TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MUAZAM ALI

PIPELINED FAST FOURIER TRANSFORM PROCESSOR

Master of Science Thesis

Examiners: Prof. Jarmo Takala and
Dr. Fahad Qureshi
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on January 4th, 2017

ABSTRACT

Muazam Ali: Pipelined Fast Fourier Transform Processor
Tampere University of Technology
Master of Science Thesis, 52 pages, 2 Appendix pages
August 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiners: Prof. Jarmo Takala and Dr. Fahad Qureshi

Keywords: FFT, SDF, VHDL, MATLAB, OFDM, Constant Multiplication

In some cases, signal processing is easier in frequency-domain and Discrete Fourier Transform (DFT) is a useful tool to convert signals between time and frequency domains. Fast Fourier Transform (FFT) is an efficient algorithm for computing DFT. FFT has a number of applications, where frequency-domain of a signal needs to be analyzed. It gained attention in communication systems because FFT is a crucial processing operation in the Orthogonal Frequency Division Multiplexing (OFDM) systems. The research work carried out in this thesis presents hardware implementation of FFT processors. The processors' architectures are designed based on Single-path Delay Feedback (SDF) FFT architecture scheme, which implements the radix-2^2 and identical radix-2^2 FFT algorithms for 2048-point FFT. The designed identical radix-2^2 FFT algorithm has similar number of non-trivial complex twiddle factor multiplication stages, identical to radix-2^2 FFT algorithm but less complex operations. The lower complexity makes it an area efficient, memory efficient, reduce the multiplication cost and power consumption. Moreover, the computational complexity of the FFT processor can be reduced by replacing the general complex multipliers by constant multipliers (shift-and-add) circuits. W_8 and W_{16} constant multipliers circuits are implemented in this thesis, which can replace three complex multipliers in the FFT processor. Finally, the FFT processors and constant multiplier circuits are implemented on Virtex-7 FPGA.

PREFACE

The research work in this thesis was carried out in the Department of Pervasive Computing, Tampere University of Technology, Finland. I would like to sincerely express my gratitude to the all the knowledgeable people who helped me the most throughout my thesis. I deeply respect and appreciate their encouragement, support and prayers.

Foremost, I express my sincere and highest gratitude to Prof. Jarmo Takala who gave me opportunity to work with him in his research group. I could not have envisioned a better professor and mentor to conduct my thesis work. In addition to it, I will always be highly indebted to him for teaching me two core courses of my M.Sc. majors, DSP Implementations and Computer Architecture. His immense knowledge and professionalism in teaching the courses helped me understand the core details of these subjects. The practical work conducted in these courses helped me a lot to understand the ideas even better and improved my skillset. I am extremely thankful to him for believing in my capabilities, giving me finances to support my work and giving me all the freedom I needed to conduct my thesis work.

Besides my professor, I gratefully thank my advisor Dr. Fahad Qureshi for his priceless supervision, guidance, encouragement, inspiration and support. The most knowledgeable efforts to polish my skillset and steer my research work towards completion are attributed to him. He was always available to help me whenever I had trouble or a question about my thesis work or writing. I strongly admit that his persisting support not only boosted confidence in me but also helped me to carry out quality research work.

I want to pay a kind gratitude to all of my teachers who put their efforts to grow me in my career. I deeply respect the valuable contribution of Prof. Timo Hämäläinen, Dr. Matti Rintala, Arto Perttula, Tero Ahtee, and Dr. Waqar Hussain in developing my skillset in the field of computer systems.

I am highly thankful to my proud father Ghulam Rasool and my dearest mother Bismillah BiBi for their support to pursue my Master's in Information Technology in Finland. I acknowledge their sacrifices for my best education since I was a child, which I can never pay back my whole life. My parents always taught me to support democracy, respect different cultures, strongly believing in love and toleration for everyone. Moreover, I am extremely thankful to my elder brother Muhammad Amjad as well as my sisters, Maria, Saima, Javeria and Sadia Rasool for their prayers and support.

A special thank of mine goes to my colleagues Jingui Li, Renjie Xie and Jakub Zadnik who always motivated me, exchange their interesting ideas and created a friendly work environment to conduct my research work.

Finally yet importantly, I thank all of my friends for their prayers and moral support in both academic and social life.

This research work has been a great learning experience for me; it strengthened my skillset in hardware design and implementation through different approaches. I gratefully thank again Prof. Jarmo Takala and Dr. Fahad Qureshi for this opportunity, believing in my capabilities and their persisting support as well as giving me all the freedom I needed to conduct my thesis.

Tampere, 12.07.2017

Muazam Ali

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

DSP	Digital Signal Processing
DFT	Discrete Fourier Transform
IDFT	Inverse Discrete Fourier Transform
FFT	Fast Fourier Transform
SFG	Signal Flow Graph
DIT	Decimation-in-Time
DIF	Decimation-in-Frequency
PE	Processing Element
BF	Butterfly
SDF	Single path Delay Feedback
MDF	Multiple path Delay Feedback
FF	Feed Forward
MDC	Multiple path Delay Commutator
FIFO	First in First out
SCM	Single Constant Multiplication
MCM	Multiple Constant Multiplication
CSD	Canonical Signed Digit
CC	Clock Cycle
BF I	2-point Butterfly
BF II	2-point Butterfly with additional circuitry of W_4 Multiplier

1. INTRODUCTION

Advancement of digital systems is continuously replacing the old analog systems. Digital systems are the backbone of modern day organizations, products, processes and services and their quality is increasingly subject to these systems. Applications of present day digital systems include communication systems, traffic systems, control systems, weather forecasting systems, internet and so forth.

Microelectronics hardware and software have turned into a fundamental vital material for leading gadgets, products, processes and services. Modern day microelectronic technology implements complete complex information processing system on a single chip. The transformation from the multi-chip systems to systems-on-a-single-chip (SoC) opens new potential outcomes and generates new challenges. Advances in microelectronic technology are exceptionally fast.

These days, digital systems are set up to confront the requirements of the most demanding digital signal processing (DSP) applications, which impose solid conditions such as, small silicon area, clock frequency, high throughput, reduced power consumption, latency and real-time computations. In order to fulfil these requirements, digital systems are being implemented on Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) devices. These devices operate at high clock frequencies and achieve high performance in computations of digital signal processing (DSP) algorithms. An algorithm can have wide range of hardware implementations, which implies algorithms does not provide any information about the hardware architecture that processes it.

Fast Fourier transform (FFT) is an efficient algorithm for fast computation of discrete Fourier transform (DFT) and its inverse. FFT is highly efficient procedure for fast computation of DFT finite series and requires less number of computations than the direct evaluation of DFT. It takes benefit of the fact that the computation of coefficients can be carried out iteratively. FFT reduces the computational complexity of DFT from $O(N^2)$ to $O(NlogN)$.

Cooley-Tukey's algorithm [4] is the most popular among all the other FFT algorithms. Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF) are two basic forms of Cooley-Tukey's algorithm. Radix-2 DIT approach decomposes input samples into even and odd samples iteratively but radix-2 DIF approach practice same procedure from output samples. In FFT algorithms, decompositions are applied recursively up to required small point DFTs called as radix. Small point DFTs can be radix-2, radix-4 and

mixed radix. After Cooley-Tukey's algorithm, various FFT algorithms were proposed like, radix-4, radix-8 and higher radices to decrease the complexity of twiddle factor multiplications. But, these FFT algorithms have complex butterfly and cannot be applied for all power-of-2 FFT sizes. For such FFTs, a mixed radix FFT algorithm (radix-2 and radix-4) was proposed, which are not power-of-4 at cost of complex.

After three decades of Cooley-Tukey's FFT algorithm, a new radix-2^2 FFT algorithm was proposed, which uses simple radix-2 butterfly structure to take advantage of radix-4. The number of complex twiddle factor multiplications in radix-2^2 is almost similar to radix-4. This algorithm is applicable for any power-of-2 FFT sizes. Based on this algorithm, later radix-2^3, 2^4 and radix-2^k FFT algorithms were proposed. However, binary tree representation is used to present the basic forms of Cooley-Tukey's algorithms as well as FFT algorithms mentioned previously are generated through this representation. It is worth mentioning that these algorithms differ from each other only in twiddle factor multiplications. The flow of data and butterfly structure remain constant.

FFT has various applications where frequency-domain of a signal is necessary to be analyzed. It gained attention in communication systems because FFT is a crucial processing operation in orthogonal frequency division multiplexing (OFDM) systems. OFDM is a leading modulation technique, which adequately broaden channel usage and abridge inter-symbol interference (ISI) along with inter-carrier interference (ICI) generated by multipath effect. It is mainly used in digital audio broadcasting (DAB), digital video broadcasting-terrestrial (DVB-T) and digital video broadcasting-handheld (DVB-H). In addition to it, there are many communication applications based on OFDM where FFT is most important processing operation such as Ultra Wide Band (UWB), 3GPP LTE, Digital Subscriber Line (DSL), WLAN, WiMax etc.

In designing communication systems, the principal factor dominating performance is throughput of the system. Because of immense data rate requirements, efficient FFT modules with high throughput but smaller chip area as well as with reduced power consumption are in demand.

In general, there are two types of FFT architectures, pipelined architectures and memory-based architectures. There are two principal types of pipelined architectures: Feedback FFT architectures and Feedforward FFT architectures. According to the number of data lines, these architectures can be further divided into single-path and multi-path pipelined FFT architectures. Multi-path pipelined architectures, where data is fed to the FFT processor using several paths, are used where the throughput has to be increased for a given clock frequency of FFT processor. This research work is particularly focused on hardware implementation of single path delay feedback (SDF) pipelined FFT architecture, where data is fed to the processor using a single path.

Lately, IC design resources in chip technology have increased, which allows the implementation of complex algorithms on a single chip, such as FFT. Especially, to fulfill the requirements of OFDM standards, it is necessary to design an efficient FFT processor. Thus, the research work carried out in this thesis presents an efficient hardware implementation of an FFT processor.

SDF pipelined FFT architectures are often used for higher throughputs and reduced chip-area. In SDF architectures, FFT is computed in stages where each stage consists of an FFT butterfly structure, a FIFO and a complex multiplier. One complex multiplication contains four real multiplications and two real additions, which is important for performance and chip-area. A lot of research has done in order to reduce the number of complex operations.

Previous work includes FFT architectures for radix – 2, radix – 2^2 and radix – 2^3 algorithms. We propose identical radix-2^2 FFT algorithm, which requires less number of complex multiplications than the previous work. Designed FFT algorithm is given the name identical radix – 2^2 due to the fact that the number of non-trivial complex multiplier stages are identical to radix-2^2. The designed FFT algorithm mapped on SDF pipelined hardware architecture for 2048-point FFT. Less complex multiplications makes it an area-efficient, memory efficient, reduce the multiplication cost, and power consumption.

FFT architectures differ in twiddle factor multiplications but flow and processing of data remains the same. This architecture uses radix-2 butterflies to take advantages of higher radix FFT i.e. radix – 2^2. In general in this case, a traditional SDF architecture requires ten complex multiplier stages. This approach extensively reduces the number of complex multipliers from 10 to two, which reduces the memory requirements to store the twiddle factors. Five trivial complex multipliers (multiplication by $-j$) are handled within the butterfly structure by swapping and sign changing. Furthermore, three complex multipliers can be replaced by constant multipliers i.e. shifters and adders. Constant multipliers implementation is done as well in this work. Through integration of those constant multipliers in the implemented pipelined FFT processor, the number if complex multipliers can be further reduced from five to two. The total required memory in this approach is less than the previous works and it decreases the chip-area as well as power consumption.

Another greater advantage of using this approach appears in reduction of round-off error as compared to previous work. The main reasons for reduction in round-off error are lower number of non-trivial multiplications and their position in the later processing stages.

1.1 Thesis Objective

The main objective of this research work was to design an efficient FFT algorithm with less number of complex multiplications, map the designed algorithm on SDF pipelined FFT architecture and implement on the FPGA.

1.2 Thesis Organization

The thesis is organized as follows. Chapter 2 introduces the DFT and FFT. Moreover, derivation of different FFT algorithm is provided and main concepts of FFT are explained in this chapter such as, difference between DIT and DIF decompositions of FFT. Lastly, improved FFT algorithms are discussed through binary tree representation.

Chapter 3 introduces FFT hardware architectures. A brief overview of memory based FFT architectures and pipelined FFT architectures is provided. However, pipelined FFT architectures are discusses in more detail including both feedback FFT architectures and feedforward FFT architectures. Finally, mapping of FFT signal flow graph (SFG) on hardware is discussed.

Chapter 4 deals with FFT twiddle factor multiplications and its different hardware implementation techniques. Twiddle factor multiplications' implementation approaches include generic complex multiplier and constant multiplier (shift-and-add).

Chapter 5 analyses the basic concepts of different number representations such as, Binary, Two's Complement and Canonical Signed Digit (CSD). Moreover, it discusses the effects of finite word length and the concepts of overflow handling.

Chapter 6 focuses on designing the hardware architecture for the proposed approach that is identical radix-2^2 pipelined FFT processor. Hardware architectures with general complex multipliers and constant multipliers are discussed.

Chapter 7 discusses the synthesis results of the proposed FFT architecture and make comparison with radix-2^2 FFT architecture synthesis results.

Finally, Chapter 8 addresses the main conclusions and future works.

2. FAST FOURIER TRANSFORM AND ITS ALGORITHMS

The discrete Fourier transform (DFT) is the Fourier representation of finite length sequences and is a complex function of frequency [1], [2]. It is a widespread algorithm and can be exploited for the implementation of the convolution of two linear sequences as well as spectral analysis of signals [3]. Therefore, it is of greater importance to explore the efficient computation methods of DFT.

N-point DFT is defined as

$$X[k] = \sum_{n=0}^{N-1} x[n]W_N^{nk}, \quad k = 0, 1, \ldots, N\text{-}1, \tag{2.1}$$

where $x[n]$ and $X[k]$ are the input and output sequences respectively. The inverse discrete Fourier transform (IDFT) is defined as

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k]W_N^{-nk}, \qquad n = 0, 1, \ldots, N\text{-}1, \tag{2.2}$$

where x[n] and $X[k]$ are both complex sequences in time domain and frequency domain respectively. Coefficient W_N is called the twiddle factor, which is defined as

$$W_N = e^{-j2\pi/N}. \tag{2.3}$$

In addition, $e^{-j2\pi/N}$ can be defined as

$$e^{-j2\pi/N} = \cos\left(\frac{2\pi}{N}\right) - j\sin\left(\frac{2\pi}{N}\right). \tag{2.4}$$

The Fast Fourier Transform[1] (FFT) [1], [2], [3] is an efficient algorithm for DFT computation. It is based on the successive decomposition of *N*-point DFT into smaller size DFTs and taking the benefit from the symmetry and periodicity properties of the twiddle factor in equation (2.3). Cooley-Tukey's algorithm [4] is the most popular among all the other FFT algorithms. FFT algorithm reduces the computational complexity of DFT from $O(N^2)$ to $O(N\log N)$. Two basic forms of Cooley-Tukey algorithm are Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF). Cooley-Tukey FFT algorithm is based on divide and conquer approach where *N*-point DFT is decomposed

[1] In FFT algorithms, we will be using an *N*-point sequence or sequence of length *N* interchangeably. Terms sample and point will also be used as interchangeably.

recursively up to required small point DFT which defines the radix. The next sections illustrate radix-2 DIT and DIF FFT algorithms.

2.1 Decimation-in-Time FFT

Algorithms, which decompose the $x[n]$ sequences into successive smaller sequences, are called Decimation-in-Time (DIT) algorithms [1]. The main principle of DIT FFT algorithm is illustrated as follows, when $N = 2^v$.

We compute $X[k]$ by dividing $x[n]$ into two ($N/2$) sequences i.e. even and odd indexed sequences. Rewrite the equation (2.1) as follows

$$X[k] = \sum_{even} x[n]W_N^{nk} + \sum_{odd} x[n]W_N^{nk}. \tag{2.5}$$

Now substitute $n = 2m$ for even and $n = 2m + 1$ for odd indices of $x[n]$ sequence as

$$X[k] = \sum_{m=0}^{\frac{N}{2}-1} x[2m]W_N^{2mk} + \sum_{m=0}^{\frac{N}{2}-1} x[2m+1]W_N^{(2m+1)k}. \tag{2.6}$$

Equation (2.6) is rewritten as

$$X[k] = \sum_{m=0}^{\frac{N}{2}-1} x[2m](W_N^2)^{mk} + W_N^k \sum_{m=0}^{\frac{N}{2}-1} x[2m+1](W_N^2)^{mk}. \tag{2.7}$$

As [1] says $W_N^2 = W_{N/2}$ because,

$$W_N^2 = e^{-2j(\frac{2\pi}{N})} = e^{-j2\pi/(\frac{N}{2})} = W_{N/2}.$$

Finally, rewrite the equation (2.7) as

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

Now simplify the above equation as

$$X[k] = G[k] + W_N^k H[k], \quad k = 0, 1, \ldots, N\text{-}1, \tag{2.9}$$

where G and H are both $N/2$ sample points even and odd DFTs respectively.

Flow of data in equation (2.9) is shown in the Fig. 2.1. In this figure, each of the $N/2$ points DFT the computational complexity is $O((N/2)^2)$ i.e. number of complex multiplications and additions. Then multiply the second part of (2.9) with W_N^k and we need N complex additions to get summation of both parts of (2.9). Hence, for once divided DFT $N + (N/2)^2$ complex multiplications and additions are needed, which is less than the

computational complexity of the traditional DFT having $O(N^2)$ computational complexity.

Equation (2.9) represents two *N*/2 point DFTs where *N* is a power-of-2. Now, further consider *N*/2 point DFT decomposition into two *N*/4 points DFTs for each *N*/2. Then add *N*/4 points DFTs together to get *N*/2-point DFT.

$$G[k] = \sum_{m=0}^{\frac{N}{4}-1} g[2m]\, W_{N/4}^{mk} + W_{N/2}^{k} \sum_{m=0}^{\frac{N}{4}-1} g[2m+1] W_{N/4}^{mk} \tag{2.10}$$

Similarly, write $H[k]$ as

$$H[k] = \sum_{m=0}^{\frac{N}{4}-1} h[2m]\, W_{N/4}^{mk} + W_{N/2}^{k} \sum_{m=0}^{\frac{N}{4}-1} h[2m+1] W_{N/4}^{mk} \tag{2.11}$$

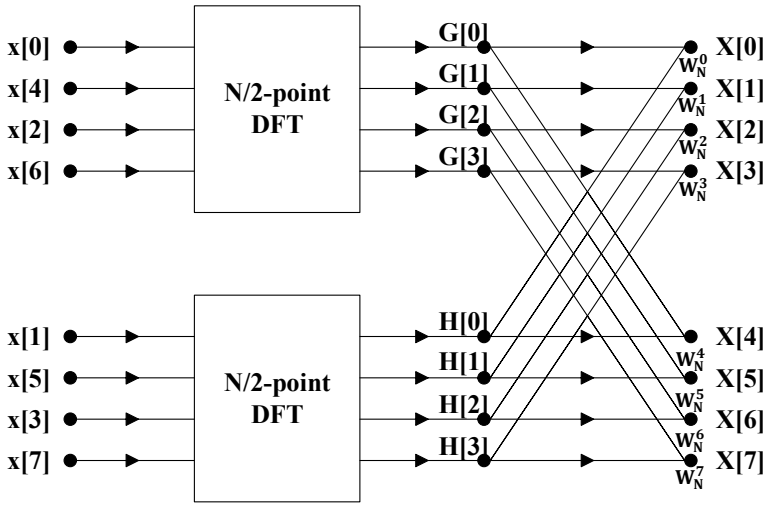


Figure 2.1 Flow graph of DIT decomposition for computation of N-point DFT into two (N/2- point) DFTs, N = 8

The computations for equations (2.10) and (2.11) are shown by green color in the Fig. 2.2.

Now, computations have become 2-point DFTs. Now, the signal flow graph of 2-point DFT is illustrated in the Fig. 2.3. This 2-point DFT structure is called as radix-2 butterfly and signal flow graph (SFG) is called as butterfly structure.

Finally, the complete SFG of 8-point DFT is illustrated in Fig. 2.4, which is replacing radix-2 butterfly structure, by *N*/4-point DFTs in Fig. 2.2.

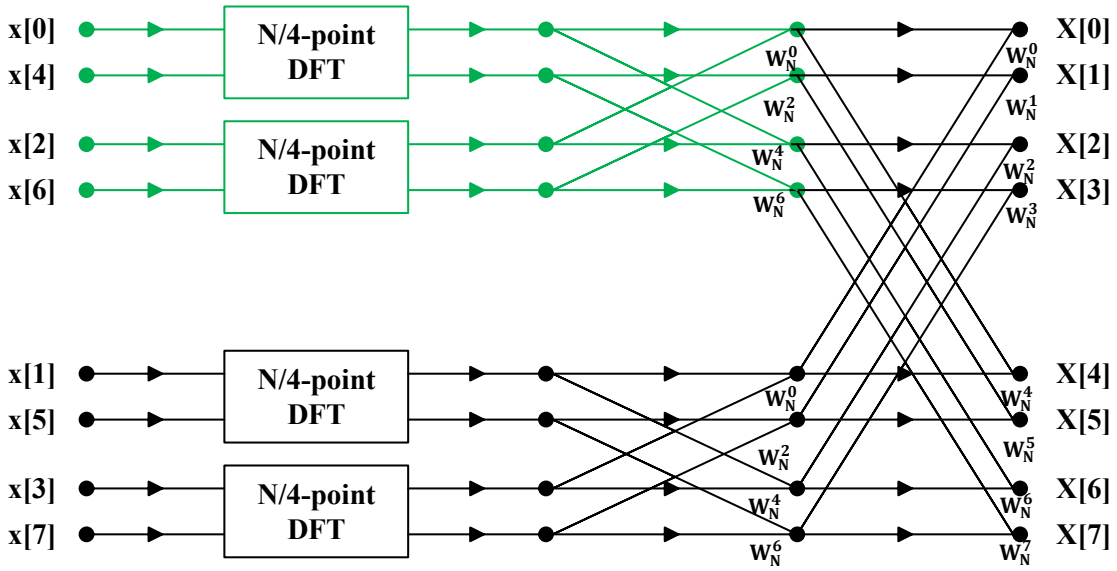


Figure 2.2 Green color represents DIT decomposition of N/2-point DFT into two N/4-point DFTs computations, N = 8

To generalize the idea for large value of N where N would still be a power of two, further decompose $N/4$-point DFTs[2] into $N/8$-point DFTs and so on recursively decompose into small DFTs until radix-2. For any value of N, which is a power-of-2, this process requires $log_2 N$ computing stages for computing an N-point DFT [1].

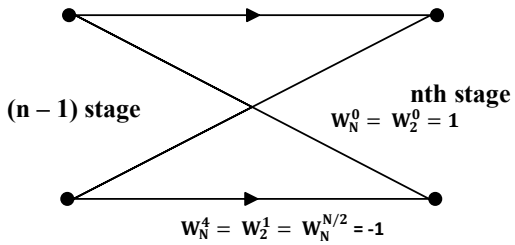


Figure 2.3 Radix-2 butterfly for 2-point DFT computation

The overall computational complexity for N-point DFT would be $N + N + 4(N/4)^2$ number of complex multiplications and complex additions. To generalize it for any number N that is a power-of-2, $log_2 N$ number of stages are needed for N-point DFT computations. Hence, total number of complex multiplications and additions are $Nlog_2 N$.

[2] Equations (2.10) and (2.11) represent the $N/4$ -point DFTs.

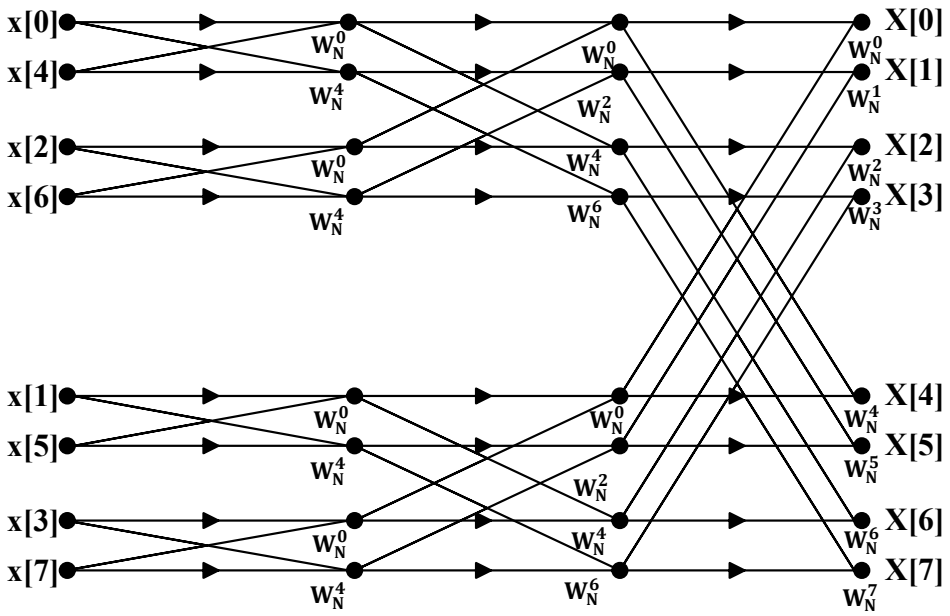


Figure 2.4 Signal flow graph of 8-point DIT FFT

The computational complexity is reduced by using the periodicity and symmetry properties of twiddle factor, which is defined in Eqn. (2.3). For example, simplified structure of radix-2 butterfly in Fig. 2.3 is redrawn in Fig. 2.5. This structure requires only one complex multiplication as comped to Fig. 2.3 structure, which demands two complex multiplications. Because of exploiting the following DFT properties:

$$\textit{Symmetry property: } W_N^{k+\frac{N}{2}} = -\,W_N^k.$$

$$\textit{Periodicity property: } W_N^{k+N} = W_N^k.$$

For example, to compute 2048-point DFT, $N = 2048 = 2^{11}$. This implies $N^2 = 4194304$ and $Nlog_2N = 22528$. Notice that the computational complexity is significantly reduced.

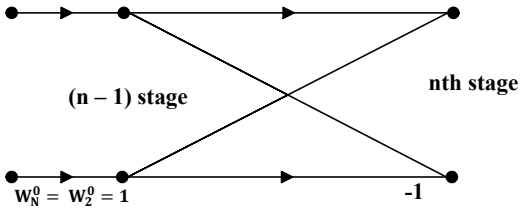


Figure 2.5 Simplified structure of radix-2 butterfly

The signal flow graph of 8-point DIT DFT using simplified structure of radix-2 butterfly is illustrated in Fig. 2.6. Now, the computational complexity of DFT illustrated by Fig. 2.6 is reduced about 50%, i.e., the number of complex multiplications are reduced by the factor 2 as compared to SFG illustrated in Fig. 2.4.

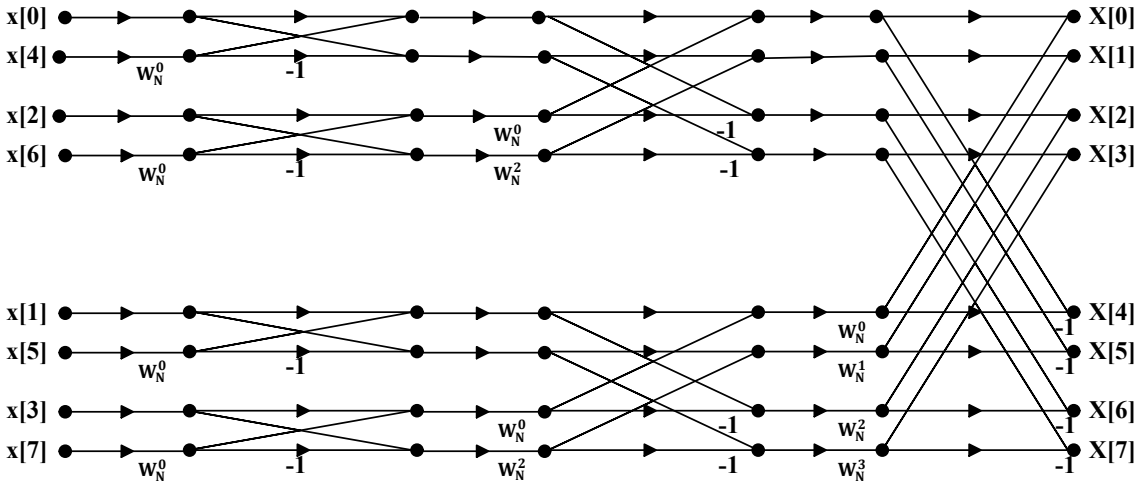


Figure 2.6 8-point DFT by using simplified radix-2 butterfly structure

2.2 Decimation-in-Frequency FFT

Decimation-in-Frequency (DIF) algorithms are built on structuring the DFT computation by decomposing $X[k]$ i.e. output sequence into smaller subsequences [1].

For DIF algorithms, reconsider N as a power-of-2 and individual computation of even and odd indexed samples. Since, the DFT $X[k]$ for an input sequence $x[n]$ is defined by the following equation.

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, \ldots, N\text{-}1 \tag{2.12}$$

and even indexed samples are

$$X[2k] = \sum_{n=0}^{N-1} x[n] W_N^{n(2k)}, \qquad k = 0, 1, \ldots, N/2\text{-}1. \tag{2.13}$$

Rewrite equation (2.13) as follows

$$X[2k] = \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^{n(2k)} + \sum_{n=N/2}^{N-1} x[n] W_N^{n(2k)} . \tag{2.14}$$

The previous equation can also be expressed as

$$X[2k] = \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^{n(2k)} + \sum_{n=0}^{\frac{N}{2}-1} x\left[n + \frac{N}{2}\right] W_N^{(n+\frac{N}{2})(2k)} . \tag{2.14}$$

Since W_N^{2nk} is periodic, we have

$$W_N^{2k(n+\frac{N}{2})} = W_N^{2kn} W_N^{kN} = W_N^{2nk} \tag{2.15}$$

and

$$W_N^2 = W_{N/2}. \tag{2.16}$$

By using Eqns (2.15) and (2.16), we can write the equation (2.14) as

$$X[2k] = \sum_{n=0}^{\frac{N}{2}-1} (x[n] + x\left[n + \frac{N}{2}\right]) W_{N/2}^{nk}, \qquad k = 0,1, \ldots, N/2 - 1. \tag{2.17}$$

Eqn (2.17) represents the $N/2$-point DFT by summation of first and second halves of the input sequence.

Now, consider the odd indexed samples:

$$X[2k + 1] = \sum_{n=0}^{N-1} x[n] W_N^{n(2k+1)}, \; k = 0, 1, \ldots, N/2 - 1 . \tag{2.18}$$

We can rewrite Eqn. (1.18) as follows

$$X[2k + 1] = \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^{n(2k+1)} + \sum_{n=\frac{N}{2}}^{N-1} x[n] W_N^{n(2k+1)} . \tag{2.19}$$

Then the second half of Eqn. (2.19) can be expressed as

$$\sum_{n=\frac{N}{2}}^{N-1} x[n] W_N^{n(2k+1)} = \sum_{n=0}^{\frac{N}{2}-1} x\left[n + \frac{N}{2}\right] W_N^{(n+\frac{N}{2})(2k+1)} \tag{2.20}$$

$$= W_N^{\frac{N}{2}(2k+1)} \sum_{n=0}^{\frac{N}{2}-1} x\left[n + \frac{N}{2}\right] W_N^{n(2k+1)}.$$

Since, $W_N^{\left(\frac{N}{2}\right)2k} = 1$ and $W_N^{\frac{N}{2}} = -1$ we have

$$W_N^{\frac{N}{2}(2k+1)} \sum_{n=0}^{\frac{N}{2}-1} x\left[n + \frac{N}{2}\right] W_N^{n(2k+1)} = - \sum_{n=0}^{\frac{N}{2}-1} x\left[n + \frac{N}{2}\right] W_N^{n(2k+1)}. \tag{2.21}$$

By using Eqn. (2.21) in Eqn. (2.19), we get

$$X[2k + 1] = \sum_{n=0}^{\frac{N}{2}-1} \left(x[n] - x\left[n + \frac{N}{2}\right]\right) W_N^{n(2k+1)}, \qquad k = 0, 1, \ldots, N-1 \tag{2.22}$$

and

$$X[2k+1] = \sum_{n=0}^{\frac{N}{2}-1} \left(x[n] - x\left[n + \frac{N}{2}\right] \right) W_N^n \, W_{N/2}^{nk}, \quad k = 0, 1, \ldots, N\text{-}1 \; . \tag{2.23}$$

Eqn. (2.23) represents the $N/2$-point DFT by subtracting the second half of the input sequence $x[n]$ from the second half of the input sequence and finally multiplying the result with W_N^n. Fig. 2.7 illustrates the mathematical expressions of equations (2.17) and (2.23).

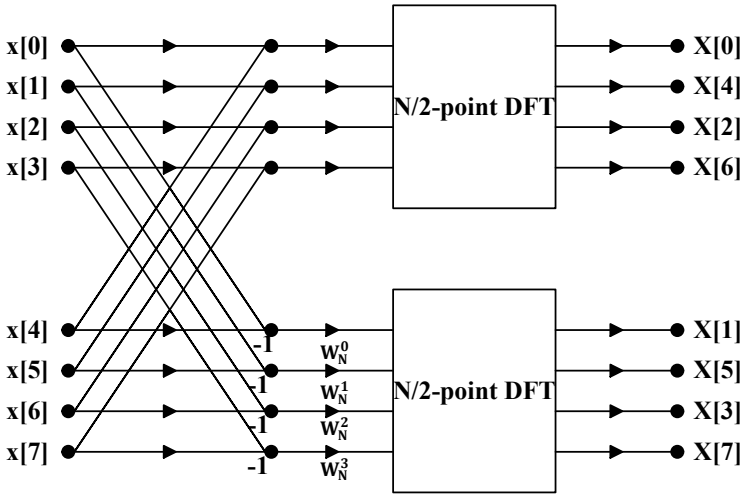


Figure 2.7 DIF decomposition of N-point DFT into two N/2-point DFTs, N = 8

Now, similar to DIT decomposition, N is a power-of-2, which makes $N/2$ even as well. Therefore, even and odd number of samples in Eqns (2.17) and (2.23) computes $N/2$-point DFT. Furthermore, $N/4$-point DFTs is computed in a similar way.

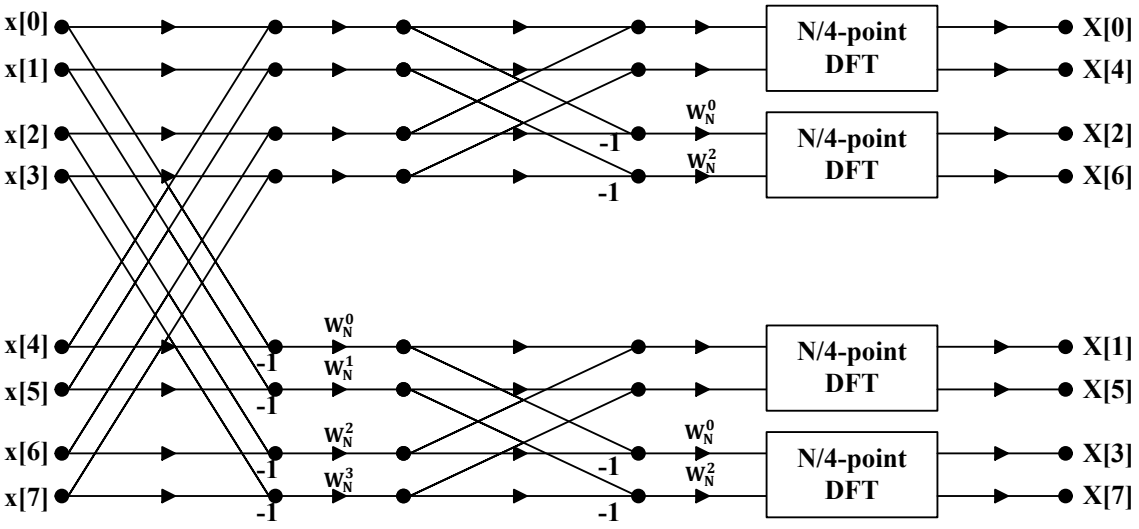


Figure 2.8 DIF decomposition of 8-point DFT into four 2-point DFTs

Fig. 2.8 illustrates that the computations of 8-point DFT has been reduced to 2-point DFTs. Now, replace the 2-point DFTs in Fig. 2.8 by 2-point butterfly structure illustrated in Fig. 2.9. Consequently, SFG of 8-point DFT is illustrated in Fig. 2.10.

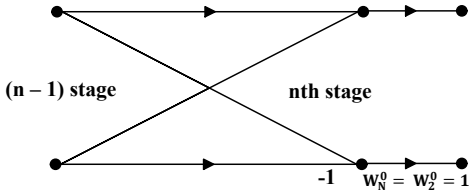


Figure 2.9 Radix-2 DIF butterfly structure

To generalize N, where N is a power-of-2, the computational complexity for DIF is same as DIT algorithm i.e. $Nlog_2N$ complex multiplications and $(N/2)log_2N$ complex additions [1]. Twiddle factor multiplication stages are the major difference between DIT and DIF approaches.

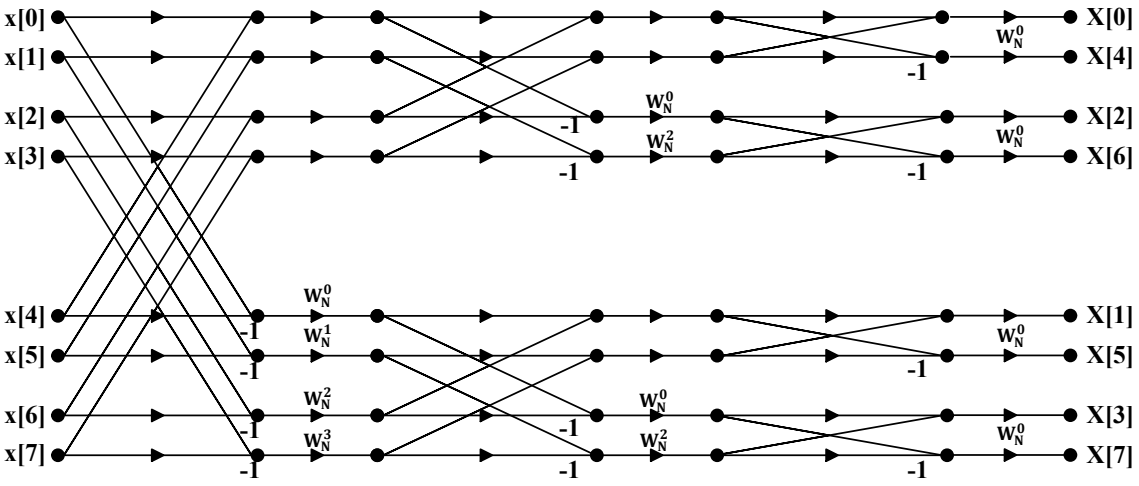


Figure 2.10 SFG of 8-point DFT using radix-2 DIF decomposition

2.3 FFT Algorithms

FFT-The Fast Fourier Transform [1], [2], [3] is an efficient algorithm for DFT computation. In FFT algorithms, decompositions are applied recursively up to required small point DFTs called as radix. Small point DFTs can be radix-2, radix-4 and mixed radix. The approach to develop an FFT algorithm have enormous effect on the computational complexity of DFT. Some of the FFT algorithms are introduced here.

2.3.1 Radix-2 FFT

N-point DFT $X[k]$ of an input sequence $x[n]$ is defined by equation (2.1) as

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, \ldots, N\text{-}1,$$

where $W_N = e^{-j2\pi/N}$.

Cooley and Tukey proposed radix-2 algorithm in 1965 [4] which is very beneficial for data vector of size *N*, where *N* is a power-of-2:

$$N = r^v, \tag{2.24}$$

where *r* and *v* both are integers and *r* is called the radix. When *r* =2, algorithm is called as radix-2 FFT algorithm. Radix-2 FFT algorithm proceeds by decomposing *N*-point DFT into two *N*/2 point DFTs, recursively iterates up to 2-point DFTs. DIT, and DIF decompositions of DFT in sections 2.2 and 2.3 are based on radix-2 FFT. In radix-2, FFT the basic unit of computation is 2-point butterfly, shown in the Fig. 2.11.

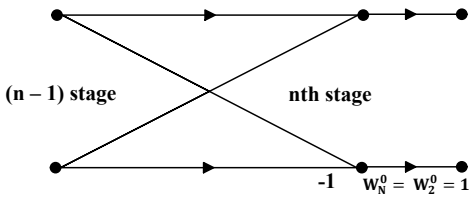


Figure 2.11 Radix-2 DIF butterfly structure

2.3.2 Radix-4 FFT

Radix-4 FFT algorithm is suitable for $N = 4^v$. Although we can also use radix-2 FFT algorithm for these data points but for this particular case we are considering radix-4 FFT algorithm. Similar to radix-2 FFT algorithm, radix-4 FFT algorithm also has both DIT and DIF decompositions of DFT. Radix-4 FFT processes 4 input samples at the same time rather than radix-2 FFT which processes 2 input samples at the same time. Hence, number of FFT stages are reduced in radix-4 algorithm than radix-2 FFT. For example, in 16-point FFT computation radix-4 needs only two FFT stages than 4 in radix-2 FFT. The concept of radix-4 algorithm resembles with the radix-2 FFT algorithm. As in radix-2 we recursively decompose *N*-point DFT computation into *N*/2-point DFT computations until 2-point DFTs which defines the radix of the FFT algorithm. Similarly, radix-4 divides the *N*-point DFT into *N*/4-point DFTs and recursively iterates until 4-point DFTs which defines the radix. The *N*-point input sequence is decomposed into four subsequences i.e. $x[n]$, $x[n + N/4]$, $x[n + N/2]$ and $x[n + 3N/4]$ where $N = 0, 1, \ldots, N/4\text{-}1$ [5].

The computations carried out by one radix-4 butterfly are equal to four radix-2 butterflies. The basic radix-4 butterfly is shown in the Fig. 2.12.

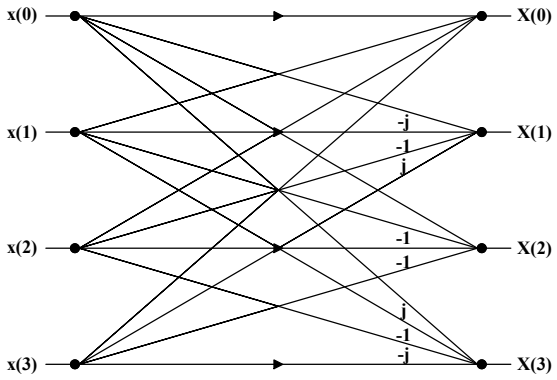


Figure 2.12 Basic butterfly for radix-4 FFT algorithm

2.3.3 Radix-2^k FFT

The radix of the FFT algorithm extremely effects the FFT processor's architecture. Small radix gains importance because of simple butterfly structure. However, high radix results in reduction in FFT twiddle factor multiplications. The concept of radix-r^k is to earn both advantages simultaneously, simple butterfly structure and minimum number of twiddle factor multiplications [6].

As the number of stages in FFT algorithms can be reduced by using higher radix but the complexity of FFT butterfly increases in high radix algorithms. One of the most popular FFT algorithms is radix-2^k, which uses simple 2-point butterfly structure to take advantage of higher radices. For instance, radix-2^2, which has similar number of non–trivial multiplications as radix – 4 FFT algorithm [6]. Fig. 2.14 shows the binary tree of radix-2^2 FFT algorithm. Furthermore, Figs. 2.13 and 2.14 depict the binary tree representation of radix 2, 4, and 2^2.

The binary trees [7 – 10] of radix – 2 DIT, radix -2 DIF and radix – 4 FFT algorithms are shown in the Fig. 2.13. In the binary tree representation, a weight is assigned to each of the nodes to represent a set of 2^n-point DFTs. Let $n(u)$ be the weight of a node u. If a 2^{l+r}-point DFT is decomposed into 2^l-point and 2^r-point DFTs then the node $(l + r)$ has left and right children of weight l and r respectively. Now the left child represents the set of 2^r inner DFTs of 2^l points and the right child represents the set of 2^l outer DFTs of 2^r points. Thus, the weight of the parent node is always equal to the sum of the weights of its children. After the binary tree structure is finalized then all the leaf nodes represent the BFs. Internal nodes represent the twiddle factor multiplication between the two sets of child DFTs. In comparison, radix – 4 algorithm has half stages than radix – 2 FFT algorithm [10, 11].

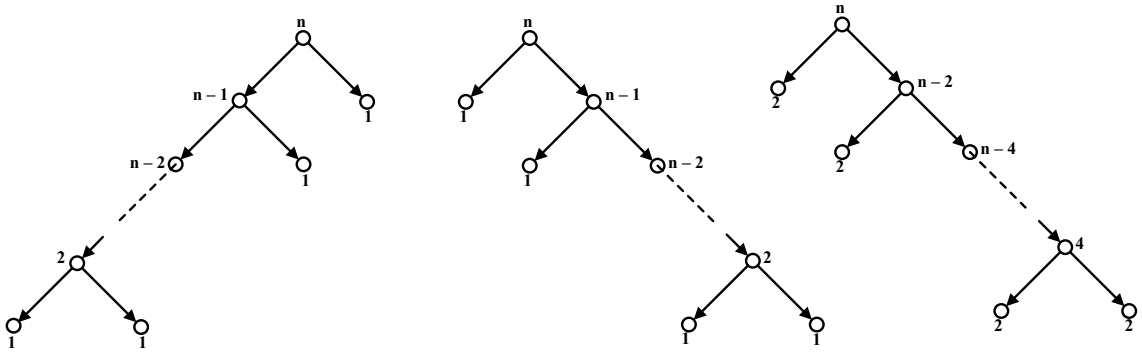


Figure 2.13 Binary trees of FFT algorithms: (a) Radix – 2 DIT, (b) radix – 2 DIF, (c) radix – 4 DIF

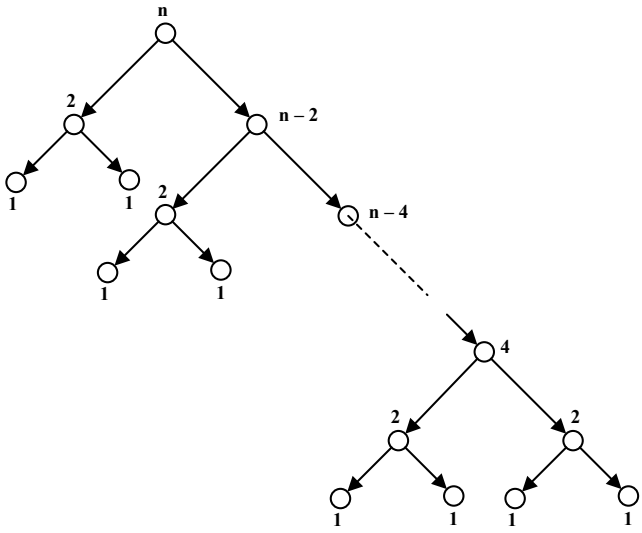


Figure 2.14 Binary tree of radix - 2^2 FFT algorithm

We can notice from the figure that every alternate stage has W_4 twiddle factor multiplication where W_4 is a trivial twiddle factor multiplication. Moreover, we can use radix-2^3, radix-2^4, …, radix-2^v to implement radix–8, radix–16, …, radix-2^v butterflies by using radix – 2. The twiddle factor stages in radix - 2^3, radix - 2^4, radix - 2^v are same as radix–8, radix–16, …, radix-2^v FFT algorithms. Twiddle factor multiplication stages of radix–2, radix-2^2, radix-2^3, radix-2^4 are written in Table 2.1.

Radix	**Twiddle Factor Multiplication Stages**						
	1	2	3	4	5	…	s
2	W_N	$W_{\frac{N}{2}}$	$W_{\frac{N}{4}}$	$W_{\frac{N}{8}}$	$W_{\frac{N}{16}}$	…	W_4
2^2	W_4	W_N	W_4	$W_{\frac{N}{4}}$	W_4	…	W_4

2^3	W_4	W_8	W_N	W_4	W_8	…	W_8
2^4	W_4	W_8	W_{16}	W_N	W_4	…	W_{16}

Table 2.1 Twiddle factor multiplication stages for radix–2, radix-2^2 , radix-2^3, radix-2^4 FFT algorithms

2.3.4 Identical Radix-2^2 FFT

We have designed this identical radix-2^2 FFT algorithm for the work presented in this thesis. Fig. 2.15 represent the binary tree of identical radix-2^2 for 2,048-point FFT. This algorithm is given name an identical radix-2^2 because the number of non-trivial complex multiplier stages are identical to radix-2^2 and alternate stages of both algorithms have W_4 complex multipliers. The overall complexity is reduced in this algorithm as compared to radix - 2^2. Table 2.1 presents the comparison between complex multiplier stages of radix-2^2 and identical radix-2^2.

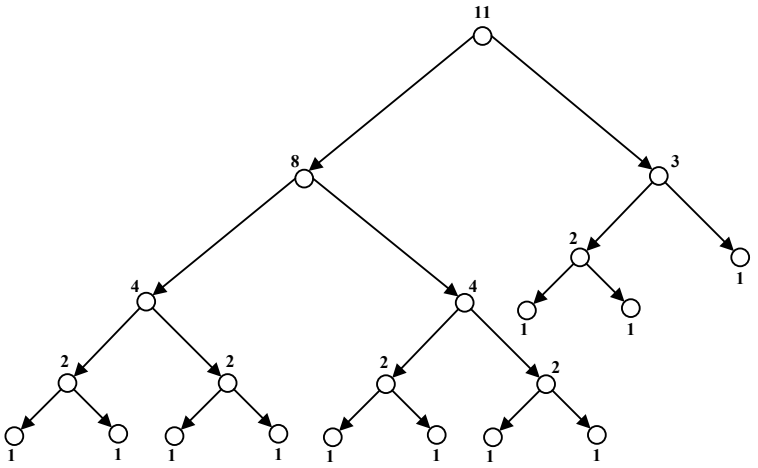


Figure 2.15 Binary tree of identical radix - 2^2 FFT algorithm

Radix	**Number of twiddle factor multiplication stages**									
	1	2	3	4	5	6	7	8	9	10
2^2	W_4	W_{2048}	W_4	W_{512}	W_4	W_{128}	W_4	W_{32}	W_4	W_8
Identical 2^2	W_4	W_{16}	W_4	W_{256}	W_4	W_{16}	W_4	W_{2048}	W_4	W_8

Table 2.1 Complex multiplier stages' comparison between radix-2^2 and identical radix-2^2, N=2,048

3. FFT ARCHITECTURES

The fundamental principle to achieve very high performance in FFT hardware is to arrange the processing elements in such a manner that the architecture matches the structure of the algorithm [10].

There are three most common approaches to design the hardware architectures for FFT algorithms' implementations, named as direct implementation, Memory-based FFT architectures and Pipeline-based architectures. Main units of an FFT processor are a butterfly-processing unit, address generation unit, twiddle factors multiplication circuit, memory units for storing twiddle factors as well as other FFT data and a control unit to generate control signals for the FFT processor [12].

3.1 Direct Implementation

The direct implementation of FFT architecture is simply an identical mapping of signal flow graph of an FFT algorithm on hardware. The number of processing elements (PEs) is exactly equals to the number of operations. Therefore, this hardware architecture may not be an efficient implementation for most of the FFT based applications [12]. This architecture is suitable for smaller size FFTs implementations or when the requirement of throughput is very high. The utilization of butterflies in this architecture is 100% and the hardware cost is proportional to $O(N\log N)$ [10, 12].

3.2 Memory-Based FFT Architectures

Memory-based FFT hardware architectures [10, 12 – 14] are also known as in-place architectures or iterative architectures. The main goal of the memory-based architectures is to improve the utilization rate of butterfly processing unit. These architectures use one or more butterfly processing units for computation of all the butterflies and twiddle factor multiplication of the algorithm. As the memory based architecture, consist of one or more butterfly processing units, so one or more memory units are required to store the data [10]. Memory-based FFT architectures depends on the utilization of memory for computations. Usually, the memory-based architectures are divided into two different categories [12]:

- Single memory FFT architectures.
- Dual memory FFT architectures.

Single memory FFT architectures consist of one memory unit and an FFT processing core, which are, connected to each other though data bus. FFT computation is achieved through fetching the input data from the memory, processing it in the FFT core and storing the results back in the same memory. FFT core includes circuits of butterfly processing element and twiddle factor multiplication. This process is iterated until all the FFT computations are done, giving it the name, iterative FFT architectures [10, 12 – 14].

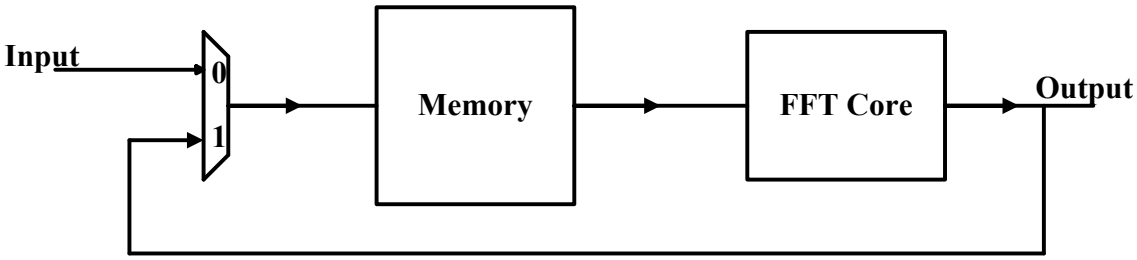


Figure 3.1 Memory-based FFT architecture

Fig. 3.1 shows a very simple memory-based FFT architecture. As noticed in the Fig. 3.1 that FFT data are stored in the memory at every iteration, which makes necessary to compute whole FFT before it receives new samples of data. Therefore, single memory-based FFT architecture is unable to compute the FFT when input data is arriving continuously [10]. This problem can be fixed by adding an extra memory block to store the input data while FFT is being computed. Dual memory architectures fix this problem as illustrated in Fig. 3.2.

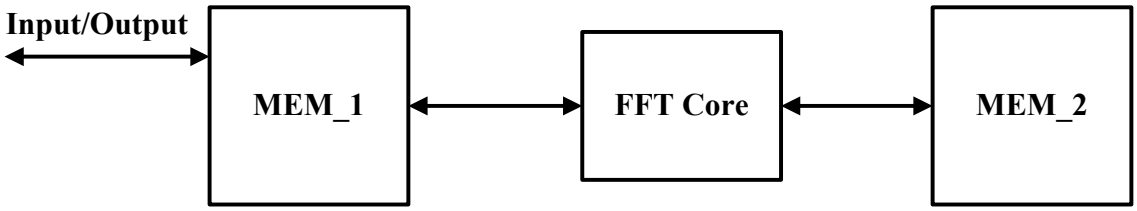


Figure 3.2 Dual memory FFT architecture

Dual memory FFT architectures [12] use two memory units and one FFT processing core. In every clock cycle, an input sample is fetched from the first memory and FFT computed sample is stored into the second memory. After the completion of one FFT stage, the second memory becomes input memory and the first memory becomes the output memory. Memories continue switching roles at every stage until the FFT computation is done. This architecture allows simultaneous reading from and writing samples into memory in one clock cycle. The FFT output can be read from either of these memories depending on the number of FFT stages.

These architectures are very hardware efficient because of low area requirements for large size FFTs. However, there is a tradeoff between the hardware cost and the pro-

cessing speed of the FFT processor because in one clock cycle the processing unit can only perform one butterfly operation, which results in, higher latency and lower throughput or decreasing the overall speed of the FFT processor.

High-radix processing elements can optimize the throughput and latency of the memory-based FFT architectures. However, memory conflict problems occur when memory-based and high-radix used together because of the improper memory accesses. Using multi bank memory can fix these memory conflict problems [15]. Parallel processing elements is another approach to optimize the throughput and latency of the memory-based architectures. The column FFT architectures are used for implementation of parallel processing approach [10, 16 – 17], where entire stage of FFT signal flow graph is computed in parallel. Fig. 3.2 shows the column FFT architecture.

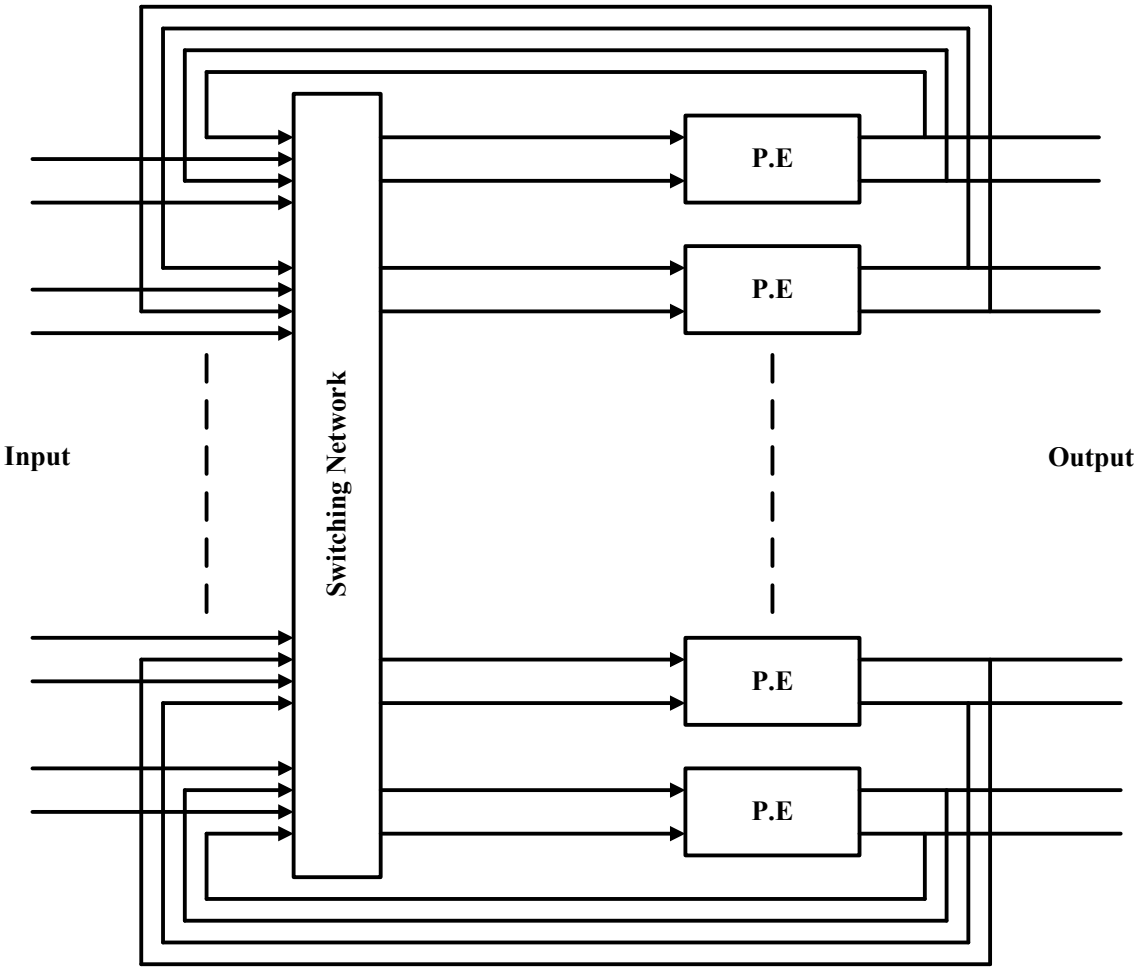


Figure 3.2 Memory-based architectures with parallel PEs

Such architectures have better throughput but more hardware cost than single processing element architectures.

3.3 Pipelined FFT Architectures

Pipelined FFT architectures [10, 18 – 20] are also known as streaming architectures and take advantage of parallel processing between the pipeline stages. Pipelined architectures consist of well-organized structure, area efficient, comparatively simple control unit and have high throughput. One of their considerable advantages is to process continuous flow of data without any additional circuitry in comparison to memory-based architectures, which makes it cost efficient as well. Moreover, it is also possible to increase the clock frequency in pipelined architectures with addition of register, which shorten the critical path and increase the throughput. Pipelined architectures are more flexible when transforms of variable lengths are to be computed with the same circuit. In general, pipelined architectures have a butterfly-processing element between the commutators at each stage. Processing element computes the butterfly operation, then twiddle factors multiplication is performed and commutator (used in feedforward FFT pipelined FFT architectures) has functionality is to rearrange the data to make the successive computations easier. As a result, pipelined architectures are best suited for real-time applications because of high throughput rates.

In pipelined architectures, one stage of signal flow graph of FFT algorithm is computed by exactly one stage of FFT architecture. At every stage, for the continuous flow of data and FFT computations the first half of input samples has to be stored in memory until the second half arrives, i.e., $x(n)$ and $x(n + N/2)$ should be available in order to start the FFT computation [10]. The general pipelined FFT architecture for N-point FFT and $log_r(N)$ number of pipeline stages is shown in Fig. 3.3., where r is the radix of FFT algorithm.

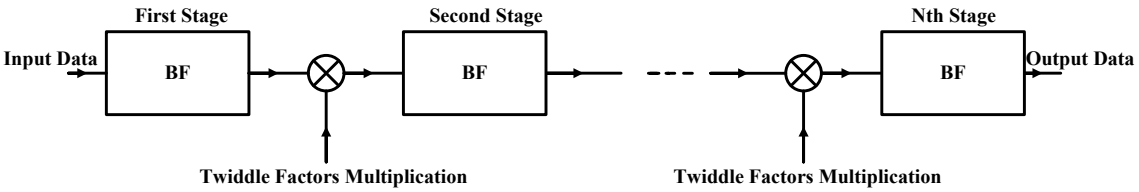


Figure 3.3 General architecture of a pipelined FFT processor

There are two main types of pipelined FFT architectures as follows:

- Feedback FFT Architectures.
- Feedforward FFT Architectures.

3.3.1 Feedback FFT Architectures

There are two types of feedback architectures, named as Single path Delay Feedback (SDF) architectures and Multi-path Delay Feedback architectures (MDF). However, we have only focused on the SDF architectures because of the target applications for this

work. In order to increase the throughput as well as to reduce the chip-area, SDF pipelined FFT architectures are often used.

SDF FFT architectures have a feedback loop at every pipeline stage [21 – 22]. SDF architectures consist of a butterfly processing unit, a feedback memory, i.e., FIFO, memory element at each stage to store the twiddle factor coefficients and a complex multiplier in each stage. Fig. 3.4 shows SDF architecture for N points FFT and $log_r(N)$ number of pipeline stages.

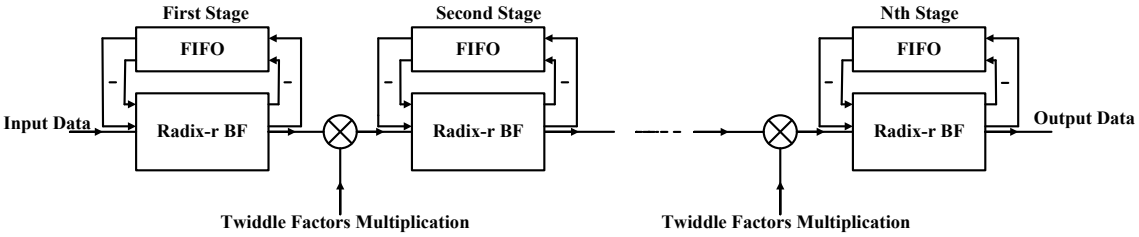


Figure 3.4 Radix-r N-point FFT SDF architecture

The utilization of butterflies in SDF architectures is 50%. These architectures get one continuous stream of input data which is one sample per clock cycle (CC), i.e., one sample per CC is fed to the system and one output sample appears at the output per CC. Fig. 3.5 shows a 16-point radix-2 FFT signal flow graph and its SDF architecture.

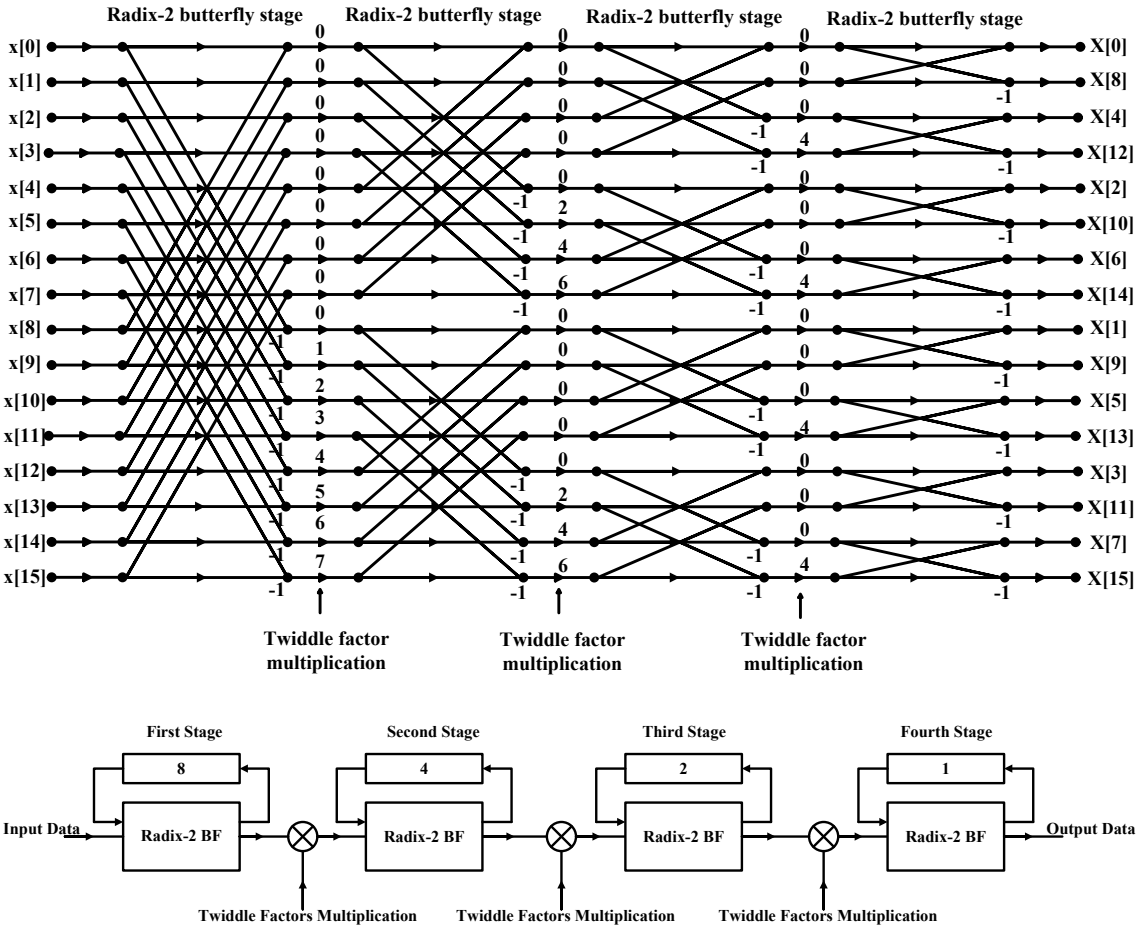


Figure 3.5 Radix-2 16-points FFT Signal Flow graph and its SDF architecture

MDF pipelined FFT architectures are also named as parallel feedback architectures because their structure consists of parallel SDF pipelined FFT architectures to process several samples in parallel [10, 49, 50, 51]. Butterfly utilization ratio of MDF FFT architectures is similar to the SDF pipelined FFT architectures, i.e., 50%. Thus, we cannot optimize the utilization ratio through parallelism. As MDF, architectures can process the several samples in parallel, which can improve the throughput.

3.3.2 Feedforward FFT Architectures

Feedforward FFT architectures are also referred as; Multi-path Delay Commutator (MDC) pipelined FFT architectures [20]. MDC FFT architecture is the most straightforward implementation of FFT algorithm. These architectures do not have any feedback loops; therefore, the FFT data is processed by the FFT butterflies and rotators, and then fed to the next stage of MDC architecture [10]. Thus, utilization ratio of butterflies in this architecture is 100% [20]. This architecture can process several samples in parallel, which means MDC architectures have higher throughput than SDF architectures. Fig. 3.6 shows radix-*r* MDC pipelined FFT architecture.

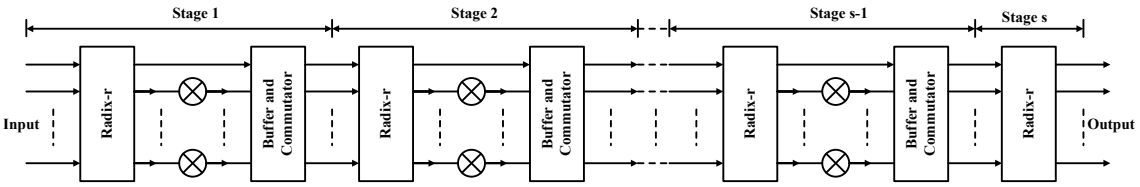


Figure 3.6 Radix-r MDC pipelined FFT architecture

3.4 Mapping of FFT SFG to Hardware

In this thesis, we have implemented SDF pipelined FFT architecture, so we only discuss the hardware design of SDF pipelined FFT processor. Referring to Fig. 3.5, the 16–point radix–2 DIF SFG is mapped to SDF pipelined FFT architecture.

We need $log_2(N)$ pipeline stages, which is four in case of a 16–point FFT. We need $log_2(N) - 1$ number of twiddle factor multiplication stages i.e., three. This is SDF architecture which gets one continuous stream of data which is one sample per CC, i.e., one sample per CC is being fed to this processor and one sample per CC is being appeared at the output. We can divide the operation of the architecture into four stages as follows

1. First Stage
   There are eight 2–point butterflies to compute the FFT in the SFG but in hardware, we need only one butterfly PE per stage because this is a pipelined proces-

sor. From SFG it can be noticed that first butterfly operation takes 1st and 9th samples as input. Second butterfly operation requires 2nd and 10th samples to perform the butterfly operation and so on. Thus, to start the butterfly operation first we will store the first 8 input samples to the FIFO and when the 9th sample appears at the input port then PE start performing butterfly operation at 9th CC on 1st and 9th input samples i.e. it takes 1st sample form the FIFO and 9th sample from the input stream. On the output side, on the same CC two outputs are generated, 1st output is passed to the second stage of the architecture while 9th output is stored back in the FIFO. At 10th CC the butterfly PE repeats the same procedure on 2nd sample i.e. coming from FIFO and 10th sample i.e. coming from the input stream. PE perform butterfly operation from 9th to 16th CC to compute all the eight 2–point butterflies in the first stage of SFG. On the same CC the PE sends first eight outputs to the second stage and last eight outputs are stored back in the FIFO. Butterfly PE go to inactive state for next 8 CCs because other eight outputs stored in the FIFO are now being send to the second stage. Control unit generates the control signals to control the functionality. Control unit of SDF architectures is very sample and can be designed using sample counters.

2. Second Stage
   Firstly, the outputs coming from the first stage are being multiplied with the FFT twiddle factors as shown in the figure and then fed to the second stage PE or FIFO. As SFG represents that, second stage PE requires 1st and 5th sample to perform the butterfly operation, so PE will be in inactive state for first 4 CCs and feed the input samples to the FIFO. In Fifth CC when the 5th samples arrives at the input of second stage PE, it will take 1st sample from the FIFO and perform the butterfly operation. On the output side, two outputs are generated, 1st is send to the third stage and 5th is stored back in the FIFO i.e. behavior is similar to first stage PE. As first stage generate 1st output at 9th CC, so second stage generates 1st output at 13th CC and so on.
3. Third Stage
   PE's behavior is similar to the first and second stage but it stores the first two samples in the FIFO and then start performing butterfly operation. PE of the third stage goes to active and inactive state for 2 CC each state. First actual output after third stage appears at 15th CC.
4. Fourth Stage
   Behavior of PE is similar to last three stages but it stores one sample to the FIFO and then perform the butterfly operation in the second CC. PE switches between active and inactive state after every CC. First actual output after fourth stage i.e. first FFT output of a 16–point radix–2 SDF pipelined processor appears at 16th CC and so on.

The butterflies' utilization ration in SDF pipelined FFT architectures is 50% as butterfly PE perform computations for 50% of the time and the other 50% it stays inactive.

3.5 Input/Output Order or Bit-Reversal

Bit-reversal is an algorithmic technique, which is very important for radix-2 Cooley-Tukey FFT algorithms [4] and has been of strong interest. This algorithm generates indexed data through reversal of bits. The output frequencies of FFT are usually in bit-reversed order and we use this algorithm to sort out those frequencies in order [10].

For *N*-point FFT, where *N* is a power-of-2, algorithm rearranges the data samples through reversal of the bits of the index. This is simply an inversion of the bits of the indices. For example any data samples with indices $I = b_{n-1}, \ldots, b_1, b_0$ are moved to the place $BR(I) = b_0, b_1, \ldots, b_{n-1}$.

4. TWIDDLE FACTOR MULTIPLICATIONS

Two key modules for mathematical operation of the FFT are the butterfly operation and the twiddle factor multiplication. Twiddle factor has defined by equation (2.3) and (2.4) in chapter 2. The butterfly block of the FFT consist of an adder and a subtractor, which perform addition and subtraction on two input data samples of FFT. Twiddle factor block usually requires a complex multiplier and a memory to store the twiddle factor coefficients. The butterfly operation of FFT is usually followed by the twiddle factor multiplication in each stage of FFT architecture for computation [23]. In this chapter, we have discussed different techniques to implement FFT twiddle factor multiplication.

4.1 Implementation Techniques

There are several techniques to implement FFT twiddle factor but we have discussed the following two techniques:

- General Multiplier.
- Constant Multiplier.

4.1.1 General Multiplier

A general twiddle factor multiplication consists of a memory to store twiddle factor coefficients and a complex multiplier as represented in the Fig. 4.1. The most common way of implementing the FFT twiddle factor multiplication is by using a complex multiplier that consist of four real multiplications and two real additions as shown in the following example.

For example, the complex $a + jb$ is the complex number.

$$(a + jb)\,(c + jd) = a.c + ja.d + jb.c + j^2 b.d$$

$$= (a.c - b.d) + j(a.d + b.c)$$

$$= X + jY,$$

where X and Y are the real and imaginary parts of the resultant complex numbers, respectively. There are different ways to implement this complex multiplication. To implement general multiplier in pipelined FFT architectures, real and imaginary parts of the twiddle factors are precomputed and stored in the memory. In pipelined FFT archi-

tectures, each stage performs N twiddle factor multiplications. Therefore, we need to store N twiddle factor coefficients in the memory.

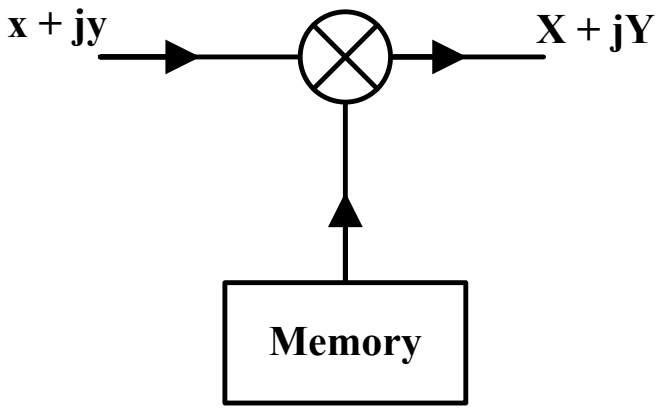


Figure 4.1 General complex multiplier

The multiplication sign in the figure represents the twiddle factor multiplication being performed. This multiplication operation sign has four real multiplications and two real additions inside. An address generation unit is required to address the twiddle factor coefficients' memory. Twiddle factor coefficients are input to the complex multiplier in order to compute the twiddle factor multiplication of the input sample.

4.1.2 Constant Multiplier

A constant multiplier is usually used when one of the operands is already known; for example, twiddle factor multiplication in FFT processors where twiddle factor coefficients are already known.

A constant multiplier is implemented by using shift-and-add operations [7]. We replace the general multiplier with shifts, adders, and subtractors and is called as constant multiplier [24 – 25]. This implementation is sometimes also referred as multiplierless implementation [8 – 9, 25 – 28]. This implementation is cost-efficient for hardware because it replaces the costly multipliers with shifts, adders, and subtractions. In addition to it, it also benefits software implementations, such as embedded processors [25]. For example, $y = 9x$ can be computed as $y = (x << 3) + x$.

Such a solution is known as a multiplier block. In the case of adders and subtractors, the complexity is similar which allows us to refer both of them as adders. Therefore, the term adder can be used to denote cost of the total number of adders and subtractors in the entire circuit of implementation [25].

Constant multiplier block is implemented for the known operands and the challenge is to find the optimal implementation of the constant multiplier block, which means block with minimum number of adders and shifters. Shifters does not have much effect on the hardware cost of the constant multiplier block and can be assumed to be cost free.

4.1.2.1 Single-Constant Multiplication

Single-constant multiplication (SCM) [24 – 25] is a simple method for multiplication of a given constant *c* using adders and shift can only be read off from the bit representation of c. This method is called as binary method.

Canonical Signed-Digit (CSD) [29], a well-known fixed-point number representation, can reduce the number of non-zero bits. Therefore, this method is based on CSD recording, which is explained in section 5.3. However, there are many other ways to implement multipliers more efficiently, which require fewer adders because shifts are assumed free. The main challenge is to reduce the number of adders in the multiplier less circuits.

For example, a constant 45, has CSD representation of $10\bar{1}0\bar{1}01$. The implementation of this pattern is shown in Fig. 4.3, where 45 is computed as 64 – 16 – 4 + 1. Left shift by 1 bit is equivalent to multiplication by 2. Fig. 4.4 optimize the implementation in Fig. 4.3 and reduces the number of adders from 3 to 2. Moreover, the number of actual shifts are also reduced from 6 to 5. Thus, 45 is implemented as $9 \times 5 = (1 + 4)(1 + 8)$.

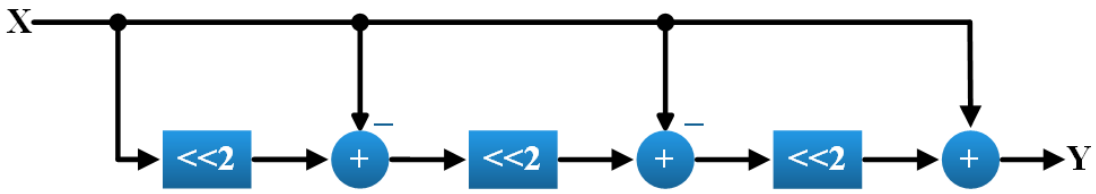


Figure 4.3 Circuit for multiplication of an input X with a constant 45 using 6 actual shifts and 3 adders

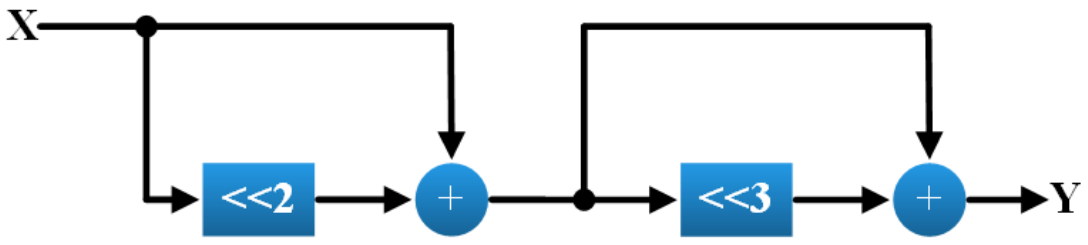


Figure 4.4 Optimized circuit for multiplication of an input X with a constant 45 through 5actual shifts and 2 adders

4.1.2.2 Multiple-Constant Multiplication

Some applications require one signal to be multiplied with a set of given constants c_1, c_2, ... , c_n. The challenge is to find the multiplier block with parallel multiplications with the given set of constants using minimum number of adders [25]. Fig. 4.5 explains the idea of multiple-constant multiplication (MCM).

Figure 4.5 The principle of multiple-constant multiplication

5. NUMBER REPRESENTATION

Number representation gives an exact numeric value to any kind of data. In the field of computers and digital electronics, we always deal with numbers and their representations. Therefore, it is very important to understand the different ways to represent numeric data systematically. Numbers can be represented by two symbols, i.e., 0 and 1; the system based on this representation is called as binary number system. However, it would be wise to discuss the decimal number representation before going through the details of the binary representation because it would much easier to understand the binary number system along their decimal representations (ten possible digits). For example, we can represent the positive one hundred and twenty-five as a decimal number as follows:

$$125_{10} = 1 \times 100 + 2 \times 10 + 5 \times 1 = 1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0,$$

where subscript 10 denotes the base of the decimal number system. To represent a number in decimal system the right most digit is multiplied by 10^0, the next digit is multiplied by 10^1 and so on. Moving from right to left, each digit has a multiplier, which is 10 times of the previous one. In the decimal number system, to multiply a number by 10 is simply achieved by sifting the number to the left by one digit and adding a zero at the place of right most digit. Similarly, to divide a number by 10 can be implemented by shifting the decimal number to the right by one digit or move the decimal place by one digit to the left. To find, how many digits we need to represent a number in decimal number system, we can take logarithm of the absolute value of the number with base 10 and add 1 to it. The integer part of the result would be the number of needed digits [30].

Fig. 5.1 represents an example for calculating required number of digits to represents a decimal number.

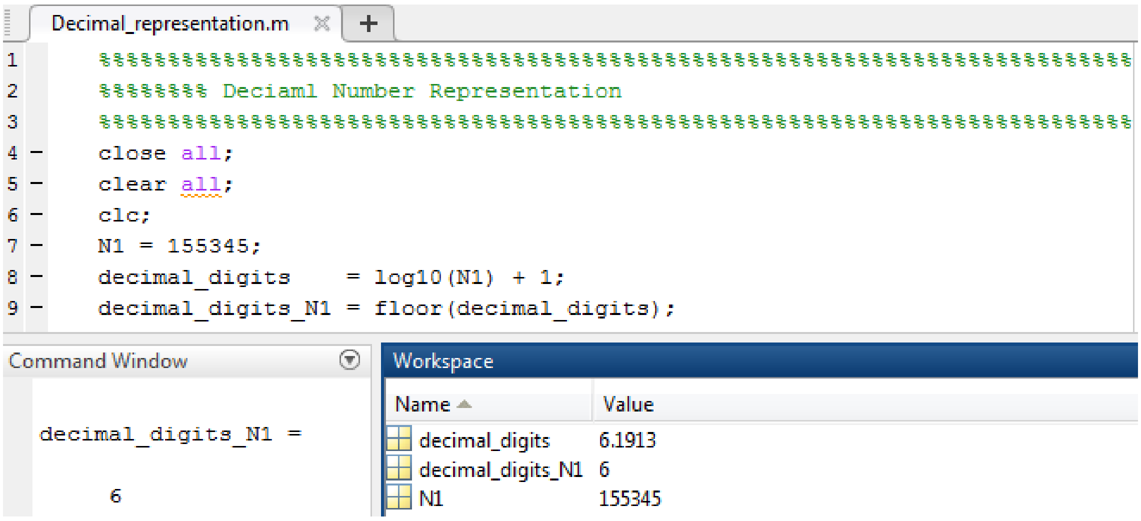


Figure 5.1 MATLAB program for calculating required number of digits to represent a decimal number

As a result, six digits are needed to represent the above number 155345. We can represent 10^n unique numbers with n number of digits. i.e. 0 to $10^n - 1$. Lastly, negative numbers can be represented by just adding a minus - sign before the number [10].

5.1 Binary Number Representation

In a binary number representation system, we have two possible digits (1 and 0) to represent a number. Binary representation of positive numbers can be understood in the similar way to their decimal representation. For example:

$$86_{10} = 1 \times 64 + 0 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1$$

$$86_{10} = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$86_{10} = (1010110)_2 \,,$$

where the subscript denotes the binary number and each digit in the binary number representation is called a bit [30]. Therefore, seven bits represent 1010110. We can represent any number in the binary format by breaking down the number as above example and finding all the powers of two, which add up to the required number.

Multiplication of a number by two in the binary system is simply shifting the number to the left by one bit and filling the right most bit with zero. Similarly, for dividing by two means shifting to the right by one bit. We can find out the number of binary digits needed for the binary representation of a number by taking logarithm of the number with base two and add one to it. For instance, Fig. 5.2 represents an example MATLAB program for binary representation and calculating the required number of digits for representation.

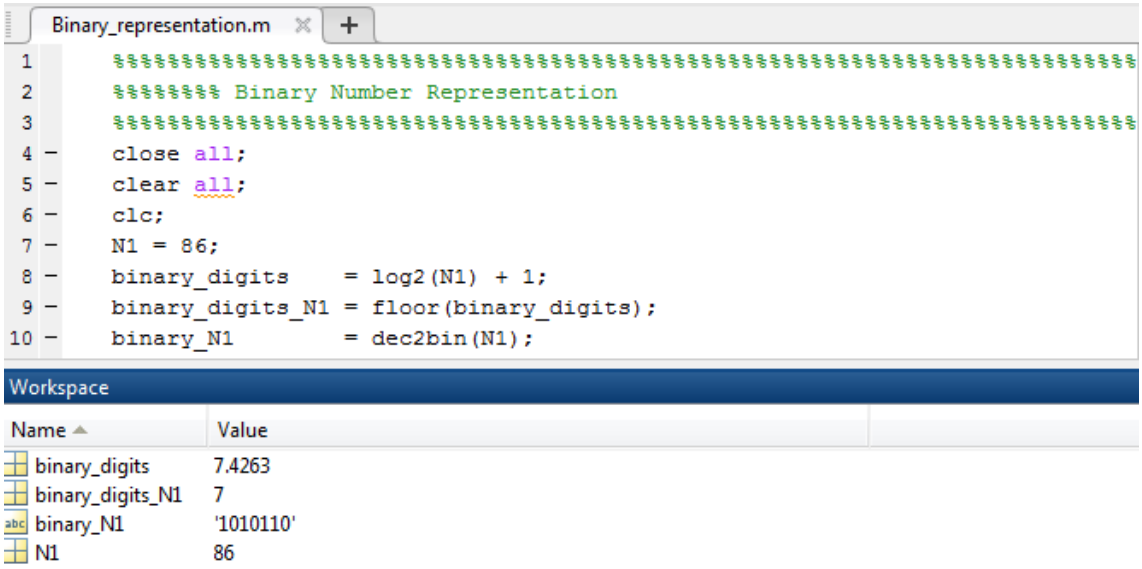


Figure 5.2 MATLAB program for binary representation and calculating required number of digits

As a result, seven digits or bits are needed to represent 86 in the binary format because the integer part is seven [30].

5.2 Two's Complement Representation

Two's complement representation is a convention, which represents the signed binary integers. As in binary number representation each binary digit has a weight which a power of two. This weight increases from right to the left:

$86_{10} = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = (1010110)_2$.

Two's complement of a positive number is a simple binary. Two's complement notation represents the negative numbers. To represent a negative number through two's complement, first we represent the number in binary, then we invert all the bits i.e. 0 becomes 1 and 1 becomes 0. Lastly, we add 1 to the inverted number. To represent the number -86, first we write simple binary of 86 as follows

$86_{10} = (1010110)_2$.

Then we invert all the bits, which results $(0101001)_2$. Now we add 1 to the inverted number, which results in tow's compliment binary of -86, $(0101001)_2 + 1 = 0101010_2 = 101001_2$.

This is the representation -86 in the 6-bit binary representation through two's complement scheme. The rightmost digit in the two's complement representation indicates the sign of the integer. 1 stands for a negative number and 0 stands for a positive number [31].

5.3 Canonical Signed Digit Representation (CSD)

Canonical signed digit (CSD) is a special technique to encode the binary values in signed digit representation. CSD is not unique because it allows a number to be represented in many different ways. CSD representation allows the encoding of a number such that it contains the fewest number of non-zero bits. It is very useful technique to represent the multiplier constants which can significantly reduce the area of hardware implementation [29]. The important properties of CSD representation are as follows

- Mapping the number to a ternary system i.e. CSD representation of a number consists of numbers 0, 1 and -1,
- The CSD representation of a number is unique,
- The number of non-zero digits are minimal, and
- No two consecutive digits in a CSD number are zero.

Let us take an example to understand the conversion of a number into its CSD representation. For example, we write the CSD representation of 287,

Step #01

First, we find the binary of 287.

$287_{10} = 256 + 16 + 8 + 4 + 2 + 1 = (100011111)_2$

Step #02

Starting from the left, the LSB (least significant bit) and moving towards right, the MSB (most significant bit), if we find more than one non-zero digits i.e. 1 or -1, take all of them plus the next zero. If there is no zero, then create a zero.

$(011111)_2$

Step # 03

Add 1 to the above number and force the right most bit to be -1.

$(011111)_2 => (100000_2 - 1)$.

Now, our original number looks like this

$(10010000_2 - 1)$

Since, we can notice that there are no consecutive non-zero digits; our conversion to CSD is complete. Hence, the CSD representation of 287 is $(10010000_2 - 1)$

5.4 Effects of Finite Word Length

Finite word length effects occur when the word length or memory is less than the precision needed to store. These word length effects introduce ´noise and non-ideal system responses.

In FFTs, Finite word length data and arithmetic are used to map FFT algorithm on hardware. This leads to the quantization of internal signals and coefficients in FFT processors' architectures. This determine the hardware resources and input dynamic range precision.

This section discusses these effects in FFTs including round – off/truncation error and overflow.

5.4.1 Overflow

Overflow occurs when the available data range becomes higher than the available finite word length [10]. In FFTs, overflow occurs because of the butterfly operations i.e. addition/subtraction. In general, it is assumed that the input data being fed to the FFT processor is within the range [-1, 1] and is represented using certain number of bits. Thus, results of butterfly operations are not under range [-1, 1] which causes the overflow. Therefore, certain techniques are used i.e. scaling the data to avoid the overflow, performing truncation to convert data to the required word length and rounding the data to minimize the error.

5.4.2 Scaling: An Overflow Handling Technique

Scaling is a technique to handle overflows in fixed-point representation of data while at the same time keeping the signal level as high as possible to reduce the round – off noise [10]. Range of data is adjusted by introducing scaling multipliers. However, the scaling should not disturb the functionality of the system of the transfer function. There are two types of scaling, named as static data scaling and dynamic data scaling.

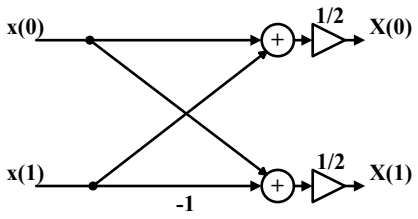


Figure 5.3 Butterfly with scaling

Static data scaling is very common in designing the hardware architectures to avoid overflows. In designing FFT architectures, static data scaling is often used and it intro-

duces a scaling factor after each butterfly operation. In FFT processors, to make sure that overflow will not occur, results after butterfly operations are multiplied by a scaling factor ½ as presented in Fig. 5.3. The reason behind this ½ scaling factor is the FFT butterfly operation i.e. addition and subtraction of data samples. As addition introduces at most one extra bit, so scaling factor of ½ can avoid it.

The basic idea about the dynamic scaling is to scale down the data when required and scale up the data when possible. The static scaling is often too prohibiting as it is statistically rather unlikely, two data signals will add up to a large value, if they do so then the other butterfly output will be small. Thus, it is of a greater interest to find ways to adapt the scaling dynamically to the signals being processed. Many approaches have been presented to perform this dynamic scaling for FFT processors [10].

Dynamic data scaling scales down the data when the butterfly results are large, scales up the data when the butterfly results are smaller and leave the data as it is if it is within the proper range. We do not even need to store the scaling factor if it is same for all data. Before making the scaling factor decision, all data must be processed in a stage. In memory based FFT processors, it is very easier to have memory with slightly longer word length as compared to that of arithmetic operator inputs and some additional logic to keep track of the stored data. For pipelined FFT processors it increases the memory requirements because most of the pipelined architectures do not have access to all intermediate data at the same time [10].

5.4.3 Round–Off and Truncation

In fixed-point representation, not all data within the range can be represented exactly. When data type and scaling cannot represent data exactly, then it must be rounded-off to a representable number.

If we multiply two *w*-bit (*w* is the word length) fixed-point numbers then the product is (2*w*-1)-bit. Eventually, we must truncate this product to w bits by rounding.

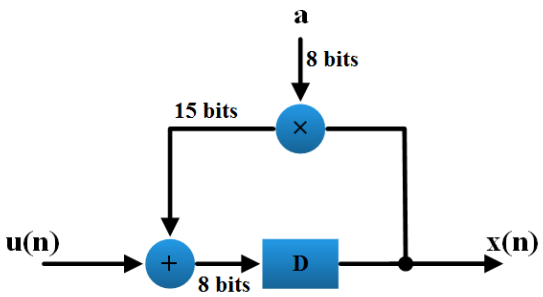


Figure 5.4 1st order IIR filter (W = 8)

For example, let us consider the first order IIR filter as shown in the Fig. 5.4. Assume that input has word length of 8-bits and the multiplier uses 8-bit coefficients. Now, to maintain the full precision at the output, per iteration we have to increase the output word-length by 8 bits, which is infeasible. Thus, we as an alternate solution we can quantize the output word-length by rounding off and truncation to its nearest 8-bit representation [32].

In pipelined FFTs, we need to perform rounding and truncation to quantize the output data to its nearest *w*-bit (word length of the FFT processor) representation at two points in every butterfly stage, after the butterfly operation, after the twiddle factors multiplications. Fig. 5.3 represents the FFT butterfly operation being performed, output word length becomes (*w*+1)-bit because addition requires 1-bit extra. We quantize the output to w-bit word length by the truncating the extra bit and rounding the data to the nearest. Twiddle factors' multiplication doubles the output word length for signed fixed-point representation, we need to truncate output data back to its nearest *w*-bit representation by truncating the extra bits and then rounding the data to the nearest. Finally, we pass *w*-bit results to the next FFT butterfly stage.

Such quantization results in introducing round-off noise, which is important to analyze. The main purpose behind the analysis of round off noise is to determine its effect at the output. If the noise variance at the output is not negligible as compared to the output signal level then we must increase the word length. That is why we compute SNR (signal-t-noise) at the output. It is important to notice that the FFT twiddle factors' multipliers are sources for round off noise in FFT processors [32].

Scale_Round_Trunc.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% Scaling, Rounding and Truncation in FFT Stage
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
x0 = complex(-0.9997, 0.87364); % input1
x1 = complex(0.29345, -0.43735); % input2
% input1 to 18-bit signed fixed-point representation, s=1, f=17, i=0, w=18
% total wordlength of x0 = 36-bits, 18-bit real part and 18-bit imaginary
x0 = fi(x0, true, 18, 17);
x1 = fi(x1, true, 18, 17);
twiddle_factor = complex(0.65434, -0.98766); %twiddle factor
twiddle_factor = fi(twiddle_factor, true, 12, 11); %total wordlength=24-bit
O1 = x0 + x1; % FFT add op: wordlength=38-bit, 1-bit extra for each r, i
O2 = x0 - x1; % FFT sub op: wordlength=38-bit, 1-bit extra for each r, i
O1 = O1/2; % scaling by 1/2: wordlength remains the same
O2 = O2/2; % scaling by 1/2: wordlength remains the same
X1 = fi(O1, true, 18, 17); % trunc and round to 36-bit word: r=18, i=18
X2 = fi(O2, true, 18, 17); % trunc and round to 36-bit word: r=18, i=18
TFmult_O = X1*twiddle_factor; % wordlength=62-bit, r=31, i=31
TFmult_O = fi(TFmult_O, true, 18, 17); % trunc and round to 36-bit word
```

Figure 5.5 MATLAB program for fixed-point scaling, truncation and rounding

The script presented in Fig. 5.5 is a model of a butterfly stage of pipelined FFT processor, which explains, how MATLAB does scaling after performing butterfly operation, how it rounds the data to nearest and truncates the data back to *w*-bit word length. Range of input data is always [-1, 1]. This script is using 36-bit fixed-point representation of input data, with 18-bits of real and imaginary parts. The MSB represents the sign-bit, lower 17-bits represent the fraction. Twiddle factor is represented 24-bit fixed-point representation, with 12-bits of real and 12-bits of imaginary part.

6. PIPELINED FFT PROCESSOR

This chapter discusses the implementation of proposed approach for a pipelined FFT architecture. This architecture consists of modified radix – 2 SDF FFT, which processes length of 2,048-point FFTs. The following pipelined FFT processor architectures are implemented in this work.

- Radix-2^2 FFT architecture
- Identical radix-2^2 FFT architecture

This chapter further discusses the implementation of W_8 and W_{16} constant multipliers. identical radix-2^2 requires less number of complex multiplications and smaller memories to store the twiddle factor coefficients than the radix-2^2 FFT processor. W_8 and W_{16} constant multipliers can be integrated in the identical radix-2^2 for reducing the complex multiplications cost even more, which can remove three complex multipliers and the twiddle memories associated to them. Finally, the chapter provides the comparison between radix-2^2 FFT architecture and identical radix-2^2 FFT architecture.

6.1 Radix-2 FFT Architecture

Fig. 6.1 depicts the radix–2 DIF FFT algorithm based SDF pipelined processor's architecture. This processor computes FFT length of 2,048-point DFT. The number of stages required to compute FFT length of 2048-point is $n=log_2(2048)$, which is 11. Total number of complex multiplier stages required for twiddle factor multiplication is $log_2(2048)–1$, which is 10. Furthermore, 10 memory blocks are required to store the twiddle factor coefficients. A complex multiplier to perform the twiddle factor multiplication as shown in the Fig. 6.1 follows each of the butterfly stage.

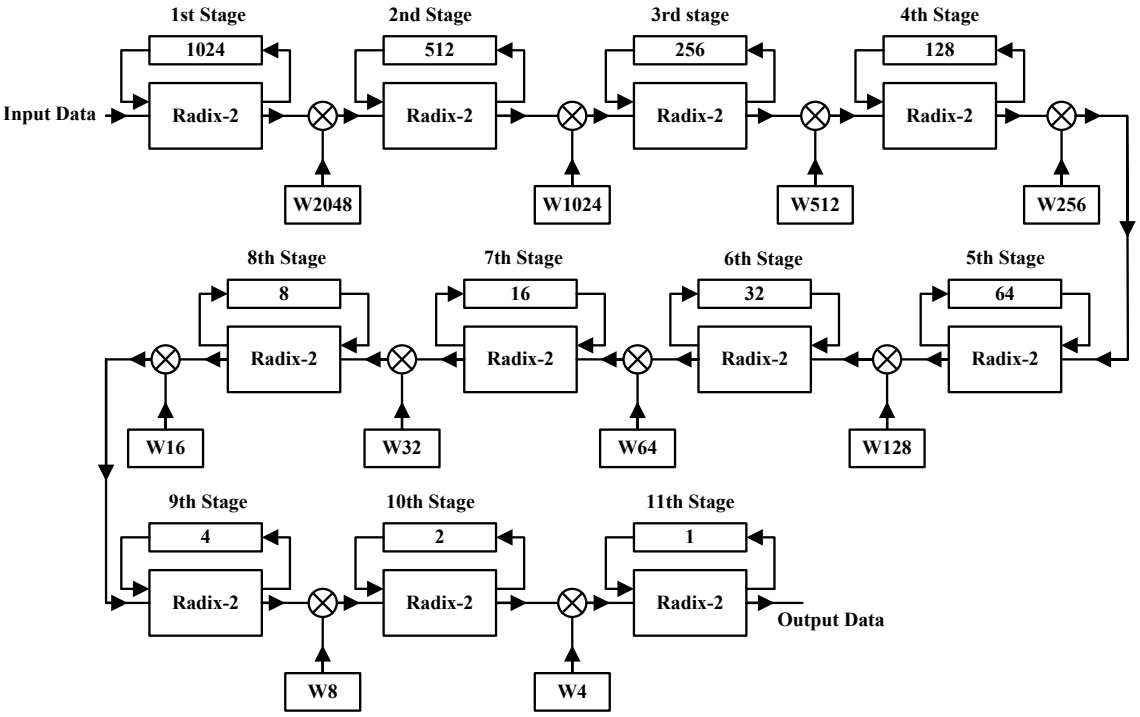


Figure 6.1 2048 points radix–2 SDF FFT architecture

6.2 Radix-2^2 FFT Architecture

This section discusses designs of radix-2^2 SDF architecture using radix-2 (BF I) FFT butterflies and BF II structures, which is an extended radix-2 butterfly structure with W^4 complex multiplier. W_4 performs trivial complex multiplication (multiplication by $-j$), which can be implemented by sign change and exchanging the real and imaginary parts of the data. Therefore, BF I extended structure with W_4 is designed, which is called as BF II structure.

6.2.1 Radix-2^2 FFT Architecture

The Fig. 6.2 represents the radix-2^2 SDF FFT architecture, which is same as radix-2. The difference comes in the stages of twiddle factor multiplication. W_4 is a trivial complex multiplier and can be implemented within radix-2 butterfly (BF I), which is called as BF II. The next section discusses the internal structures of BF II and I.

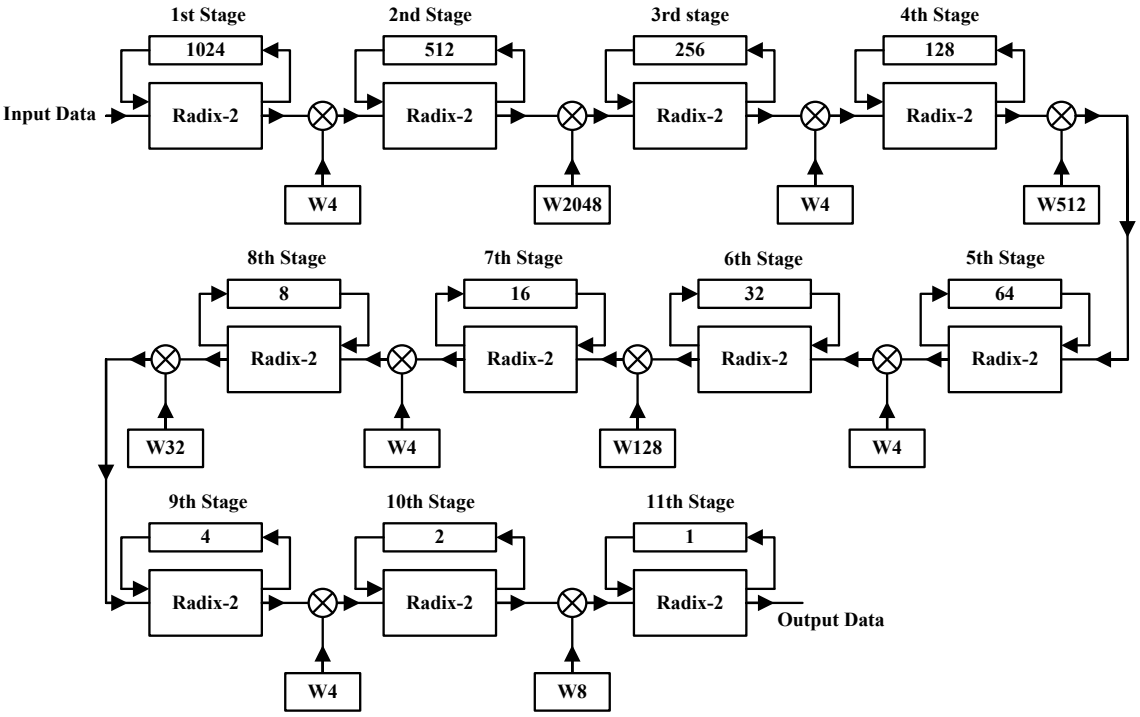


Figure 6.2 2048- point radix-2^2 SDF FFT Architecture

6.2.2 Internal Structures of BF I and BF II

Radix–2 butterfly (BF I) is basic 2-point FFT butterfly, which computes 2-point DFT.

6.2.2.1 Internal Structure of BF I

Fig. 6.3 depicts the internal structure of radix–2 butterfly, which is named as BF I. It simply takes two input samples and computes 2-point FFT. Butterfly operation means performing addition and subtraction operations on the input samples as shown in Fig. 6.3.

S represents the selection line of the multiplexers to choose between performing the butterfly operation on the inputs and to keep the butterfly to inactive mode.

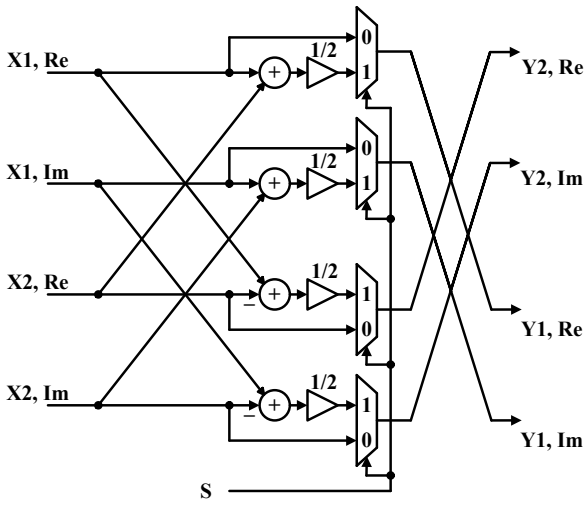


Figure 6.3 Internal structure of a radix–2 butterfly or BF I

6.2.2.2 Internal Structure of BF II

BFII is similar to BF I structure with additional circuitry of W_4. Fig. 6.4 depicts the internal structure of BF II. W_4 multiplication is nothing but the multiplication with a complex number '*-j*'. Thus, multiplication of a complex number with '*-j*' results in effecting the signs of real and imaginary parts of the complex number and swapping the real and imaginary parts.

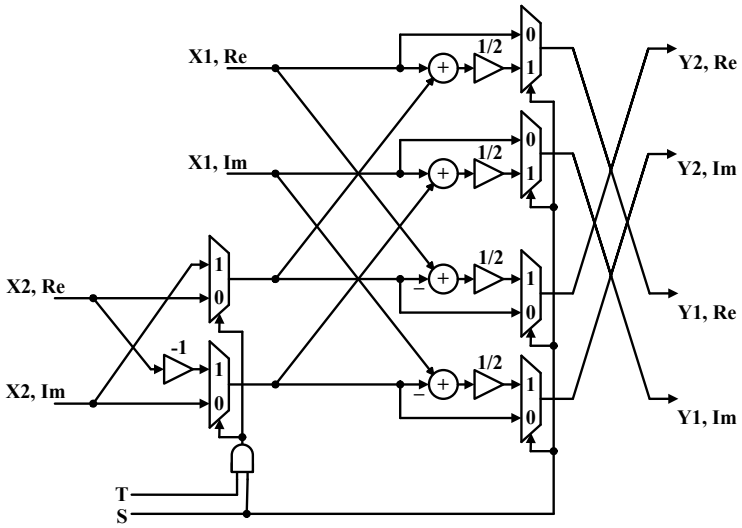


Figure 6.4 Internal Structure of BF II

For example, we have a complex number ***x + jy*** then its multiplication with '*-j*' will result as follows

$$-j * (x + jy) = -j * x + (-j) * jy$$

$$= -jx + (-j^2) * y$$

$$= -jx + (-(-1)) * y$$

$$= -jx + y$$

$$= y - jx.$$

There, the implementation of this W_4 logic can be seen as additional circuitry to BF I in the Fig. 6.4. The signals *S* and *T* are the control signals of the BF II structure to select between BF II and I behaviors.

6.2.3 Improved radix-2^2 FFT Architecture

Fig. 6.5 shows the improved radix-2^2 SDF FFT architecture, which uses BF II and I structures. This resulted in removing all of the W_4 multipliers from the architecture. This improved the architecture because of reduction in twiddle factor coefficients' memories and smaller chip-size as well as reduction in power consumption.

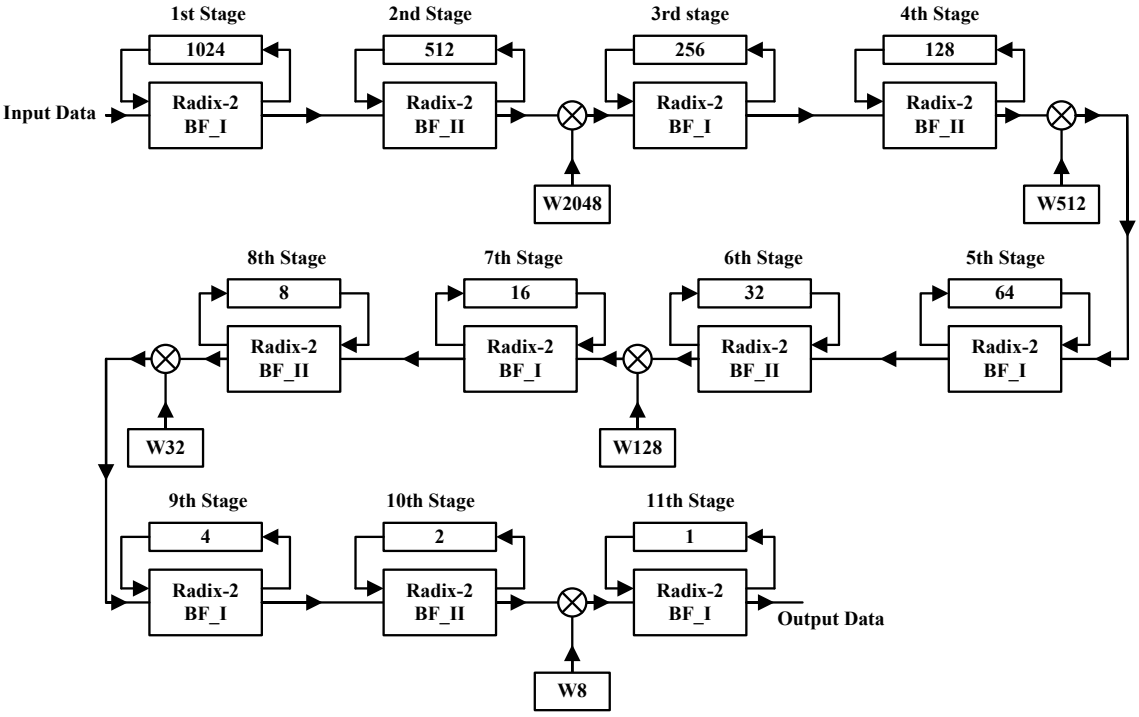


Figure 6.5 Improved 2048-point radix-2^2 FFT architecture

Improved processor's architecture only requires five complex multipliers instead of ten as depicted in Fig. 6.2.

Finally, Fig. 6.6 depicts improved architecture for *N*–point radix-2^2 FFT.

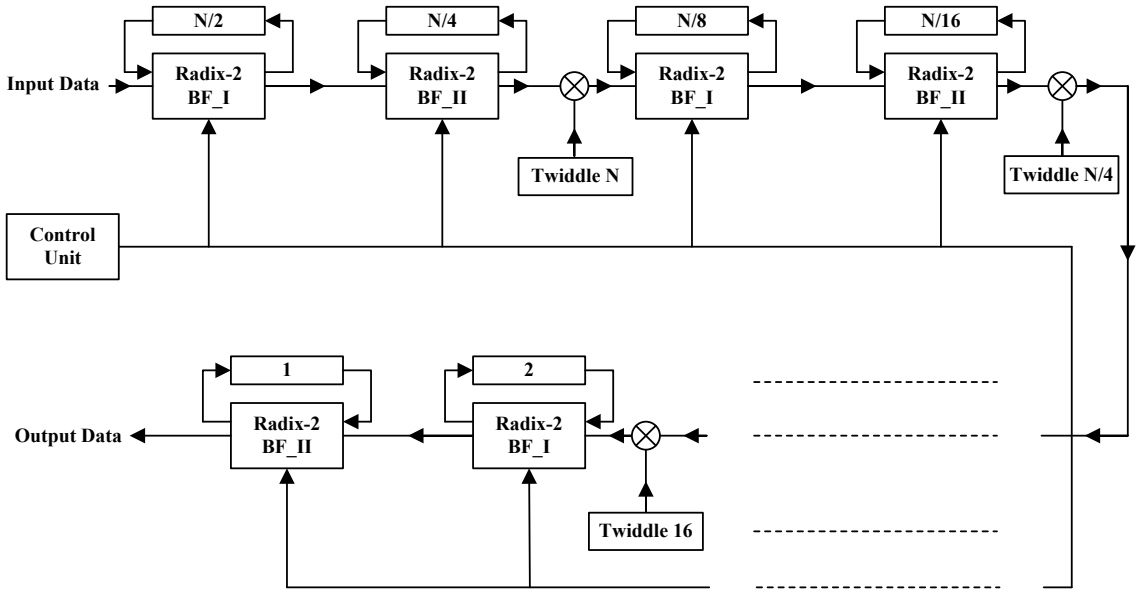


Figure 6.6 N–point SDF FFT Architecture using BF II

The control unit controls the processing and flow of FFT data in the processor as shown in the above figure.

6.3 Identical Radix-2^2 FFT Architecture

This section discusses the identical radix-2^2 SDF FFT architecture. Fig. 6.7 illustrates the SDF architecture for 2048-point FFT for identical radix-2^2 algorithm. The number of complex multipliers are still the same in identical radix-2^2 architecture but the total number of complex twiddle factors' operations are reduced in comparison to radix-2^2 depicted in Fig. 6.5.

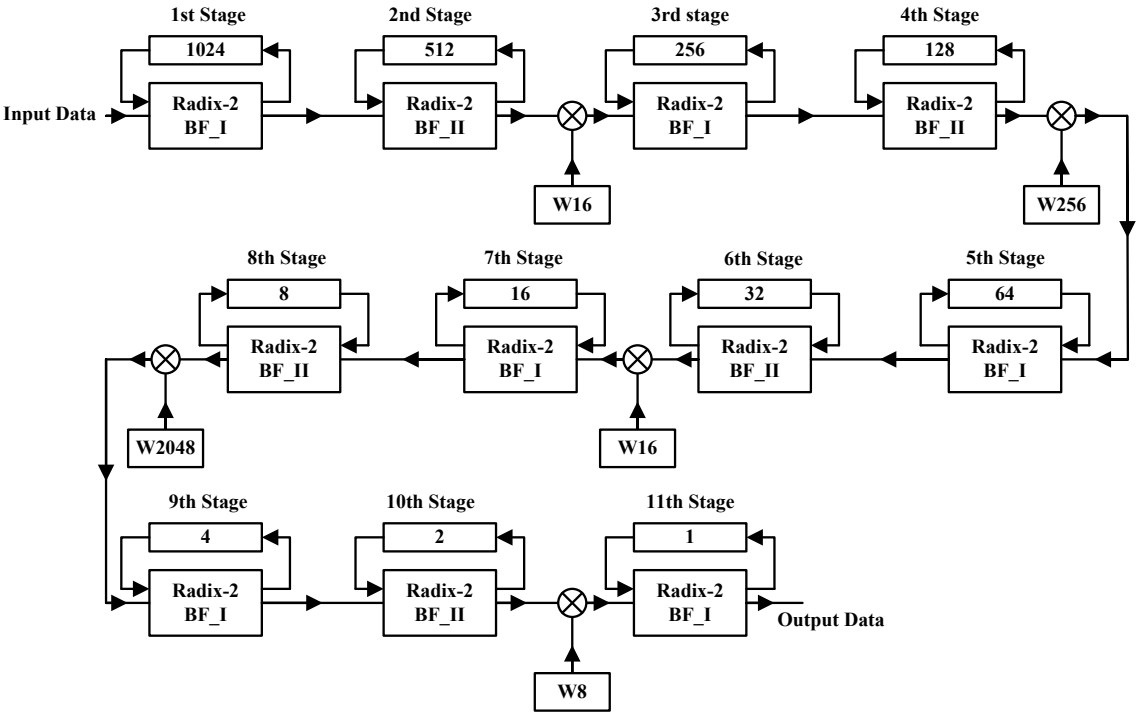


Figure 6.7 Identical radix-2^2 pipelined FFT architecture

6.4 Implementation of Constant Multipliers

Shift-and-add operations implement multiplication by a constant efficiently (section 5.1.2). Shifts, adders, and subtractors [7] replace the general complex multiplier to avoid the multiplier cost. However, we are referring subtractors as adders because of similar hardware complexity. Shifts do not have much effect on the hardware cost and assumed to be cost free. For FFT computation, precomputation of twiddle factors is carried out. A complex multiplier to perform the twiddle factor multiplication follows each stage of the FFT architecture. Constant multiplication circuits replace the complex multipliers to reduce the hardware cost of the FFT architectures. In implementation of constant multipliers, the main goal is to obtain the coefficients with minimum error and smallest number of adders.

FFT computes the twiddle factors by $W_L^i = e^{-j.2\pi i/L}$, $i = 0, \ldots, L-1$. The coefficient sets W_4, W_8, and W_{16} are very common twiddle factors in the FFT architectures, where W_4 is trivial complex multiplier which is very simple to implement. Radix-2 FFTs of size greater than or equal to 16, demand W_8 and W_{16} twiddle factors. For example, 2048-point identical radix-2^2 FFT requires W_{16} at two stages and W_8 at one stage. Twiddle factors are computed from specific sets of angles, which are generated by dividing the circumference in L equal parts. This division of circumference results in multiple symmetries in the complex plane, which defines the concept of octave symmetry. According

to octave symmetry for an L-point twiddle factor, only $M = L/8 + 1$ angles in the range $[0, \pi/4]$ are required [28]. The idea of octave symmetry is illustrated in Fig. 6.8.

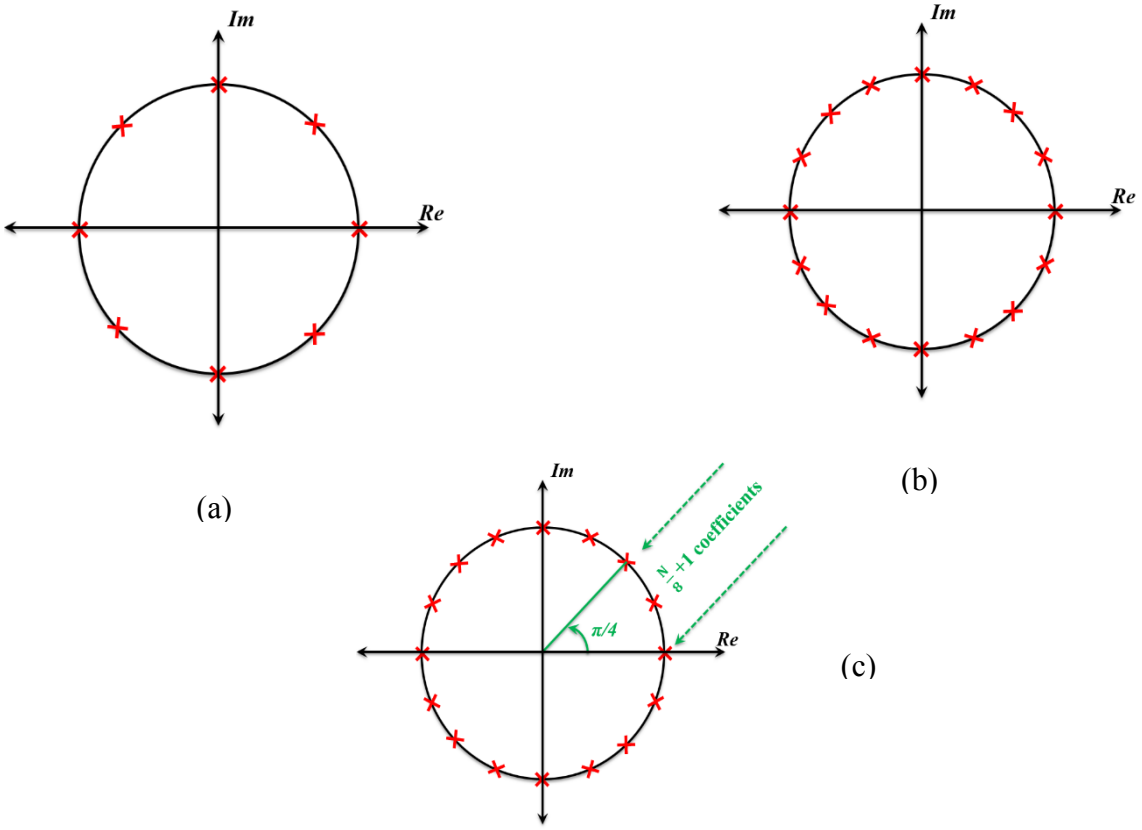


Figure 6.8 W_8, W_{16} and octave symmetry in (a), (b) and (c), respectively

Only $[0, \pi/4]$ angles are considered and rest of the twiddle factors can be computed from those in range $[0, \pi/4]$ through swapping of real and imaginary parts of input and output data samples as well changing their signs [28]. Table 6.1 provides the twiddle factors designed with low complexity with range $[0, \pi/4]$.

TF		Coefficients		Properties	
	0	$\pi/8$	$\pi/4$	W_L	Adders
W_8	577	-	$408 + j408$	11	6
W_{16}	349093	$322520 + j133592$	$246846 + j246846$	20	10

Table 6.1 Designed W_8 and W_{16} twiddle factors for FFT with low complexity [28]

In this thesis work, we have implemented W_8 and W_{16} constant multipliers [28] that can replace the complex multipliers. Figs. 6.9 and 6.10 depicts the schematics of W_8 and W_{16} multipliers. The schematics of both W_8 and W_{16} designed using shifters, adders and multiplexers. The output configurations are represented by ● , ▲ and ■. This constant

multiplication implementation also removes the memories to store the twiddle factor coefficients for W_8 and W_{16} complex multipliers, which makes the FFT architecture memory efficient as well as cost efficient.

6.4.1 W_8 Constant Multiplier

Fig. 6.8 represents the implementation of W_8 constant multiplier. This circuit considers the angles 0 and $\pi/4$ and implements the kernel [577, 408 + j408]. Configuration of the multiplexers decides the multiplication of the input data sample by 577 or 408 + j408 [28].

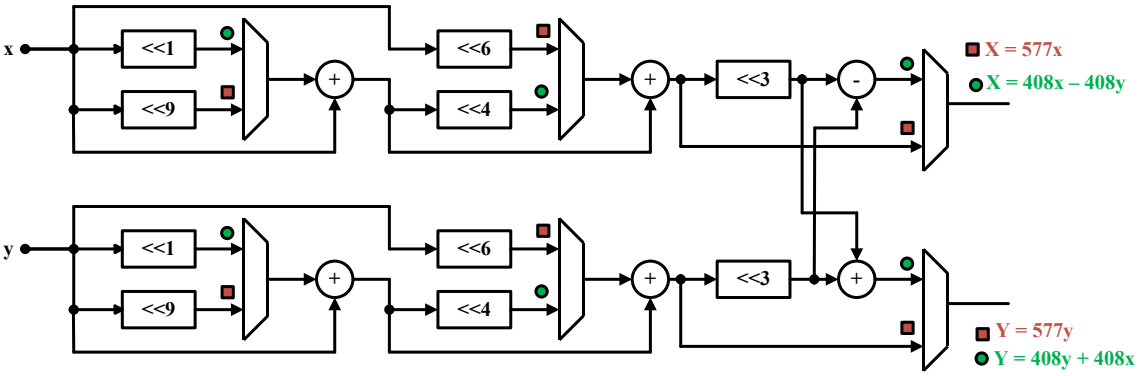


Figure 6.9 W_8 constant multiplier, kernel [577, 408 + j408], [28]

6.4.2 W_{16} Constant Multiplier

The Fig. 6.10 shows the implementation of W_{16} constant multiplier, which considers the angles 0, $\pi/8$ and $\pi/4$ [28]. The kernel implemented is [349093, 322520 + j133592, 246846 + j246846].

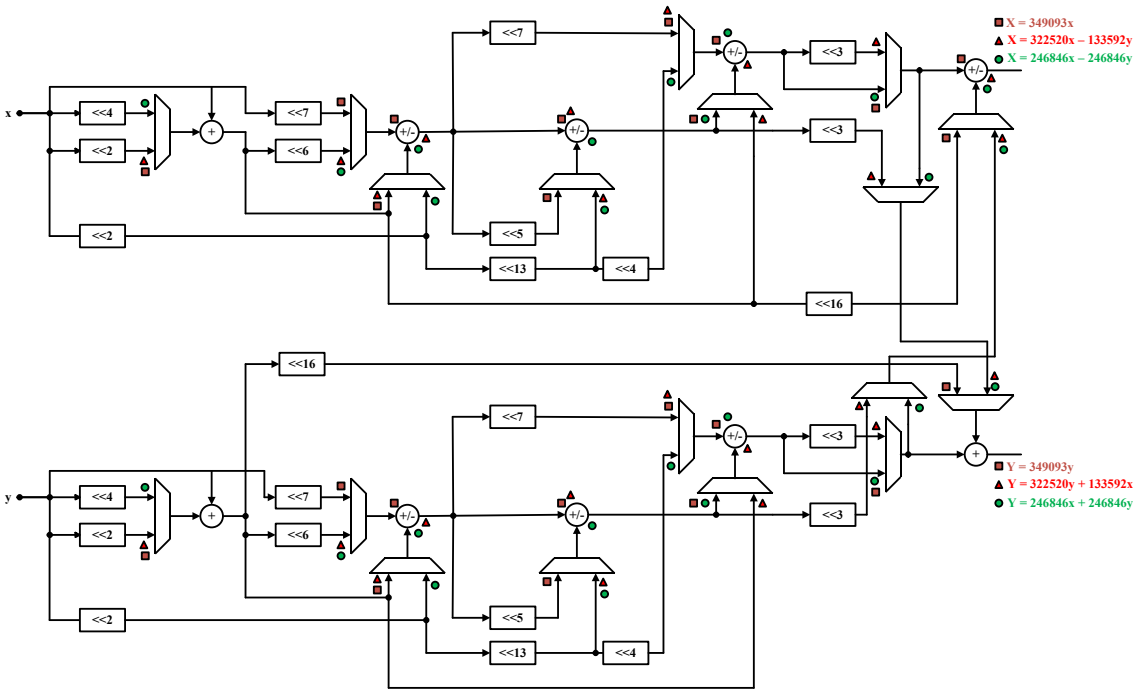


Figure 6.10 W_{16} Constant multiplier, kernel [349093, 322520 + j133592, 246846 + j246846], [28]

7. SYNTHESIS RESULTS

This chapter talks about the verification, synthesis and presents the FPGA implementation results. FFT processors were modeled in MATLAB. VHDL was used for the implementation of the FFT processor using Vivado (Xilinx) 2017.1 tool. Vivado's simulator was used for waveform monitoring and verification of functionality. Finally, the FFT processor were implemented on Virtex-7 FPGA in synthesis.

7.1 Verification

Verification is perhaps the most challenging aspect of any design, there is no excuse for omitting the verification because a design without verification cannot be claimed correctly designed. It is extremely important to verify the functionality of the design. There are different verification methodologies being used in the industry for verification of digital designs e.g. UVM, OVM etc. Another way to verify your design is to use test benches. We have used VHDL test benches for the functional verification of our designed FFT processors.

TB is a piece of code meant to verify the functional correctness of VHDL model. Test bench instantiate the design under test (DUT), feed the stimuli to the DUT, generate the output waveforms and make comparison with the reference outputs. Finally, test bench provides a pass or fail indication.

We Modeled the FFT processor in MATLAB, generated the stimuli and reference in MATLAB using two's compliment binary in fixed-point representation. Data word-length used was 36-bit to represent a complex data sample, MSB 18-bits were used to represent the real part and LSB 18-bits for imaginary part. Data was in range [-1, 1], where no integer bits were involved, i.e., MSB represents the sign and 17 bits represent the fraction.

Stimulus and expected response data samples are vectors stored as constants in an array in TB. TB read the stimulus vector and feed the data samples to the FFT processor as well as it reads the expected response vector and compare with processor's response.

7.2 Synthesis

The main goal of writing VHDL to design digital systems is to generate synthesizable description. The quality of the design is much affected by the coding styles. We must be

able to select coding structures that synthesize the best on FPGA device. We used Virtex-7 (xc7vx1140tflg1930-1) FPGA device for synthesis in this thesis work.

7.2.1 Comparison of FFT Architectures

Table 7.1 presents the comparison between twiddle factor multiplication resources of radix-2^2 and identical radix-2^2 architectures illustrated in Figs. 6.5 and 6.7 respectively. The main goal was to reduce the number of general complex multipliers and memories to store the twiddle factor coefficients. Both radix-2^2 and identical radix-2^2 FFT algorithms were mapped on the radix-2 SDF pipelined FFT architecture. When we are using general complex multipliers, identical radix-2^2 FFT algorithm has reduced twiddle factors memory than radix-2^2, thus less number of complex operations for 2048-point FFT computations. Further, general complex multiplier can be replaced by using the W_8 and W_{16} constant multipliers. Even, twiddle factor memories is reduced further because the constant multipliers do not require any coefficient memory. As a result, the twiddle factor memory can be reduced up to 15% as shown in Table 7.1.

Resources	**Radix-2^2 Fig. 6.5**	**Identical Radix-2^2**	
		General Multipliers Fig. 6.7	**Const. Multipliers W_8 and W_{16}**
# of complex multipliers	5	5	2
# of constant multipliers	0	0	3
Twiddle factor coefficients	2728	2344	2304
Constant Multiplier Name		**# of Constant Coefficients**	
W_8		2	
W_{16}		3	

Table 7.1: Comparison of radix-2^2 and identical radix-2^2 FFT architectures

7.2.2 FFT Processors Synthesis Results

The following table provides the comparison between synthesis results of radix-2^2 and identical radix-2^2 architectures illustrated in Figs. 6.5 and 6.7 respectively.

Resource Type	**Available**	**Radix-2^2 Fig. 6.5**	**Identical Radix-2^2 Fig. 6.7**

Slice LUTs	712000	15911	15896
• LUT as Logic	712000	14687	14672
• LUT as Memory	283200	1224	1224
○ LUT as Shift Registers		1224	1224
Slice Registers	1424000	38329	38345
• Registers as Flip Flop	1424000	38329	38345
F7 Muxes	437600	4896	4896
F8 Muxes	218800	2448	2448
Block RAMs	1880	90 Kb	72 Kb
• RAMB36/FIFO (36 Kb)	1880	36 Kb	36 Kb
• RAMB18 (18 Kb)	3760	54 Kb	36 Kb
DSP48E1	3360	20	20

Table 7.2: Radix-2^2 and identical radix-2^2 synthesis results

7.2.3 Constant Multipliers Synthesis Results

The following table provides the synthesis results of W_8 and W_{16} constant multipliers.

Resource Type	**Available**	**W_8** **Fig. 6.9**	**W_{16}** **Fig. 6.10**
Slice LUTs	712000	231	776
• LUT as Logic	712000	231	776

Table 7.3: W_8 and W_{16} constant multipliers synthesis results

7.2.4 Results Analysis

Referring to Figs. 6.5 and 6.7 of radix-2^2 and identical radix-2^2 FFT algorithms, both algorithms are mapped on the same SDF architecture. The non-trivial complex multiplier stages are identical in both architectures. Without using the constant multipliers, the main difference in the both FFT processors is the number of complex operations, size of memories to store the twiddle factor coefficients and the memory address generation logic. Table 7.1 presents the comparison between the architectures of both FFT processors. The reduction in twiddle factor memories is clearly shown in the Table, which also means the less number of complex operations for computation of 2048-point FFT.

In both implementations, DSP48 blocks are used for general multipliers. Generally, one multiplier requires one DSP48 block, which means four DSP48 blocks are required for one general complex multiplier. These architectures require five general complex multipliers, so 20 DSP48 blocks are used for both implementations as shown in Table 7.2.

There were two different sizes of block RAMs available on Virtex-7 FPGA used, i.e. 36 Kb and 18 Kb. Radix-2^2 used one block RAM of 36 Kb and three block RAMs of 18 Kb. On the other hand, identical radix-2^2 used one block RAM of 36 Kb similar to radix-2^2 but less number of 18Kb block RAMs, which is two as compared to three in radix-2^2. Overall, 20% reduction in the memories is achieved in the identical radix-2^2 as compared to radix-2^2. In addition to it, difference can be seen in the slice LUTs, which is because of the memory address generation logic in the FFT processors.

Moreover, Table 7.3 represents the synthesis results of W_8 and W_{16} constant multipliers. These can be used in identical radix-2^2 for further reduction in memory and general complex multiplier as shown in Table 7.1.

8. CONCLUSIONS AND FUTURE WORK

Signal processing is easier in the frequency domain and DFT is a method that provides transforms between time and frequency domains of signals. FFT is an efficient algorithm for faster computation of DFT.

In this thesis work, we have designed an efficient FFT algorithm named as identical radix-2^2 FFT algorithm that has same number of non-trivial complex multiplier stages as radix-2^2 but less number of complex multiplications. Both radix-2^2 and identical radix-2^2 FFT algorithms were mapped on SDF pipelined FFT architectures.

FFT processors were modeled in MATLAB. VHDL was used for implementations of the FFT processor using Vivado (Xilinx) 2017.1 tool. Functional correctness of processors was verified by writing testbenches, where Vivado's simulator was used for the simulations and generated output samples were compared with the expected output samples of the MATLAB models. Finally, the designs were implemented on Virtex-7 (xc7vx1140tflg1930-1) FPGA device in synthesis. In synthesis, without using the constant multipliers, 20% of reduction in memories is achieved in the identical radix-2^2 than radix-2^2. identical radix-2^2 used less number of block RAMs and slice LUTs as compared to radix-2^2. Reduction in memories and area can be achieved even more by replacing general complex multipliers with W_8 and W_{16} constant multipliers.

Radix-2^2 and identical radix-2^2 FFT architectures are implemented in this thesis work. In comparison, identical radix-2^2 FFT processor performs less number of complex multiplications than radix-2^2 for computation of 2048-point FFT. In addition to it, identical radix-2^2 requires smaller memories to store the twiddle factors than radix-2^2. Furthermore, W_8 and W_{16} constant multipliers are implemented that can replace the two W_{16} and one W_8 general complex multipliers in the identical radix-2^2 FFT processor. Integration of constant multipliers in place of general complex multipliers can further reduce the cost of complex twiddle factor multiplications in the FFT processor as well make the processor even more memory efficient by removing twiddle factor coefficients' memories for W_8 and W_{16} multipliers.

As a result, the work done in this thesis sets a very strong base for the implementation of identical radix-2^2 FFT processor with constant multipliers in extension of this work in future.

BIBLIOGRAPHY

[1] A. V. Oppenheim, R. W. Schafer, “Discrete – Time Signal Processing”, Published by Prentice-Hall, 1999.

[2] L. Wanhammar, “DSP Integrated Circuits”, Published by Academic Press, March 4,1999.

[3] S. K. Mitra, “DIGITAL SIGNAL PROCESSING: A COMPUTER-BASED APPROACH”, Published by Mcgraw Hill Higher Education, 4th International edition, 2010.

[4] J. W. Cooley, J. W. Tukey, “An algorithm for machine computation of complex Fourier series,” in proc. Mathematic of Computation, April 1965, vol. 19, no. 90, pp. 297-301.

[5] Prof. S. S. Belsare, A. S. Padekar, “Radix-4 FFT Architecture”, in proc. International Journal of Advance Research in Computer Science and Software Engineering, 05 May. 2014, vol. 4, no. 5, pp. 337 – 340.

[6] A. Cortes, I. Velez, J. F. Sevillano, "Radix r^{k} FFTs: Matricial Representation and SDC/SDF Pipeline Implementation," in IEEE Transactions on Signal Processing, vol. 57, no. 7, pp. 2824-2839, July 2009.

[7] F. Qureshi, O. Gustafsson, "Generation of all radix-2 fast Fourier transform algorithms using binary trees", in proc. 20th European Conference on Circuit Theory and Design (ECCTD), Linkoping, IEEE, 29 – 31 Aug. 2011, pp. 677-680. 13 Oct. 2011.

[8] M. Garrido, R. Andersson, F. Qureshi, O. Gustafsson, "Multiplierless Unity-Gain SDF FFTs", in proc. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Sept. 2016, vol. 24, no. 9, pp. 3003-3007.

[9] F. Qureshi, O. Gustafsson, "Low-complexity reconfigurable complex constant multiplication for FFTs", in proc. IEEE International Symposium on Circuits and Systems, Taipei, 24 – 27 May 2009, pp. 1137-1140.

[10] F. Qureshi, “Optimization of Rotations in FFTs”, Linköping Studies in Science and Technology, Dissertations, no. 1423, Linköping, 2012.

[11] H. Y. Lee, I. C. Park, "Balanced Binary-Tree Decomposition for Area-Efficient Pipelined FFT Processing", in proc. IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 54, no. 4, pp. 889-900, April 2007.

[12] S. M. Joshi, “FFT Architecture: A Review”, in proc. International Journal of Computer Applications (0975-8887), Apr. 2015, vol. 116, no. 7, pp. 33 – 36.

[13] Sang-Chul Moon, In-Cheol Park, "Area-efficient memory-based architecture for FFT processing", in proc. International Symposium on Circuits and Systems, 25 – 28 May 2003. ISCAS '03, IEEE, vol. 5, pp. V-101-V-104.

[14] C. L. Wey, W. C. Tang, S. Y. Lin, "Efficient VLSI Implementation of Memory-Based FFT Processors for DVB-T Applications", in proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07*)*, Porto Alegre, 9 – 11 Mar. 2007, pp. 98-106.

[15] B. G. Jo, M. H. Sunwoo, "New continuous-flow mixed-radix (CFMR) FFT Processor using novel in-place strategy", in proc. IEEE Transactions on Circuits and Systems I: Regular Papers, May 2005, vol. 52, no. 5, pp. 911-919.

[16] H. S. Stone, "Parallel Processing with the Perfect Shuffle", in proc. IEEE Transactions on Computers, Feb. 1971, vol. C-20, no. 2, pp. 153-161.

[17] P. Philipov, V. Lazarov, Z. Zlatev, M. Ivanova, "A Parallel Architecture for Radix-2 Fast Fourier Transform", in proc. IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)*,* Sofia, 3 – 6 Oct. 2006, pp. 229-234.

[18] P. P. Boopal, M. Garrido, O. Gustafsson, "A reconfigurable FFT architecture for variable-length and multi-streaming OFDM standards", in proc. IEEE International Symposium on Circuits and Systems (ISCAS2013), Beijing, 19 – 23 May 2013, pp. 2066-2070.

[19] M. Garrido, K. K. Parhi, J. Grajal, "A Pipelined FFT Architecture for Real-Valued Signals", in proc. IEEE Transactions on Circuits and Systems I: Regular Papers, Dec. 2009, vol. 56, no. 12, pp. 2634-2643.

[20] M. Garrido, J. Grajal, M. A. Sanchez, O. Gustafsson, "Pipelined Radix – 2(k) Feedforward FFT Architectures", in proc. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Jan. 2013, vol. 21, no. 1, pp. 23-32.

[21] F. Qureshi, O. Gustafsson, "Analysis of twiddle factor memory complexity of radix-2^i pipelined FFTs", in proc. Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers*,* Pacific Grove, CA, 1 – 4 Nov. 2009, pp. 217-220.

[22] F. Qureshi, S. A. Alam, O. Gustafsson, "4k-point FFT algorithms based on optimized twiddle factor multiplication for FPGAs", in proc. Asia Pacific Conference

on Postgraduate Research in Microelectronics and Electronics (PrimeAsia), Shanghai, 22 – 24 Sept. 2010, pp. 225-228.

[23] H. J. Kang, J. Y. Lee, J. H. Kim, "Low-complexity twiddle factor generation for FFT processor", in proc. Electronics Letters, 07 Nov. 2013, IEEE, vol. 49, no. 23, pp. 1443-1445.

[24] J. Thong, N. Nicolici, "An Optimal and Practical Approach to Single Constant Multiplication", in proc. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Sept. 2011, vol. 30, no. 9, pp. 1373-1386.

[25] "Multiplerless Constant Multiplication", Access date: 20 Oct. 2016, Available online: http://www.spiral.net/hardware/multless.html

[26] O. Gustafsson, F. Qureshi, "Addition Aware Quantization for Low Complexity and High Precision Constant Multiplication", in proc. IEEE Signal Processing Letters, Feb. 2010, vol. 17, no. 2, pp. 173 – 176.

[27] F. Qureshi, O. Gustafsson, "Low-Complexity Constant Multiplication Based on Trigonometric Identities with Applications to FFTs", in proc. IEICE Transactions on Fundamentals of Electronics, Communication and Computer Sciences" Nov. 2011, vol. 94, no. 2, pp. 2361 – 2368.

[28] M. Garrido, F. Qureshi, O. Gustafsson, "Low-Complexity Multiplierless Constant Rotators Based on Combined Coefficient Selection and Shift-and-Add Implementation (CCSSI)", in proc. IEEE Transactions on Circuits and Systems I: Regular Papers, Jul. 2014, vol. 61, no. 7, pp. 2002-2012.

[29] R. Guo, L. S. DeBrunner, "A novel fast canonical-signed-digit conversion technique for multiplication" in proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP*)*, Prague, 22 – 27 May 2011, pp. 1637-1640.

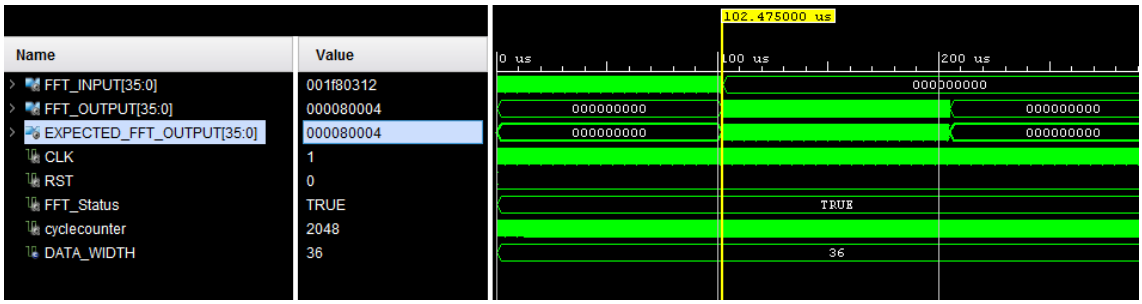
[30] "Representation of Numbers", Access date: 10 Oct. 2016. Available online: http://www.swarthmore.edu/NatSci/echeeve1/Ref/BinaryMath/NumSys.html

[31] "Two's Complement" Access date: 10 Oct. 2016. Available online: http://www.tfinley.net/notes/cps104/twoscomp.html

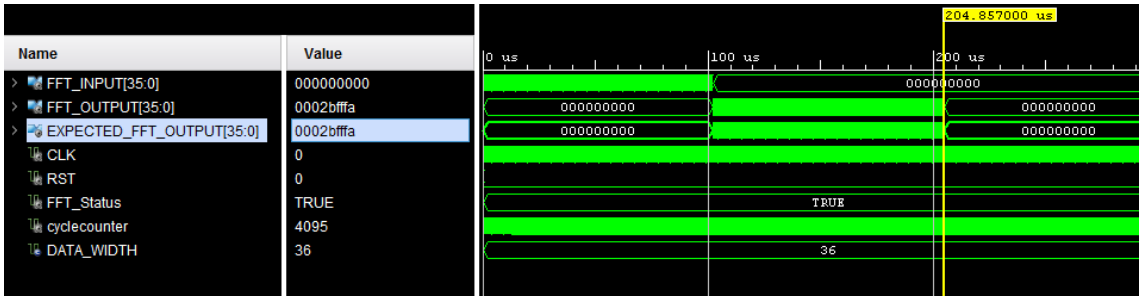
[32] K. K. Parhi, "VLSI digital Signal Processing Systems – Design and Implementation", A Wiley-Interscience Publication, John Wiley & Sons, Inc. 1999, pp. 380 – 382.

APPENDIX A: SIMULATION WAVEFORMS

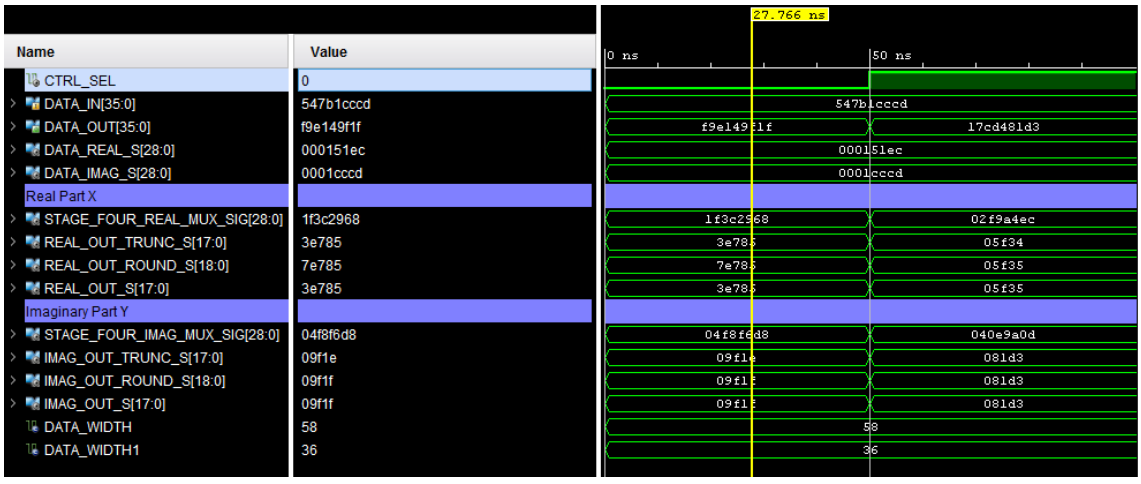
❖ 2048-Point Radix-2^2 FFT Simulation Waveforms



❖ 2048-Point Identical Radix-2^2 Simulation Waveforms



❖ W_8 Constant Multiplier Simulation Waveforms



❖ W_{16} Constant Multiplier Simulation Waveforms