ASIC Implementation of a High Throughput, Low Latency, Memory Optimized FFT Processor

A Thesis
Submitted for the Degree of

Master of Science (Engineering)
in the
Faculty of Engineering

by

Kala S.

Centre For Nano Science and Engineering
INDIAN INSTITUTE OF SCIENCE
Bangalore - 560012

December 2012

To Nalesh for his endless love and support...

Acknowledgements

I am deeply indebted to many people for their help and support during my days at IISc. I take this opportunity to thank all of them. First and foremost, I express my deepest gratitude to my advisors Prof. S K Nandy and Prof. H S Jamadagni for their guidance, motivation and support throughout my research period in CAD Lab. Without their encouragement and persistent help, this thesis would not have been possible. Prof. Nandy has been constantly guiding me to follow the right direction.

I am also indebted to Dr. Ranjani Narayan and Dr. S. Balakrishnan (Morphing Machines Pvt. Ltd.) for their valuable and in-depth comments and suggestions which greatly enhanced my understanding.

I would like to thank all the members of CAD Lab especially Sanjay, Madhav, Nandhini, Ramesh, Farhad, Gopinath, Saptarsi, and Gaurav for creating an enthusiastic and dynamic environment. I am thankful to Adarsha Rao for helping me with scripts and libraries for physical design. I also thank Arka Maity of NIT Durgapur, who started with the matlab simulations during his internship in CAD Lab.

I am also grateful to Research Scholars from VLSI Lab (ECE) and Ms.Vedavalli (DESE) for helping me out with library and licenses for synthesis. I thank Ms. Mallika of CAD Lab and Dr.Narahari of CeNSE for their co-operation in the administrative matters.

Without the joyous company of my friends especially all SIMA members, my life at IISc would not have been colorful. I am extremely thankful to all of them for all the fun we had.

No words are sufficient to express my gratitude to my parents and sisters for their love, care and support throughout my life. I owe everything to my father for what I have achieved so far. I am so lucky to share my life with Nalesh who stands with me in all ups and downs in my academic and family life. His infinite patience, love and support have enriched my life.

Kala S.
12th December, 2012

Publication based on this Thesis

Kala S., Nalesh S., Arka Maity, S. K. Nandy and Ranjani Narayan, **High Throughput, Low Latency, Memory Optimized 64K Point FFT Architecture using Novel Radix-4 Butterfly Unit**, in Proceedings of *IEEE International Symposium on Circuits and Systems* (ISCAS), Beijing, pp. 3034-3037, May 2013.

Abstract

The rapid advancements in semiconductor technology have led to constant shrinking of transistor sizes as per Moore's Law. Wireless communications is one field which has seen explosive growth, thanks to the cramming of more transistors into a single chip. Design of these systems involve trade-offs between performance, area and power. Fast Fourier Transform is an important component in most of the wireless communication systems. FFTs are widely used in applications like OFDM transceivers, Spectrum sensing in Cognitive Radio, Image Processing, Radar Signal Processing etc. FFT is the most compute intensive and time consuming operation in most of the above applications. It is always a challenge to develop an architecture which gives high throughput while reducing the latency without much area overhead. Next generation wireless systems demand high transmission efficiency and hence FFT processor should be capable of doing computations much faster. In this thesis, a fully parallel unrolled FFT architecture based on a novel radix-4 engine is proposed which is catered for wide range of applications. Different size of FFTs are verified and implemented in ASIC based on the proposed architecture. This architecture is more suitable for computing FFT of large sizes. The main features of our architecture are high throughput and low latency with significant reduction in memory.

Contents

List of Figures

List of Tables

Acronyms

ADSL	**A**symmetric **D**igital **S**ubsciber **L**oop
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
CDMA	**C**ode **D**ivision **M**ultiple **A**ccess
DAB	**D**igital **A**udio **B**roadcasting
DFT	**D**iscrete **F**ourier **T**ransform
DIF	**D**ecimation **I**n **F**requency
DIT	**D**ecimation **I**n **T**ime
DSP	**D**igital **S**ignal **P**rocessing
DVB-T	**D**igital **V**ideo **B**roadcasting **T**errestrial
FEC	**F**orward **E**rror **C**orrection
FFT	**F**ast **F**ourier **T**ransform
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPP	**G**eneral **P**urpose **P**rocessor
IFFT	**I**nverse **F**ast **F**ourier **T**ransform
LTE	**L**ong **T**erm **E**volution
MDC	**M**ulti-path **D**elay **C**ommutator
OFDM	**O**rthogonal **F**requency **D**ivision **M**ultiplexing
PS	**P**arallel to **S**erial
RAM	**R**andom **A**ccess **M**emory
ROM	**R**ead **O**nly **M**emory
RTL	**R**egister **T**ransfer **L**ogic
SDC	**S**ingle-path **D**elay **C**ommutator
SDF	**S**ingle-path **D**elay **F**eedback
SDR	**S**oftware **D**efined **R**adio

SFG	**S**ignal **F**low **G**raph
SRAM	**S**tatic **R**andom **A**ccess **M**emory
SQNR	**S**ignal to **Q**uantization **N**oise **R**atio
TDMA	**T**ime **D**ivision **M**ultiple **A**ccess
TF	**T**widdle **F**actor
UWB	**U**ltra **W**ide **B**and
VDSL	**V**ery high speed **D**igital **S**ubsciber **L**oop
VLSI	**V**ery **L**arge **S**cale **I**ntegration

Chapter 1

Introduction

Over the last few decades as semiconductor technology has improved both in terms of performance and cost, most of the DSP applications demand flexible functionality with change in user requirements or system features. One of the mainstream wireless technologies that has emerged from internet revolution is Software Defined Radio (SDR). SDR provides compatibility with new standard releases as well as existing standards. Second and third generation wireless systems are mainly based on TDMA or CDMA technologies. But next generation broadband wireless systems are based on OFDM. OFDM is considered to be the most suitable modulation technique to support high data rates in wireless technologies. FFT and IFFT are the major compute intensive and power consuming blocks in modern OFDM based transceivers.

Fast Fourier Transform (FFT) has become a key component in many of the digital signal processing and telecommunication systems. Efficient FFT computation is required in Polyphase Channelization, modems of discrete multi-tone communications, Spectrum analysis and other Image processing applications. Most of the FFT applications demand low latency, less memory requirement, low power consumption and high throughput. Hence speeding up of FFT algorithm has become an important issue.

Several communication applications require FFT/IFFT calculation of large complex data. For example, in radar systems for detecting small cross sectional targets, FFT computation of 64K points or more is required [1]. ADSL/VDSL modems also require higher order FFTs. Also, next generation wireless standards demand high data rates, the

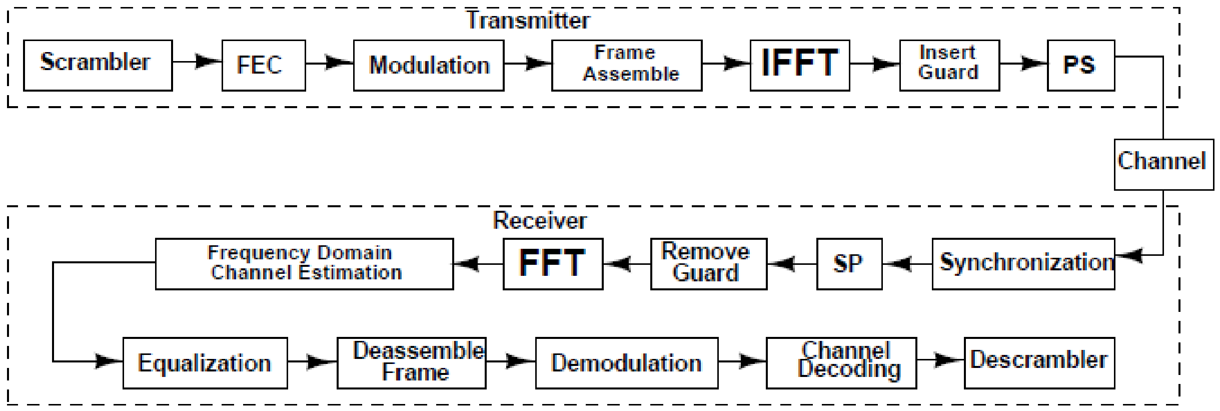


Figure 1.1: Baseband Processing of Wireless Systems

Table 1.1: FFT Requirements of Various Standards

Standard	FFT Size (complex)
802.11 a/g	64
802.11 n	64-128
802.16 d/e	128-2048
DAB	256-2048
DVB-T/H	2048-8192
UWB	128
LTE	128-2048

FFT processor must be capable of delivering high throughput. To increase the transmission bandwidth and hence the transmission efficiency, higher order FFTs are required in OFDM systems.

1.1 Overview of OFDM

Orthogonal Frequency Division Multiplexing has become a most favored technique for broadband wireless system due to susceptibility to signal dispersion under multipath conditions. OFDM effectively decomposes the wideband channel into a set of narrowband orthogonal subchannels with a different symbol sent over each subchannel [2]. Simultaneous data transmission and reception over these subcarriers are carried out independently. The block diagram of baseband processing of wireless standards is shown in figure 1.1. FFT requirement of different wireless standards is shown in Table 1.1 [3].

In OFDM based SDR environments, we therefore need to perform FFT for certain N point, and for each chosen value of N, there are issues of latency and throughput that must be respected, in order to meet the required Quality of Service (QoS). Communication applications based on OFDM require high throughput FFT processors for supporting high data rates. Moreover, a memory optimized FFT architecture is required, since majority of area in OFDM implementation is dominated by FFT. The output of FFT has to be in natural order before giving to next block. Otherwise the next block, i.e. channel estimation and equalization has to take care of input reordering for maintaining the orthogonality of signals which ensures that signals are not corrupted.

1.2 Overview of Polyphase Channelizer

Channelization is a process where a single, few or all channels from a certain frequency band are separated for further processing. Polyphase Channelizer is the most efficient approach in terms of computations and required hardware resources as compared to standard channelizer [4] [5]. A Polyphase Channelizer consists of the following components:

- Commutator to down-sample the data rate
- Polyphase partitioned filter
- Complex phase rotators to extract the individual channels which is equivalent to the M point FFT calculation

Figure 1.2 shows the block diagram of Polyphase Channelizer. The size of FFT i.e. M varies according to the number of channels extracted. Polyphase Channelizers used in Radar signal processing demand higher order FFTs like 64K, 128K, 256K and 512K point.

1.3 Choosing a Solution for Implementation

For implementing today's real-time signal processing applications, several technology solutions are available. Most popular among them are ASIC, DSP, configurable processors, FPGA and GPP. There exist several design trade-offs between different solutions and hence a suitable solution has to be considered which will meet our requirement. DSP

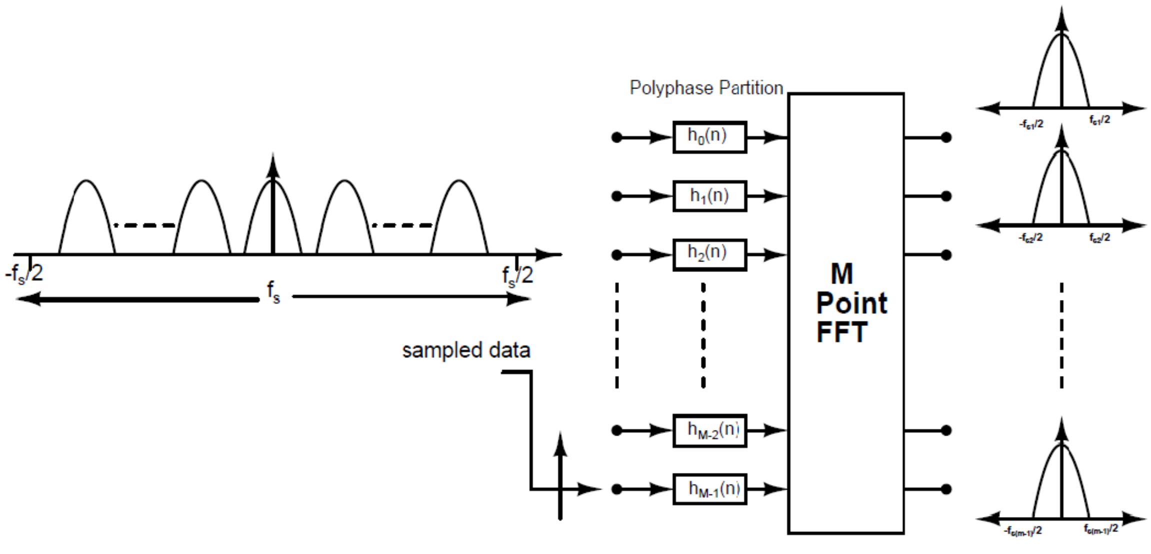


FIGURE 1.2: Polyphase Channelizer

chips are flexible but are not suited to meet high throughput. So software solutions like DSPs and GPPs are not considered. Hence a flexible hardware is required for implementing FFT processor. FPGAs are relatively not power efficient but have a good performance. ASICs are excellent in performance where we can tune the hardware specific to our application which results in high application specific performance. If the design is targeted for power efficiency, ASICs can achieve decent power efficiency similar to configurable processors. ASICs allows realization of a high performance FFT processor which fulfill our requirements of high throughput and less area.

1.4 Contribution of the Thesis

There are very few hardware implementations of higher order FFTs in literature. The purpose of our work is to develop a high throughput, memory optimized higher order FFT processor with good performance in terms of latency. A fully parallel FFT architecture based on parallel unrolled Radix-4 engine using Radix-4^3 and Radix-4^4 FFT algorithms are presented in this thesis. The proposed architecture gives a significant improvement in latency by reducing intermediate buffering, without degrading the performance or area. In existing Cooley-Tukey [6] architectures, the outputs from each stage has to be reordered before the next stage can start computation. This needs intermediate storage after each stage. Another application of our architecture, is to implement lower order FFTs with reordered outputs without any extra logic. A 64 point, 128 point and 4K point FFTs are implemented in ASIC using the proposed architecture which are based on Radix-4^3 algorithm. Based on Radix-4^4 algorithm, 256 point and 64K point FFTs are implemented with less memory requirement and comparable throughput with the state-of-art implementations.

1.5 Organization of the Thesis

The main theme of this work is to develop an efficient FFT architecture for high throughput applications, with low latency and memory requirement.

Chapter 2 covers the FFT algorithm and various FFT architectures in literature. Hardware complexity analysis of different architectures are discussed in this chapter.

Chapter 3 discusses different techniques for optimizing the area of an SDF pipelined architecture. A 128 point radix-2^2 FFT is implemented in ASIC and comparison is done between optimized and conventional FFT architectures.

In Chapter 4, Radix-4^3 FFT algorithm and architecture in the literature is discussed. Merits and demerits of this architecture and the need for a new architecture is also briefly explained.

In Chapter 5, Radix-4^4 FFT algorithm is discussed. An efficient architecture for Radix-4^3 and Radix-4^4 FFT is proposed in this chapter. Extending this architecture to develop a 4K and 64K point FFTs are described.

Output reordering of Radix-4^3 and Radix-4^4 FFTs are discussed in Chapter 6. A reordered 128 point FFT architecture is also explained in this chapter.

In Chapter 7, the Matlab and Modelsim simulation details are explained. ASIC Implementation of the proposed FFT Processor and the results are discussed. Comparisons with the state-of-art implementations are also done in this chapter.

Chapter 8 presents the conclusions and future directions of this research work.

Chapter 2

FFT Algorithms and Architectures

In this chapter, FFT algorithms based on different radices and various FFT architectures in literature are discussed. Hardware complexity of these architectures are analyzed and evaluated.

2.1 FFT Algorithms

Discrete Fourier Transforms are useful in frequency analysis of discrete time signals. Generally DFTs are not computed directly due to the computational complexity. There are several methods for computing DFT [7]. DFTs are efficiently calculated using Fast Fourier Transform algorithms. Among various FFT algorithms, Cooley-Tukey is very popular because of its reduced complexity i.e. O(Nlog_2N) instead of O(N^2) when compared to DFT [7]. The formula for finding DFT of a sequence $x(n)$ is given in equation (2.1).

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \quad k = 0, 1, 2, ...N-1 \tag{2.1}$$

$$W_N = e^{-2\pi i/N} \tag{2.2}$$

where W_N, the Twiddle Factor, denotes the N^{th} primitive root of unity, with its exponent evaluated to modulo N and is introduced by the equation (2.2). Inverse DFT also consumes the same type of computational algorithm as that of DFT. So the discussion is restricted to DFT. Inverse DFT of $X(k)$ is given equation (2.3).

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)W_N^{-nk} \quad n = 0, 1, 2, ...N-1 \tag{2.3}$$

The FFT algorithm uses a divide-and-conquer approach to reduce the number of computations to arrive at the same result of DFT. The size of an FFT decomposition is generally referred as ‘radix’. Radix-k FFT is efficient for computing an N point DFT, if N=k^M, where k and M are integers. Based on different decompositions, different algorithms are used to compute FFT.

2.1.1 Radix-2 FFT

FFT can be computed using Decimation-in-Time (DIT) or Decimation-in-Frequency (DIF) algorithms. In the case of DIT algorithms, the input sequence is stored in bit- reversed order and the output sequence will be in normal order. But in the case of DIF algorithms, the input sequences are in natural order and the output sequences are obtained in bit-reversed order. For deriving DIF algorithm from the DFT formula given in equation (2.1), X(k) is split into odd and even samples as given in equation (2.4) [7].

$$\begin{aligned} X(2k) &= \sum_{n=0}^{(N/2)-1} [x(n) + x(n+\frac{N}{2})]W_{N/2}^{nk} \quad k = 0, 1, 2, ...\frac{N}{2}-1 \\ X(2k+1) &= \sum_{n=0}^{(N/2)-1} [x(n) - x(n+\frac{N}{2})]W_N^{nk}W_{N/2}^{nk} \quad k = 0, 1, 2, ...\frac{N}{2}-1 \end{aligned} \tag{2.4}$$

This procedure can be repeated and for realizing an N point FFT, log_2N stages of decimation is required. Butterfly computations can be done “in-place” [7] which means that once butterfly operation is done on a pair of data, the results are stored in the same locations as that of input data. The butterfly operation in DIF FFT algorithm is shown in Figure 2.1.

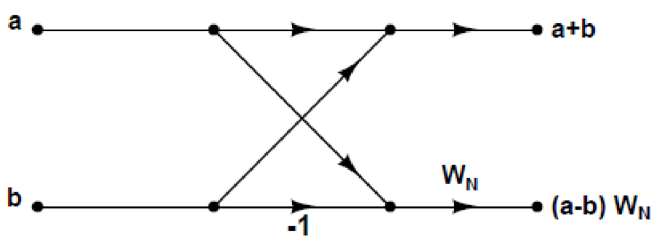


FIGURE 2.1: Radix-2 DIF FFT Butterfly

2.1.2 Radix-4 FFT

If the number of input data sequence N is a power of 4, instead of using radix-2, more efficient way of FFT computation is radix-4 FFT algorithm. For realizing an N point FFT, log_4N stages are required. Here, an N point DFT sequence X(k) in equation (2.1) is broken into four smaller DFTs, X(4k), X(4k+1), X(4k+2) and X(4k+3), where k=0,1,2,...$\frac{N}{4}-1$. The radix-4 DIF FFT is obtained as follows.

$$
\begin{aligned}
X(4k) &= \sum_{n=0}^{(N/4)-1}[x(n)+x(n+\frac{N}{4})+x(n+\frac{N}{2})+x(n+3\frac{N}{4})]W_{N/4}^{nk} \\
X(4k+1) &= \sum_{n=0}^{(N/4)-1}[x(n)-jx(n+\frac{N}{4})-x(n+\frac{N}{2})+jx(n+3\frac{N}{4})]W_N^{n}W_{N/4}^{nk} \\
X(4k+2) &= \sum_{n=0}^{(N/4)-1}[x(n)-x(n+\frac{N}{4})+x(n+\frac{N}{2})-x(n+3\frac{N}{4})]W_N^{2n}W_{N/4}^{nk} \\
X(4k+3) &= \sum_{n=0}^{(N/4)-1}[x(n)+jx(n+\frac{N}{4})-x(n+\frac{N}{2})-jx(n+3\frac{N}{4})]W_N^{3n}W_{N/4}^{nk} \qquad (2.5)
\end{aligned}
$$

Signal Flow Graph of Radix-4 butterfly operation is given in Figure 2.2. Radix-4 FFT computation results in digit reversed output where each digit is with regard to a number scheme of base 4 [8].

2.1.3 Radix-2^2 FFT

Change of traditional step-by-step twiddle factor decomposition into cascaded twiddle factor decomposition at every alternate stage has given rise to radix-2^2 FFT [9]. Number of stages required for FFT computation is same as that of radix-2 i.e. log_2N. Further the number of multipliers required is same as that of radix-4 and number of adders required is same as that of radix-2. There are two types of butterflies used in this architecture viz.,

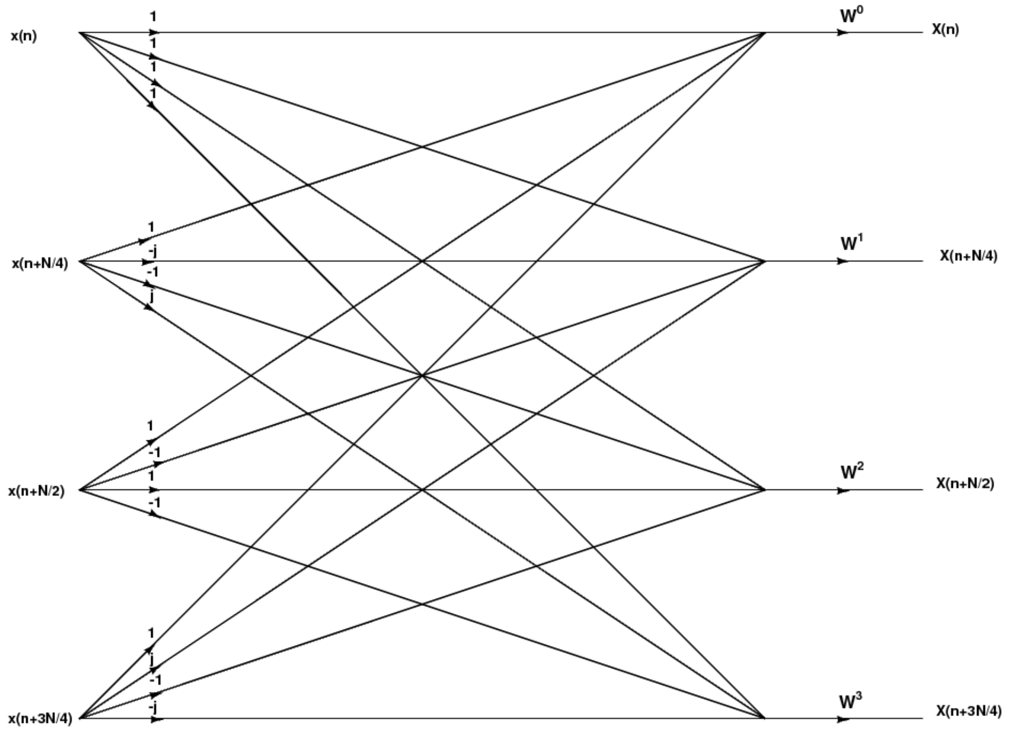


FIGURE 2.2: Radix-4 DIF FFT Butterfly

BF2 I and BF2 II. However, both the butterfly operations retain the same structure as radix-2 butterfly operation. Steps involved in radix-2^2 DIF FFT algorithm are as follows [9]. Applying the 3-dimensional linear index map to n and k in equation (2.1),

$$\begin{aligned} n &= \frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3 \\ k &= k_1 + 2k_2 + 4k_3 \end{aligned} \tag{2.6}$$

Apply equation (2.6) to the DFT formula in (2.1),

$$\begin{aligned} X(k_1+2k_2+4k_3) &= \sum_{n_3=0}^{\frac{N}{4}-1}\sum_{n_2=0}^{1}\sum_{n_1=0}^{1} x(\frac{N}{2}n_1+\frac{N}{4}n_2+n_3)W_N^{(\frac{N}{2}n_1+\frac{N}{4}n_2+n_3)(k_1+2k_2+4k_3)} \\ &= \sum_{n_3=0}^{\frac{N}{4}-1}[B(\frac{N}{4}n_2+n_3)(W_N^{(\frac{N}{4}n_2+n_3)k_1})]W_N^{(\frac{N}{4}n_2+n_3)(2k_2+4k_3)} \end{aligned} \tag{2.7}$$

where the butterfly structure (BF2 I) has the form given in equation (2.8),

$$B(\frac{N}{4}n_2+n_3) = x(\frac{N}{4}n_2+n_3) + (-1)^{k_1}x(\frac{N}{4}n_2+n_3+\frac{N}{2}n_4) \tag{2.8}$$

Decomposing the twiddle factors in (2.7),

$$W_N^{(\frac{N}{4}n_2+n_3)(2k_2+4k_3)} = (-j)^{n_2(k_1+2k_2)} W_N^{n_3(k_1+2k_2)} W_N^{4n_3k_3} \tag{2.9}$$

Substituting equation (3.1) in (2.7),

$$X(k_1+2k_2+4k_3) = \sum_{n_3=0}^{\frac{N}{4}-1} [H(k_1,k_2,k_3) W_N^{n_3(k_1+2k_2)}] W_{\frac{N}{4}}^{n_3k_3} \tag{2.10}$$

where the second butterfly (BF2 II) is given in (2.11),

$$H(k_1,k_2,k_3) = [x(n_3)+(-1)^{k_1}x(n_3+\frac{N}{2})] + (-j)^{(k_1+2k_2)}[x(n_3+\frac{N}{4})+(-1)^{k_1}x(n_3+\frac{3}{4}N)] \tag{2.11}$$

Equation (2.11) can be written as

$$BF2II = BF2I + (-j)^{(k_1+2k_2)} BF2I \tag{2.12}$$

Signal flow graph for N=32 is shown in Figure 2.3. Signal flow is same as that of radix-2, but the butterfly structures are different and twiddle factors are multiplied in alternate stages as shown in Figure 2.3. Number of stages is same as that of radix-2. For N=32, 5 stages are required.

2.1.4 Radix-2^r FFT

Radix-2^2 utilizes best of both radix-2 and radix-4 FFTs. Followed by radix-2^2 introduced by He and Torkelson [9] in 1996, similar decomposition and cascading approach is used in several radix-2^r FFT algorithms. Radix-2^3 algorithm is also developed by the same group in 1998 which is described in [10]. Later on Radix-2^4, Radix-2^5 Radix-2^n algorithms were formulated [11][12][13]. Several modifications were made to the existing algorithms and architectures for better performance and simplicity.

2.1.5 Mixed Radix FFT

Unusual sized FFTs are used in problems where we cannot control the size of the data record to be analyzed. In such cases FFTs are computed using mixed radix algorithm.

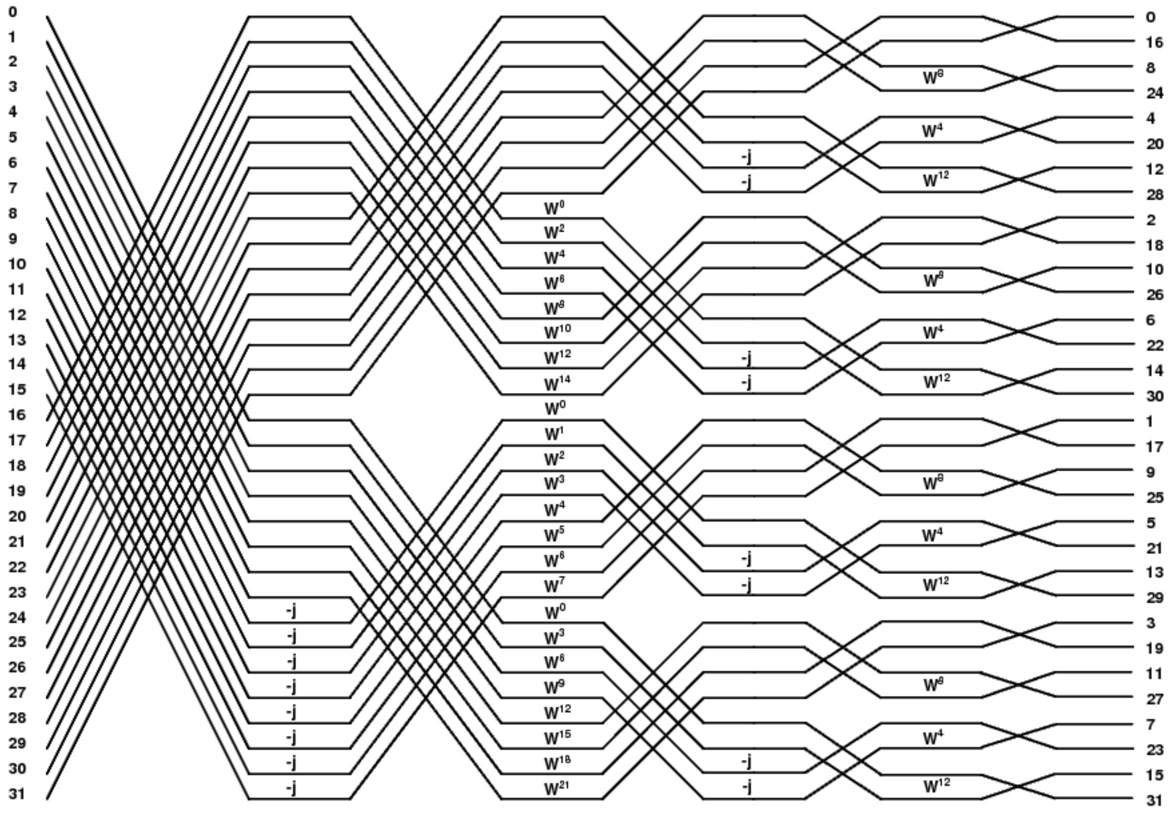


Figure 2.3: Signal Flow Graph of Radix-2^2 DIF FFT for N=32

For an N point FFT, if N $=k^M\ p^L$, where k, p, M and L are all integers, decompose the data set according to the factors of k, and then according to the factors of p. Generally this algorithm is not preferred for generic FFTs, as it does not scale well.

2.1.6 Split Radix FFT

Split radix is a variant of the Cooley-Tukey FFT algorithm that uses a blend of radix-2 and radix-4 FFTs. It recursively expresses a DFT of length N in terms of one smaller DFT of length N/2 and two smaller DFTs of length N/4. The split-radix algorithm can only be applied when N is a multiple of 4, but since it breaks a DFT into smaller DFTs it can be combined with any other FFT algorithm as desired. Since this FFT has an irregular structure, implementation is more complicated.

2.1.7 Higher Radix Algorithms

For the FFT computation of large size of complex input sequence, smaller radix FFTs are inefficient. This is mainly due to more number of stages, processing speed and more hardware blocks. Higher radix FFTs are preferred in such cases. However design complexity increases if higher radices are used. But FFT can be calculated more efficiently in terms of speed, throughput and power. Several higher radix algorithms are available in literature. In [14][15], FFTs based on radix-4^2 and radix-4^3 are discussed which can compute FFTs starting from 16 and 64 point onwards. Parallel or cascaded connection of these FFTs result in longer FFTs.

2.2 FFT Architectures

This section discusses the various architectures for implementing FFT. Decimation-In-Frequency algorithm is followed throughout this thesis, in which input is in natural order and output is in reversed order. FFT algorithms and architectures are chosen according to the area, power, throughput and processing speed requirements [16] [17]. Fully unfolded architectures consume more area, but gives maximum throughput [18]. FFT architectures are mainly classified into memory based and pipeline based architectures. Cascaded architectures (pipeline based) are not efficient for higher radix algorithms. They require large memory within each butterfly unit and so they require more area when compared to unfolded FFTs.

Memory based FFTs [19].

In single memory architectures there is an arithmetic module which is connected to a single memory. Here, the butterfly input is read from the memory and processed output is written back to the same memory. In dual memory architecture, two memory blocks are connected to the arithmetic module separately. Data is read and written to different memory blocks (ping pong).

Pipeline based FFTs [9] [16].

Pipeline structure is highly regular and flexible. Pipeline architectures have been observed to be faster to cater to very large bandwidth and has better hardware utilization. For high throughput applications, pipelining and parallelism is utilized. There are mainly two types of pipelined architectures.

- Single-path Delay Feedback (SDF) architectures
- Multipath Delay Commutator (MDC) architectures

In [9][20] different schemes of pipelined architectures such as Radix-2 MDC, Radix-2 SDF, Radix-4 SDF, Radix-4 SDC (Single-path Delay Commutator) are discussed. MDC supports M parallel input data and it provides high throughput compared to SDF. Delay feedback based approaches are more efficient in terms of their memory requirements since the output of a butterfly operation and input shares the same storage. Some of the SDF Architectures are discussed in the following sections.

2.2.1 Radix-2 SDF Architecture

In this structure, an N-point FFT/IFFT is realized as a log_2N stage pipeline. Each stage has one radix-2 butterfly element along with the buffer. The size of the buffer for a particular stage is N/2^i, where i is the stage number. There will be totally $2log_2$N adders, 2((log_4N)-1) Multipliers, and 4 load/store operations per stage. Radix-2 SDF architecture for a 16 point FFT is shown in Figure 2.4.

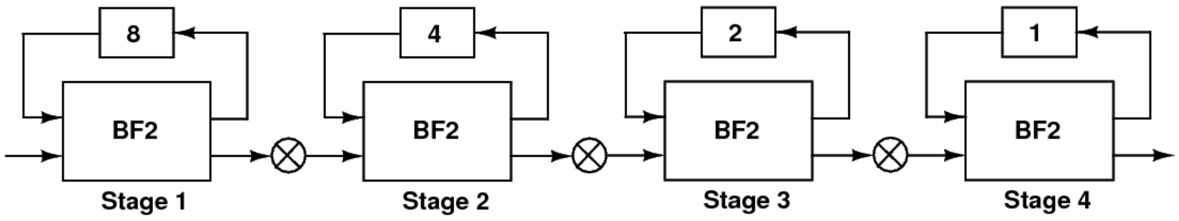


FIGURE 2.4: Radix-2 SDF Architecture for N=16

A total of N/8+1 Twiddle Factors are required to be stored in ROM for any N point FFT in case of radix-2. Other Twiddle Factors may be computed dynamically from these values as it involves `complement` or/and `swap` operation. We use the term "`unique TF`" for these N/8+1 values.

2.2.2 Radix-4 SDF Architecture

In this structure, an N-point FFT/IFFT is realized as a log_4N stage pipeline. The number of stages is 50% less compared to radix-2. Each stage in radix-4 has one radix-4 butterfly element along with the buffer. The size of the buffer for a particular stage is 3N/4^i locations where i is the stage number. Though the buffer requirement per stage is increased, the overall buffer requirement is N-1 locations for both radix-2 and radix-4. There will be totally 4log_2N adders, (log_4N)-1 multipliers, and 8 load/store operations per stage. Number of adders is twice that of radix-2 and the number of multipliers is half of radix-2. Radix-4 SDF pipelined architecture for a 64 point FFT is shown in Figure 2.5.

A total of 3N/4 TFs are required to be stored in ROM for any N point FFT. But after

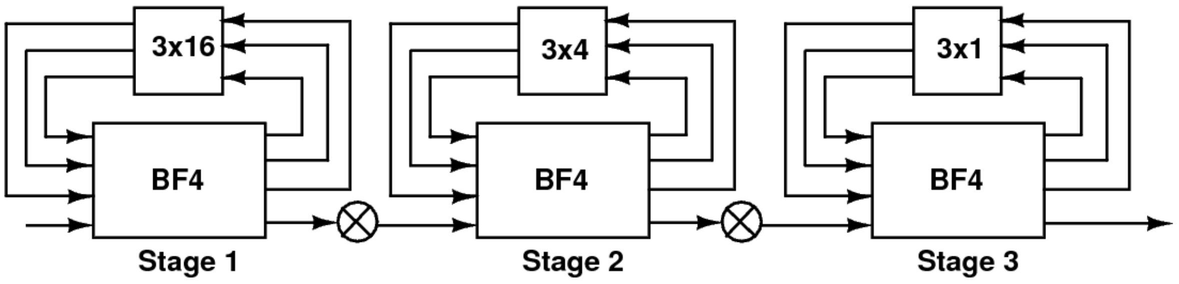


FIGURE 2.5: Radix-4 SDF Architecture for N=64

removing all the duplicate entries the number of TFs can be reduced to N/8+1. However the addressing complexity increases manifold to identify the operations `swap` and/or `complement`.

2.2.3 Radix-2^2 SDF Architecture

Figure 2.6 shows the radix-2^2 SDF architecture for 64 point FFT. Radix-2^2 has the property that it has same adder complexity as that of radix-2 and same multiplier complexity as that of radix-4. N-1 locations is the memory requirement for the radices mentioned above. Details of butterfly structures in radix-2^2 are given in chapter 3.

Hardware complexity analysis of the SDF architectures is carried out and it is observed that Twiddle factor (TF) memory dominates the major portion of the area in an FFT processor. Figures 2.7 and 2.8 indicates the adder and multiplier complexity of the above

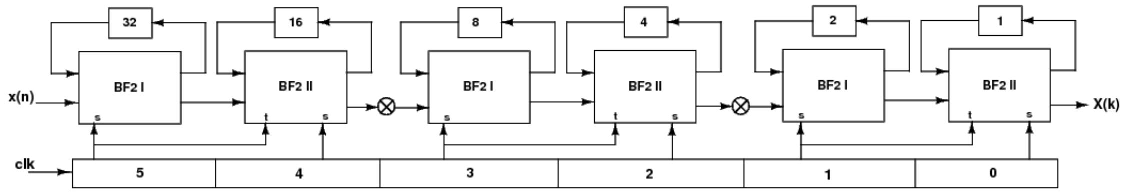


FIGURE 2.6: Radix-2^2 SDF Architecture for N=64

discussed SDF architectures.

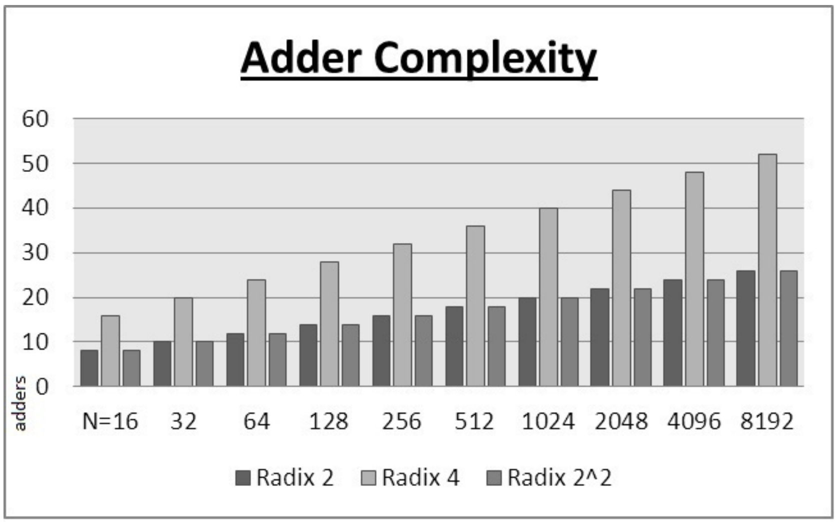


FIGURE 2.7: Adder Complexity

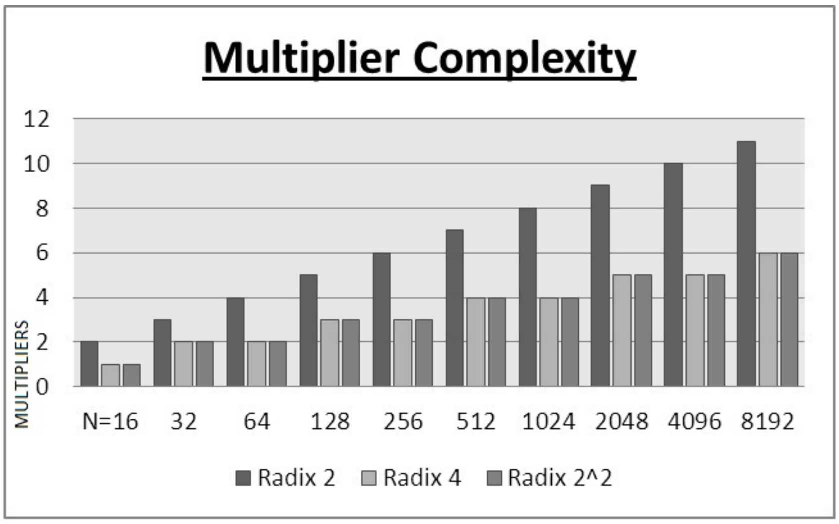


FIGURE 2.8: Multiplier Complexity

Figures 2.9 and 2.10 indicates the twiddle factor requirement and memory access for these architectures. As N increases, the hardware complexity increases and the memory required for storing twiddle factors is also increased.

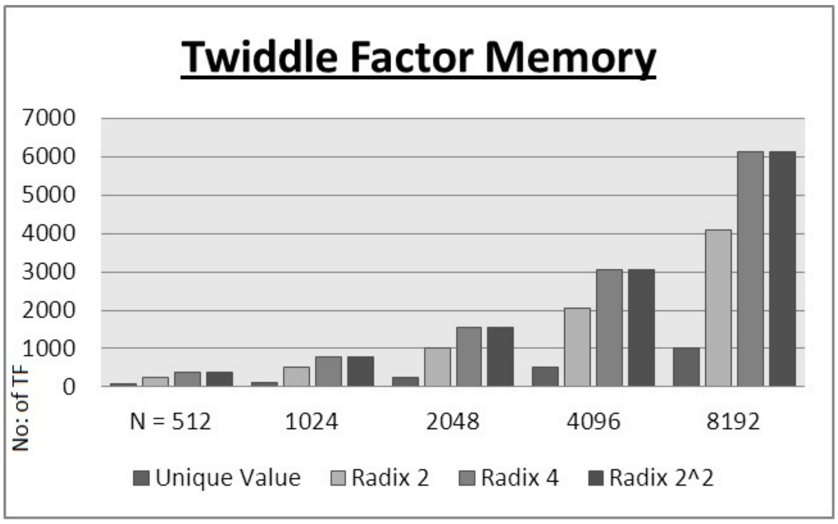


Figure 2.9: Twiddle Factor Memory Requirement

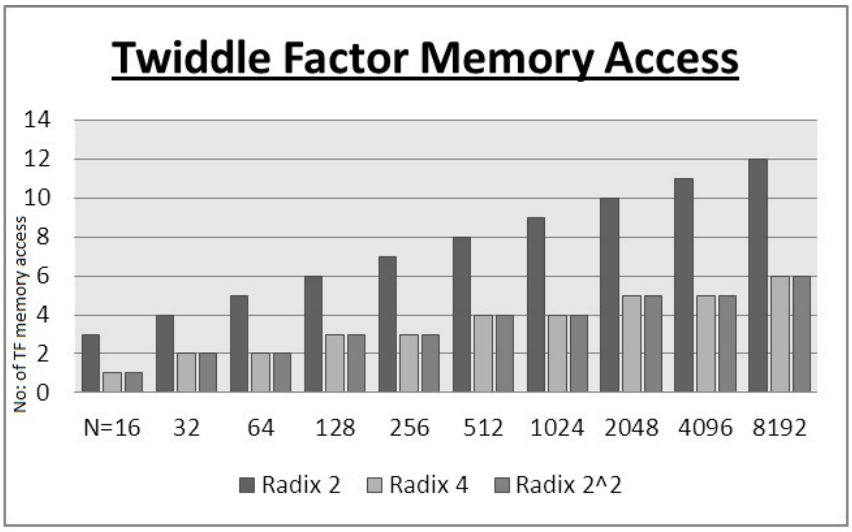


Figure 2.10: Twiddle Factor Memory Access

2.2.4 Systolic FFT Architecture

When compared to cascaded architecture, systolic architecture is more efficient for high throughput applications. Here, the memory organization is simple and improves the latency and can achieve scalability of the pipeline depth [17]. Advantages of both cascaded and unfolded architectures are utilized in systolic architecture. A 64 point radix-2 systolic

Table 2.1: Hardware Complexity of FFT Architectures

	Complex Multipliers	Complex Adders	Memory	Control
R2SDF	$2log_4$N -1	$4log_4$N	N-1	Simple
R4SDF	log_4N -1	$8log_4$N	N-1	Medium
R2^2SDF	log_4N -1	$4log_4$N	N-1	simple
R2^3SDF	2(log_8N -1)	$6log_8$N	N-1	simple
R2MDC	$2log_4$N -1	$4log_4$N	3N/2 -2	Simple
R4MDC	3(log_4N -1)	$8log_4$N	5N/2 -4	Simple
R2^2MDC	log_2N -2	$2log_2$N	3N/2-2	Simple
R4SDC	log_4N -1	log_4N	2N -2	Complex
R4^3 Systolic	log_4N -1	$3log_4$N	(N/3)log_4N	simple

FFT processor is shown in Figure 2.11 [21]. It uses seven computational elements (CE) and six reordering elements (RE). In case of radix-4 systolic architecture, data streams will be four.

In [15] a fully systolic FFT architecture for high throughput applications using Radix-4^3

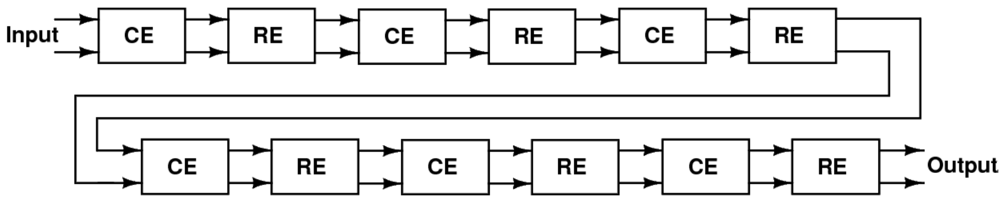


Figure 2.11: 64 Point Radix-2 Systolic FFT Processor

algorithm is discussed. This is efficient for longer FFT computations like 16K, 64K and 256K points.

Hardware complexity of different architectures in literature [15] [9] [14] is given in Table 2.1. It is observed from the Table 2.1 that R2^2SDF is an efficient FFT architecture with simple control logic. In the case of Radix-2^3 architectures, though the complexity is less, N should be a power of 8. So in cases where small FFTs like 64, 256, 1024 point etc. are to be computed, we use R2^2 architectures. But for FFT computation of large values of 'N', higher radices like R4^3 are suitable, which reduces the number of stages. Hence the hardware complexity is reduced. For high throughput applications, MDC is better than SDF. But memory requirement and hardware cost is higher in MDC [20].

2.3 Output Reordering in FFT

Various FFT architectures based on different algorithms have been discussed in literatures [9] [18] [20]. But only few of these architectures could address the output reordering issues. In DIF FFT algorithm, the output is in bit reversed order. For applications like OFDM the outputs are to be in order. Therefore extra logic is required for reordering the outputs into normal order. However, the problem can be solved if the channel estimation and equalization blocks, which are connected with the output of FFT processor, can be designed to process the input data in bit reversed order [8]. In this way, the output of FFT will serve as the input to the channel decoding stage. Some work has already been done for the bit reversed order output problem. In [22], an output reordering architecture for 128 point FFT with maximum throughput of 1 GSPS is proposed. In most of the FFT architectures in literature, a separate hardware unit is required for reordering which increases the latency and complexity. In [23] a signal reordering unit for OFDM systems which reorders the FFT output is proposed. But it requires extra hardware for implementation. In [24] a part of OFDM system, i.e. FFT and cyclic prefix blocks are implemented for addressing the reordering issue.

These solutions are very complex to implement for wide range of N. An output reorder logic is given in Figure 2.12 [8]. In Figure 2.12, dual port SRAM (one read port and one write port) of size N is used for storing and retrieving the FFT output. There are two address generators, one for writing the FFT output into memory and the other for reading the FFT output. Address generated by write address generator is bit reversed for first N/2 cycles (i.e., even outputs). In the second N/2 cycles, the address generated by write address generator is directly used for storing into memory. The read process starts after N/2 +1 cycles. In the case of read address generator, every even address is taken as it is, to retrieve data from memory and the odd addresses are bit reversed and retrieved from the memory. Writing FFT output in to memory and reading ordered FFT output from memory is done simultaneously from N/2+1 cycle onwards. The process described above is continued as long as the output is obtained from the last pipelined FFT stage. This ensures the ordered output of FFT data. The memory required to reorder the output for radix-2, radix-4 and radix-2^2 SDF pipelined architectures is N and the latency due to reordering is N+N/2 cycles.

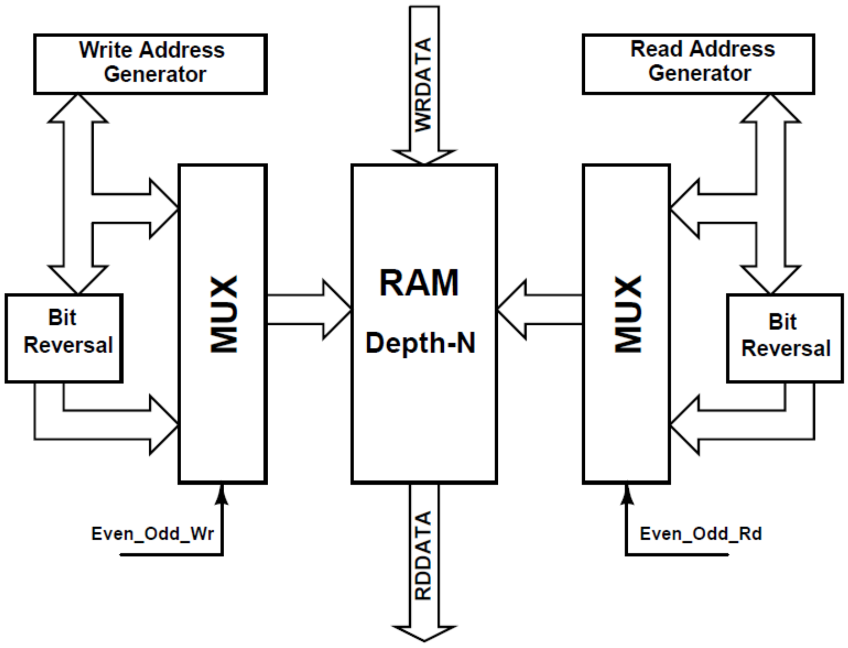


FIGURE 2.12: Output Reorder Logic

Chapter 3

Area Optimization of SDF Pipelined FFT Architecture

The merits and de-merits of various architectures are discussed in chapter 2. Twiddle Factor memory consumes majority of the area in SDF architectures. As FFT size increases, memory requirement also increases. This chapter suggests effective area optimization techniques for a radix-2^2 SDF architecture.

Signal flow graph of radix-2^2 algorithm for N=32 is shown in Figure 3.1. In Figure 3.1, we observe that the TF for every odd stage (stage 1, stage 3) is j. This trivial multiplication can be implemented by swapping real and imaginary terms and sign inverting it. The multiplication operations are such that for every even stage, non-trivial multiplications have to be performed. The delay in fetching TF for radix-2^2 from memory and processing it for a particular stage can be done over the period of two cycles in radix-2^2. Since the architecture is similar to radix-2 structure, the TFs of radix-2^2 in the subsequent even stages (stage 4, stage 6) are subset of stage 2 TFs.

From Figure 3.1 we refer that every even stage can be grouped into odd and even blocks. In an even stage, the odd blocks have one set of TF i.e.W_N^{2n} (series 2), and the even blocks have two sets of TF i.e.W_N^{n} (series 1) and W_N^{3n} (series 3). Analyzing the TFs, there is regularity in the addresses of TF that need to be fetched from memory for all desired ranges of N. We can also note that the number of TF coefficients required in stage 4 is a subset of stage 2. As the value of N increases we observe that the TF of

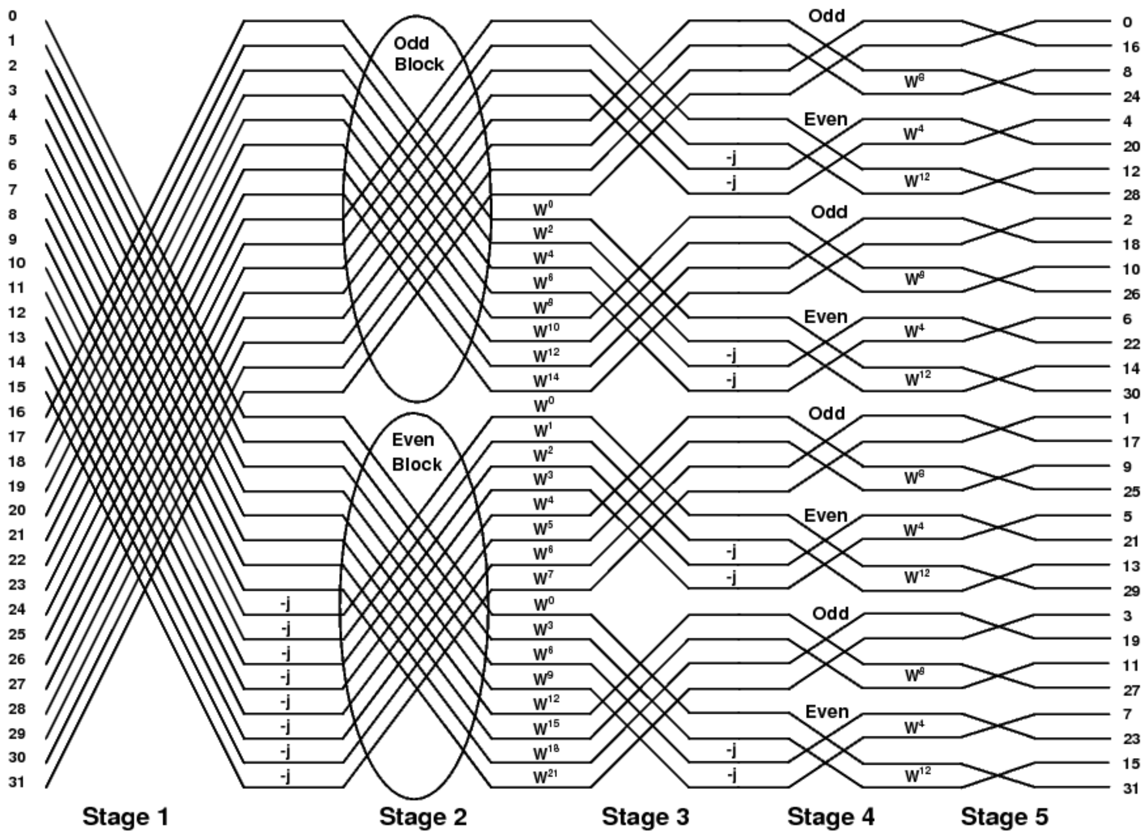


FIGURE 3.1: Radix-2^2 SDF Architecture for N=32

TABLE 3.1: TF Formula for Radix-2^2 for N Point

TF is W^{nk} where nk can be calculated as follows					
Stage	2	4	6	...	log_2N
Odd I	0	0	0	...	0
Odd II	2n	8n	32n	...	(N/2)n
Even I	n	4n	16n	...	(N/4)n
Even II	3n	12n	48n	...	(3N/4)n
Value of n	0 to N/4 -1	0 to N/16 -1	0 to N/64 -1		0

higher order even stages are subset of first even stage. Hence a suitable logic circuitry can be designed to copy the desired TF of a particular stage from stage 2 much before the evaluation of a butterfly of that stage. Table 3.1 shows the TF multiplication to I and II input of butterfly in odd and even blocks for every even stage in radix-2^2. i.e. how to obtain series 0, series 1, series 2 and series 3 set of TFs.

As discussed in chapter 2, there are two types of butterflies used in radix-2^2 architecture viz., BF2I and BF2II. However, both the butterfly operations retain the same signal flow as radix-2 butterfly operation. Figure 3.2 and Figure 3.3 show the butterfly structures used in radix-2^2. Here the control signal CS1 is used to select between the direct path and the added/subtracted path and CS2 is used to swap and complement between the real and imaginary parts of the inputs for the trivial '$-j$' multiplication.

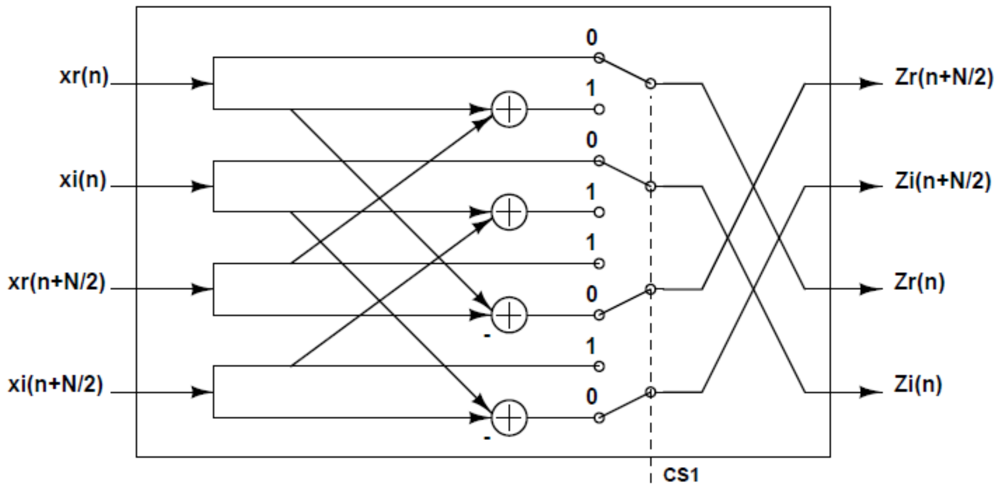


FIGURE 3.2: Butterfly I in Radix-2^2 FFT

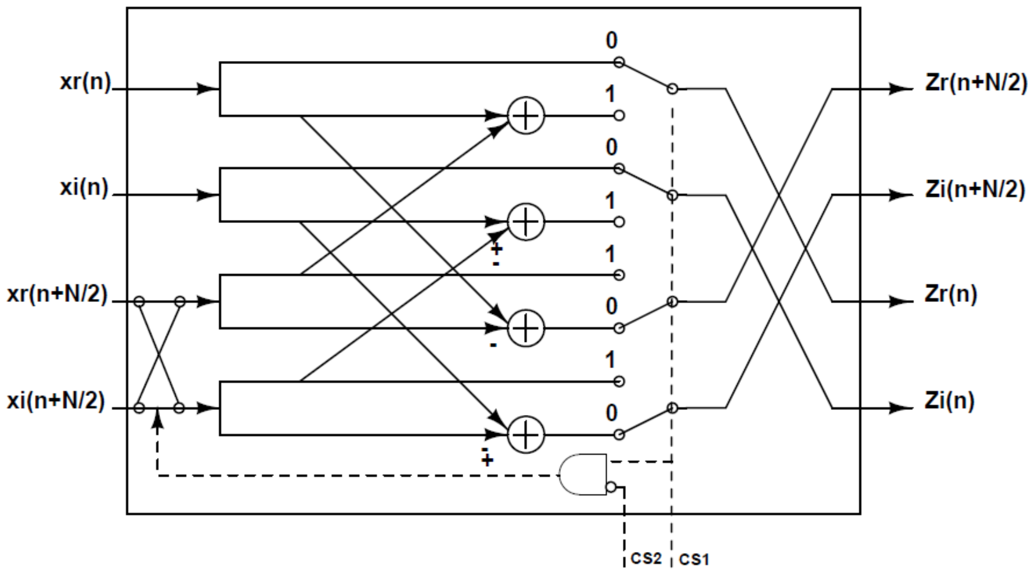


FIGURE 3.3: Butterfly II in Radix-2^2 FFT

Table 3.2: TF in last stage for N=128

Real	Imaginary
1	0
0	-1
-0.707	0.707
0.707	0.707

3.1 Optimizations for Radix-2^2 FFT

Since memory requirement for storing twiddle factors is observed to be an important issue while implementing an FFT processor, we discuss a method for TF memory reduction, and an optimization in the last stage.

3.1.1 TF Memory Reduction

From the TF complexity analysis it is clear that only N/8+1 TFs are to be stored in the memory. An effective coefficient memory reduction scheme is presented in [25]. In the case of a 128 point radix-2^2 FFT, only 17 twiddle factors are needed, due to the symmetrical characteristics of TFs with respect to Real, Imaginary and -$\pi/4$ axes. All the coefficient indices of radix-2^2 FFT are derived from the `unique TF` coefficient indices.

We observe that only eight combinations of operations are involved for computing TFs from basic unique set for any N. We can code these operations into 3 bits and store only those values instead of complete 3N/4 values. Control circuitry can be accordingly implemented to calculate TFs. The basic operations that can be encoded are real/imaginary `swapping` and/or `sign inversion` of real/imaginary values.

3.1.2 Optimization in the last stage

A complex multiplier consists of four real multipliers, an adder and a subtractor. Consider the last stage twiddle factors for N=128, which are one of the inputs to the complex multiplier. Out of 8 TFs in the last stage, four distinct TFs are given in Table 3.2.

Let a+jb be the output from previous butterfly unit which has to be multiplied with TF.

(a+jb)(0.707+j 0.707) = 0.707(a-b) + j 0.707(a-b)

Since this operation can be performed using two real multipliers and an adder and a subtractor as shown in Figure 3.4, an area reduction can be achieved in the last stage. This optimization can be applied to last stage multiplication for all values of N.

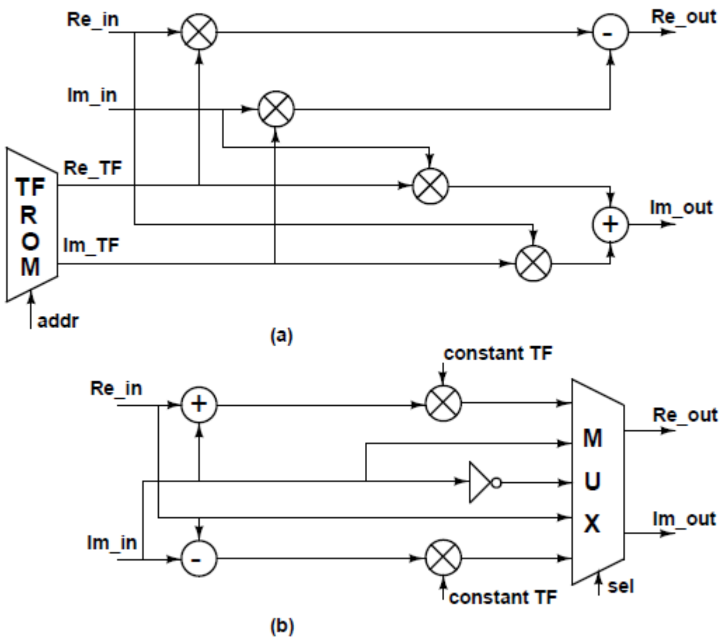


Figure 3.4: Last Stage Optimization: (a) Complex Multiplier (b) Optimized Multiplier

3.2 Implementation

The TF memory optimization for the first stage and multiplier optimization for the last stage for 128 point FFT is carried out. To implement and evaluate the effect of area optimization techniques mentioned, we have selected parallel-pipeline FFT processor structure based on the Radix-2^2 SDF architecture which is presented in [19]. FFT processor in [19] was developed for Multi-band OFDM UWB (Ultra Wide Band) systems for N=128 complex points. The architecture given in [19] is shown in Figure 3.5. A two-path parallel architecture is used in this processor. Exploiting parallelism effectively increases throughput. Here, the streaming input data is separated into even and odd parts and are

processed synchronously. In the last stage they are mixed. The twiddle factors are also separated into even and odd and are processed.
Since multiplier is one of the largest area consuming block, the multiplier used in this FFT processor is Booth-Wallace multiplier to meet the timing constraints while minimizing the area. Han-Carlson adder, one of the fastest adders is chosen as the adder in this processor. The data representation used in our implementation is 16 bit fixed point format.

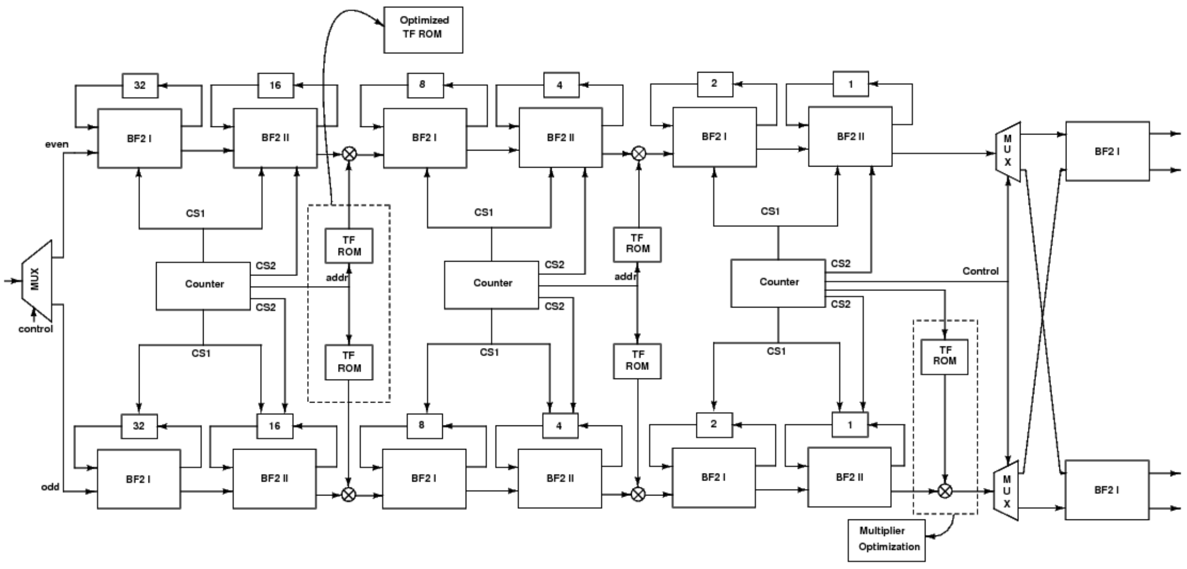
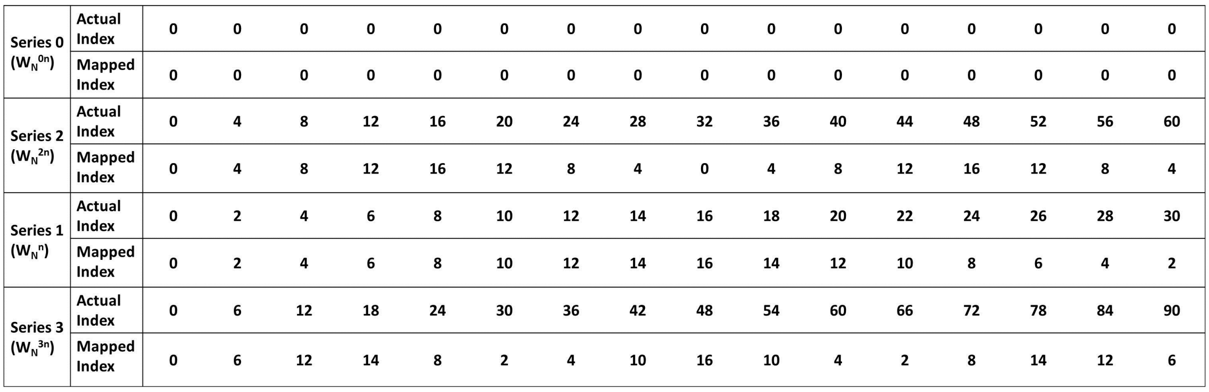


FIGURE 3.5: Radix-2^2 Parallel FFT Architecture for N=128 [19]

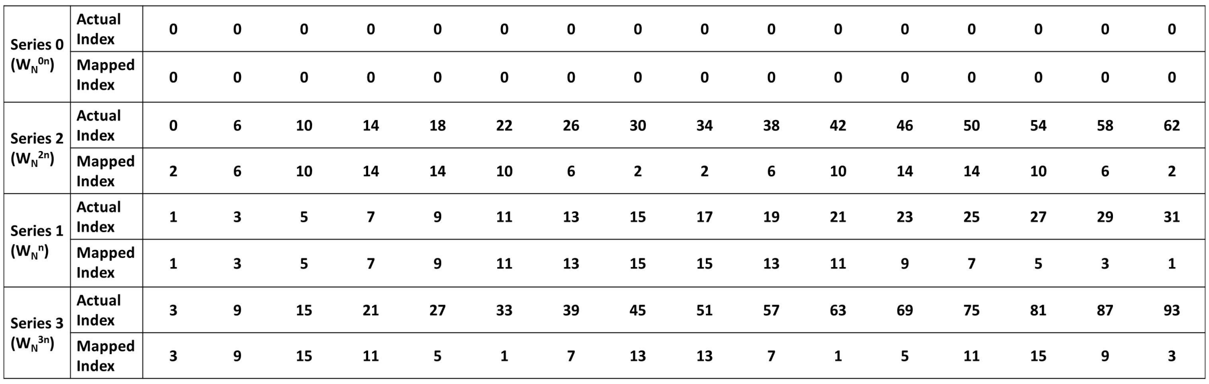
TF memory reduction for N=128 using index mapping is shown in figure 3.6 and figure 3.7 for even and odd paths respectively. This technique is possible due to the symmetric characteristics with respect to real, imaginary and ($-\pi/4$) axes and thus `swapping` and `sign inversion` has to be considered [25]. Index derivation and mapping is done separately for even and odd paths. TFs with indices 0 to 16 are to be stored. `Swapping` and/or `complementing` of real/imaginary parts are shown in figure 3.8. A single control circuit can perform these operations for both even and odd paths. The memory block of first stage occupies a considerable portion of the total memory. Here, coefficient memory reduction of first stage for both even and odd paths and an optimization for the last stage multiplier is done.

The control circuitry for memory optimization is shown in figure 3.9. The control signals CS1 and CS2 selects between 'series 0', 'series 1', 'series 2' and 'series 3' addresses. The N/8th index (ie.,16) and address addr[3:0] are subtracted if the value goes



Series 0 (W_N^{0n})	Actual Index	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Mapped Index	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Series 2 (W_N^{2n})	Actual Index	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
	Mapped Index	0	4	8	12	16	12	8	4	0	4	8	12	16	12	8	4
Series 1 (W_N^{n})	Actual Index	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
	Mapped Index	0	2	4	6	8	10	12	14	16	14	12	10	8	6	4	2
Series 3 (W_N^{3n})	Actual Index	0	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90
	Mapped Index	0	6	12	14	8	2	4	10	16	10	4	2	8	14	12	6

Figure 3.6: Derived TF indices for Even path



Series 0 (W_N^{0n})	Actual Index	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Mapped Index	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Series 2 (W_N^{2n})	Actual Index	0	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
	Mapped Index	2	6	10	14	14	10	6	2	2	6	10	14	14	10	6	2
Series 1 (W_N^{n})	Actual Index	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
	Mapped Index	1	3	5	7	9	11	13	15	15	13	11	9	7	5	3	1
Series 3 (W_N^{3n})	Actual Index	3	9	15	21	27	33	39	45	51	57	63	69	75	81	87	93
	Mapped Index	3	9	15	11	5	1	7	13	13	7	1	5	11	15	9	3

Figure 3.7: Derived TF indices for Odd path

beyond 16, else the same address is considered. Thus the fifth bit of address acts as a select signal to the multiplexer. The real and imaginary values from the TF ROM are `swapped` and/or `complemented` according to the control signals 'swap', 'Re_complement' or 'Im_complement', which are generated from the Address generator. This logic is applicable to both even and odd paths. This technique is very effective for higher order FFTs.

3.3 ASIC Results

The architecture of FFT processor is modeled in Verilog and the word length is 16 bits for both real and imaginary parts. The FFT processor has been synthesized with the Synopsys Design Compiler using UMC 90nm technology (Low K process). The implementation has an output latency of 67 clock cycles. Based on the synthesis results, the processor can run at a maximum frequency of 157MHz. For this library, the gate density is 400k/mm^2

Series 2 (W_N^{2n})			Series 1 (W_N^{n})			Series 3 (W_N^{3n})		
Even	Odd		Even	Odd		Even	Odd	
0	2	R,I	0	1	R,I	0	3	R,I
4	6		2	3		6	9	
8	10		4	5		12	15	-I,-R
12	14		6	7		14	11	
16	14	-I,-R	8	9		8	5	
12	10		10	11		2	1	I,-R
8	6		12	13		4	7	
4	2		14	15		10		
0	2	I,-R	16	15	-I,-R	16	13	-R,I
4	6		14	13		10	13	
8	10		12	11			7	
12	14		10	9		4	1	-R,-I
16	14	R,-I	8	7		2	5	
12	10		6	5		8		
8	6		4	3		14	11	I,R
4	2		2	1		12	15	
						6	9	
							3	

FIGURE 3.8: Swapping/Sign Inversion in even and odd paths

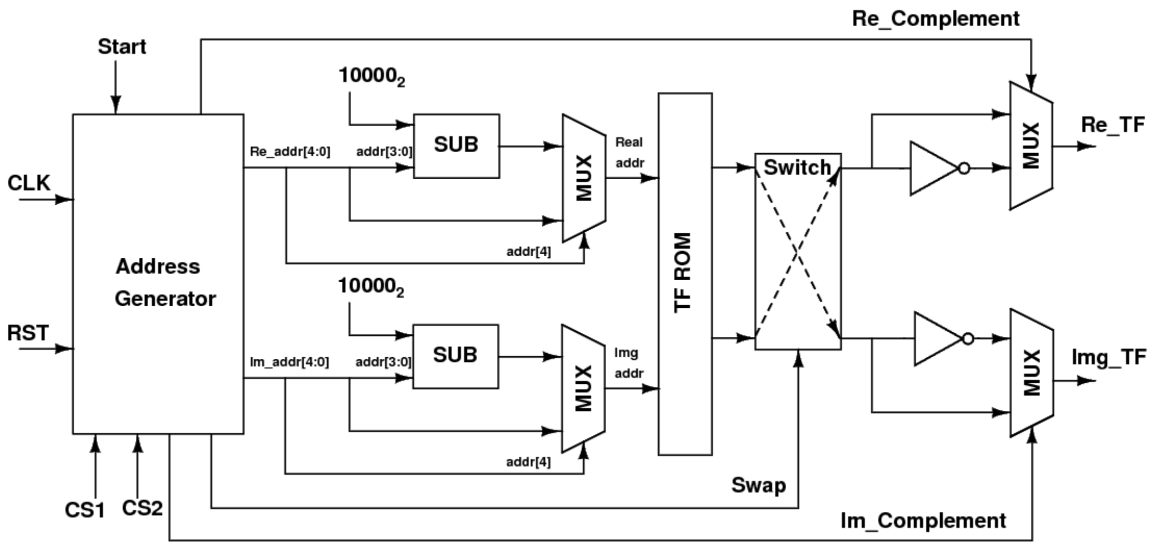


FIGURE 3.9: Control Logic for Index Mapping

and has a gate size of $4.7\mu m^2$. The gate count for optimized FFT is 63882 and that of conventional radix-2^2 parallel FFT is 68250. The results show that after first stage TF memory reduction and last stage multiplier optimization, there is a reduction of 6.4% in the total area. Here TF memory optimization resulted in 1.9% reduction and last

Table 3.3: Comparison of Radix-2^2 parallel FFTs (N=128)

Parameter	Conventional FFT(3N/4 TFs)	Optimized FFT
Word length(bits)	16	16
Frequency(MHz)	157	157
Area(μm^2)	320778	300247

stage optimization gave 4.5% reduction. A comparison of conventional radix-2^2 parallel-pipelined FFT (with 3N/4 TFs) and FFT with first and last stage optimization is shown in Table 3.3.

The area reduction due to optimization in the last stage is more effective for lower order FFTs (N=64,128). Area reduction due to TF memory optimization becomes more significant as N increases. We have implemented the TF generation module separately for higher order FFTs to analyze the area reduction and the results are shown in Figure 3.10. For 64 point FFT, the savings in memory is offset by the additional control circuit and so the area increases for the optimized case. Significant area reduction can be seen from 128 point FFT onwards . Since higher order FFTs are derived from the lower order FFTs, the reduction in area is cumulative as N increases.

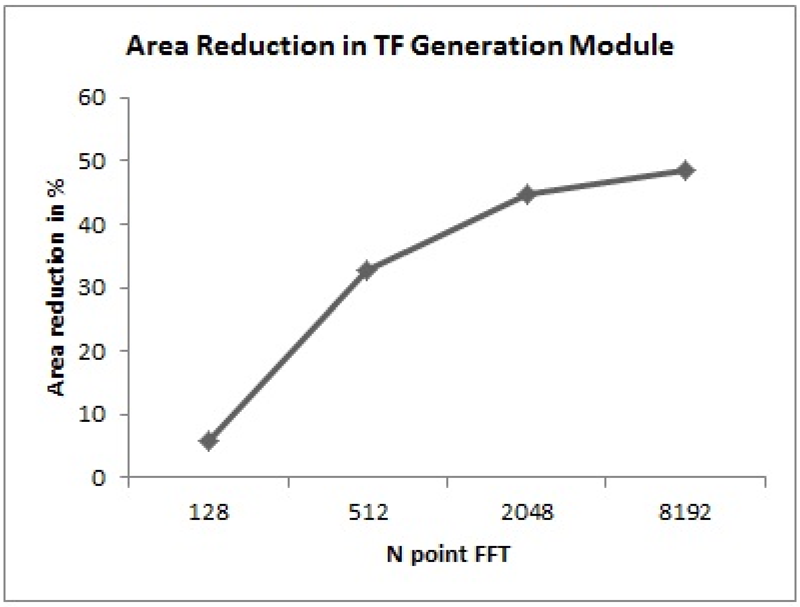


Figure 3.10: Area Reduction for Higher order Radix-2^2 Parallel FFTs

Chapter 4

Computation of Longer FFTs using Radix-4^3 Algorithm

In this chapter, FFT algorithms which are used to compute longer FFTs are discussed. The need for higher radix FFTs, existing higher radix algorithm and its architecture are provided in this chapter. Merits and demerits of this architecture is also discussed.

4.1 Higher Radix FFTs

For high throughput applications such as next generation wireless systems, Radar signal processing, spectrum sensing of Cognitive Radio etc. FFT computation of large complex data is required. In such cases radix-2, radix-4, radix-8 FFTs are not efficient in terms of processing speed and performance. Higher Radix techniques reduce the number of stages of the FFT at an increased cost in terms of VLSI area for each stage. In order to meet the real-time constraints at low area and power, [15], [17] developed an architecture based on Radix-4^3 FFT algorithm. The derivation of this algorithm is described in following section. The FFT architecture in [15] is efficient for high throughput applications.

4.2 Radix-4^3 Algorithm

In [15], radix-4^3 algorithm which is equivalent to radix-64 is derived using a four dimensional index mapping of n and k to the basic DFT formula given in equation (4.1). From the Discrete Fourier Transform formula in [7],

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} \quad k = 0, 1, 2, ...N-1 \tag{4.1}$$

$$\begin{aligned} n &= n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{N}{4}n_4 \\ k &= 64k_1 + 16k_2 + 4k_3 + k_4 \end{aligned} \tag{4.2}$$

Apply equation (4.2) to the DFT formula in (4.1),

$$X(64k_1 + 16k_2 + 4k_3 + k_54) = \sum_{n_1=0}^{\frac{N}{64}-1} \sum_{n_2=0}^{3} \sum_{n_3=0}^{3} \sum_{n_4=0}^{3} x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{N}{4}n_4)W_N^{nk} \tag{4.3}$$

Decomposing the twiddle factors,

$$W_N^{nk} = W_{\frac{N}{64}}^{n_1k_1} W_N^{n_1(16k_2+4k_3+k_4)} W_{64}^{n_2(4k_3+k_4)} W_{16}^{n_3k_4} (-j)^{(n_2k_2+n_3k_3+n_4k_4)} \tag{4.4}$$

Substituting (4.4) in (4.3) and expanding the summation with index n_4,

$$\begin{aligned} X(64k_1 + 16k_2 + 4k_3 + k_4) = \sum_{n_1=0}^{\frac{N}{64}-1} \sum_{n_2=0}^{3} \sum_{n_3=0}^{3} [B(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3)](-j)^{(n_2k_2+n_3k_3)} \\ W_{\frac{N}{64}}^{n_1k_1} W_N^{n_1(16k_2+4k_3+k_4)} W_{64}^{n_2(4k_3+k_4)} W_{16}^{n_3(k_4)} \end{aligned} \tag{4.5}$$

The first butterfly unit, B is given by

$$\begin{aligned} B = x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3)(-j)^{k_4} x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{N}{4}) + \\ (-1)^{k_4} x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{N}{2}) + (j)^{k_4} x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{3N}{4}) \end{aligned} \tag{4.6}$$

Expanding the summation in (4.5) with index n_3,

$$X(64k_1 + 16k_2 + 4k_3 + k_4) = \sum_{n_1=0}^{\frac{N}{64}-1} \sum_{n_2=0}^{3} [H(n_1 + \frac{N}{64}n_2)] \times \ (-j)^{(n_2k_2} W_{\frac{N}{64}}^{n_1k_1} W_N^{n_1(16k_2+4k_3+k_4)} W_{64}^{n_2(4k_3+k_4)} W_{64}^{n_3(4k_4+k_5)} \tag{4.7}$$

The second butterfly unit, H is given by

$$H = B(n_1 + \frac{N}{64}n_2) + (-j)^{k_3}[B(n_1 + \frac{N}{64}n_2)]W_{16}^{k_4} + (-1)^{k_3}[B(n_1 + \frac{N}{64}n_2 + \frac{N}{8})]W_8^{k_4} + (j)^{k_3}[B(n_1 + \frac{N}{64}n_2 + \frac{3N}{16})]W_{16}^{k_4}W_8^{k_4} \tag{4.8}$$

Expanding the summation in (4.7) with index n_2,

$$X(64k_1 + 16k_2 + 4k_3 + k_4) = \sum_{n_1=0}^{\frac{N}{64}-1} [T(n_1)W_N^{n_1(16k_2+4k_3+k_4)}]W_{\frac{N}{64}}^{n_1k_1} \tag{4.9}$$

The third butterfly unit, T is given by

$$T = H(n_1) + (-j)^{k_2}[H(n_1 + \frac{N}{64})]W_{16}^{k_3}W_{64}^{k_4} + (-1)^{k_2}[H(n_1 + \frac{N}{32})]W_{16}^{k_3}W_8^{k_4} + (j)^{k_2}[H(n_1 + \frac{3N}{64})]W_{64}^{3(4k_3+k_4)} \tag{4.10}$$

Equations (4.6), (4.8) and (4.10) are the three butterfly units in radix-4^3 algorithm. Each of these butterfly operations are based on radix-4 butterfly and hence the name Radix-4^3.

4.3 Radix-4^3 Architecture

In [15], an efficient, high throughput, high speed, pipelined radix-4^3 FFT architecture is proposed. A brief review of the architecture in [15] is given here. A 64 point FFT is organized using three radix-4 FFT engines. This results in a radix-4^3 FFT which is equivalent to radix-64 FFT. Using this radix-4^3 as the basic block, higher order FFTs are constructed. A radix-4^3 (R4^3) block consists of three radix-4 (R4) engines.

Radix-4 Engine

Radix-4 engine used in [15] is shown in Figure 4.1. Inputs to R4 engine is streamed at one complex point per cycle. After 4 cycles, the inputs from "Reg A" is loaded into "Reg B". The four parallel outputs from "Reg B" forms the inputs for radix-4 butterfly operation. The control unit will set the add/sub units to compute the four radix-4 outputs in four consecutive cycles.

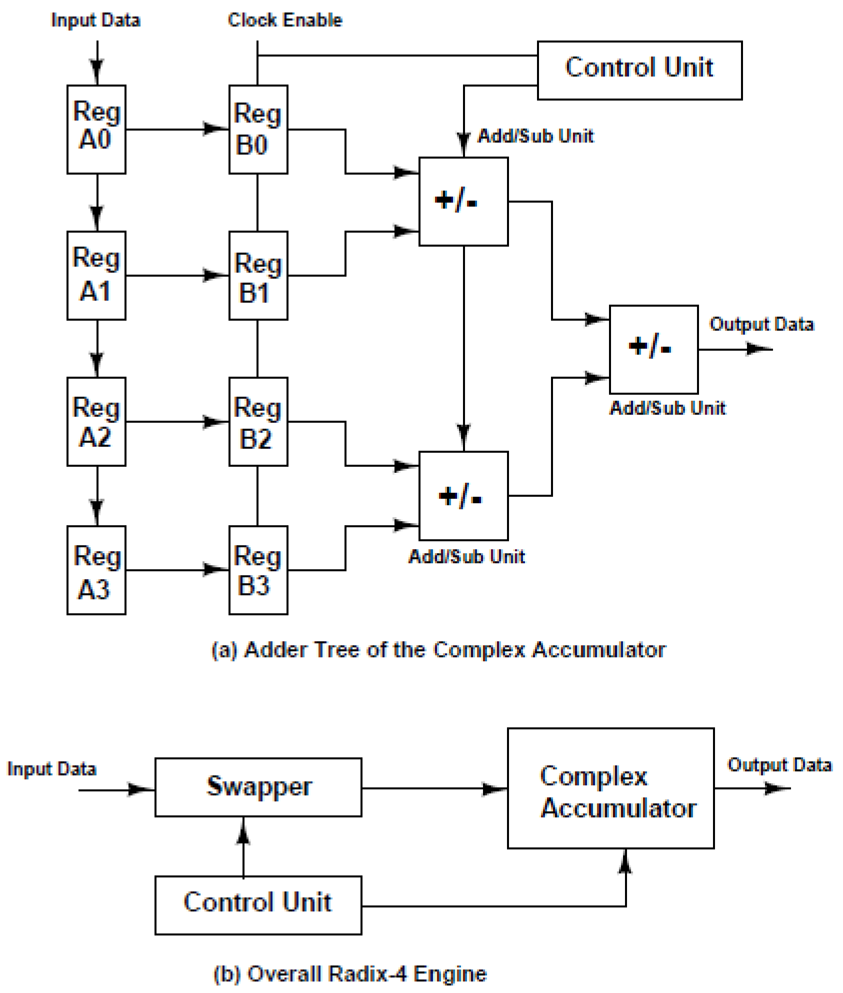


FIGURE 4.1: Radix-4 Engine used in [15]

The R4^3 architecture in [15] is a fully systolic architecture for giga sample applications. It consists of three R4 butterfly units, two complex multipliers and two dual bank memories as shown in Figure 4.2. The twiddle factors are stored in two separate ROMs. An R4^3 block is a 64 point FFT processor in standalone case. Each R4 engine used in R4^3 block uses an unrolled architecture. Since the outputs from R4 engine comes in consecutive cycle, they have to be reordered before applying to the next stage. The dual bank memories are used for this reordering purpose.

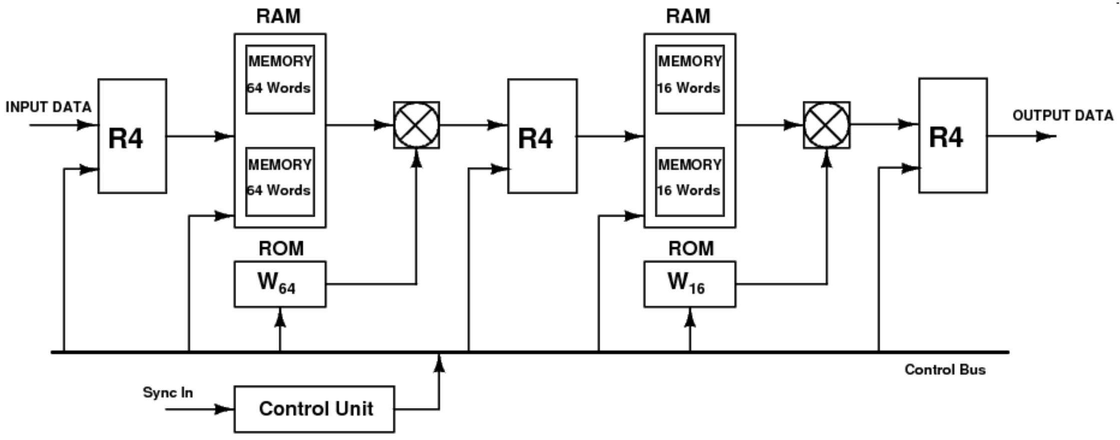


FIGURE 4.2: Radix-4^3 Architecture in [15]

The advantage of using a R4^3 instead of radix-64 FFT is the simplicity of the circuit due to basic R4 blocks, and regularity of the structure. The complex points are 14 bits wide and for high precision, each of the internal stages takes 16 and 20 bits width. Final outputs are truncated to 14 bits. Twiddle factors are 18 bit complex points.

The FFT architecture in [15], [17] uses radix-4^3 as the basic block to implement higher order FFTs like 4K, 16K, 64K and 256K point FFT. Two radix-4^3 engines are used in cascade for a 4K point FFT. After every radix-4^3 block, outputs has to be reordered before giving it to the next stage. The radix-4^3 in [15], [17] is sequential. Internal to the radix-4^3, there are 144 words of intermediate buffers. After the first radix-4^3 block in the 4K FFT, 8K intermediate RAM is required to store the complete 4K points before reordering. This 4K FFT is used in a sequential-parallel architecture to give 16K FFT. Four 16K FFTs in parallel forms a 64K point FFT. Cascading three R-4^3 blocks will result in a 256K point FFT. As the FFT order increases, the RAMs form a significant part of the total area and results in longer latencies. For example, for the 64K FFTs in [15], total RAM required is 192K words, out of which 32K is at input, while 160K is after the first set of radix-4^3 block.

4.4 Summary of R4^3 FFT Architecture in [15]

The radix-4^3 block implemented using the butterfly unit given in Figure 4.1 will take in 64 inputs sequentially and produce 64 outputs sequentially in a `fixed order`. By using this architecture for the R4^3 engine in [15], the outputs from each stage of R4 needs to be reordered before applying to the next stage. Similarly if two R4^3 engines are cascaded, the intermediate outputs also have to be reordered. Since the subsequent blocks will require the inputs to be reordered before processing, intermediate RAMs are required between every cascaded stage in this implementation. This re-ordering needs intermediate memory and the memory size grows as the FFT length increases. So for higher order FFTs, this implementation results in considerable increase in area which can be optimized by our implementation presented in Chapter 5.

Chapter 5

Proposed Architecture for Computation of Longer FFTs

In this chapter, a parallel unrolled FFT architecture based on Radix-4^3 and Radix-4^4 algorithms which are used to compute FFTs of large size, is proposed. Compared to the state-of-art implementations our architecture results in comparable throughput, low latency and significant memory reduction. The basic building block of this architecture is a novel Radix-4 butterfly unit. Replacing the radix-4 engine in [15] which is explained in chapter 4 with the novel parallel radix-4 butterfly unit is the basis of the proposed architecture. Moreover, $R4^3$ and $R4^4$ blocks are fully parallel architectures in this design. This $R4^3$ and $R4^4$ can be cascaded to get 4K and 64K point FFTs are also explained here.

5.1 A Parallel Radix-4 Butterfly architecture

The proposed FFT architecture in this work is based on the parallel Radix-4 butterfly unit. Each radix-4 butterfly unit produces four outputs. The basic operations in a radix-4 butterfly are given in equation (5.1).

$$\begin{aligned} X(0) &= x(0) + x(1) + x(2) + x(3) \\ X(1) &= x(0) - jx(1) - [x(2) - jx(3)] \\ X(2) &= x(0) - x(1) + x(2) - x(3) \\ X(3) &= x(0) + jx(1) - [x(2) + jx(3)] \end{aligned} \tag{5.1}$$

Three complex adders/subtractors are required to produce one output. In [15], the radix-4 butterfly architecture consists of 3 adder/subtractors and the swap unit. But there, the inputs to the butterfly units are applied in serial, and takes four clock cycles to start the operation. All the four outputs comes out in consecutive clock cycles. For every set of four inputs, four outputs are generated one after the other.

The radix-4 engine implementation in this work is shown in Figure 5.1. The radix-4 butterfly unit is fully parallel in this case. The four complex inputs are in parallel and are loaded to the input registers in a single clock cycle. Multiplication with 'j' can be done

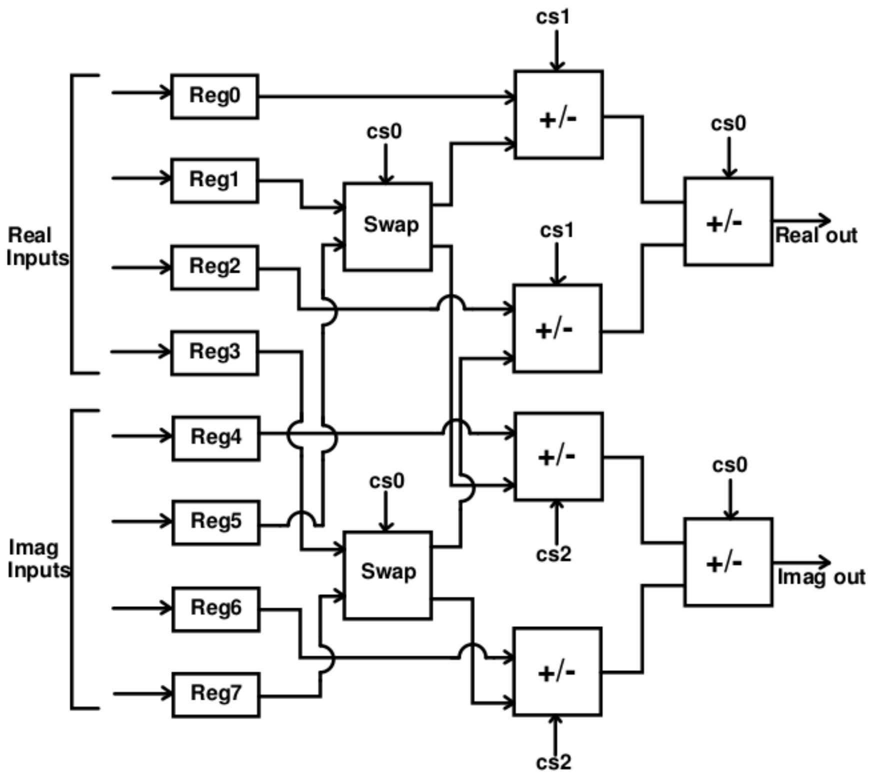


FIGURE 5.1: Architecture of Radix-4 Engine

using a swap unit which performs swapping of real and imaginary values of the input. The

TABLE 5.1: Control Signals in Radix-4 Engine

Mode	Output	cs0	cs1	cs2
0 0	X(0)	0	0	0
0 1	X(1)	1	0	1
1 0	X(2)	0	1	1
1 1	X(3)	1	1	0

control signal in `swap` module selects whether to swap real and imaginary inputs or not. The `adder/subtractor` control signal selects adder or subtractor. From equation (5.1), radix-4 engine can generate one of the four outputs based on the control signals. The output to be generated for each set of inputs is controlled by a two bit '`mode select`' signal. The individual control signals cs0, cs1 and cs2 are derived from mode select signal and is shown in Table 5.1.

A nodal representation of radix-4 butterfly unit is shown in Figure 5.2. Here, the butterfly

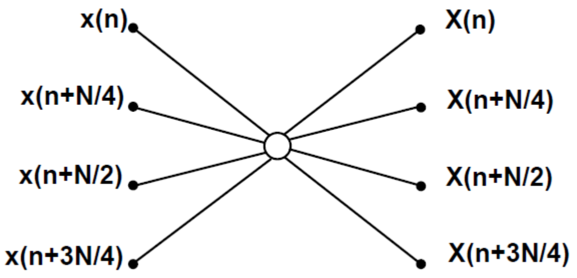


FIGURE 5.2: Radix-4 Butterfly node

outputs are not consecutive. Based on the '`mode select`' signal, outputs can be generated in any order. This is an advantage in the case where this radix-4 engine has to be used in building higher blocks for an FFT architecture. The `mode select` signal decides the generation of one output out of the four.

5.2 Proposed Radix-4^3 Architecture using Novel R4 Engine

The Radix-4^3 FFT require 64 inputs to produce any of the 64 outputs in a single set of computation. If we can implement the radix-4^3 block such that the outputs generated are already `reordered`, there can be significant savings in the intermediate RAMs and latency. This will need a parallel unrolled architecture which can accept all 64 inputs simultaneously and produce whichever output required based on some '`select inputs`'.

So the basic radix-4 butterfly unit also has to be parallel.
Signal Flow Graph of a Radix-4^3 FFT is shown in Figure 5.3. From Figure 5.3 it is clear that stage1 needs 16 radix-4 butterfly units, stage2 needs 4 radix-4 butterfly units and 1 radix-4 butterfly unit for the last stage. Each node represents a radix-4 butterfly unit. The first butterfly takes in the inputs x(1), x(16), x(32) and x(48).

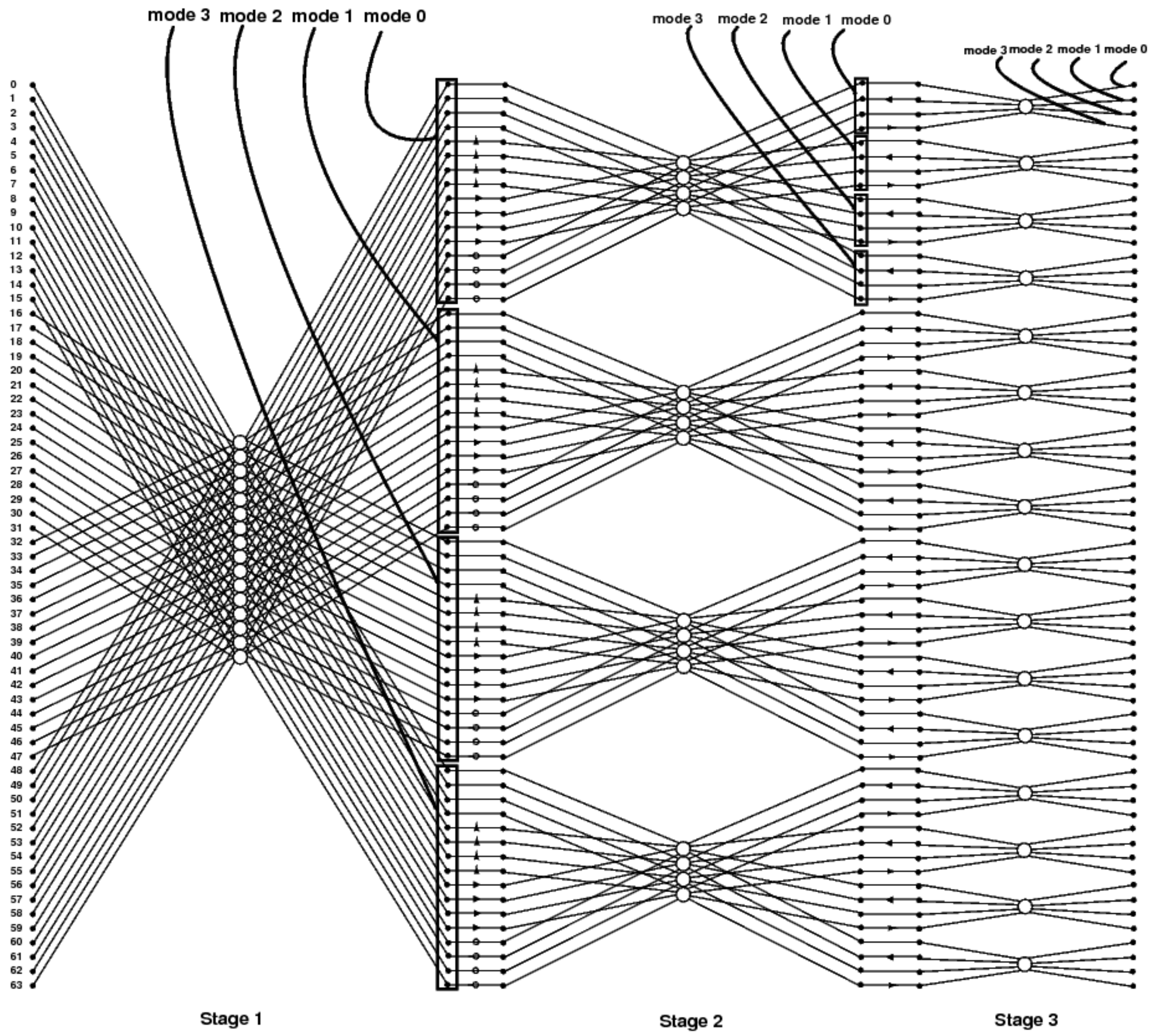


FIGURE 5.3: Signal Flow Graph of Radix-4^3 FFT

Mode Selection in R4^3 Block

In the SFG shown in Figure 5.3, the first 16 outputs which are grouped together, are the first output of all 16 butterflies. This is obtained by configuring the `mode select` of all radix-4 engines as `mode 0`. These outputs are multiplied with the corresponding twiddle factors. Similar operation is performed in the second stage. Here, four radix-4 engines

TABLE 5.2: Mode Selection in Different Stages of R4^3 Block

Stage 1	Stage 2	Output generated
0 0	0 0	0 1 2 3
0 0	0 1	4 5 6 7
: :	: :	: :
1 1	1 1	60 61 62 63

are required to process the 16 outputs that are obtained from first stage. Again, the first four outputs are generated by configuring the `mode select` of all four radix-4 engines as `mode 0`, keeping the first stage radix-4 engines in `mode 0` itself. Now using this four outputs, we can generate the output required in final stage with another mode selection in final stage. In this work, since a fully parallel radix-4 engine is used in the last stage, mode selection is not needed there. For obtaining the next set of four outputs, keeping the first stage radix-4 engines in `mode 0`, change the mode of second stage to `mode1`. This procedure is repeated to get all the 64 outputs. Mode select signals in different stages of Radix-4^3 Block are given in Table 5.2. First four outputs in Figure 5.3 are grouped as 0,1,2,3 which are mentioned in Table 5.2 and the next four are 4,5,6,7 and so on. The final outputs are in digit reversed order.

From Figure 5.3, it can be seen that the consecutive outputs in the final stage are from the same butterfly node using the same set of outputs from the previous stage. So instead of using a `mode select` signal for the last stage, a fully parallel radix-4 engine is used. This generates all four outputs in parallel and requires eight adder/subtractors. This results in a four fold increase in the throughput.

The proposed Radix-4^3 FFT architecture is shown in Figure 5.4. This fully parallel unrolled architecture uses the novel radix-4 engine as its basic block. In the first stage 16 R4 engines are used. The inputs to these R4 engines are reordered inputs. There are 16 TF ROMs for storing the twiddle factors which are denoted as `W16`. Each ROM stores four TF values. In the second stage only four R4 engines are required. TF values (`W64`), are stored in four ROMs, each ROM storing 16 TF values. The last stage R4 engine is fully parallel which produces all four outputs in a single clock cycle.

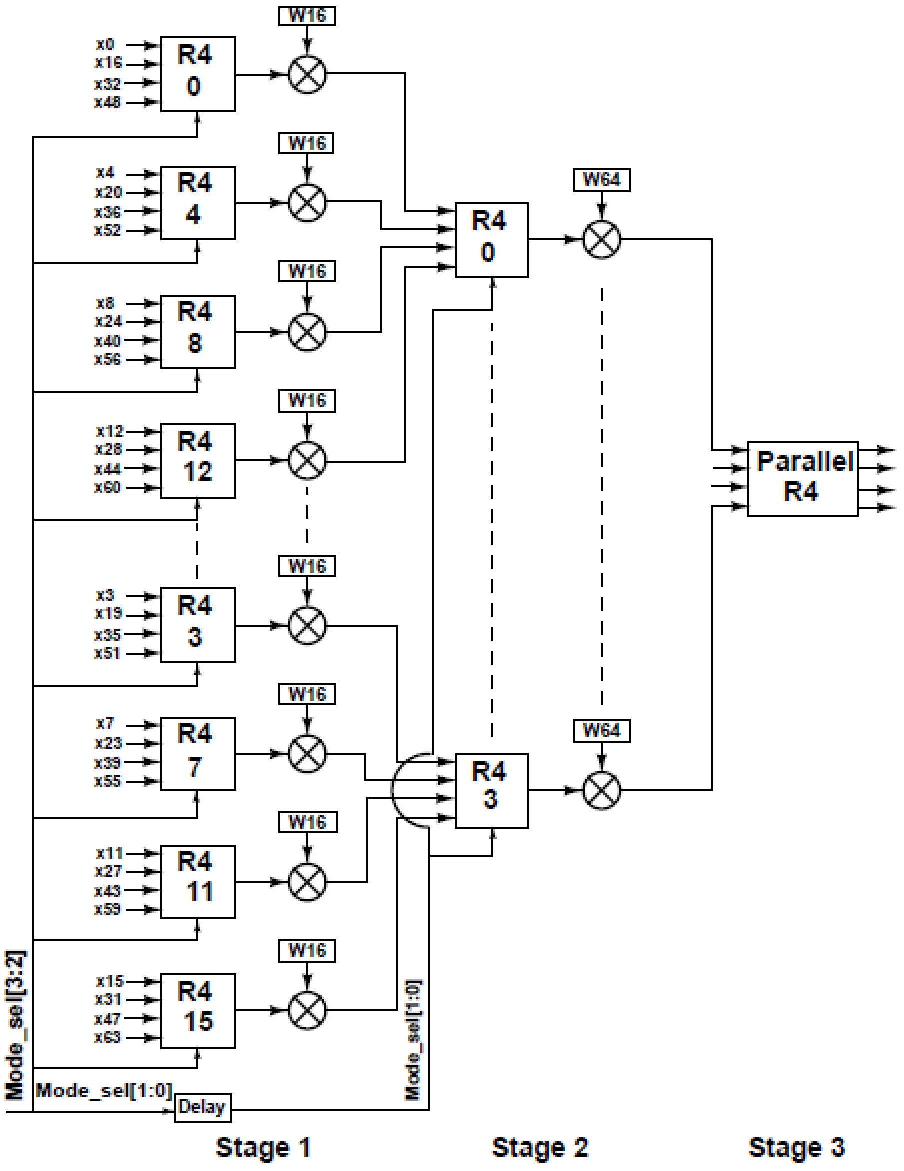


FIGURE 5.4: Proposed Radix-4^3 Architecture

Mode selection bits are denoted as `mode_sel[3:0]`. The mode select bits to the first stage are `mode_sel[3:2]`. The next two bits i.e. `mode_sel[1:0]` are delayed for synchronisation, and are given to the next stage. Thus all intermediate outputs are already reordered. Considerable memory savings is obtained here, by without using any intermediate RAMs. The entire 64 point FFT computation takes 16 clock cycles.

Out of all the N multipliers in each stage, the first N/4 multipliers always get the twiddle factor input as 1. So these N/4 multipliers can be removed from the actual implementation. Consequently, first stage has 12 multipliers instead of 16 and next stage has 3 multipliers instead of 4.

`Mode Select` signal is implemented using a simple four bit up-counter which is the only control circuitry used in this architecture.

5.2.1 4K Point FFT Architecture using R4^3 Block

Radix-4^3 FFT architecture presented in section 5.2 can be extended to develop a 4K point FFT. This can be done by cascading two R4^3 blocks [15].

DFT equation for a 4K point FFT is given in the following equation [7],

$$X(k) = \sum_{n=0}^{4095} x(n) W_{4096}^{nk} \quad k = 0, 1, 2, ...4095$$
$$X(k) = \sum_{n=0}^{64.64-1} x(n) W_{64.64}^{nk} \tag{5.2}$$

Decomposing the indices n and k,

$$n = n_1 + 64n_2 \quad where \quad n_1, n_2 = 0, 1, 2, ...63$$
$$k = 64k_1 + k_2 \quad where \quad k_1, k_2 = 0, 1, 2, ...63 \tag{5.3}$$

The DFT equation in (5.2) becomes

$$X(64k_1 + k_2) = \sum_{n_1=0}^{63} \sum_{n_2=0}^{63} x(n_1 + 64n_2) W_{64.64}^{(64n_1k_1 + 4096n_2k_1 + 64n_2k_2 + n_1k_2)}$$
$$X(64k_1 + k_2) = \sum_{n_1=0}^{63} \{[\sum_{n_2=0}^{63} x(n_1 + 64n_2) W_{64}^{n_2k_2}] W_{4096}^{n_1k_2}\} W_{64}^{n_1k_1} \tag{5.4}$$

Equation (5.4) shows that to get a 4K point FFT, a 64 point FFT computation is performed which is multiplied with the twiddle factors of 4K point and then another 64 point FFT calculation is required. Thus two R4^3 blocks are cascaded to form a 4K point FFT. Architecture of a 4K point FFT is shown in Figure 5.5. The implementation consists of two R4^3 blocks in cascade. This can take in 4 sets of inputs in a single clock cycle which is stored in the input memory block. The input memory block consists of 128, 16×128 RAMs which can feed the 64 inputs of the first R4^3 block in parallel. The first R4^3 block outputs four complex points which are multiplied with the W_{4k} twiddle factors using four complex multipliers. The intermediate register based buffering stage buffers the first stage outputs to be fed to the second R4^3 block. The final output will be at a throughput of four complex points per clock cycle.

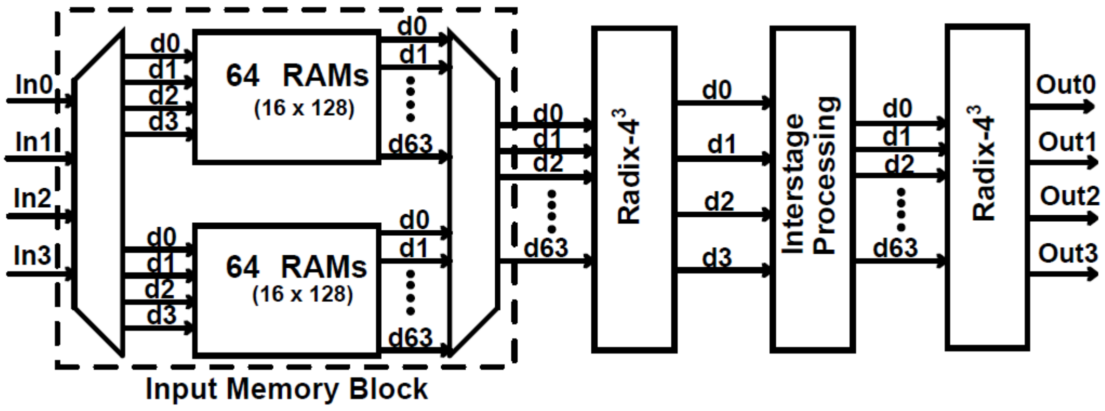


FIGURE 5.5: 4K Point FFT Architecture using R4^3 Block

Data Scheduling and Addressing [15]

The first R4^3 block can compute 4 complex points of a 64 point FFT per clock cycle. Each of this 64-tuple contains elements with address of $64i+k$ where i=0 to 63 and k is the tuple index ranging from 0 to $\frac{4K}{64}$-1. Within each 64-tuple, the processing is done in terms of quadruples with one R4 engine per quadruple. The quadruples are formed by elements with address $16i_1+k_1$, i_1= 0 to 3 and k_1= 0 to $\frac{64}{4}$-1. Thus there are total 16 quadruples and corresponding 16 R4 engines. For each 64-tuple input, the first R4^3 block, depending on the 4 bit `mode select`, can produce 16 different sets of outputs per clock cycle. So it takes 16 clock cycles to produce all the 64 outputs from a single 64-tuple input.

The second R4^3 block has to compute $\frac{4K}{64}$ FFTs with each transform of 64-tuple from the outputs of first R4^3 block. The elements in each of the 64-tuple is of the form $64j+l$, where j=0 to 63 and l is the tuple index ranging from 0 to $\frac{4K}{64}$-1. So each 64-tuple input to the first R4^3 block contributes only one element out of 64 for the second R4^3 block. Therefore if the first R4^3 block computes all 64 outputs from a single 64-tuple consecutively, the next processor has to wait till all the 4K points are generated. This will require 4K intermediate buffering [15].

This buffering can be significantly reduced if the first block is scheduled to compute all the 64 elements required by second block in consecutive cycles. In this way only 64×4 buffering is required, 'four times' because four outputs are generated per clock cycle. The $R4^3$ architecture proposed in our work enables this. In each clock cycle a different input 64-tuple is fed to first block and one element is produced. The `mode select` is held at `0` for 64 consecutive cycles, so that all 64 elements required by second block are produced. Since the first stage produces four outputs, four sets of input 64-tuple required for second block are computed in 64 cycles. The second block takes 16 clock cycles to compute 64 point FFT from a single 64-tuple and can complete computation for all four 64-tuple in 64 cycles. This is done by changing the `mode select` as mentioned in Table 5.2.
After the first 64 cycles, the inputs to first block again comes from k=0 tuple. Then mode select to first block is changed to `1` and the computation is repeated. This is repeated for all `mode select` values from `0` to `F`. So it takes 16×64 clock cycles to produce the complete 4K points.

Interstage Processing

Details of Interstage Processing block is given in Figure 5.6. The Interstage Processing Stage consists of Twiddle Factor ROM for storing the 4K twiddle factors, Complex Multipliers and intermediate buffers as shown in Figure 5.6.

Twiddle factor word length is chosen as 18 bits for both real and imaginary values. In each clock cycle four TF multiplications take place. The ROM size is $1K \times 144$ i.e. $(1K \times (4 \times 36)\ )$. The 10 bit control signal `addr[9:0]`, for TF ROM is generated by the address generator. The complex multiplier takes in 16 bit stage1 outputs and 18 bit twiddle factors and generate 16 bit outputs.
The Interstage Processing Stage consists of four sets of 64 shift registers into which the first stage outputs are shifted in. After every 64 cycles, the shift register outputs are loaded in parallel into a second set of registers using a control signal,'`load`', which is generated taking into account the delay for first $R4^3$ block. These will be the inputs for the next block.

Input Memory Block

Since the inputs require reordering before giving to first $R4^3$ block, 4K buffers is required at input. The buffers have to be duplicated so that while one set reads in the input data while the other one can read out the output. This is shown in Fig 5.7. Each set consists

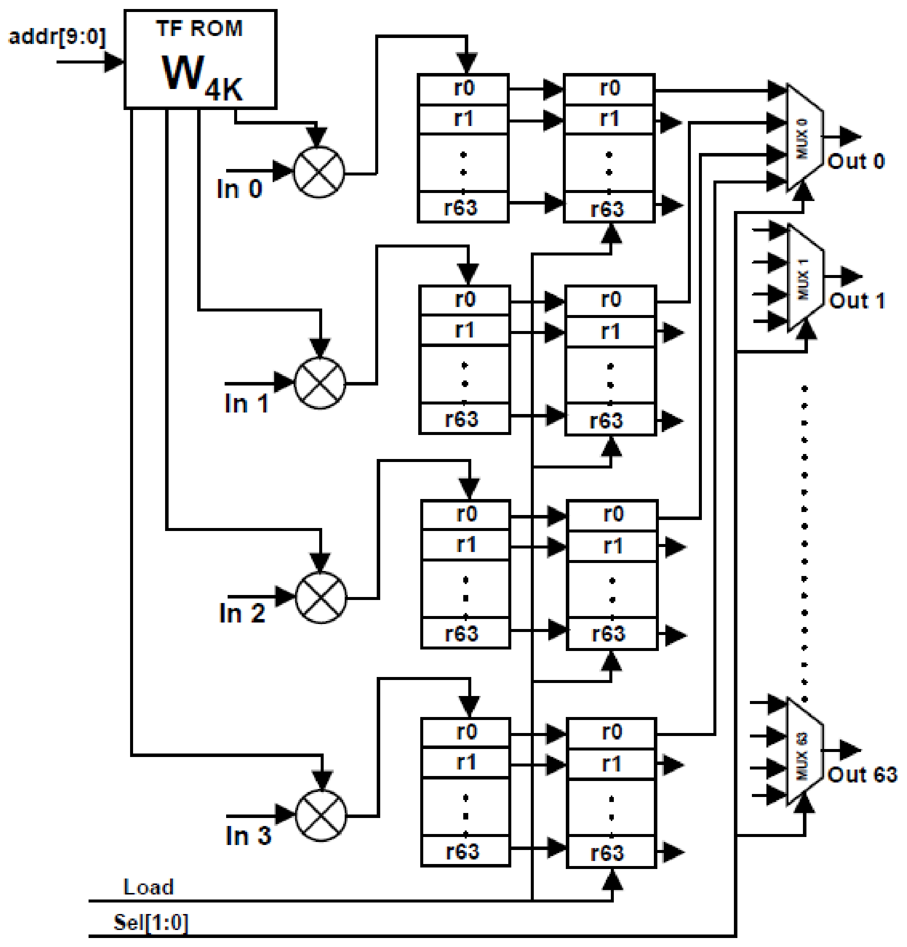


FIGURE 5.6: Interstage Processing Block in 4K FFT

of 64 RAMs of 16×128 size. Four complex inputs come in a single clock cycle. These are made into a 128 bit word and written into one location. The write addresses are given sequentially and the first 16 sets of data are written to the first RAM before moving to the next RAM.

The read from the RAMs are done in parallel with each RAM giving one 128 bit word per clock cycle. The address to each RAM will be same so that the same index is read from the same RAM. This make sure the required address rearrangement for the first $R4^3$ block.

Control circuitry

The control circuitry consists of a 11 bit counter. The input RAMs need 10 bit addressing and there are two sets of RAMs. The read, write and chip select signals have to be generated and the mode select signals of both $R4^3$ blocks are also generated from the counter. These signals are synchronized with respect to the delays of the previous stage.

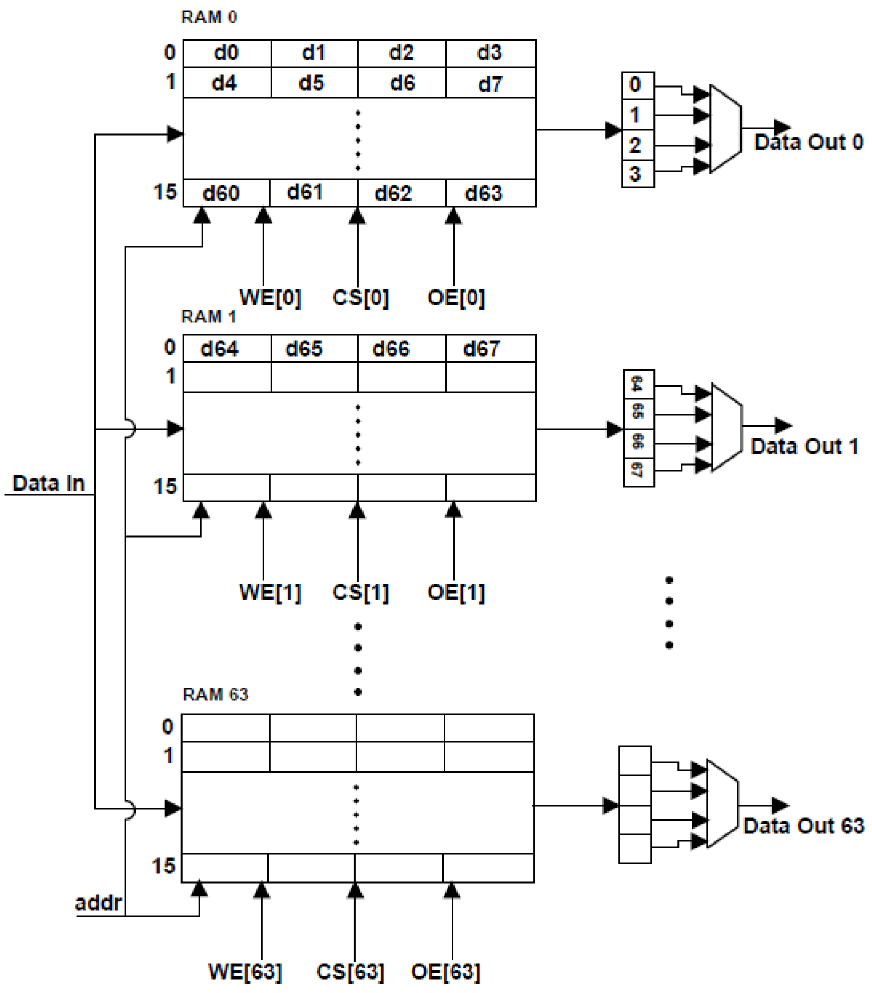


FIGURE 5.7: Input Memory Block in 4K FFT

Resource Utilization

In the 4K architecture, the inputs to the first R4^3 block are changing every clock cycle. Hence here the resources are fully utilized. The second R4^3 block in the 4K FFT processor, takes a different input once in 16 clock cycles. Thus here the 16 R4 engines at first stage will have the same inputs for 16 clock cycles while the mode select changes once in every 4 clock cycles. The inputs to the next 16 multipliers also will change only once in every 4 clock cycles. For the next set of 4 R4 engines in stage 2 and complex multipliers, the input change is once in every clock cycle. Thus the resources at the first stage is significantly underutilized. This is not the case with the first R4^3 block since inputs are changed in every clock.

The resource utilization for the second R4^3 block can be made 100% by changing the architecture. This will result in 4 R4 engines in first stage and an R4 engine in the second stage. But this will result in the two blocks of R4^3 having different architectures. With the current architecture although the resources are not 100% utilized, we still do not have any significant area overhead and for scalability we will go with identical architectures for both the blocks.

Motivation for Radix-4^4 FFT Algorithm and Architecture
There are very few hardware implementations of 64K and above FFTs in literature. The purpose of this work is to develop a 64K FFT processor with good performance in terms of latency and throughput. In [15], [17] a fully systolic [21] FFT processor based on radix-4^3 algorithm for giga-sample applications is described. Cascading two radix-4^3 engines resulted in a 4K point FFT. This 4K FFT is used in parallel to extend to 64K point FFT. Although the basic radix-4^3 engine has less adder and multiplier complexity, extending this in parallel to 64K results in an increase in the complexity. Here, we implement a 64K point FFT by using two radix-4^4 engines in cascade. The basic radix-4^4 architecture is such that no memory is required between the two cascaded stages which makes the 64K FFT area efficient and with low latency while achieving higher throughput. A high throughput, low latency FFT architecture based on radix-4^4 algorithm which can be used to implement area and memory efficient FFTs of 64K point and above is presented here. The radix-4^4 engine in the architecture cascades four radix-4 engines which results in a 256 point FFT. The parallel pipelined architecture of a radix-4^4 engine is utilized to improve the throughput and reduce the latency of FFT computation.

5.3 Radix-4^4 Algorithm

In this section, organization of a Radix-4^4 block using four radix-4 engines is described. We will refer the radix-4 and radix-4^4 units as R4 and R4^4. Radix-4^4 algorithm is an extension to radix-4^3 algorithm in [15]. The derivation of R4^4 algorithm follows the same procedure as R4^3 in [15]. But here we use a five dimensional linear index map instead of the four dimensional map in [15]. A radix-4^4 unit consists of four butterfly stages which can be obtained from the common factor algorithm. The cascade decomposition lies in four steps which are explained below. A five dimensional linear mapping is applied to the indices n and k, to equation 4.1,

$$\begin{aligned} n &= n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{N}{16}n_4 + \frac{N}{4}n_5 \\ k &= 256k_1 + 64k_2 + 16k_3 + 4k_4 + k_5 \end{aligned} \tag{5.5}$$

Apply equation (5.5) to the DFT formula in (4.1),

$$\begin{aligned} X(256k_1 + 64k_2 + 16k_3 + 4k_4 + k_5) &= \sum_{n_1=0}^{\frac{N}{256}-1} \sum_{n_2=0}^{3} \sum_{n_3=0}^{3} \\ \sum_{n_4=0}^{3} \sum_{n_5=0}^{3} [x(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{N}{16}n_4 + \frac{N}{4}n_5)]W_N^{nk} \end{aligned} \tag{5.6}$$

Decomposing the twiddle factors,

$$\begin{aligned} W_N^{nk} = W_{\frac{N}{256}}^{n_1k_1} W_N^{n_1(64k_2+16k_3+4k_4+k_5)} W_{256}^{n_2(16k_3+4k_4+k_5)} \\ W_{64}^{n_3(4k_4+k_5)} W_{16}^{n_4k_5} (-j)^{(n_2k_2+n_3k_3+n_4k_4+n_5k_5)} \end{aligned} \tag{5.7}$$

Substituting (5.7) in (5.6) and expanding the summation with index n_5,

$$\begin{aligned} X(256k_1 + 64k_2 + 16k_3 + 4k_4 + k_5) &= \sum_{n_1=0}^{\frac{N}{256}-1} \sum_{n_2=0}^{3} \\ \sum_{n_3=0}^{3} \sum_{n_4=0}^{3} [BF1(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{N}{16}n_4)] \times \\ (-j)^{(n_2k_2+n_3k_3+n_4k_4)} W_{\frac{N}{256}}^{n_1k_1} W_N^{n_1(64k_2+16k_3+4k_4+k_5)} \\ W_{256}^{n_2(16k_3+4k_4+k_5)} W_{64}^{n_3(4k_4+k_5)} W_{16}^{n_4k_5} \end{aligned} \tag{5.8}$$

The first butterfly unit, BF1 is given by

$$\begin{aligned} BF1 = x(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{N}{16}n_4) + \\ (-j)^{k_5} x(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{N}{16}n_4 + \frac{N}{4}) + \\ (-1)^{k_5} x(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{N}{16}n_4 + \frac{N}{2}) + \\ (j)^{k_5} x(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{N}{16}n_4 + \frac{3N}{4}) \end{aligned} \tag{5.9}$$

Expanding the summation in (5.8) with index n_4,

$$\begin{aligned} X(256k_1 + 64k_2 + 16k_3 + 4k_4 + k_5) = \sum_{n_1=0}^{\frac{N}{256}-1} \sum_{n_2=0}^{3} \\ \sum_{n_3=0}^{3} [BF2(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3)] \times \ (-j)^{(n_2k_2+n_3k_3)} \\ W_{\frac{N}{256}}^{n_1k_1} W_N^{n_1(64k_2+16k_3+4k_4+k_5)} W_{256}^{n_2(16k_3+4k_4+k_5)} \\ W_{64}^{n_3(4k_4+k_5)} \end{aligned} \tag{5.10}$$

The second butterfly unit, BF2 is given by

$$\begin{aligned} BF2 = BF1(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3) + \\ (-j)^{k_4}[BF1(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{N}{16})]W_{16}^{k_5} + \\ (-1)^{k_4}[BF1(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{N}{8})]W_{8}^{k_5} + \\ (j)^{k_4}[BF1(n_1 + \frac{N}{256}n_2 + \frac{N}{64}n_3 + \frac{3N}{16})]W_{16}^{k_5} W_{8}^{k_5} \end{aligned} \tag{5.11}$$

Expanding the summation in (5.10) with index n_3,

$$\begin{aligned} X(256k_1 + 64k_2 + 16k_3 + 4k_4 + k_5) = \sum_{n_1=0}^{\frac{N}{256}-1} \sum_{n_2=0}^{3} \\ BF3(n_1 + \frac{N}{256}n_2) \times \ (-j)^{(n_2k_2)} W_{\frac{N}{256}}^{n_1k_1} \\ W_N^{n_1(64k_2+16k_3+4k_4+k_5)} W_{256}^{n_2(16k_3+4k_4+k_5)} \end{aligned} \tag{5.12}$$

The third butterfly unit, BF3 is given by

$$\begin{aligned} BF3 = BF2(n_1 + \frac{N}{256}n_2) + \\ (-j)^{k_3}[BF2(n_1 + \frac{N}{256}n_2 + \frac{N}{64})]W_{64}^{4k_4+k_5} + \\ (-1)^{k_3}[BF2(n_1 + \frac{N}{256}n_2 + \frac{N}{32})]W_{64}^{2(4k_3+k_4)} + \\ (j)^{k_3}[BF2(n_1 + \frac{N}{256}n_2 + \frac{3N}{64})]W_{64}^{3(4k_4+k_5)} \end{aligned} \tag{5.13}$$

Expanding the summation in (5.12) with index n_2,

$$X(256k_1 + 64k_2 + 16k_3 + 4k_4 + k_5) = \sum_{n_1=0}^{\frac{N}{256}-1} BF4(n_1) \times \; W_{\frac{N}{256}}^{n_1k_1} W_N^{n_1(64k_2+16k_3+4k_4+k_5)} \tag{5.14}$$

The fourth butterfly unit, BF4 is given by

$$\begin{aligned} BF4 = BF3(n_1) &+ \\ (-j)^{k_2}[BF3(n_1 + \frac{N}{256})]W_{16}^{k_3}W_{64}^{k_4}W_{256}^{k_5} &+ \\ (-1)^{k_2}[BF3(n_1 + \frac{N}{128})]W_{8}^{k_3}W_{32}^{k_4}W_{128}^{k_5} &+ \\ (j)^{k_2}[BF3(n_1 + \frac{3N}{256})]W_{256}^{3(16k_3+4k_4+k_5)} & \end{aligned} \tag{5.15}$$

Equations (5.9), (5.11), (5.13) and (5.15) describe the four different butterfly units of radix-4^4 block, which are based on radix-4 butterfly operation. The R4^4 block requires log_4N-1 complex multipliers and 3log_4N complex adders, where N denotes number of complex input sequences. Each R4^4 block uses four radix-4 stages and each of the radix-4 unit requires 3 complex adders for obtaining an output.

5.4 Radix-4^4 Architecture

In the proposed architecture, the inputs to R4 engine are applied in parallel and the output to be generated can be selected using the mode select. By using this novel R4 engine, we can design a R4^4 FFT, where, by applying a set of 256 inputs, any one of the 256 outputs can be computed in a single set of computations. This will result in zero intermediate memory within the R4^4 block. For developing FFT architectures of large size, this will considerably reduce the intermediate memory between two cascaded R4^4 blocks.

Radix-4^4 Architecture is shown in Fig 5.8. There are four stages of R4 butterfly units. All the 256 complex inputs are fed in parallel to the R4 engines. 64 R4 butterfly operations (BF1) are performed in `stage 1` using 64 R4 engines. In `stage 2`, the 64 outputs from `stage 1` are connected to 64 complex multipliers for twiddle factor multiplications given in equation (5.11) and fed to 16 R4 units. `Stage 3` consists of 16 complex multipliers and four R4 engines and `stage 4` has four complex multipliers and one R4 unit respectively. The last stage uses a `parallel` R4 engine which produces all four butterfly outputs in a single clock cycle. Once all the 256 inputs are loaded, four outputs are generated per clock cycle.

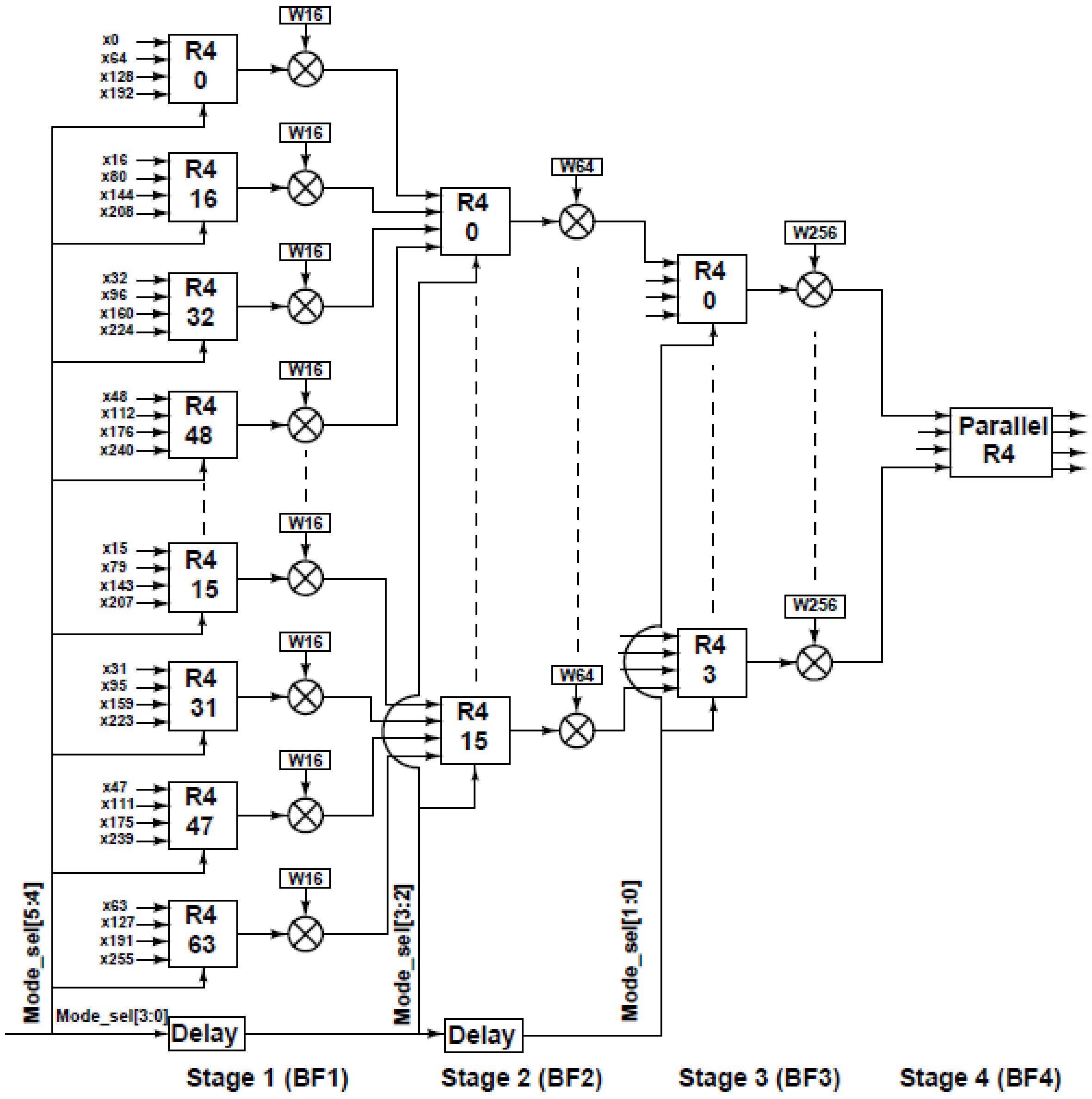


FIGURE 5.8: Proposed Radix-4^4 Architecture

The twiddle factors at each stage are stored in separate ROMs. Each radix-4 engine is associated with a ROM. Each `W16` stores 4 complex values, `W64` stores 16 TFs and each `W256` stores 64 twiddle factors. Out of all the N multipliers in each stage, the first N/4 multipliers always get the twiddle factor input as 1. So these N/4 multipliers can be removed from the actual implementation. Consequently, `stage 2` has 48 multipliers

TABLE 5.3: Mode Selection in Different Stages of R4^4 Block

Stage 1	Stage 2	Stage 3	Output generated
0 0	0 0	0 0	0 1 2 3
0 0	0 0	0 1	4 5 6 7
: :	: :	: :	: :
1 1	1 1	1 1	252 253 254 255

instead of 64, `stage 2` has 12 multipliers instead of 16 and `stage 3` has 3 multipliers instead of 4, at the input side.

Mode Selection in R4^4 Block

In the Figure 5.8, stages 1-3 have `mode select` signal which is denoted as `Mode_sel[5:0]`. By controlling this signal, outputs from 0-255 can be generated. Different mode selection bits in each stage are `Mode_sel[5:4]`, `Mode_sel[3:2]` and `Mode_sel[1:0]`. Various combinations of mode selection in different stages are mentioned in Table 5.3.

For the first three stages, the outputs generated by the R4 engines are controlled by the mode select signals. For one cycle of computation, all the R4 engines in a single stage requires the same two bit mode select signal. For the three stages together, six bits of mode select are required. These six bits are fed together at input and for each stage, the mode selects are delayed to match the pipeline delays of the previous stages. For each set of 256 inputs, the six bit mode select signals change from '000000' to '111111' in 64 cycles. In each cycle, the fourth stage computes four outputs and thus the entire 256 point FFT computation takes 64 clock cycles. Apart from six bit mode select signal, no other control signal is required and the control circuit can be implemented as a simple 6 bit counter. One full 256 point FFT takes 64 clock cycles + pipeline delay. The next set of inputs can be loaded after 64 clock cycles. The `mode select` signals of each stage has to be pipeline matched with the data path of the previous stages.

5.4.1 64K Point FFT Architecture using R4^4 Block

Figure 5.9 represents the architecture of a 64K Point FFT. Here, two R4^4 blocks are cascaded as in the case of a 4K point FFT explained in Section 5.2.1. The input memory block consists of 512, 64×128 RAMs which can feed the 256 inputs of the first R4^4 block in parallel. The first R4^4 block outputs four complex points which are multiplied with the W_{64k} twiddle factors using four complex multipliers. The Interstage processing block buffers the first stage outputs to be fed to the second R4^4 block. The final output will be at a throughput of four complex points per clock cycle.

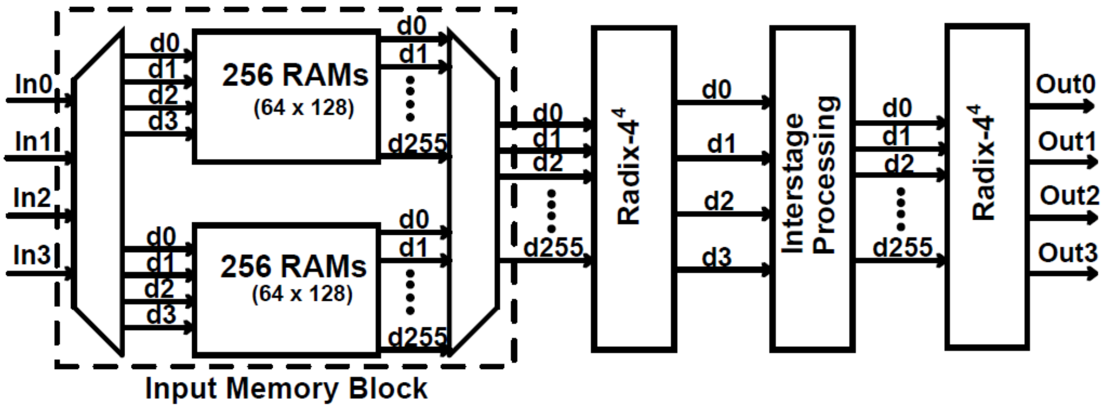


FIGURE 5.9: 64K Point FFT Architecture using R4^4 Block

Data Scheduling and Addressing

The first R4^4 block can compute 4 complex points of a 256 point FFT per clock cycle. Each of this 256-tuple contains elements with address of $256i+k$ where i=0 to 255 and k is the tuple index ranging from 0 to $\frac{64K}{256}$-1. Within each 256-tuple, the processing is done in terms of quadruples with one R4 engine per quadruple. The quadruples are formed by elements with address $64i_1+k_1$, i_1= 0 to 3 and k_1= 0 to $\frac{256}{4}$-1. Thus there are total 64 quadruples and corresponding 64 R4 engines. For each 256-tuple input, the first R4^4 block, depending on the 6 bit `mode select`, can produce 64 different sets of outputs per clock cycle. So it takes 64 clock cycles to produce all the 256 outputs from a single 256-tuple input.

The second R4^4 block has to compute $\frac{64K}{256}$ FFTs with each transform of 256-tuple from the outputs of first R4^4 block. The elements in each of the 256-tuple is of the form $256j+l$, where j=0 to 255 and l is the tuple index ranging from 0 to $\frac{64K}{256}$-1. So each 256-tuple input to the first R4^4 block contributes only one element out of 256 for the second R4^4 block. Therefore if the first R4^4 block computes all 256 outputs from a single 256-tuple consecutively, the next processor has to wait till all the 64K points are generated. This will require 64K intermediate buffering.

To avoid this buffering, the first block is scheduled to compute all the 256 elements required by second block in consecutive cycles. In this way only 256×4 buffering is required, 'four times' because four outputs are generated per clock cycle. So in each clock cycle a different input 256-tuple is fed to first block and one element is produced. The `mode select` is held at `00` for 256 consecutive cycles, so that all 256 elements required by second block are produced. Since the first stage produces four outputs, four sets of input 256-tuple required for second block are computed in 256 cycles. The second block takes 64 clock cycles to compute 256 point FFT from a single 256-tuple and can complete computation for all four 256-tuple in 256 cycles. This is done by changing the `mode select` as mentioned in Table 5.3.

After the first 256 cycles, the inputs to first block again comes from k=0 tuple. Then mode select to first block is changed to `01` and the computation is repeated. This is repeated for all `mode select` values from `00` to `3F`. So it takes 64×256 clock cycles to produce the complete 64K points.

The Interstage Processing Stage shown in Figure 5.10 consists of four sets of 256 shift registers into which the first stage outputs are shifted in. After every 256 cycles, the shift register outputs are loaded in parallel to a second set of registers. This will be the inputs for next block as explained in the case of 4K FFT. Here, Twiddle factor ROM size is $16K \times 128$ and address generator produces the 14 bit control signal `addr[13:0]` for TF ROM.

Input Memory Block

Here 64K buffers are needed at the input for reordering before giving to first R4^4 block. The buffers have to be duplicated so that one set takes in the input data while the other one gives the output. This is shown in Fig 5.11. Each set consists of 256 RAMs of 64×128 size. Four complex inputs come in a single clock cycle. These are made into a 128 bit word and written into one location. The write addresses are given sequentially and the first 64 sets of data are written to the first RAM before moving to the next RAM. The read from the RAMs are done in parallel with each RAM giving one 128 bit word per clock cycle. The address to each RAM will be same so that the same index is read from the

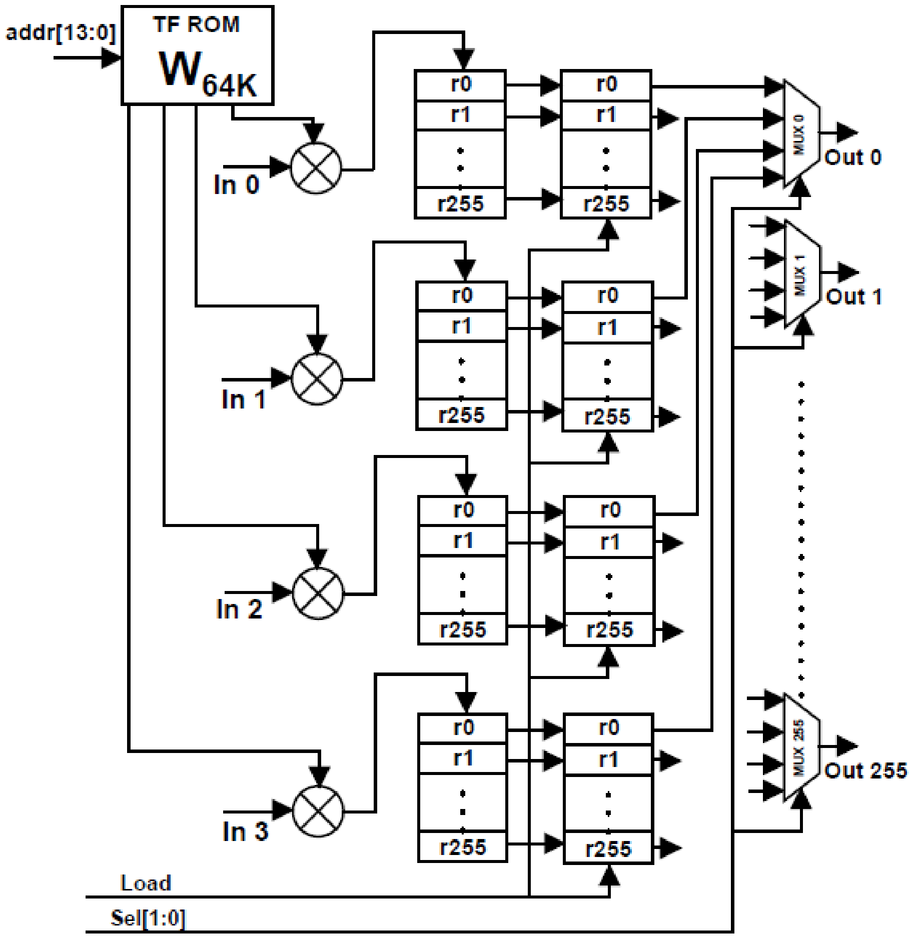


FIGURE 5.10: Interstage Processing Block in 64K FFT

same RAM. This make sure the required address rearrangement for the first R4^4 block.

Resource Utilization

Similar to the case of 4K architecture in Section 5.2.1, the resources are not fully utilized in 64K architecture. Still there is no area overhead. Resources for the first R4^4 block are fully utilized, since the inputs are changing every clock cycle. The second R4^4 block in the 64K FFT processor, takes a different input once in 64 clock cycles. Thus here the 64 R4 engines at first stage will have the same inputs for 64 clock cycles while the mode select changes once in every 16 clock cycles. The inputs to the next 64 multipliers also will change only once in every 16 clock cycles. In the second stage, the input changes once in 4 clock cycles for the set of 16 R4 engines and complex multipliers. Thus the resources at these two stages are significantly underutilized. This is not the case with the first R4^4 block since inputs are changed in every clock.

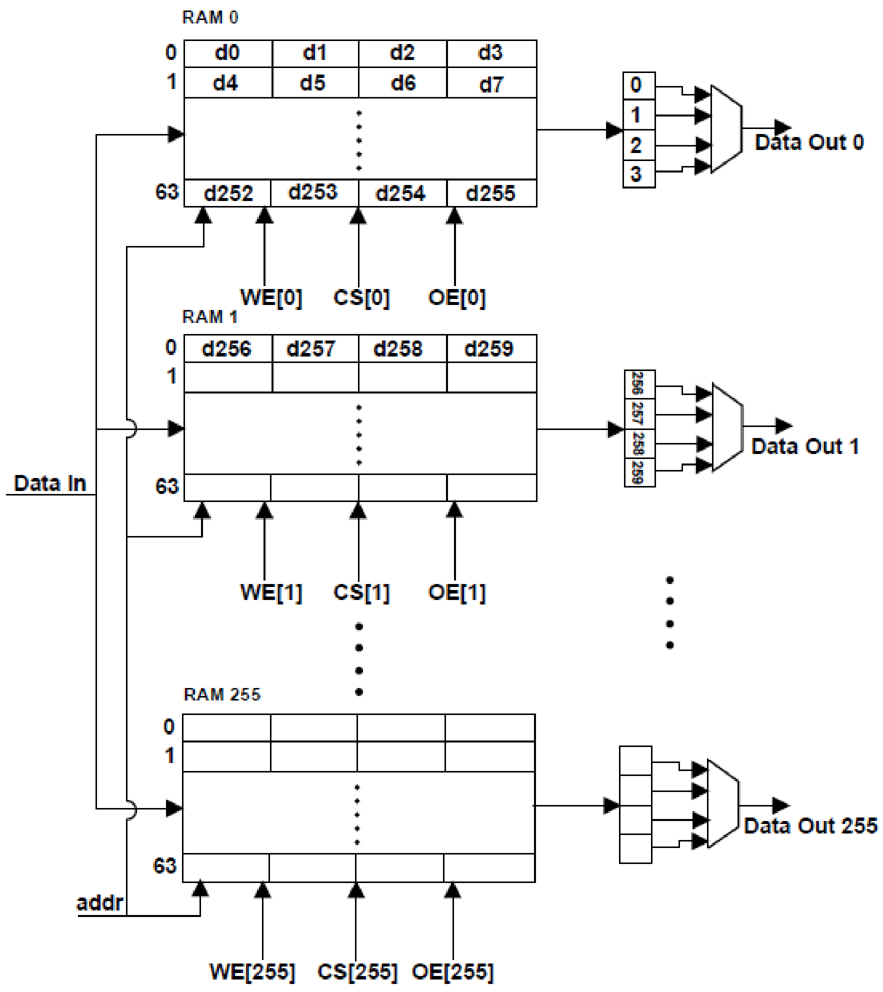


FIGURE 5.11: Input Memory Block in 64K FFT

By changing the architecture of the second R4^4 block with 16 R4 engines in first stage and 4 R4 engines in the second stage, resources can be fully utilized. Since area is not significantly affected, the same architecture is followed in two R4^4 blocks for scalability.

5.5 Complex Multiplier

A complex multiplier consists of four real multipliers and two adders. The real multiplier used in our design is Signed Booth Multiplier [26]. This is implemented as a pipelined multiplier with two stage pipeline. The adders are taken from Synopsys Designware library. The entire Complex multiplier has three clock latency.

Chapter 6

Output Reordering in R4^3 and R4^4 FFT

The output of an in-place DIF FFT will not be in natural order. In case of radix-2 FFT, bit reversal and in case of radix-4 FFT, digit reversal in the last stage has to be performed. Provision has to be made to solve output reordering issue in the last stage. Output reordering problems are discussed in many of the existing literatures [22], [23], [24] which are already mentioned in section 2.3. All the existing implementations require an extra hardware block to perform reordering. Moreover, there are latency issues that has to be taken into account, if reordering is included in the FFT processor itself.
In this chapter, a solution for reordering problem in Radix-4^3 and Radix-4^4 FFT architecture is described. By using the R4^3 and R4^4 FFT architecture explained in chapter 5, output reordering can be obtained by changing the `mode select` signal.

6.1 Reordered 64 point FFT

Signal Flow Graph for a 64 point FFT using Radix-4^3 is shown in Figure 5.3. From Figure 5.3, the flow of mode selection in each stage can be explained. First output can be obtained by setting the select signal to `mode 0` in all stages. Setting `mode 1` to last stage by keeping all other stages in `mode 0` will result in second output and so on.
Reordered output can be obtained by reversing the `mode select` signal. The `mode select`

TABLE 6.1: Mode Selection for Output Reordered 64 point FFT

Stage 1	Stage 2	Stage 3	Output generated
00	00	00	0
01	00	00	1
10	00	00	2
11	00	00	3
: :	: :	: :	: :
01	11	11	61
10	11	11	62
11	11	11	63

bits are reversed from stage 3 to stage 1. The last two stages are kept in `mode 0` and the mode of first stage is changed from `mode 0` to `mode 3` in every clock cycle. Then mode select of second stage is also changed from `mode 0` to `mode 3` keeping stage 3 in `mode 0`. This is repeated to obtain all 64 outputs. Mode selection in different stages for an output reordered R4^3 FFT which computes 64 point FFT is given in Table 6.1.

From Figure 5.4, for a 64 point FFT, four bits are required for mode selection which are represented as `mode_sel[3:0]`. The output obtained will be digit reversed in this case. Here, the last stage uses a '`parallel R4`' engine and thus mode selection is not required in the last stage. In the first stage, two bits `mode_sel[3:2]` are used for mode selection. Second stage uses `mode_sel[1:0]` signal for mode selection.
A slight variation in the architecture in Figure 5.4 is required for performing reordering. The '`parallel R4`' engine in the architecture has to be replaced with a sequential radix-4 engine, so that mode selection operation can be performed in last stage also. Thus six bits are required for mode selection.
Figure 7.2 shows the mode selection signals in different stages for an output reordered 64 point FFT using R4^3 architecture. The flow of six bits of `mode_sel[5:0]` are changed to get the digit reversed output. Thus in first stage, `mode_sel[1:0]` is the select signal and `mode_sel[3:2]` is the mode selection in second stage. The last stage uses `mode_sel[5:4]` as the control signal for mode selection. Figure 7.2 differentiates the `mode select` in a 64 point FFT without and with reordered output.

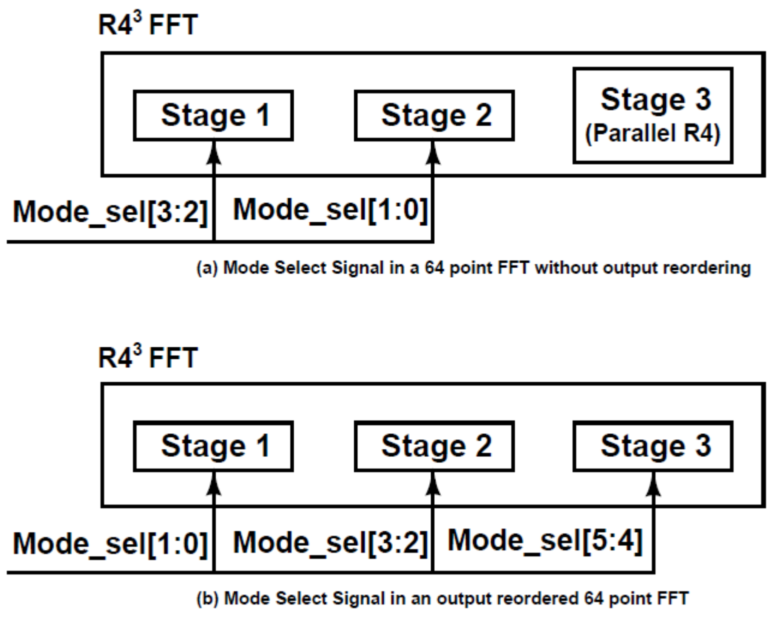


FIGURE 6.1: Mode Selection for 64 Point FFT (a) Without Output Reordering (b) Output Reordered

6.2 Reordered 128 Point FFT Architecture using R4^3 Block

Radix-4^3 architecture presented in section 6.1 is exploited to build a 128 point FFT. This can be done using two R4^3 blocks connected in parallel. Here, we explain the architecture of a reordered 128 point FFT using `mode select` signal. Steps for obtaining a 128 point FFT from 64 point FFT is explained in the following equations.

DFT equation for a 128 point FFT is given by

$$X(k) = \sum_{n=0}^{127} x(n) W_{128}^{nk} \quad k = 0, 1, 2, ...127$$

$$X(k) = \sum_{n=0}^{2.64-1} x(n) W_{2.64}^{nk} \tag{6.1}$$

Decomposing the indices n and k,

$$n = n_1 + 2n_2 \quad where \quad n_1 = 0, 1 \quad n_2 = 0, 1, 2, ...63$$

$$k = 64k_1 + k_2 \quad where \quad k_1 = 0, 1 \quad k_2 = 0, 1, 2, ...63 \tag{6.2}$$

The DFT equation in (6.1) becomes

$$X(k) = \sum_{n_1=0}^{1}\sum_{n_2=0}^{63} x(n_1+2n_2)W_{2.64}^{(64n_1k_1+128n_2k_1+2n_2k_2+n_1k_2)}$$

$$X(k) = \sum_{n_1=0}^{1}\{[\sum_{n_2=0}^{63} x(n_1+2n_2)W_{64}^{n_2k_2}]W_{128}^{n_1k_2}\}W_2^{n_1k_1} \tag{6.3}$$

From equation (6.3) it is observed that, to obtain a 128 point FFT, first perform the 64 point FFT and multiply with twiddle factors corresponding to 128 point FFT. Finally a radix-2 FFT is needed for completing the 128 point FFT. The architecture of 128 point FFT is shown in Figure 6.2.

The R4^3 block takes 64 inputs in parallel from the memory. Here the outputs of R4^3 is

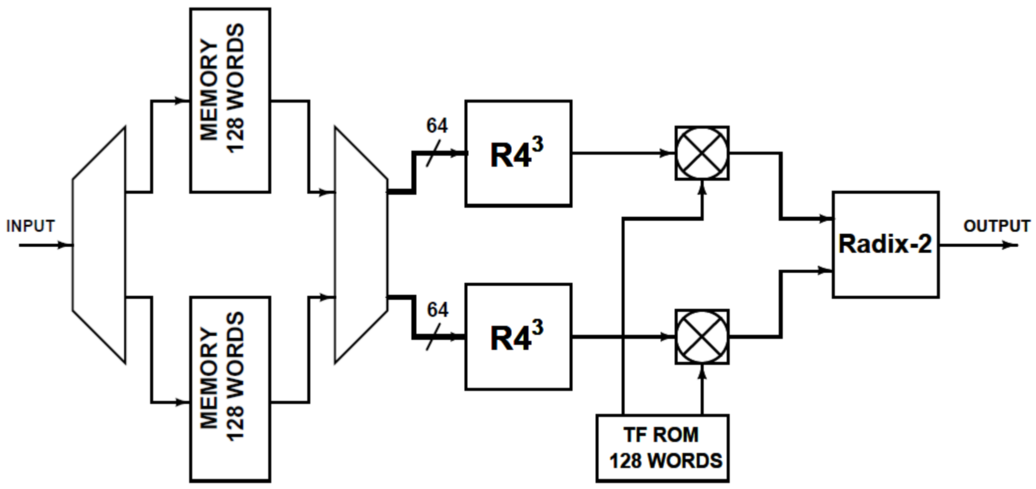


FIGURE 6.2: 128 point FFT Architecture using R4^3 Blocks

sequential and are multiplied with the twiddle factors. A radix-2 butterfly operation is performed in the final stage. The R4^3 FFT used in this architecture produces reordered output. All the stages of R4^3 block has mode select signal. The last stage R4 engine is sequential in both R4^3 blocks.

The 128 point output reordered FFT architecture require seven bits for the mode selection. The control signals for mode selection are shown in Figure 6.3. Six bits are required for mode selection of a R4^3 block and one bit for radix-2 block.

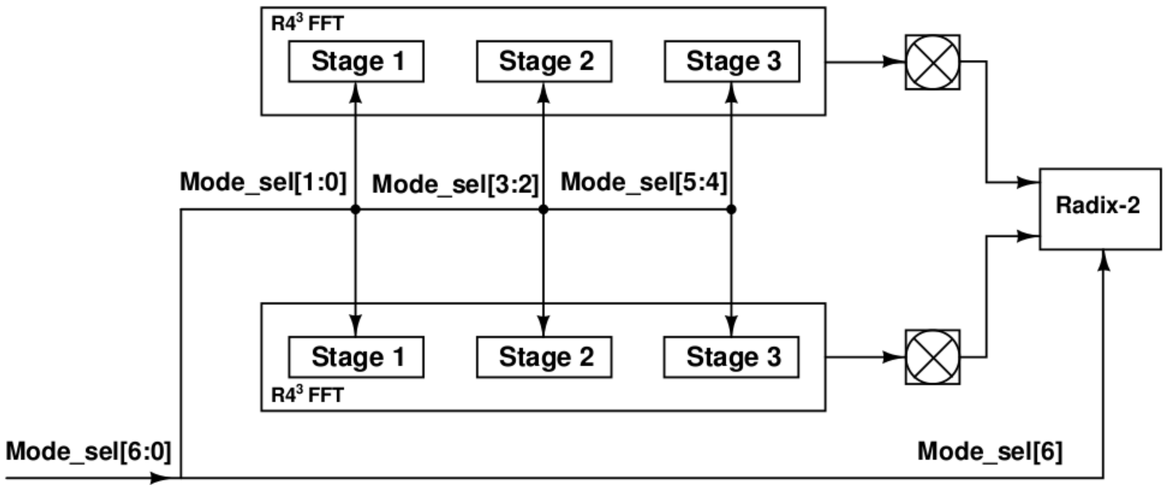


FIGURE 6.3: Mode Selection for an Output Reordered 128 Point FFT using R4^3 Blocks

6.3 Reordered 256 point FFT

Output reordering in the case of R4^4 FFT is similar as that of R4^3 FFT. Here the number of stages for mode selection are four. R4^4 FFT architecture in Figure 5.8 uses a parallel R4 engine in the last stage. For output reordering, this should be replaced with a sequential R4 engine, so that all stages utilizes `mode select` signal. Mode selection is performed using an eight bit signal. Figure 6.4 shows the different stages and corresponding `mode select` signal in a 256 point FFT using R4^4 architecture.

From the Figure 6.4 it is clear that the `mode select` bits are reversed to obtain reordered output.

The architecture proposed here, achieves Output reordering without any additional hardware requirement. Only the control signals need to be changed. The same control circuit which is used in the digit reversed FFT can be used, but the order in which control signal is connected is changed. The increase in latency and memory required which is associated with other output reordering implementations are also avoided here.

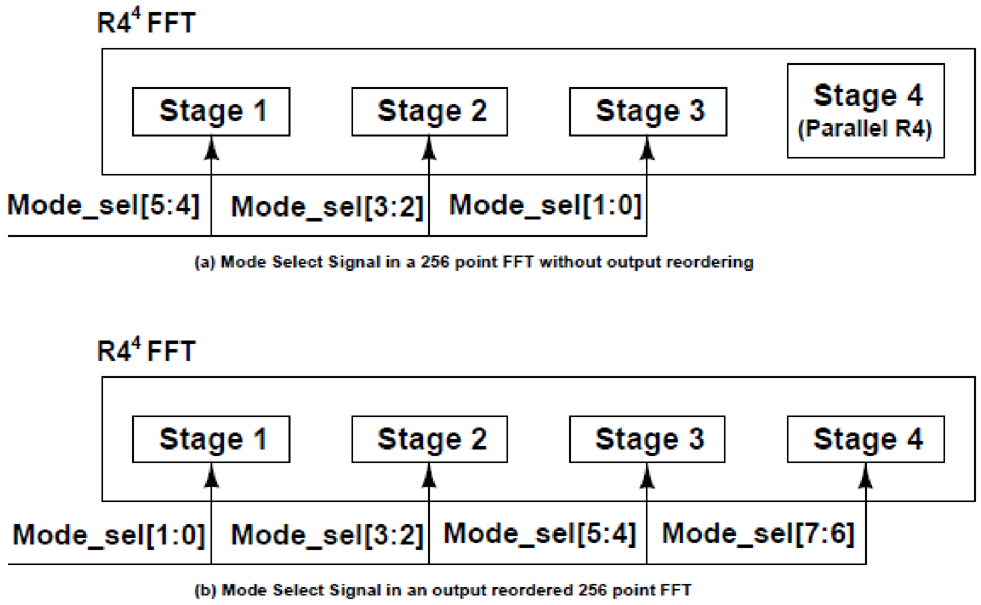


FIGURE 6.4: Mode Selection for 256 Point FFT (a) Without Output Reordering (b) Output Reordered

Chapter 7

Implementation and Results

In this chapter, functional verification of the proposed FFT processor and the results obtained from ASIC implementation of the proposed parallel unrolled architecture used to compute FFTs of 64, 128, 256, 4K and 64K points are described. Comparison of the proposed architecture with existing state-of-art implementations is also done.

7.1 Matlab Simulations

The data representation used in this work is Fixed point format. Selection of word length is a major design issue before implementing the FFT processor in hardware. Appropriate word length has to be chosen to get maximum Signal to Quantization Noise Ratio (SQNR). Selection of bit length varies in several literatures based on the FFT size and SQNR [16] [19] [20] [27]. [28] uses 16 bits for implementing 64 point FFT and in [20] 10 bits are chosen for implementing 128 point FFT. From Fixed point Matlab simulations, we have selected 16 bits as the word length for both real and imaginary values. SQNR changes with FFT size [16] and in this work we are targeting for FFTs of large size, we selected 16 bits as the word length since it gives precision nearer to floating point, eventhough hardware cost increases. Out of 16 bits, 1 bit is for sign and 15 bits for fraction giving a range of -1 to +1. The Twiddle factors are also 16 bit, but here 1 bit is for integer, 1 bit for sign and 14 fractional bits. While doing the matlab simulations we have seen that, at each radix-4 butterfly operation, to avoid overflow, the four inputs has to be scaled down by 0.25.

TABLE 7.1: SQNR of Various FFTs

FFT Size	SQNR in dB	
	Matlab Fixed point Implementation	Verilog Implementation
64	74.84	73.5
256	69.07	68.36
Output Reordered FFTs		
64	75.17	73.45
128	70.09	69.96
256	68.66	68.30

Matlab simulations for 64 point ($R4^3$), 256 point ($R4^4$), 4K and 64K point FFTs without output reordering, and 64,128 and 256 point reordered FFTs are done in 16-bit fixed point format. These are compared with the in-built FFT function in matlab. The SQNR is found to be above 60dB in all the cases.

7.2 Verification

Input data are generated by Matlab. This input is given to both Modelsim and Matlab environments. Matlab simulations for Fixed point implementation of different FFTs explained in Chapter 5 and Chapter 6 are performed. Similarly Floating point simulations using inbuilt FFT function in Matlab is also done. These are compared to check whether the SQNR requirements are met. Since in the implementation the inputs at each butterfly node are scaled down by 0.25, the floating point FFT outputs has to be scaled down by the FFT size before performing the comparison. The test bench structure is shown in Figure 7.1.

Table 7.1 shows the SQNR obtained from the Modelsim outputs and Fixed point Matlab Simulation outputs for various FFTs when compared with Matlab inbuilt function. From literatures [16] [20] [29], minimum SQNR of 40 dB is required for wireless applications which is satisfied by our processor.

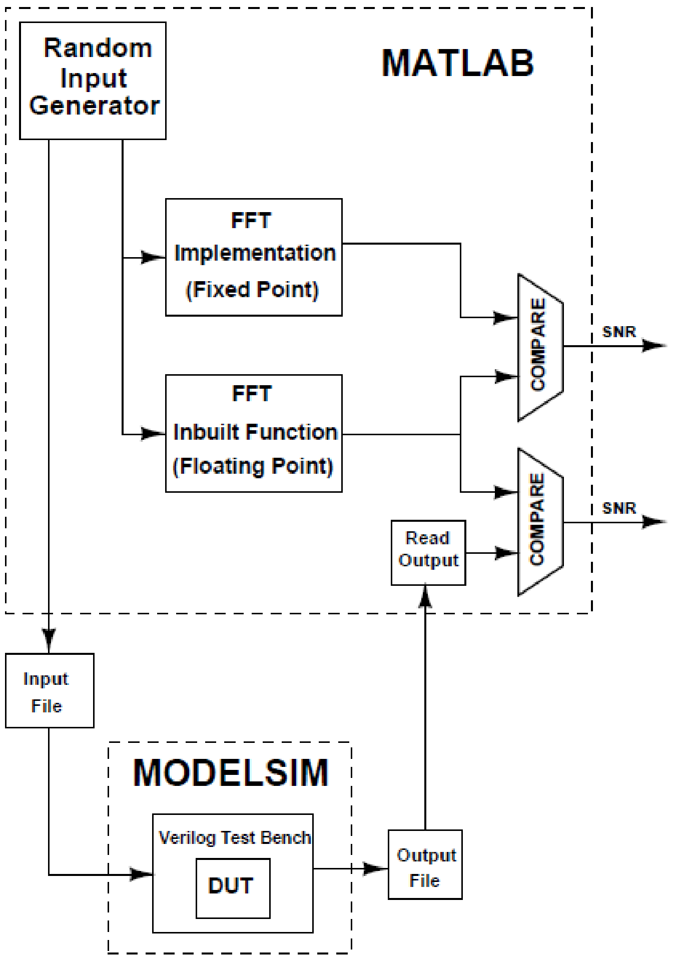


FIGURE 7.1: Test bench Structure

7.3 ASIC targeted Results

The proposed FFT architecture has been implemented in RTL using Verilog and simulated using Modelsim. The RTL has been synthesized with Synopsys Design Compiler using Faraday130 nm standard cell library which is tailored for UMC's 130nm 1.2V/3.3V 1P8M HS Logic Process. Gate density for this library is 250,000 gates/mm^2.

After synthesis, the DUT in test bench structure shown in Figure 7.1 is replaced with the synthesized Verilog netlist and Post Synthesis simulation is done using Modelsim. Modelsim generates the 'activity factor' associated with each net in the netlist into a VCD (Value Change Dump) file. This along with the standard cell library is given as input to Synopsys PrimeTime power simulator to get the power dissipation for the design. Physical Synthesis (Floorplanning, Placement and Routing) of the FFT Processor is done

TABLE 7.2: Synthesis Results for various FFTs

FFT Size	Frequency (MHz)	Throughput (MSPS)	Latency (μs)	Latency (cycles)	Area (mm^2)	Power (Watts)
64	350	1400	0.097	34	2.13	0.175
256	350	1400	0.254	89	9.0	0.680
4K	333	1332	3.38	1127	13.86	0.9174
64K	333	1332	50.1	16689	171.78	10.7
Output Reordered FFTs						
64	350	350	0.237	83	2.12	0.176
128	350	350	0.434	152	4.54	0.372
256	350	350	0.805	282	8.99	0.727

using Cadence SoC Encounter. This has been carried out for 64 and 128 point FFTs. Logic synthesis results for various FFTs are given in Table 7.2.

The layout of 64 point FFT processor is shown in Figure 7.2. Figure 7.3 and Figure 7.4 shows the Floorplan and Routed layout for 128 point reordered FFT.

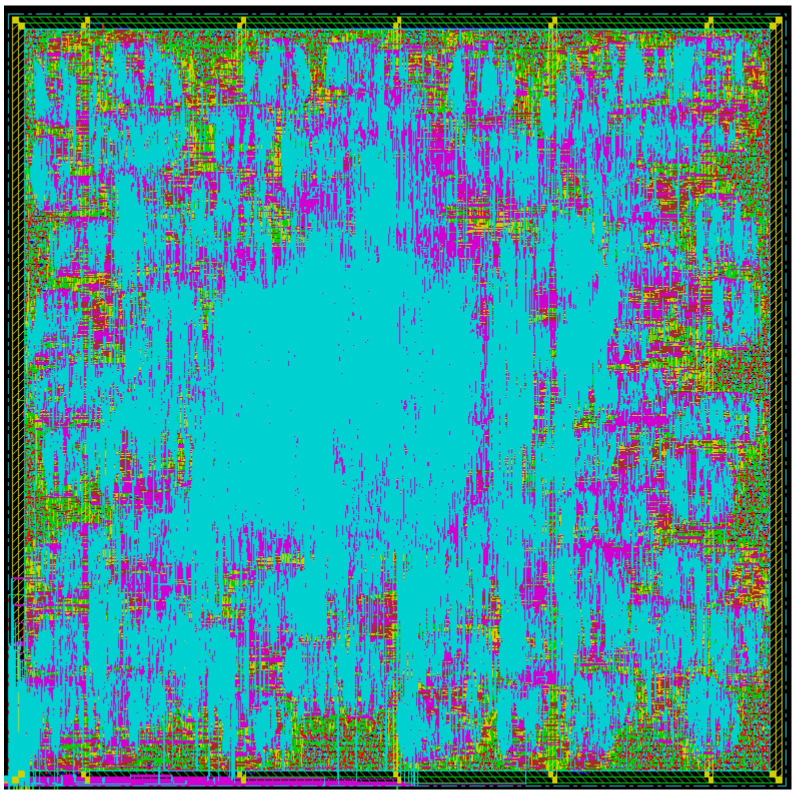


FIGURE 7.2: Layout of 64 Point FFT

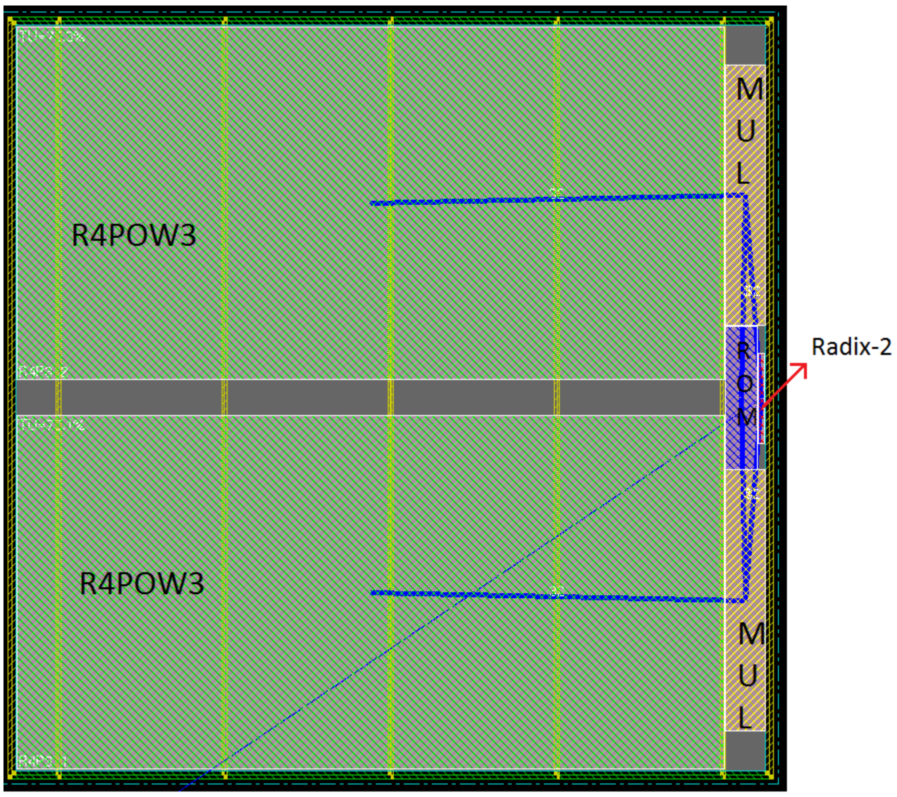


FIGURE 7.3: Floorplan of 128 Point Reordered FFT

7.4 Comparison with Existing Work

The purpose of our architecture is to optimize area and performance for large size FFTs. To achieve this, we have implemented the radix-4^3 and radix-4^4 blocks using an unrolled architecture, which leads to a significant reduction in intermediate memory requirement. But this also increases the hardware complexity in terms of adders and multipliers, compared to existing implementations. This may not be efficient for smaller FFTs like 64, 128 and 256 point FFTs. To provide an idea about how our implementation compare with existing works, we have evaluated for 64, 128, 256, 4K and 64K FFTs which are given in the following sections.
In the comparisons, Normalized Area refers to area which is normalized for 130nm CMOS technology. It is calculated as

$$Normalized\ \ Area = \frac{Area}{(process/0.13)^2}$$

Our implementation takes in four input samples per clock cycle and outputs four samples per clock cycle. So the throughput in Samples Per Second will be 'four times' the clock

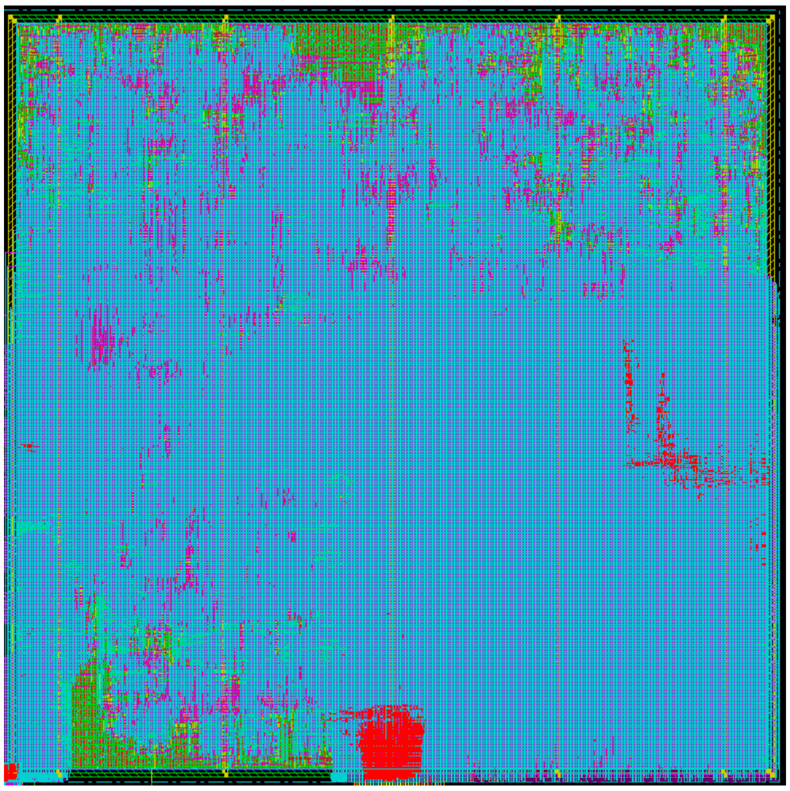


FIGURE 7.4: Layout of 128 Point Reordered FFT

frequency. For the output reordered FFTs, only one output sample is produced per clock cycle. So in this case, the throughput is same as the clock frequency.

Latency Calculation

For calculating latency, we should count the number of clock cycles required for producing the first set of samples after the first set of inputs are presented. In our architecture, this mainly includes two sets of delays. We can start evaluating the FFT only after a full set of inputs are present in the input memory block. In addition to this, the radix-4^3 / radix-4^4 implementations are pipelined and wil give rise to additional delay. Latency calculation for specific FFTs are given in the following sections.

7.4.1 64 Point FFT

Table 7.3 shows the comparison of 64 point FFT with existing architectures. Compared to other implementations, we have the highest throughput and lowest latency while the

TABLE 7.3: Comparison of 64 Point FFTs

	[28]	[30]	[15]	Our Work
Word length (bits)	16	16	14	16
Algorithm	R 2	R 8	R 4^3	R 4^3
Process (μm)	.25	.13	.13	.13
Frequency (MHz)	20	20	604.5	350
Throughput (MSPS)	16.6	20	1052	1400
Latency (μs)	3.2	3.6	0.33	0.091
Area (mm^2)	13.5	1.66	5.09	2.13
Normalized Area (mm^2)	3.65	1.66	5.09	2.13
Power (mW)	41	22.36	78.2	175

TABLE 7.4: Comparison of 256 Point FFTs

	[31]	[32]	Our Work
Word length (bits)	10	9	16
Algorithm	R2^4	R2^4	R4^4
Architecture	MMDF	SDF	Unrolled
Process (nm)	90	180	130
Frequency (MHz)	447	51.5	350
Throughput (MSPS)	2400	51.5	1400
Area (mm^2)	3.53	1.6	9.0
Normalized Area (mm^2)	7.36	0.83	9.0
Power (mW)	145.5	162.7	680

area and power dissipation are inferior. Compared to [15] although we have less area, power dissipation is more since we have more logic while [15] has more memory. The total latency here includes 17 clock cycles for input memory block and 17 clock cycles in the radix-4^3 block which gives a total of 34 clock cycles. The radix-4^3 block latency includes four cycles each in the two R4 engines, three cycles each in the two complex multipliers and three cycles in the parallel R4 engine.

7.4.2 256 Point FFT

Table 7.4 shows the comparison of 256 point FFT with existing architectures.
The input memory block has a latency of 65 clock cycles and the radix-4^4 block has latency of 24 clock cycles.

TABLE 7.5: Comparison of 4K Point FFTs

	[21]	[16]	[15]	Our Work
Word length (bits)	16	16	14	16
Algorithm	R2	R2^2	R4^3	R4^3
Architecture	Split systolic	SDF	Unrolled	Unrolled
Process (μm)	.25	.35	.13	.13
Frequency (MHz)	100	9.1	604.5	333
Throughput (MSPS)	200	18.2	1052	1332
Latency (μs)	-	-	13.8	3.39
Area (mm^2)	-	18.7	13.48	13.86
Normalized Area (mm^2)	-	2.57	13.48	13.86
Power (mW)	2600	114.65	4414.1	917.4

7.4.3 4K Point FFT

Table 7.5 shows the comparison of 4K point FFT with existing architectures.
Latency of input memory block here is 1025 clock cycles, the two the radix-4^3 blocks take 17 clock cycles each and the intermediate processing stage takes 68 clock cycles, which gives a total of 1127 clock cycles. From the Table 7.5, our architecture achieves highest throughput among all implementations. When compared to [15], our implementation achieves 75% reduction in latency with the same area and less power. Our work as well as [15] uses "Flop based memory" for implementation. When compared to "SRAM based memory", this is area and power intensive. There can be further reduction in area and power if "SRAM based memory" is used. This reduction will be more for [15] since that implementation is more memory intensive. We have done this analysis for 64K FFT which is given in the next section.

7.4.4 64K Point FFT

There are very few hardware implementations of 64K FFT in literature. Table 7.6 shows the comparison of 64K point FFT with an existing architecture.

For the RAMs, we used "Flop based asynchronous single port RAM" from Synopsys DesignWare Library. Compared to [15], our architecture gives 50.8% reduction in area and reduces the latency by 301μs as observed from Table 7.6. The increase in clock frequency and throughput is due to two reasons. [15] gives a complete placed and routed

TABLE 7.6: Comparison of 64K Point FFT

	[15]	Our Work
Word length (bits)	14	16
Algorithm	R4^3	R4^4
Architecture	Unrolled	Unrolled
Process (μm)	.13	.13
Frequency (MHz)	256.5	333
Throughput (MSPS)	1000	1332
Latency (μs)	351.1	50.1
Area (mm^2)	348.9	171.78
Power (W)	9.8	10.7

TABLE 7.7: Hardware Complexity of 64K FFTs

	[15]	Our Work
Adders	92	512
Complex Multipliers	40	130
RAM (words)	192K	128K
ROM (words)	128K	64K

TABLE 7.8: Area Analysis of 64K FFTs

Area	Our Work		[15]
	Flop based RAM	SRAM based RAM	Estimated with SRAM based RAM
Total(mm^2)	171.78	56.04	50.14
RAM(mm^2)	121.34	27.65	41.47
Logic+ROM(mm^2)	50.44	28.39	8.67
RAM(%)	70.64	49.34	83.21

implementation while we are giving the synthesis results. Also the adder and multiplier logics used are different. The hardware complexity of our architecture when compared to [15] is given in Table 7.7. We wanted to analyze the logic area and memory area of our design compared with [15]. From Table 7.8, when using flop based RAMs, the RAMs form 70.64% of the total area. Since our design reduces the RAMs required, we are able to get a 50.8% overall area advantage. We wanted to make a comparison when using SRAM based RAMs. Since [15] does not elaborate on the type of multipliers and adders and the control circuitry used, we estimated the area for [15] using our adders and multipliers. We have ignored the control circuitry and the memory inside R4^3 blocks in[15] in this estimation. This is given in Table 7.8. Based on this estimate, our architecture shows an increase in area by 12% while giving the throughput and latency as given in Table 7.6.

TABLE 7.9: Comparison of 128 Point FFTs

	[20]	[33]	[29]	Our Work
Word length (bits)	10	I:10/O:13	16	16
Algorithm	R2-R8	$R2/R2^2$-R4	$R2^3$	$R4^3$
Architecture	MRMDF	Pipeline	SDF	Unrolled
Process (μm)	.18	.13	.18	.13
Frequency (MHz)	250	125	75	350
Throughput (MSPS)	1000	500	75	350
Latency (μs)	-	-	3.44	0.433
Area (mm^2)	3.09	2.16	2.1	4.54
Normalized Area (mm^2)	1.6	2.16	1.09	4.54
Power (mW)	175	-	65	372

7.4.5 128 Point Output Reordered FFT

Since we could not find any existing FFT processors with reordered outputs for evaluating our work, the comparison of 128 point output reordered FFT with existing architectures without reordering is given in Table 7.9.

Output reordered 64 and 256 point FFTs give the same results as given in Table 7.3 and Table 7.4. So they are not mentioned separately.

Chapter 8

Conclusions and Future Work

In this chapter, summary and further directions for the research work in this thesis are presented. A fully parallel unrolled FFT architecture used to compute FFTs of 64, 128, 256, 4K and 64K points is proposed in this work. The ASIC implementation of the FFT processor gives high throughput and low latency with significant reduction in memory.

Summary of the Thesis

- From the literature survey, challenges in developing higher order FFTs are defined. Hardware complexity of different FFT algorithms and architectures are compared.
- Memory optimization techniques for Radix-2^2 FFT are suggested. ASIC Implementation of Memory optimized FFT architecture shows that this technique is more effective for FFTs of large size.
- A fully parallel unrolled FFT architecture is proposed, which is based on a novel parallel radix-4 engine. The proposed R4^3 and R4^4 architecture gives a significant improvement in throughput and latency by reducing intermediate buffering, without degrading the performance or area.
- Our architecture can support output reordering without any additional hardware cost.
- The proposed FFT solution is verified using Matlab and Modelsim. ASIC synthesis results for 64, 128, 256, 4K and 64K point FFTs are presented.

Future Work

- Extending this architecture to realize 256K point FFT to evaluate and validate the optimization in memory and latency. Cascading three Radix-4^3 units can result in a 256K point FFT.
- The proposed Radix-4 engine can be used as a building block for Programmable FFTs of various sizes. This requires further studies on existing programmable FFT architectures.
- Integrating the FFT block with the Polyphase Channelizer require deeper analysis and research in each stage of the Channelizer.

Bibliography

[1] Thomas M. Hopkinson and G. Michael Butler. A Pipelined, High-Precision FFT Architecture. In *Proc. 35th Midwest Symposium on Circuits and Systems*, October 1992.

[2] Andrea Goldsmith. *Wireless Communications*, Cambridge University Press 2005.

[3] Sergio Saponara, Massimo Rovini, Luca Fanucci, Athanasios Karachalios, George Lentaris and Dionysios Reisis. Design and Comparison of FFT VLSI Architectures for SoC Telecom Applications with Different Flexibility, Speed and Complexity Trade-Offs. *Circuits Syst Signal Process (2012)*, 31:627-649.

[4] Mehmood Awan and Muhammad Mahtab Alam. Design and Implementation of FPGA-based Multi-standard Software Radio Receiver. In *Proc. IEEE Norchip Conference on Microelectronic*, Nov. 2007

[5] Fredric J. Harris, C. D. and Rice, M. Digital receivers and transmitters using polyphase filter banks for wireless communications. *IEEE Transactions on Microwave Theory and Techniques (2003)*, 51(4).

[6] J.W.Cooley and J.W.Tukey. An algorithm for machine computation of complex fourier series. *Math Comput (1965)* vol 9, pp.297-301.

[7] John G Proakis and Dimitris G Manolakis. *Digital Signal Processing Principles, Algorithms and Applications*, Prentice-Hall 1996.

[8] Ramesh Chidambaram. A Scalable and High-performance FFT processor, Optimized for UWB-OFDM. *M.S. thesis, Delft University of Technology*, 2005.

[9] Shousheng He and Torkelson, M. A new approach to pipeline FFT processor. In *Proc. The 10th International Parallel Processing Symposium, IPPS*, pp.766-770, 15-19 Apr 1996.

[10] Shousheng He and Torkelson, M. Designing Pipeline FFT Processor for OFDM (de)Modulation. In *Proc. Signals, Systems and Electronics*, 1998.

[11] Jeesung Lee, Hanho Lee, Sang-in Cho and Sang-Sung Choi. A High-Speed, Low-Complexity Radix-2^4 FFT Processorfor MB-OFDM UWB Systems. In *Proc. ISCAS*, 2006.

[12] Taesang Cho, Hanho Lee, Jounsup Park and Chulgyun Park. A High-Speed Low-Complexity Modified Radix-2^5 FFT Processor for Gigabit WPAN Applications. In *Proc. ISCAS* , 2011.

[13] H.-L. Lin, H. Lin, R. C. Chang, S.-W. Chen, C.-Y. Liao, C.-H.Wu. A High-Speed Highly Pipelined 2^n-point FFT Architecture for Dual OFDM Processor. In *Proc. Intl. Conf. Mixed Design, MIXDES*, 2006.

[14] Chin-Teng Lin and Yuan-Chu Yu. Cost-Effective Pipeline FFT/IFFT VLSI Architecture for DVB-H System. In *Proc. National Symposium on Telecommunications*, October 2007.

[15] K. Babionitakis, V. A. Chouliaras, K. Manolopoulos, K. Nakos, D. Reisis and N. Vlassopoulos. Fully Systolic FFT Architecture for Giga-sample Applications. *Journal Signal Process Syst (2010)*, 58:281-299.

[16] Cortes, A., Velez, I., Zalbide, I., Irizar, A. and Sevillano,J. F. An FFT core for DVB-T/DVB-H receivers. In *Proc. ICECS (2006)* pp. 102- 105.

[17] K. Manolopoulos, K. Nakos, D.Reisis and N. Vlassopoulos and V. A. Chouliaras. High Performance 16K, 64K, 256k VLSI Systolic FFT Architectures. In *Proc. IEEE Intl Conf on Electronics, Circuits and Systems* December 2007. pp. 146-149.

[18] Bernard Gold and Lawrence R.Rabiner. *Theory And Application of Digital Signal Processing* Prentice Hall 1975.

[19] Nuo Li. ASIC FFT Processor for MB-OFDM UWB System. *MSc. thesis, Delft University of Technology*, 2008.

[20] Lin, Y. N., Liu, H. Y., Lee, C. Y. A 1-GS/s FFT/IFFT processor for UWB applications. *IEEE Journal of SSC, 40(8), 2005.*

[21] Swartzlander, E. E. Jr. Systolic FFT processors: A personal perspective. *Journal of VLSI Signal Processing, 2007*, 53, 314.

[22] Tuhin Subhra Chakraborty and Saswat Chakrabarti. On Output Reorder Buffer Design of Bit Reversed Pipelined Continuous Data FFT Architecture. In *Proc. IEEE Asia pacific Conf. on Circuits and systems, APCCAS* December 2008. pp. 1132-1135.

[23] Jong-Yeol Lee and Seung-Won Yang. Area and Power Efficient Signal Reordering unit for OFDM Systems. *IEICE Electronics Express (2008)* Vol.5, No.24, pp. 1049-1053.

[24] Fredrik Kristensen, Peter Nilsson and Anders Olsson. Reduced transceiver-delay for OFDM systems. In *Proc. IEEE 59th Vehicular Technology Conference, VTC-2004 Spring.* vol.3, pp. 1242-1245.

[25] Huirae Cho, Myung-Soon Kim, Duk-Bai Kim and Jin-Up Kim. R2^2 SDF FFT Implementation with Coefficient Memory Reduction Scheme. In *Proc. IEEE 64th Vehicular Technology Conference, VTC-2006 Fall*, vol., no., pp.1-4, 25-28 Sept. 2006

[26] Neil Weste and David Harris *CMOS VLSI Design; A Circuits and Systems Perspective (4th Edition)*, Addison-Wesley 2011.

[27] Yu-Wei Lin, Hsuan-Yu Liu and Chen-Yi Lee. A Dynamic Scaling FFT Processor for DVB-T Applications. *IEEE Journal of Solid State Circuits* Vol. 39, No. 11, Nov 2004.

[28] Koushik Maharatna, Eckhard Grass and Ulrich Jagdhold. A 64-Point Fourier Transform Chip for High-Speed Wireless LAN Application Using OFDM. *IEEE Journal of Solid State Circuits* Vol. 39, No. 3, March 2004.

[29] Bo Fu and Paul Ampadu. An Area Efficient FFT/IFFT Processor for MIMO-OFDM WLAN 802.11n. *Journal Signal Process Syst (2009)*, 56:59 - 68.

[30] Lin, C. T., Yu, Y. C., and Van, L. D. A low power 64-point FFT/IFFT design for IEEE 802.11a WLAN application. In *Proc. of ISCAS 2006*, pp. 4523 - 4526.

[31] Y. Chen, Y.-W. Lin, Y.-C. Tsao and C.-Y. Lee, A 2.4-Gsample/s DVFS FFT processor for MIMO OFDM communication systems. In *IEEE Journal of Solid State Circuits*, Vol. 43, no. 5, pp. 1260 - 1273, May 2008.

[32] Chih-Peng Fan, Mau-Shih Lee and Guo-An Su. Efficient Low Multiplier Cost 256-Point FFT Design With Radix-2^4 SDF Architecture. In *Proc. IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2006*, pp. 1935 - 1938.

[33] Simeng Li, Huxiong Xu, Wenhua Fan, Yun Chen, Xiaoyang Zeng. A 128/256-Point Pipeline FFT/IFFT Processor for MIMO OFDM System IEEE 802.16e. In *Proc. ISCAS, 2010*, pp. 1488- 1491.

[34] Chu Yu, Mao-Hsu Yen, Pao-Ann Hsiung and Sao-Jie Chen. A Low-Power 64-point Pipeline FFT/IFFT Processor for OFDM Applications. *IEEE Transactions on Consumer Electronics*, Vol. 57, no. 1, pp. 40 - 45, February 2011.

[35] Dionysios Reisis and Nikolaos Vlassopoulos. Conflict-Free Parallel Memory Accessing Techniques for FFT Architectures. *IEEE Transactions on Circuits and Systems-I*, Vol. 55, No. 11, pp.3438 - 3447, December 2008

[36] Song-Nien Tang, Jui-Wei Tsai, and Tsin-Yuan Chang. A 2.4-GS/s FFT Processor for OFDM-Based WPAN Applications. *IEEE Transactions on Circuits and Systems-II*, Vol. 57, No. 6, pp. 451 - 455, June 2010.

[37] Huggett, C., Maharatna, K. and Paul, K. On the implementation of 128-pt FFT/IFFT for high-performance WPAN. In *Proc. ISCAS, 2005*, pp. 5513 - 5516.

[38] Chin-Teng Lin, Yuan-Chu Yu, and Lan-Da Van. A Low-Power 64-Point FFT/IFFT Design for IEEE 802.11a WLAN Application. In *Proc. ISCAS, 2006*, pp. 4523 - 4526.

[39] Y.-N. Chang and K. K. Parhi. An efficient pipelined FFT architecture. *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, Vol. 50, No. 6, pp. 322 - 325, June 2003.

[40] M. Hasan, T. Arslan and J.S. Thompson. A Novel Coefficient Ordering based Low Power Pipelined Radix-4 FFT Processor for Wireless LAN Applications. *IEEE Transactions on Consumer Electronics*, Vol. 49, No. 1, pp. 128 - 134, February 2003.

[41] K.K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation.* New York, John Wiley and Sons, 1999.

[42] Daisuke Takahashi. An Extended Split-Radix FFT Algorithm. *IEEE Signal Processing Letters*, Vol. 8, No. 5, May 2001.

[43] Saad Bouguezel, M. Omair Ahmad and M. N. S. Swamy. A New Radix-2/8 FFT Algorithm for Length-q x 2^m DFTs. *IEEE Transactions on Circuits and Systems-I*, Vol. 51, No. 9, pp. 1723 - 1733, September 2004.

[44] Chao Cheng and Keshab K. Parhi. High-Throughput VLSI Architecture for FFT Computation. *IEEE Transactions on Circuits and Systems-II: Express Briefs*, Vol. 54, No. 10, pp. 863 - 868, October 2007.

[45] Encounter^{TM} User Guide Product Version 7.1 October 2007.

[46] Wang Chao and HAN Yun Peng. Twin Butterfly High Throughput Parallel Architecture FFT Algorithm. In *Proc. Intl. Symposium on Information Science and Engineering*, 2008.

[47] Ayman M. El-Khashab and Earl E. Swartzlander, Jr. An Architecture for a Radix-4 Modular Pipeline Fast Fourier Transform. In *Proc. Application-Specific Systems, Architectures, and Processors (ASAP)*, 2003.

[48] A.V. Oppenheim and R.W. Schafer. *Discrete - Time Signal Processing, 2nd ed.* New Jersey, Prentice-Hall, 1999.

[49] L. Wanhammar. *DSP Integrated Circuits*, Academic Press, 1999.

[50] S. He and M. Torkelson. Design and Implementation of a 1024-point pipeline FFT processor. In *Proc. IEEE Custom Integrated Circuits. Conf.* pp. 131 -134, May 1998.

[51] Jimson Mathew, K Maharatna and D K Pradhan. Pseudo-parallel Datapath Structure for Power Optimal Implementation of 128-pt FFT/IFFT for Wireless Personal Area Networks. *Journal Circuits, Syst and Signal Process (2011)*, 30: 871- 882

[52] Nuo Li and van der Meijs, N.P. A Radix2^2 based parallel Pipeline FFT Processor for MB-OFDM UWB System. In *Proc. SOC Conference, 2009 (SOCC). IEEE International*, pp. 383 - 386, September 2009.

[53] Pandey, R. and Bushnell, M.L. Architecture for Variable- Length Combined FFT, DCT, and MWT Transform Hardware for a Multi- Mode Wireless System. In *Proc. 20th Intl. Conf. on VLSI Design* pp.121-126, January 2007.