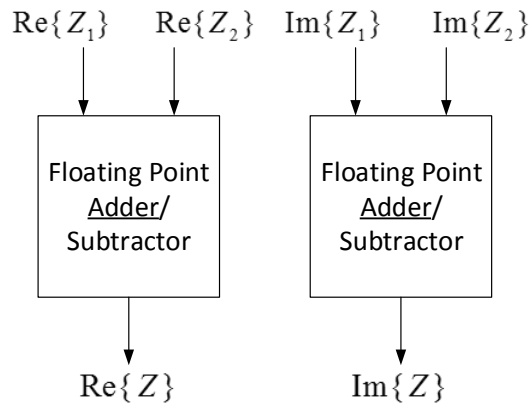

Figure 6.4 : Complex Adder Architecture

6.3.2 Complex Multiplier

The multiplication of any two complex numbers Z_1, Z_2, where $Z_1 = X_1 + jY_1$ and $Z_2 = X_2 + jY_2$, can be performed as indicated by Equations 6.2 and 6.3.

$$\begin{aligned} Z &= (X_1 + jY_1) \times (X_2 + jY_2) \\ \mathrm{Re}\{Z\} &= X_1 \times (X_2 + Y_2) - Y_2 \times (X_1 + Y_1) \\ \mathrm{Im}\{Z\} &= X_1 \times (X_2 + Y_2) - X_2 \times (X_1 - Y_1) \end{aligned} \tag{6.2}$$

$$\begin{aligned} \mathrm{Re}\{Z\} &= X_1 \times X_2 - Y_1 \times Y_2 \\ \mathrm{Im}\{Z\} &= X_1 \times Y_2 + Y_1 \times X_2 \end{aligned} \tag{6.3}$$

The result shown in Equation 6.2 requires three multipliers and five adders. And the result shown in Equation 6.3 requires four multipliers and two adders. Figure 6.5 and 6.6 show the two different architectures. Since our floating point adder consumes approximately twice more resources and has the half of the latency of the multiplier, we chose the first architecture because the second consumes 13/8 times more resources and it spends 5 cycles less than the first one.

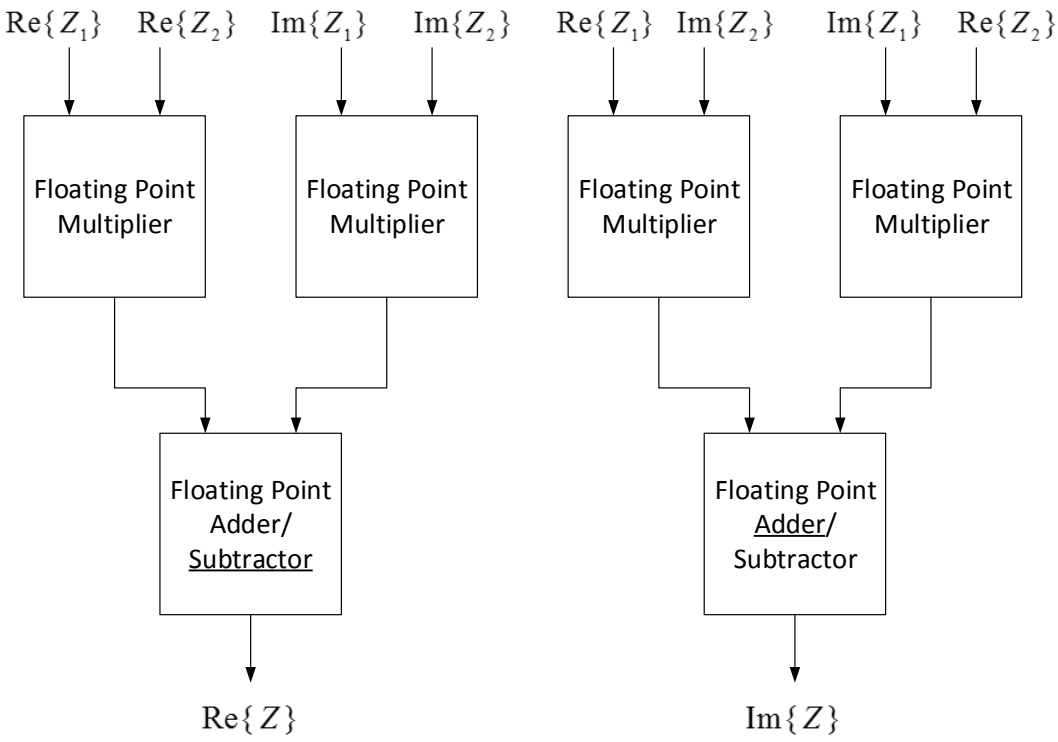

Figure 6.5 : Complex Multiplier Architecture 1

6.4 Memory Organization

Two different memory units were needed in this implementation. The first stores the signal in the time domain, the intermediate data during calculation, and the signal in the frequency domain. The second stores the phase factors.

Since every butterfly needed two different values at the same time, the data memory in our design consisted of a bank of 2β memory blocks. For simplicity, the signal in the time domain was first stored in the same order it was produced. Since our implementation was intended to use an "in place" strategy [5], every memory had $N/(2\beta)$ locations. The memory access process using this strategy consisted on first fetching the data, processing them, and then writing them into the same locations. This process is depicted in Figure 6.7 . Since the write and read operation may

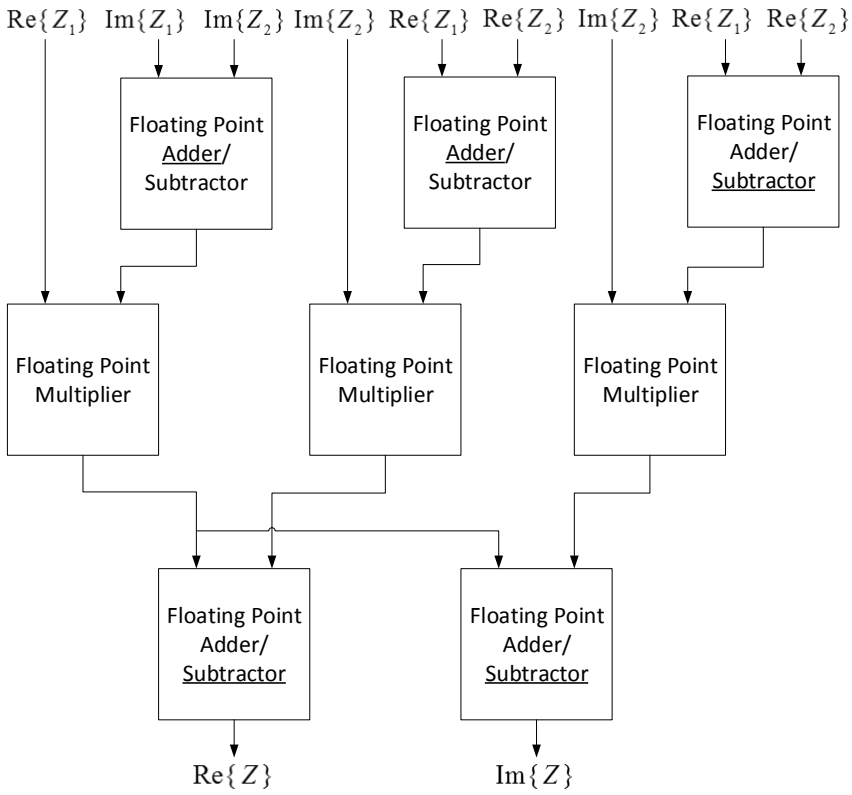

Figure 6.6 : Complex Multiplier Architecture 2

overlap, every memory unit had two different address ports to allow for this operations to be performed simultaneously.

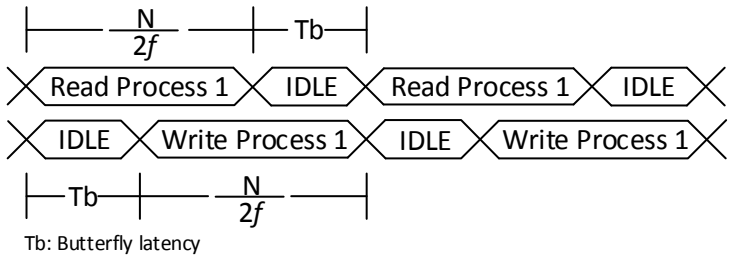

Figure 6.7 : Memory Access Process

The phase factor memory size is different depending on the folding factor. As every butterfly needs a different phase factor, when the structure is completely folded,

a single port memory is used. When the structure is not completely folded, a dual-port memory is used to minimize the memory usage. Regarding to the memory usage, we took advantage of the periodicity and symmetry properties of the phase factor. Therefore, only $N/4$ numbers were stored instead of $N/2$ as an FFT implementation normally requires [16]. Hence, the phase factor required for a FFT of N points was,

$$W_N = e^{-2\pi jk/N} \quad 0 \leq k \leq N/4 - 1, \tag{6.4}$$

and the other $N/4$ number needed were obtained using Equation 6.5

$$W_N^{k+N/4} = \mathrm{Im}\{W_N^k\} - j\mathrm{Re}\{W_N^k\} \tag{6.5}$$

6.5 Data Switch Design

The main task of the data switches was to route the data between the memories and the arithmetic units. The data address generator is in charge of generating the correct addresses, but the data is not always in the order required by the arithmetic units. The Read Switch reorders the data as required by the arithmetic units. The Write Switch reorders the data in such a way that all the points to be fetched in the next stage be in different memories.

6.5.1 Data Switch (Read)

The Read Data Switch is located between the memory bank and the butterflies. This block partly contributed to performing the permutation between the stages, specifically, when data were being read. The behaviour described here is consistent for any folding factor. Figure 6.8 shows the three different permutations to be performed during calculation when four butterflies are used. Figure 6.8 a. shows the one which is performed in the first stage and is called a bit reversal permutation. Figure 6.8 b. and c show the ones which are performed in the rest of the stages in the first and

second half, respectively. These are called stride-2 and modified stride-2 permutation, respectively.

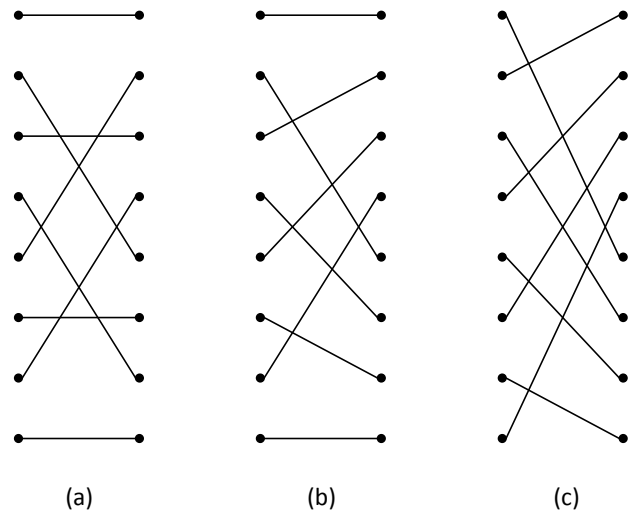

Figure 6.8 : (a) Bit Reversal Permutation. (b) Stride-2 Permutation. (c) Modified stride-2 Permutation.

In hardware, this switch is implemented as an array of multiplexers like that shown in Figure 6.9 . As can be seen in the Figure, this block has $2\beta - 1$ data inputs and outputs, allowing to make any permutation over the data.

To perform the permutations, every multiplexer must choose the correct input. The three different sequences are generated with bitwise operations over a normal ordered sequence, like circular shifting and simple logical operations. Figures 6.10 , 6.11 , and 6.12 show how to perform the Bit-reversal permutation, Stride-2 Permutation, and Modified Stride-2 Permutation, respectively. A bit-reversal sequence is obtained reversing the binary representation of a normally ordered sequence. A Stride-2 permutation is obtained making a left circular shifting in one position. And a Modified Stride-2 Permutation is obtained making a left circular shift by one position too, but negating the least significant bit.

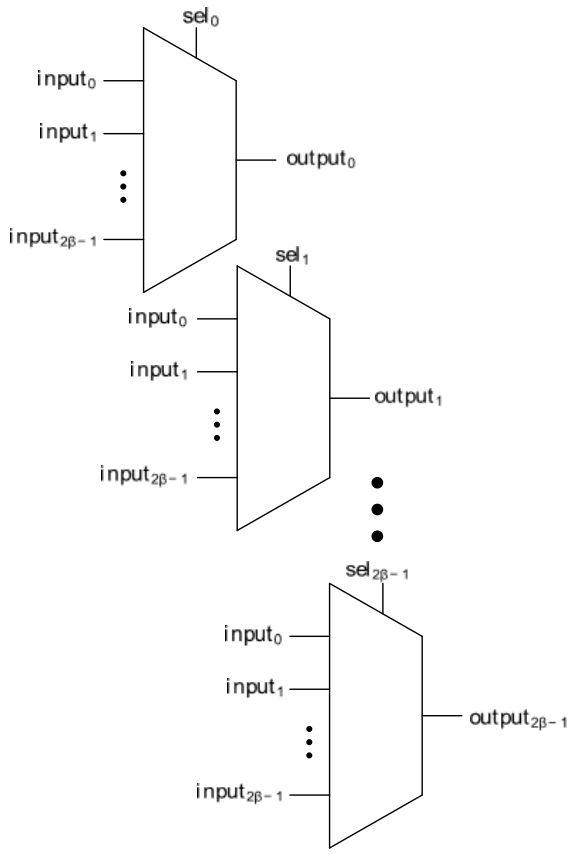

Figure 6.9 : Data Switch (Read)

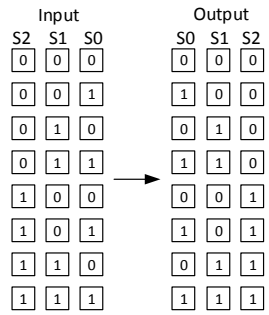

Figure 6.10 : Bit-reversal Permutation

6.5.2 Data Switch (Write)

The Write Data Switch is the one located between the butterflies and the memory bank. This block also partially contributes to performing of the permutation between

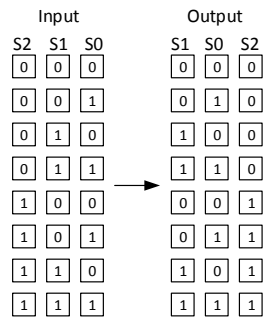

Figure 6.11 : Stride-2 Permutation

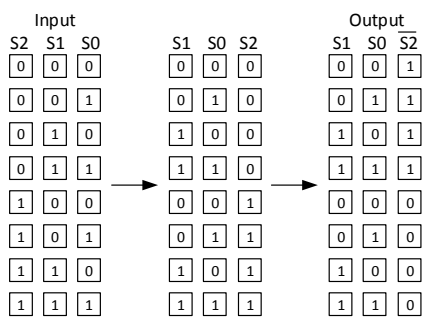

Figure 6.12 : Modified Stride-2 Permutation

stages, specifically, when data are being written. Figure 6.13 shows the two different permutations that need to be performed during the calculation. Figure 6.13 a. shows the one performed on the even indexed butterflies of every stage. Figure 6.13 b shows the one which is performed in the odd indexed butterflies of every stage.

Since every output can have only two values, it is no necessary to include all input values in the multiplexers, just 2. Figure 6.14 shows the basic architecture of this Data Switch.

6.6 Address Generation Schemes

A scalable folding of a Pease structure requires two major considerations: the ability of folding the data permutation completely or partially and the ability of

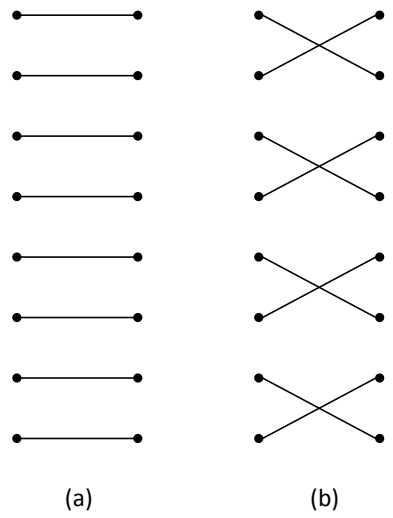

Figure 6.13 : (a) First Permutation. (b) Second Permutation.

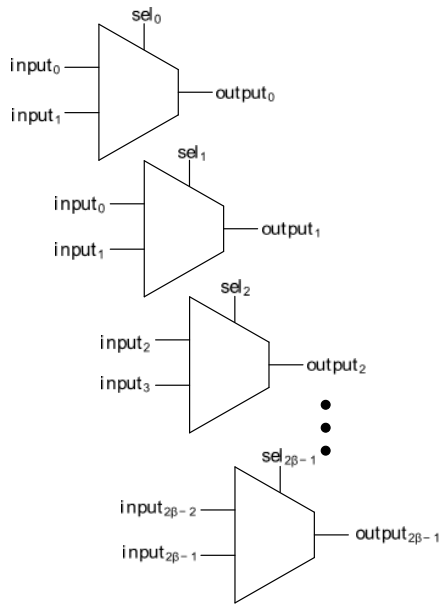

Figure 6.14 : Data Switch (Write)

producing the correct phase factor schedule at each stage. These task were performed by two address generators which are explained next.

6.6.1 Data Address Generator Design

The Data Address Generator is one of the most important blocks in this work and is the one that has the highest complexity. First at all, this block had to perform Bit-reversal permutation in the first stage and Stride-2 permutation in the next stages. Also, the design had to be scalable to support different sizes and any folding factor. Furthermore, these permutations had to be performed assuring the minimum requirements of memory. The only way to guarantee this requirement was by writing into the same locations being read.

Since 2β memories were needed, the same number of simultaneous addresses had to be generated. Nonetheless, for any folding factor there were only two different sequences. Therefore, the hardware implementation only required two different address generator blocks.

In the work developed by Polo, et al, the first access and the data arrangement depended on the number of butterflies [16]. For example, with one 1 butterfly, the data in the two memories had to be arranged in stride-2 organization and the data was accessed in normal order. With two butterflies, the data in the four memories had to be arranged in stride-4 organization and the data was accessed in normal order. To introduce regularity to the system, we proposed to arrange the data in normal order in all cases and the first access to each memory made it in bit-reversal order. Thus, the first permutation needed was performed automatically and the first access was always the same.

Figures 6.15 to 6.17 show the addressing sequences for different numbers of butterflies and transform sizes. These sequences were obtained running a core emulation in MATLAB. By analysing these sequences, it can be seen that the addressing of the even indexed memories are obtained making a Stride-2 Permutation of the previous sequence and of the odd indexed memories are obtained making a modified Stride-2 Permutation of the previous sequence. This addressing sequence applied for

any number of butterflies and any power-of-2 transform size. Following the last statements, we can formally express mathematically the address patterns, that is, for an N-point FFT with vertical folding factor ϕ the addressing sequences can be expressed as:

$$X_{i+1} = L_2^{2^{\frac{\phi}{2}}} X_i \quad X_0 = R_{2^{\log_2(\phi)}} \begin{bmatrix} 0 \\ 1 \\ 2 \\ \vdots \\ \phi - 1 \end{bmatrix} \quad 0 \leq i \leq \log_2(N) - 1 \tag{6.6}$$

$$Y_{i+1} = \begin{bmatrix} \emptyset & I_{\frac{\phi}{2}} \\ I_{\frac{\phi}{2}} & \emptyset \end{bmatrix} L_2^{2^{\frac{\phi}{2}}} Y_i \quad Y_0 = R_{2^{\log_2(\phi)}} \begin{bmatrix} 0 \\ 1 \\ 2 \\ \vdots \\ \phi - 1 \end{bmatrix} \quad 0 \leq i \leq \log_2(N) - 1 \tag{6.7}$$

where X_i is the addressing sequence of the even indexed memories, Y_i is the addressing sequence of the odd indexed memories, i is the stage identifier, and L_n^{nm} and R_{2^n} is a stride permutation and a bit-reversal operation respectively [27].

ST1	M0	0	4	2	6	1	5	3	7
	M1	0	4	2	6	1	5	3	7
ST2	M0	0	2	1	3	4	6	5	7
	M1	4	6	5	7	0	2	1	3
ST3	M0	0	1	4	5	2	3	6	7
	M1	6	7	2	3	4	5	0	1
ST4	M0	0	4	2	6	1	5	3	7
	M1	7	3	5	1	6	2	4	0

Figure 6.15 : Data Addressing Sequence with $N = 16$ and $\beta = 1$

ST1	M0	0	4	2	6	1	5	3	7
	M1	0	4	2	6	1	5	3	7
	M2	0	4	2	6	1	5	3	7
	M3	0	4	2	6	1	5	3	7
ST2	M0	0	2	1	3	4	6	5	7
	M1	4	6	5	7	0	2	1	3
	M2	0	2	1	3	4	6	5	7
	M3	4	6	5	7	0	2	1	3
ST3	M0	0	1	4	5	2	3	6	7
	M1	6	7	2	3	4	5	0	1
	M2	0	4	2	6	1	5	3	7
	M3	6	7	2	3	4	5	0	1
ST4	M0	0	4	2	6	1	5	3	7
	M1	7	3	5	1	6	2	4	0
	M2	0	4	2	6	1	5	3	7
	M3	7	3	5	1	6	2	4	0
ST5	M0	0	2	1	3	4	6	5	7
	M1	3	1	2	0	7	5	6	4
	M2	0	2	1	3	4	6	5	7
	M3	3	1	2	0	7	5	6	4

Figure 6.16 : Data Addressing Sequence with $N = 32$ and $\beta = 2$

The hardware implementation of this block consisted mainly of a single counter in normal order and two units which modify the counter values. There were two units due to it was needed one for the even indexed memories and another one for the odd indexed memories. The even indexed memories were addressed performing Stride-2 permutations only. Thus, the Stride-2 permutation was calculated throughout the stages just making a left circular shift over the previous sequence. An example of how to generate the entire sequence is shown in Figure 6.18 . Similarly, the odd indexed memories were addressed performing Modified Stride-2 permutation. This Permutation was calculated throughout the stages by making a left circular shift and negating the least significant bit over the previous sequence. An example of

ST1								ST2							
M0	M1	M2	M3	M4	M5	M6	M7	M0	M1	M2	M3	M4	M5	M6	M7
0	0	0	0	0	0	0	0	0	2	0	2	0	2	0	2
2	2	2	2	2	2	2	2	1	3	1	3	1	3	1	3
1	1	1	1	1	1	1	1	2	0	2	0	2	0	2	0
3	3	3	3	3	3	3	3	3	1	3	1	3	1	3	1
ST3								ST4							
M0	M1	M2	M3	M4	M5	M6	M7	M0	M1	M2	M3	M4	M5	M6	M7
0	3	0	3	0	3	0	3	0	1	0	1	0	1	0	1
2	1	2	1	2	1	2	1	1	0	1	0	1	0	1	0
1	2	1	2	1	2	1	2	2	3	2	3	2	3	2	3
3	0	3	0	3	0	3	0	3	2	3	2	3	2	3	2

ST5							
M0	M1	M2	M3	M4	M5	M6	M7
0	0	0	0	0	0	0	0
2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1
3	3	3	3	3	3	3	3

Figure 6.17 : Data Addressing Sequence with $N = 32$ and $\beta = 4$

how to generate all the sequence is shown in Figure 6.19 . The Figures also show a decimal representation and a decimal Bit-reversal representation. Since it was easier to calculate the decimal representation, this was the one actually implemented. Nevertheless, the addresses actually needed a Decimal Bit-Reversal representation. Therefore, all the Address Generator outputs were bit-reversed to produce the correct values.

6.6.2 Phase Factor Scheduler Design

The first step to calculate the addresses was to calculate the phase factor matrices for different transform sizes and recognize the pattern they follow throughout the stages. After the analysis, it was established that the sequence needed for a single butterfly was similar to that in Figure 6.20 . Therefore, for the implementation with

N	Binary Equivalent			B
	S2	S1	S0	
0	0	0	0	0
1	0	0	1	4
2	0	1	0	2
3	0	1	1	3
4	1	0	0	1
5	1	0	1	5
6	1	1	0	3
7	1	1	1	7

→

N	Binary Equivalent			B
	S1	S0	S2	
0	0	0	0	0
2	0	1	0	2
4	1	0	0	1
6	1	1	0	3
1	0	0	1	4
3	0	1	1	6
5	1	0	1	5
7	1	1	1	7

→

N	Binary Equivalent			B
	S0	S2	S1	
0	0	0	0	0
4	1	0	0	1
1	0	0	1	4
5	1	0	1	5
2	0	1	0	2
6	1	1	0	3
3	0	1	1	6
7	1	1	1	7

N: Decimal Normal Representation
B: Decimal Bit-Reversal Representation

Figure 6.18 : How Stride-2 Permutation is calculated throughout the stages

more butterflies, the address sequences must be the ones shown in the Figure 6.21 and 6.22 . It should be noted that the addresses shown in Figures 6.20 to 6.22 are supposed to be memories of $N/2$ locations, but we had established that the phase factor memories were of $N/4$ locations. This is done because the Most Significant Bit is for applying the Equation 6.5, and in this way generate other $N/4$ twiddle Numbers.

Analysing the sequences in Figures 6.20 to 6.22 , it can be seen that there are three main parameters: the step S, the frequency F, and the init value I. The step refers to the amount to be added for obtain the next number. The frequency refers to how often this step has to be added. And the init value is the first number in the sequence. For the calculation of this init values, we can see it as another sequence. These main parameters behave as shown in Figure 6.23 . From the Figure it can be seen that the frequency always starts at $N/(2\beta)$ or ϕ, decreases dividing by two until the value is one and stays in there until the calculation is over. The step always starts at $N/2$ and, decreases dividing into two until the value is equals to the number of butterflies and stays in that way until the calculation is over. And finally, the

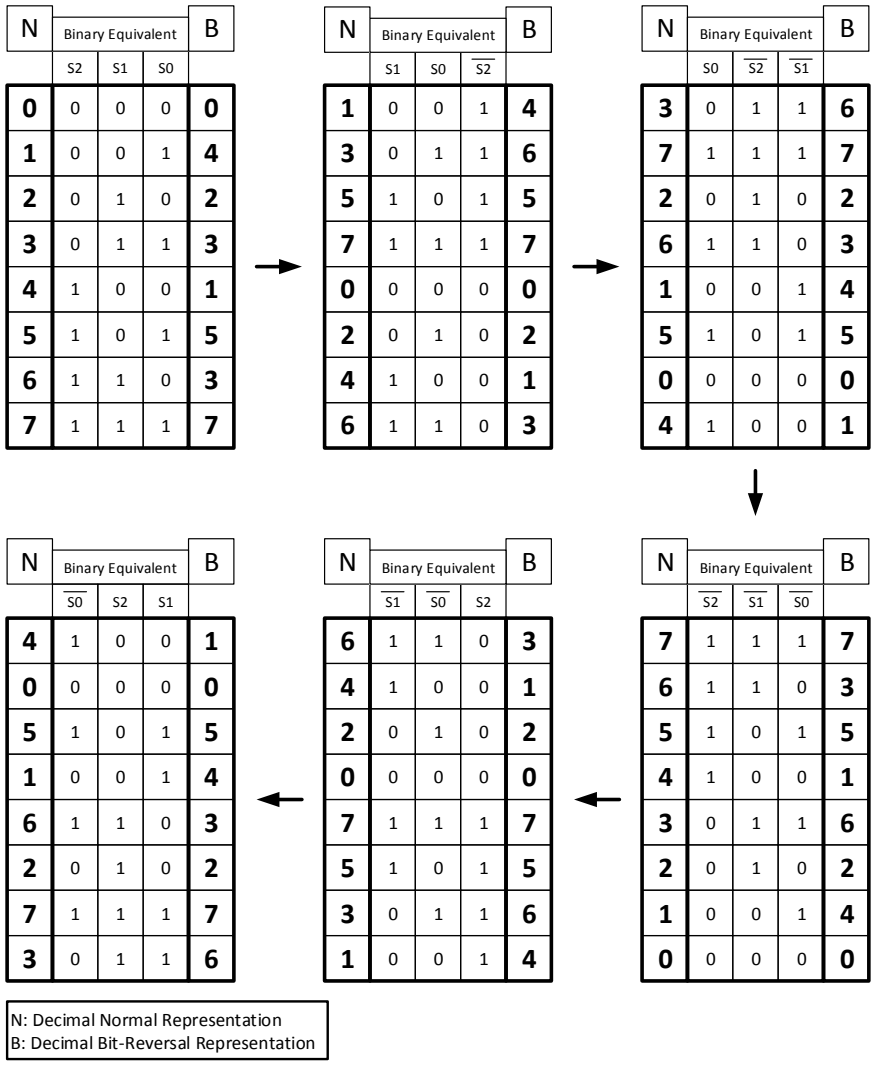

Figure 6.19 : How Modified Stride-2 Permutation is calculated throughout the stages

initial value variable, which represents the frequency and the step at the same time of the sequence of init values, starts always at $N/4$, decreases dividing by two until the value is zero. In the hardware implementation, for simplicity the variables use the same counter due to their similarity.

From the explanation given above, we designed the algorithm 1 that generates the twiddles pattern for an implementation of size N and β number of butterflies.

N = 32 and β = 1																
Stage	Twiddle Address															
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	8	8	8	8	8	8	8	8
3	0	0	0	0	4	4	4	4	8	8	8	8	12	12	12	12
4	0	0	2	2	4	4	6	6	8	8	10	10	12	12	14	14
5	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 6.20 : Phase Factor Scheduling for $N = 32$ and $\beta = 1$

N = 32 and β = 2								
Stage	Twiddle Address							
1	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
2	0	0	0	0	8	8	8	8
	0	0	0	0	8	8	8	8
3	0	0	4	4	8	8	12	12
	0	0	4	4	8	8	12	12
4	0	2	4	6	8	10	12	14
	0	2	4	6	8	10	12	14
5	0	2	4	6	8	10	12	14
	1	3	5	7	9	11	13	15

Figure 6.21 : Phase Factor Scheduling for $N = 32$ and $\beta = 2$

In this Algorithm, T is the Twiddle Address, F is the parameter frequency, S is the parameter step, and I is the init value. The variable i represent the current stage, and j and k are counter variables to initialize the j-sequence per stage and to generate the addresses at the stage respectively. Finally, R is a temporary variable to store previous values of T. The algorithm generates the addresses in the same order as they appear in Figures 6.20 to 6.22 , row per row.

N = 32 and β = 4				
Stage	Twiddle Address			
1	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
2	0	0	8	8
	0	0	8	8
	0	0	8	8
	0	0	8	8
3	0	4	8	12
	0	4	8	12
	0	4	8	12
	0	4	8	12
4	0	4	8	12
	0	4	8	12
	2	6	10	14
	2	6	10	14
5	0	4	8	12
	1	5	9	13
	2	6	10	14
	3	7	11	15

Figure 6.22 : Phase Factor Scheduling for $N = 32$ and $\beta = 4$

β = 1			
St	F	S	I
1	16	16	8
2	8	8	4
3	4	4	2
4	2	2	1
5	1	1	0

β = 2			
St	F	S	I
1	8	16	8
2	4	8	4
3	2	4	2
4	1	2	1
5	1	2	0

β = 4			
St	F	S	I
1	4	16	8
2	2	8	4
3	1	4	2
4	1	4	1
5	1	4	0

Legend			
St: Stage	F: Frequency	S: Step	I: Init Value

Figure 6.23 : Principal parameters behaviour of the phase factor scheduling for $N = 32$

Algorithm 1 TWIDDLE_ADDRESS

```
T = 0
R = 0
F = N/(2β)
S = N/2
I = N/2
for i = 1 to log_2 N do
    R = 0
    for j = 0 to β − 1 do
        T = R
        if j + 1 is divisible by I and T + I < N/2 then
            R = T + I
        end if
        for k = 2 to N/(2β) do
            if k − 1 is divisible by F then
                T = T + S
            end if
        end for
    end for
    I = I/2
    if F > 1 then
        F = F/2
    end if
    if S > β then
        S = S/2
    end if
end for
```

Chapter 7
RESULTS AND ANALYSIS

This chapter describes the procedure followed to validate the design and also shows the results obtained after completing the implementation of the proposed address generation scheme. The target hardware was a Xilinx SP605 Evaluation Platform. This specific board is based on a XC6SLX45T-3 FPGA, which is a Spartan-6 family device. Resource consumption results are referenced for this specific chip.

7.1 Core Validation

The address generation schemes first went through a high level verification. At this point, a MATLAB program served to validate the correctness of the strategy by implementing the same dataflow that would be then applied to the hardware version. After the hardware design was completed in VHDL, another program in MATLAB was written with the purpose of generating random test data. These data were generated using the MATLAB function *rand*, which generates uniformly distributed pseudorandom numbers. The data were then applied to the design by means of a VHDL testbench and the output values obtained after simulating the testbench were exported into MATLAB, where they were converted into a suitable format, for comparison with the reference FFT. Figure 7.1 illustrates the process described above. After completing this procedure, we obtained the Mean Percent Error of different implementations applying the formula:

$$MPE = 100 \times \frac{1}{N} \sum_{i=1}^{N} \frac{|g_i - y_i|}{y_i} \tag{7.1}$$

where N is number of points, g_i is data generated by MATLAB, and y_i is data generated by our FFT core. This procedure was done for power of two transform sizes between 2^4 and 2^{14} points, using one-Butterfly, two-Butterfly, and four-Butterfly version, and using single precision floating point format. Figure 7.2 illustrates the mean absolute error of our core compared with the FFT calculated with MATLAB. The Figure shows that there is a minimum percent error of 1.5×10^{-5} on an FFT core of 16 points and a maximum percent error of 4.5×10^{-5} for an FFT 16384 points. That information means an incrementing percent error while the number of points is increased. This is caused by two reasons: MATLAB used double precision floating point format, and rounding errors produced by the large number of operations involved in the calculation. These results confirm that our scalable FFT core implemented using address generation scheme works as expected.

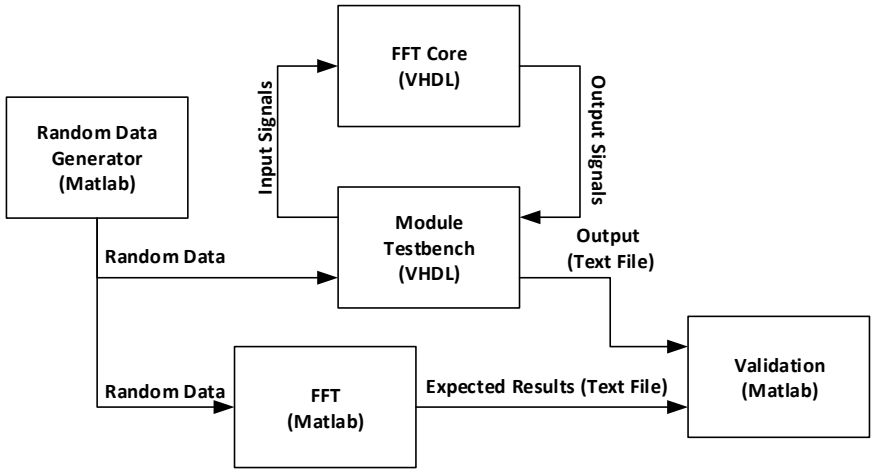

Figure 7.1 : Structure used to validate the simulation of the FFT

7.2 Timing Performance

Figure 7.3 present the calculation times, measured in clock cycles for the different FFT cores, implemented using address generation schemes. In general, when

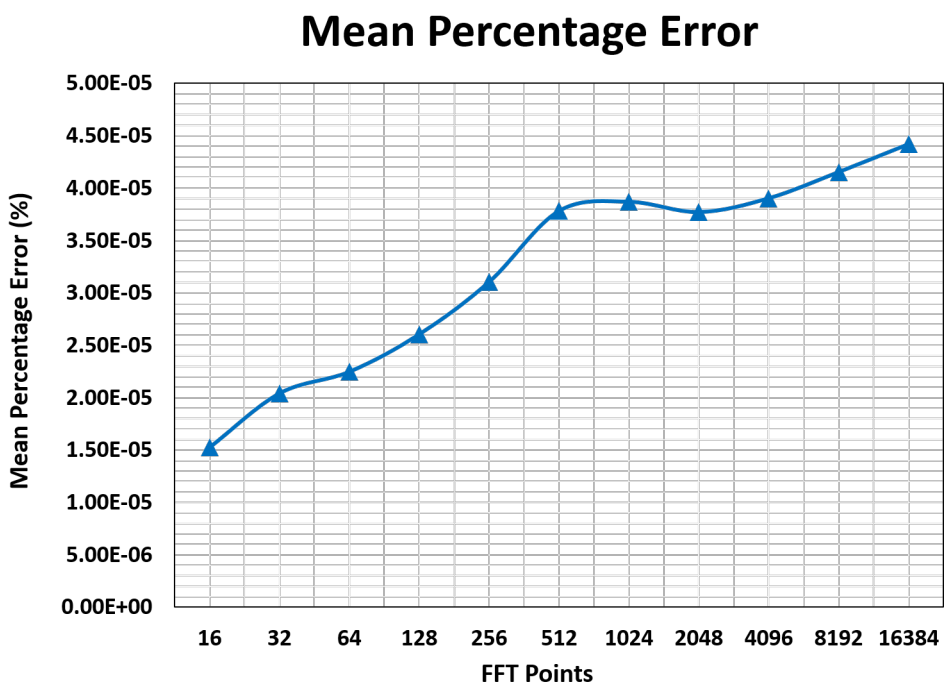

Figure 7.2 : Mean Percentage Error of our core compared with MATLAB

applying the developed strategy, the clock cycles required to complete the operation is a well defined function of the transform size N and the number of butterflies being used. This is, for a given transform of size N, number of butterflies β and a butterfly latency T_b, the cycles required to complete the calculation would be:

$$Cycles = \left(\frac{N}{2\beta} + T_b\right) \times (\log_2 N) \tag{7.2}$$

The maximum working frequency that the core can operate for different transform sizes is also provided in Figure 7.4 .

In order to provide a comparisons reference for the latencies of the Xilinx FFT Radix-2 Burst I/O core v8.0 were also included. Although the Xilinx FFT core supports various FFT architectures, a Radix-2 Burst I/O was chosen because it falls into the same category as the cores designed in this work. Latencies for Xilinx Core were extracted from Xilinx Core Generator.

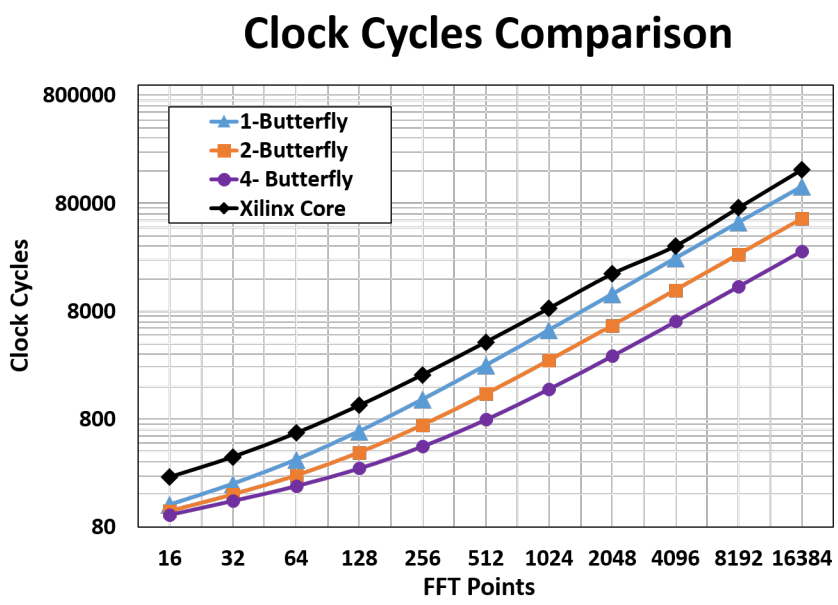

Figure 7.3 : Clock Cycles Comparison

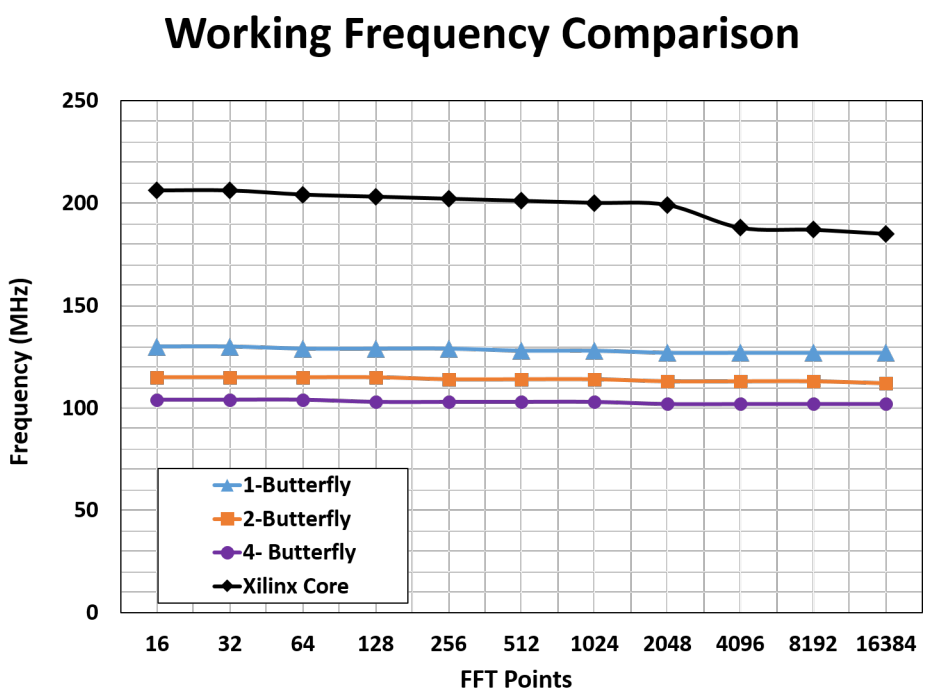

Figure 7.4 : Maximum Working Frequency Comparison

7.3 Resource Consumption

Figure 7.6 show the slice register consumed by the different implementations, starting with the one-butterfly to the four-butterfly version, Figure 7.5 present the slice LUTs used, Figure 7.7 show the DSP48 Blocks used, and finally, Figure 7.8 show the Memory usage in terms of RAM blocks. The number of slices reflects the amount of logical resources spent, Flip-Flops are contained within the slices and give information on how much resource is spent in sequential logic specifically, the number of DSP48 blocks represents the amount of special arithmetic units from the FPGA dedicated to implement the butterfly, and the total memory reflects the resources spent to store both data and phase factors. The XC6SLX45T has 54576 slice registers, 27288 Slice LUTs, and 56 DSP48 blocks.

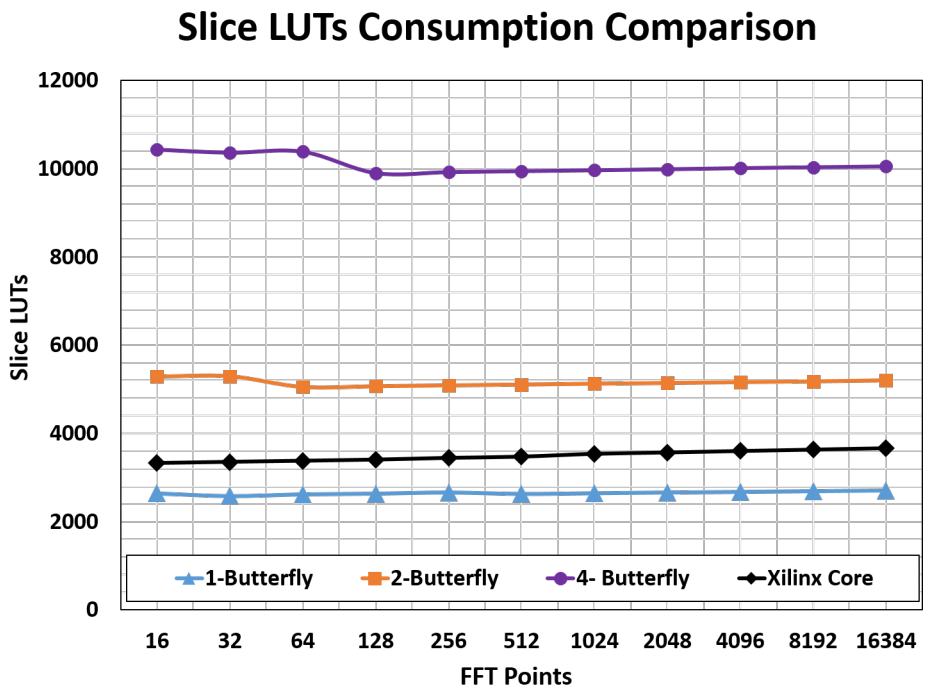

Figure 7.5 : Slice LUTs Consumption Comparison

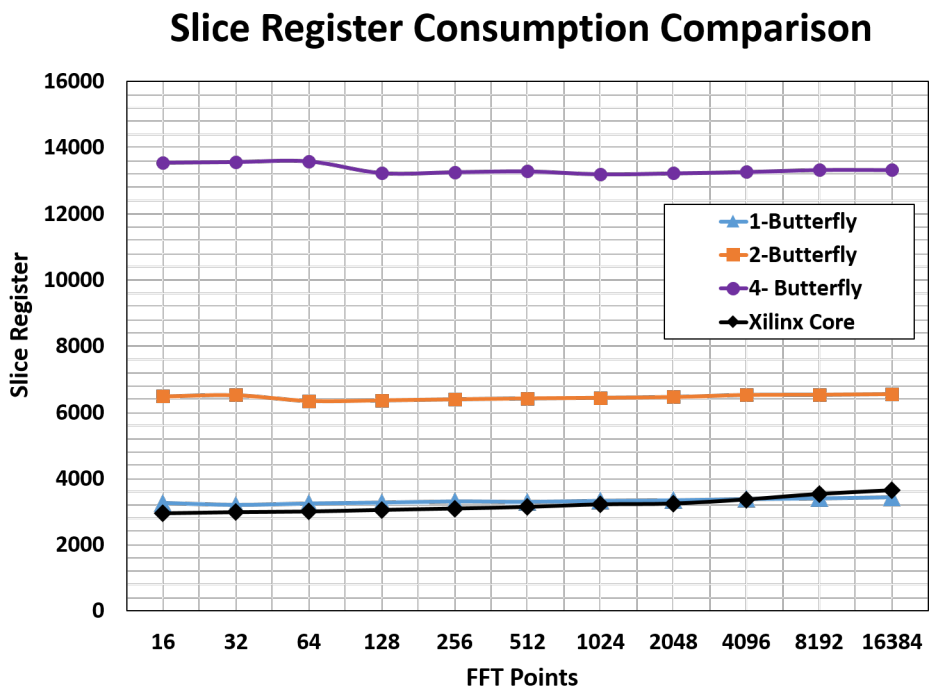

Figure 7.6 : Slice Register Consumption Comparison

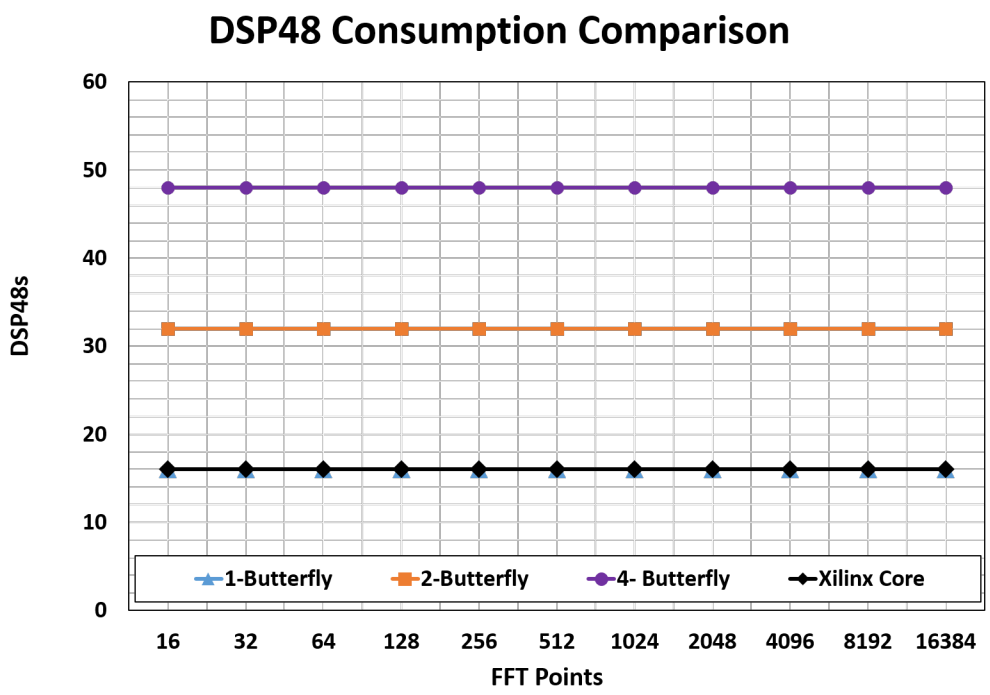

Figure 7.7 : DSP48 Consumption Comparison

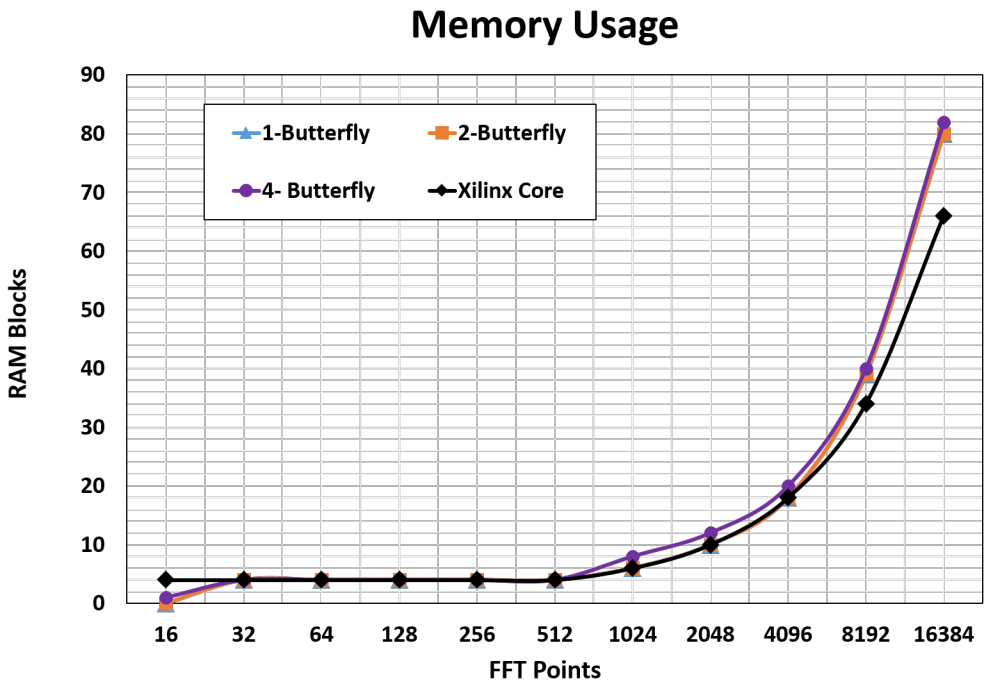

Figure 7.8 : Memory Usage Comparison

7.4 Analysis and Comparisons

Giving another look to the Figures, the timing performance of our core against Xilinx's, the latency in our implementation is around 39% better with one butterfly, 64% with two butterflies and 76% with four butterflies. On the other hand, the maximum frequency in our design is around 42% worst with one butterfly, 58% with two butterflies and 58% with four butterflies. Analysing these results, we can establish that our core is in average better regarding the computation time. The computation time is calculated with the formula $T = L/F$, where F is the frequency and L is the latency.

Regarding the consumed resources, we can observe from Figure 7.8 that the most heavily affected resource would be the memory, at its usage increases in proportion with the transform size. However, using our addressing scheme did not affect the consumption of memory resources. Figure 7.8 highlights this fact by showing that for increasing levels of parallelism the memory requirements are unaffected. Moreover,

for a specific input size and phase factor precision, the data bus and address bus width increased proportionally to $\log_2(N)$, therefore, the system blocks manipulating these buses will not grow aggressively. The one-butterfly version has a lower slice register consumption, almost the same slice LUTs used, and the same DSP48 consumed than the FFT Xilinx Core. However, the two-butterfly and four-butterfly version have a higher slice register, slice LUTs and DSP48 consumption. This is the price we have to pay to obtain a low latency.

Although in this research we did not perform an experiment to quantify the impact in energy consumption, we can infer that the strategy used favors this parameter. The base of this conjecture is that the addressing strategy reduced the computation time and the consumed resources with respect to the Xilinx Core. Therefore, we could expect a reduction in the energy consumption. Performing an experiment for verifying this assumption is left as future work.

Figure 7.9 shows the computation times of the core implemented with one-butterfly to the four-butterfly version against the number of points. The single butterfly version has a computation time close to the FFT Xilinx Core but worst, while the two-butterfly and four-butterfly version have a much better computation time to the FFT Xilinx Core.

7.5 Limitations

In the target Xilinx FPGA, the number of butterflies that can be implemented is limited by the number of DSP48 blocks available on chip. If this happened to be an inconvenient, the arithmetic hardware could be implemented using a mixture of slices and DSP48 blocks. This will give more room to implement more butterflies though at the expense of higher logic consumption.

On the other hand, the general rule governing the pattern of the address sequence for data point and phase factor regardless the folding factor do not satisfies higher

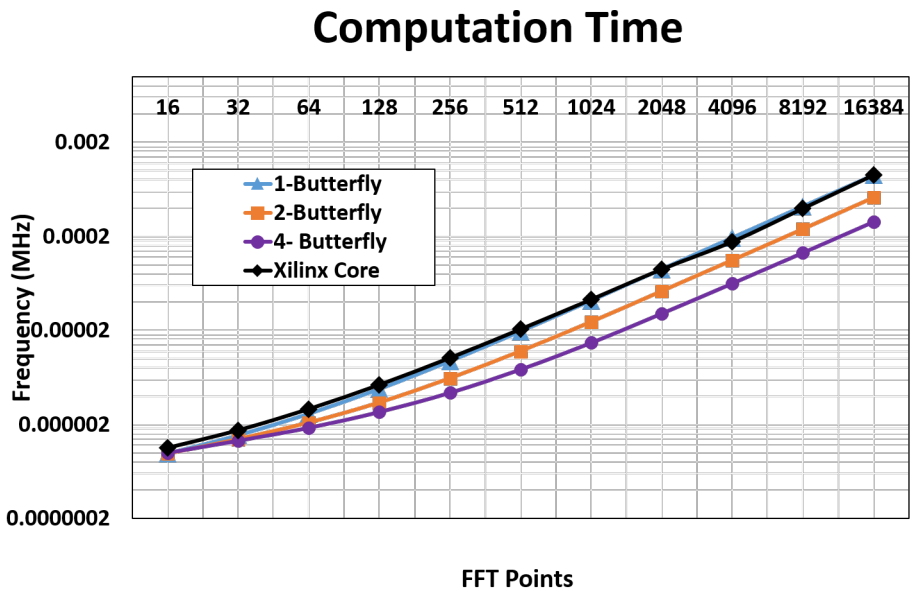

Figure 7.9 : Computation Time

radix implementation. Also, this addressing is only applicable to Decimation In Time (DIT) Pease Architectures.

Chapter 8
CONCLUSIONS

This thesis presents a method to design the permutation block when the folding factor is scaled in FFT cores based on an address generation scheme for FPGA implementation. The address generator approach described in this work is an effective alternative to achieve a scalable folding of the Pease FFT structure. For a given butterfly configuration, the impact of scaling the transform size mostly affects memory usage, causing low impact on the logical consumed resources. The approach is complete in the sense that it takes into account the address sequences required to access data points as well as twiddle factors.

Chapter 9
CONTRIBUTIONS AND FUTURE WORK

The contributions that this project makes to this area include the following:

1. A general formula to generate the address patterns regardless the number of point, folding factor, and numeric format using addressing schemes.
2. Algorithm and hardware to reproduce the phase factor address sequence for the Pease FFT radix-2 factorization.
3. A general expression relating the cycle count to the transform size and number of butterflies.
4. The detailed behaviour of the read and write switches regardless of the number of points and folding factor.
5. A scalable hardware implementation of the address generator including number of points and folding factor.
6. A fully scalable FFT core including number of points, folding factor and numeric format.

The following future directions were identified that could further improve the current state of the presented work:

1. To explore the applicability to higher radix algorithms, since this can reduce the number of stages required to complete calculation of some transform sizes. To achieve this goal, it would be necessary to study the resulting data and twiddle address patterns and then evaluate the complexity and feasibility of the implementation.

2. Modify the architecture to obtain a continuous flow FFT core based on the developed strategy.
3. Study the requirements needed to implement a structure that allows a non power-of-two points FFT.
4. Exploit the phase factor range to develop a simpler butterfly.
5. Perform an experiment that study the impact in the energy consumption using this strategy .

Bibliography

[1] J.W. Cooley and J.W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics Computation*, 19:297–301, 1965.

[2] Marshall C. Pease. An Adaptation of the Fast Fourier Transform for Parallel Processing. *J. ACM*, 15(2):252–264, April 1968.

[3] D. G. Korn and J. J. Lambiotte. Computing the Fast Fourier Transform on a Vector Computer. *Mathematics of Computation*, 33:977–992, 1979.

[4] Thomas G. Stockham, Jr. High-Speed Convolution and Correlation. In *Proceedings of the April 26-28, 1966, Spring joint computer conference*, AFIPS '66 (Spring), pages 229–233, 1966.

[5] L. G. Johnson. Conflict Free Memory Addressing for Dedicated FFT Hardware. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing.*, 39(5):312–316, 1992.

[6] Bingrui Wang, Qihui Zhang, Tianyong Ao, and Mingju Huang. Design of Pipelined FFT Processor Based on FPGA. In *Second International Conference on Computer Modeling and Simulation, 2010. ICCMS '10.*, volume 4, pages 432–435, 2010.

[7] N. Polychronakis, D. Reisis, E. Tsilis, and I. Zokas. Conflict free, parallel memory access for radix-2 FFT processors. In *2012 19th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 973–976, 2012.

[8] V. Gautam, K.C. Ray, and P. Haddow. Hardware efficient design of Variable Length FFT Processor. In *2011 IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 309–312, 2011.

[9] Liu Hongxia and Huang Shitan. High Performance Algorithm for Twiddle Factor of Variable-size FFT Processor and its Implementation. In *2012 International Conference on Industrial Control and Electronics Engineering (ICICEE)*, pages 1078–1081, 2012.

[10] Chu Chad, Zhang Qin, Xie Yingke, and Han Chengde. Design of a High Performance FFT Processor Based on FPGA. In *Design Automation Conference,*

2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific, volume 2, pages 920–923 Vol. 2, 2005.

[11] G. Szedo, V. Yang, and C. Dick. High-Performance FFT Processing Using Reconfigurable Logic. In *Conference Record of the Thirty-Fifth Asilomar Conference on Signals, Systems and Computers, 2001.*, volume 2, pages 1353–1356 vol.2, 2001.

[12] K.M. Ramesh and D.S. Sumam. Comprehensive Address Generator for Digital Signal Processing. In *2009 International Conference on Industrial and Information Systems (ICIIS).*, pages 325–330, 2009.

[13] Yingtao Jiang, Ting Zhou, Yiyan Tang, and Yuke Wang. Twiddle-Factor-Based FFT Algorithm with Reduced Memory Access. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 8 pp–, 2002.

[14] Pei-Yun Tsai and Chung-Yi Lin. A Generalized Conflict-Free Memory Addressing Scheme for Continuous-Flow Parallel-Processing FFT Processors With Rescheduling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems.*, 19(12):2290–2302, 2011.

[15] Xin Xiao, E. Oruklu, and J. Saniie. An Efficient FFT Engine With Reduced Addressing Logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 55(11):1149–1153, 2008.

[16] A. Polo, M. Jimenez, D. Marquez, and D. Rodriguez. An Address Generator Approach to the Hardware Implementation of a Scalable Pease FFT Core. In *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 832–835, 2012.

[17] S.K. Shome, A. Ahesh, D.K. Gupta, and S. Vadali. Architectural Design of a Highly Programmable Radix-2 FFT Processor with Efficient Addressing Logic. In *2012 International Conference on Devices, Circuits and Systems (ICDCS)*, pages 516–521, 2012.

[18] Yao-Xian Yang, Jin-Fu Li, Hsiang-Ning Liu, and Chin-Long Wey. Design of Cost-Efficient Memory-Based FFT Processors Using Single-Port Memories. In *2007 IEEE International SOC Conference.*, pages 29–32, 2007.

[19] Chin-Long Wey, Shin-Yo Lin, and Wei-Chien Tang. Efficient Memory-Based FFT Processors for OFDM Applications. In *2007 IEEE International Conference on Electro/Information Technology*, pages 345–350, 2007.

[20] J.A. Vite-Frias, Rd.J. Romero-Troncoso, and A. Ordaz-Moreno. VHDL Core for 1024-Point Radix-4 FFT Computation. In *International Conference on Reconfigurable Computing and FPGAs, 2005. ReConFig 2005.*, pages 4 pp.–24, 2005.

[21] M. Ayinala, Yingjie Lao, and K.K. Parhi. An in-place fft architecture for real-valued signals. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 60(10):652–656, Oct 2013.

[22] K. Babionitakis, K. Manolopoulos, K. Nakos, D. Reisis, N. Vlassopoulos, and V. A. Chouliaras. A High Performance VLSI FFT Architecture. In *ICECS '06. 13th IEEE International Conference on Electronics, Circuits and Systems, 2006.*, pages 810–813, 2006.

[23] Gijung Yang and Yunho Jung. Scalable FFT Processor for MIMO-OFDM Based SDR Systems. In *2010 5th IEEE International Symposium on Wireless Pervasive Computing (ISWPC).*, pages 517–521, 2010.

[24] V. Montaño and M. Jimenez. Design and Implementation of a Scalable Floating-point FFT IP Core for; Xilinx FPGAs. In *2010 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 533–536, 2010.

[25] Shuai Chen, Jialin Chen, Kanwen Wang, Wei Cao, and Lingli Wang. A Permutation Network for Configurable and Scalable FFT Processors. In *2011 IEEE 9th International Conference on ASIC (ASICON)*, pages 787–790, 2011.

[26] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on fpga based custom computing machines. In *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pages 155–162, Apr 1995.

[27] J.R Johnson, R.W. Johnson, D. Rodriguez, and R. Tolimieri. A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures. *Circuits Systems Signal Process*, 9(4):450–500, 1990.