UNLV Retrospective Theses & Dissertations

1-1-2005

Design and implementation of a fast Fourier transform architecture using twiddle factor based decomposition algorithm

Bhaarath Kumar
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation

Kumar, Bhaarath, "Design and implementation of a fast Fourier transform architecture using twiddle factor based decomposition algorithm" (2005). *UNLV Retrospective Theses & Dissertations*. 1837.
http://dx.doi.org/10.25669/uqf7-sxec

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

DESIGN AND IMPLEMENTATION OF A FAST FOURIER TRANSFORM ARCHITECTURE USING TWIDDLE FACTOR BASED DECOMPOSITION ALGORITHM

by

Bhaarath Kumar

Bachelor of Engineering
University of Madras, India
2002

A thesis submitted in partial fulfillment
of the requirement for the

Master of Science Degree in Electrical Engineering
Department of Electrical and Computer Engineering
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2005

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 1429711

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 1429711

Copyright 2006 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval
The Graduate College
University of Nevada, Las Vegas

July 12, 2005

The Thesis prepared by

Bhaarath Kumar

Entitled

"Design and Implementation of a Fast Fourier Transform Architecture Using Twiddle Factor Based Decomposition Algorithm"

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

Design and Implementation of a Fast Fourier Transform Architecture using Twiddle Factor Based decomposition Algorithm

by

Bhaarath Kumar

Dr. Yingtao Jiang, Examination Committee Chair
Assistant Professor
Department of Electrical & Computer Engineering
University of Nevada, Las Vegas

With the advent of signal processing and wireless communication mobile platform devices, the necessity for data transformation from one form to another becomes an unavoidable aspect. One such mathematical tool that is widely used for transforming time and frequency domain signals is Fourier Transform. Fast Fourier Transform (FFT) is perhaps the fastest way to achieve transformation. Many algorithms and architectures have been designed over the years in an attempt to make FFT algorithms more efficient and to target many applications.

The main objective of our work is to design, simulate and implement an architecture based on the Twiddle-Factor-Based decomposition FFT algorithm. The significant feature of the algorithm is its effective memory access reduction that accounts to be as much as 30% lesser than in any other conventional FFT algorithms. As a result of this memory reduction, this algorithm is said to be more power efficient and is said to compute in much lesser number of clock cycles than other algorithms developed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

Figure 1.1 Data Flow graph of an 8-point DFT calculated by splitting the *N* point input into two *N/2* parts containing even and odd components and using *N* twiddle factors..9
Figure 1.2 Data Flow graph of an 8-point DFT calculated by splitting the *N* point input into two *N/2* parts containing even and odd components and using only *N/2* twiddle factors..10
Figure 1.3 Data Flow graph of an 8-point FFT, showing the entire decomposition until 2-point DFT stage is reached. The stage with W_2^0 is the last stage of decomposition with all butterflies being 2-point DFTs............... 11
Figure 2.1 A radix-2 Decimation-In-Time (DIT) butterfly structure. 18
Figure 2.2 Flow graph of an 8-point DIT FFT structure using butterflies................. 18
Figure 2.3 A radix-2 Decimation-In-Frequency (DIF) butterfly structure 20
Figure 2.4 Flow graph of an 8-point DIF FFT structure using butterflies................. 20
Figure 2.5 Flow graph of a 16-point FFT structure based on Twiddle-Factor-Based algorithm 23
Figure 3.1 Flow graph of a 16-point FFT structure based on Twiddle-Factor-Based algorithm 27
Figure 3.2 Pseudo code for the *(log_2N-1)* stages decomposition.............................. 34
Figure 3.3 Block design representation of *(log_2N-1)* stages based on Twiddle Factor decomposition algorithm 36
Figure 3.4 *(log_2N)th* stage decomposition structure for *N* = 16.. 39
Figure 3.5 Pseudo code for the *(log_2N-1)* stages decomposition 40
Figure 3.6 VHDL Block implementation of *(log_2N)th* stage based on Twiddle factor decomposition algorithm 42
Figure 3.7 Block implementation of *(log_2N-1) stages and (log_2N)th* stage based on Twiddle factor decomposition algorithm.. 43
Figure 3.8 Block level representation of 1-word RAM cell for storing *32*-bit data ... 45
Figure 3.9 Block representation of a RAM decoder... 46
Figure 3.10a Basic group formation of RAM cells using decoders.............................. 47
Figure 3.10b Hierarchical group formation of RAM cells using decoders.................... 47
Figure 3.11 Block representation of Read Only Memory (ROM) and its controller.... 50
Figure 3.12 Complex multiplication implementation using method 1 53
Figure 3.13 Complex multiplication implementation using method 2. 54
Figure 3.14 Real multiplication implementation stages. 55
Figure 3.15 Complex multiplication implementation using method 2......................55
Figure 3.16 Block level Data-path for group1 for twiddle factor based architecture... 57
Figure 3.17 Block level Data-path for group2 for twiddle factor based architecture... 58
Figure 3.18 Block level Data-path for group3 for twiddle factor based architecture....59

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The real focus of the design is to build architecture to map this efficient algorithm on to hardware retaining the maximum efficiency of the algorithm. The complete design, simulation and testing is done using Active-HDL tool which is a VHDL package designed. The architecture designed is found to retain the memory savings capability of the algorithm thus enabling power efficiency.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 4.1 Graphical comparison for ROM access between twiddle factor based architecture and conventional DIT or DIF based architectures ... 69
Figure 4.2 Graphical comparison for RAM access between twiddle factor based architecture and conventional DIT or DIF based architectures ... 71
Figure 4.3 Graphical comparison for multiplier operation usage between twiddle Factor based architecture and conventional DIT or DIF based architectures ... 72
Figure A-I1 VHDL block representation of *(log_2N-1)* stages RAM Address block based on Twiddle factor decomposition algorithm generating *10*-bit RAM address sequences qa and qb ... 80
Figure A-I2 VHDL block representation of $(log_2N)^{th}$ stage RAM Address Generation block based on Twiddle factor decomposition algorithm generating *10*-bit RAM address sequences qa_logn2 and qb_logn and *6*-bit Temporary register bank address qaddr ... 81
Figure A-I3 VHDL design of a *1*-word RAM cell for storing imaginary part of data ... 82
Figure A-I4 VHDL design of a *1*-word RAM cell for storing real part of data ... 82
Figure A-I5 VHDL design of a 2x4 Decoder block choosing one among four outputs based on a *2*-bit input. The input port is given as *de_inp_2x4_real(1:0)* and output ports are represented by e*_out_2x4_real(3:0)* ... 83
Figure A-I6 VHDL design of a *64*-words ROM and a decoder along with interconnects outputting *32*-bits real and imaginary twiddle factors ... 83
Figure A-I7 VHDL design of a *64*-words RAM using four *16*-words RAM cells And a decoder along with interconnects outputting *32*-bits real and imaginary data ... 84
Figure A-I8 VHDL design block showing the data-paths for group1, group2 and group3. The output signals for each group are represented by *b_real_out_groupX* and *b_imaginary_out_groupX* with *X* representing individual group numbers ... 85
Figure A-I9 VHDL design block showing the resolution function designed for the data-path to choose between the three data-path groups and also for determining the *Read* and *Write* operations of the RAM block and other resolution blocks ... 86
Figure A-I10 VHDL design block showing RAM Address generation block with resolution functions that aid in enabling and disabling various signals at different instances of time ... 87
Figure A-I11 VHDL design block showing RAM Address generation block, the RAM blocks, the ROM memory controller block, the ROM blocks and all the other interconnects ... 88
Figure A-I12 VHDL design block partially showing interconnection between all the blocks of the Twiddle factor based FFT architecture ... 89

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

I take this opportunity to thank my advisor Dr. Yingtao Jiang, for his guidance and support.

I am grateful to professors in my thesis committee for all their valuable suggestions. I would like to thank Mr. Stan Hanel for providing access to Active-HDL whenever I needed it most.

I wish to thank all my friends and family members.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF FIGURES v

ACKNOWLEDGEMENTS vii

CHAPTER 1 INTRODUCTION I
- 1.1 Thesis Objective 1
- 1.2 Fourier Analysis 2
  - 1.2.1 The Fourier Series 2
  - 1.2.2 Fourier Transform 4
  - 1.2.3 Discrete Fourier Transform 5
  - 1.2.4 Fast Fourier Transform 7
    - 1.2.4.1 Mathematical Calculation for FFT Computational Complexity 8
  - 1.2.5 Power Aware Design 12
- 1.3 Organization of the Thesis Write-up 13

CHAPTER 2 FAST FOURIER ALGORITHMS AND ARCHITECTURE 15
- 2.1 Introduction 15
- 2.2 Decimation-In-Time (DIT) FFT Algorithm 17
- 2.3 Decimation-In-Frequency (DIF) FFT Algorithm 18
- 2.4 Decimation Based on Twiddle Factors 20
  - 2.4.1 Decimation Procedure 22

CHAPTER 3 ARCHITECTURAL DESIGN AND IMPLEMENTATION 25
- 3.1 Overview 25
- 3.2 Design Target 26
- 3.3 Algorithm Setup 26
- 3.4 Random Access Memory (RAM) Address Generation Block Design 28
  - 3.4.1 Formulae of Computational Complexity for the Loop Structures for (log_2N-1) Stages of Decomposition 30
  - 3.4.2 Pseudo Code Implementation 33
  - 3.4.3 Architectural Block Design and Implementation 34
- 3.5 Formulae of Computational Complexity for the Loop Structures for $(log_2N)^{th}$ Stage of Decomposition 37
  - 3.5.1 Memory Reduction Technique 38

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.5.2 Pseudo Code Implementation 40
3.5.3 Architectural Block Design and Implementation 40
3.6 Random Access Memory Design 44
3.6.1 Read Only Memory (ROM) design 48
3.7 Data-Path Design 50
3.7.1 Complex Multiplier Implementation 52
3.7.2 Group1 Design and Implementation 56
3.7.3 Group2 Design and Implementation 56
3.7.4 Group3 Design and Implementation 58
3.8 Summary 59

CHAPTER 4 ARCHITECTURAL SIMULATION RESULTS AND DISCUSSION 61
4.1 Output Simulation Results 61
4.2 RAM Address Generation Block Parameters 61
4.3 ROM Address Generation Block Parameters 63
4.4 Random Access Memory Parameters 63
4.5 Read Only Memory Block Parameters 64
4.6 Data-Path Block Parameters 64
4.7 Clock Generation and Frequency Calculations 65
4.8 Memory Access Feature 67
4.9 Multiplier Operational Savings Using Twiddle Factor Based Architecture 71
4.10 Design Issues and Challenges 73
4.11 Architectural Salient Features and Drawbacks 76

CHAPTER 5 CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK 77
5.1 Summary of Work 77
5.2 Suggestions for the Future 77

APPENDIX VHDL Design of Various Architectural Blocks 80
BIBLIOGRAPHY 90
VITA 93

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 1

INTRODUCTION

1.1 Thesis Objective

With increasing demand for mobile computing devices, conversion of data between the time and frequency domain has become vital. The Fast Fourier Transform (FFT) is one of the widely used digital signal processing algorithms for this purpose. Numerous Fast Fourier Transform algorithms have been developed over the years. Architectures, which help realize these algorithms, have found applications in diverse areas as: communications, signal processing, instrumentation, biomedical engineering, Sonics and acoustics to name a few. The goal of our work is the architectural design and implementation of a Fast Fourier Transform (FFT) processor, mapping an algorithm whose decomposition is uniquely based on Twiddle factors unlike the conventional Decimation-In-Time (DIT) or Decimation-In-Frequency (DIF). With numerous architectures already in existence, one main aspect that differentiates this work from other architectures is the algorithm that is used for the mapping purpose. The Twiddle Factor based decomposition algorithm ensures a reduction in memory access by as much as 30% [1] [10]. Hence, an architecture utilizing this algorithm is expected to have lower power consumption than any other conventional algorithm based architecture. The memory reduction comes in the form of twiddle factor access from the Read Only

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Memory (ROM), where it is generally stored and retrieved. The architecture is designed, simulated and tested in Very large-scale integrated circuits Hardware Description Language (VHDL) environment.

1.2 The Fourier Analysis

1.2.1 The Fourier Series

Consider a sequence $\tilde{x}(n)$ that is periodic with a period N so that $\tilde{x}(n) = \tilde{x}(n+kN)$ for any integer value of *k*. Such a sequence cannot be represented by its *z*-transform, since there is no value of *z* for which the *z*-transform will converge. In such situations, this sequence can be expressed using the Fourier Series (FS) tool [4] [13] [14] [15] [16]. Any periodic signal can be expressed as a sum of sinusoidal and co-sinusoidal oscillations. This decomposition is termed as Fourier Series (FS). By reversing this procedure, a periodic signal can be generated by superimposing sinusoidal and co-sinusoidal waves. Fourier series make use of the orthogonal relationships of the sine and cosine functions. The Fourier series is extremely useful in breaking down an arbitrary periodic function into a set of simple terms that can be plugged in, solved individually and then recombined to obtain the solution of the original problem. The general function is as follows:

$$f(x) = a_0 + \sum_{n=1}^{\infty}\left(a_n \cos\frac{n\pi x}{L} + b_n \sin\frac{n\pi x}{L}\right) \tag{1.2.1.1}$$

where a_0, a_n, b_n are the Fourier magnitude coefficients of the corresponding sinusoidal and co-sinusoidal waves and $2L$ represents the fundamental frequency given by $2\pi/T$ rad/sec. The Fourier Coefficients can be determined from the following integrals:

$$a_0 = \frac{1}{2L}\int_{-L}^{L} f(x)dx \tag{1.2.1.2}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$a_n = \frac{1}{L}\int_{-L}^{L} f(x)\cos\frac{n\pi x}{L}dx \qquad n = 1,2,\ldots \tag{1.2.1.3}$$

$$b_n = \frac{1}{L}\int_{-L}^{L} f(x)\sin\frac{n\pi x}{L}dx \qquad n = 1,2,\ldots \tag{1.2.1.4}$$

For non-periodic functions, one can argue that they are periodic with an infinite period, which is $L \rightarrow \infty$. The Fourier series then becomes Fourier Integral given as follows:

$$f(x) = \int_{0}^{\infty}[a(\omega)\cos\omega x + b(\omega)\sin\omega x]d\omega \tag{1.2.1.5}$$

where,

$$a(\omega) = \frac{1}{\pi}\int_{-\infty}^{\infty} f(x)\cos\omega x dx \tag{1.2.1.6}$$

$$b(\omega) = \frac{1}{\pi}\int_{-\infty}^{\infty} f(x)\sin\omega x dx \tag{1.2.1.7}$$

Thus, Fourier series are made up of sinusoids, all of which have frequencies that are integer multiples of some fundamental frequency. A great thing about using Fourier series on periodic function is that the first few terms often are a pretty good approximation to the whole function, not just the region around a special point. Fourier series are used extensively in many major engineering applications, especially for image processing and signal processing applications. They are also used in solving ordinary and partial differential equations (heat conduction, wave theory) and also for various kinds of spectroscopy. Finding the coefficients of a Fourier series is similar to performing the spectral analysis of that function.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.2.2 Fourier Transform

The Fourier Transform (FT) is basically generalization of the Fourier series. The Fourier Transform provides the means of transforming a continuous time signal into its corresponding frequency domain. Instead of sinusoidal and co-sinusoidal terms used in Fourier series, Fourier Transform uses exponentials and complex numbers. The Fourier transform X(*f*) of a continuous time function *x(t)* can be expressed as follows:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2f.t.\pi}\,dt \qquad (1.2.2.1)$$

In general, *X(f)* and *x(t)* are complex valued functions, *j* being imaginary unity number, defined as the square root of -1 and $2\pi f$ being the angular frequency range associated with the signal. From the definition of the Fourier integral, not every function *x(t)* has a transform *X(f)* [2] [20] [21]. While the exact conditions for convergence of functions are not known, two conditions that are widely considered sufficient for convergence (Bracewell, 1986) are:

Condition 1:

The integral of $|f(x)|$ from $-\infty$ to ∞ exists [2] [20]. That is,

$$\int_{-\infty}^{\infty} |f(x)|dx < \infty \qquad (1.2.2.2)$$

Condition 2:

Any discontinuities in *f(x)* are bounded [2].

Some of the properties associated with Fourier Transform are Linearity, Scaling, Time shifting, Frequency shifting, Symmetry, Modulation, Differentiation in time and Convolution. The inverse Fourier Transform that converts frequency domain signal into

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

time domain performs the exact opposite functionality of Fourier Transform and has the same complexity as the earlier. The Inverse Fourier Transform is defined as

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(f) e^{j2\pi ft} \, d\omega \qquad (1.2.2.3)$$

The Fourier Transform is used widely for image analysis, image filtering, and image reconstruction and image compression.

1.2.3 Discrete Fourier Transform

The Fourier Transform described in the previous section can be compared to an analog tool as it mainly deals with continuous signals and this is evident from the integral used in equation (1.2.2.2). For the special case in which the sequence to be represented is of finite duration, it is possible to develop an alternative Fourier representation, referred to as Discrete Fourier Transform (DFT). DFT is thus the Fourier representation of finite-length sequence which is itself a sequence rather than a continuous function, and it corresponds to samples equally spaced in frequency of the Fourier Transform of the signal [4]. Thus the Discrete Fourier Transform is used in the case where both the time and frequency variables are discrete. In short, Discrete Fourier Transform (DFT) also known as Finite Fourier Transform is widely used to analyze the frequencies contained in a sampled signal, solve partial differential equations, and to perform other operations such as convolutions [6] [17] [18] [19]. The two important reasons for using Discrete Fourier Transform over Fourier Transform are:

- The input and the output of the DFT are both discrete values making it convenient for computer manipulations.
- There is an algorithm called Fast Fourier Transform, which is a speedy way of computing the Discrete Fourier Transform.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The Discrete Fourier Transform is defined as:

$$X(k)=\sum_{n=0}^{N-1}x(n)\,e^{-2\pi ikn/N}\ ,\ 0\le k\le N-1 \qquad (1.2.3.1)$$

equation (1.2.3.1) can be rewritten as

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{kn}\ ,\ 0\le k\le N-1 \qquad (1.2.3.2)$$

where

$$W_N=e^{-i2\pi/N} \qquad (1.2.3.3)$$

with N being the number of time samples or number of frequency samples, $x(n)$ being input signal amplitude at time n and W_N^{kn} termed as Twiddle Factor. Thus, calculus is not needed to define the DFT or its inverse, and with finite summation limits, we do not encounter difficulties with infinities. Moreover, in the field of Digital Signal Processing, signals and spectra are processed only in sampled form, so that the DFT is what we really need for computational purpose [7]. In simple terms, DFT is computationally less intensive to compute than the Fourier Transform as can be seen below in this section. At the same time, the basic concepts are the same [7]. From equation (1.2.3.1) that for each value of k, direct computation of $X(k)$ involves N complex multiplications (4N real multiplications) and N-1 complex additions (4N-2 real additions) [8]. Consequently, it takes N^2 complex multiplications and N^2-N complex additions to compute all N values of the DFT [8]. Therefore, roughly $2N^2$ or $O(N^2)$ are required to calculate the DFT of length-N sequence [2]. The Inverse Discrete Fourier Transform (IDFT) performs the opposite functionality that of the DFT and involves the same complexity and the same number of computations as the later. The Inverse Discrete Fourier Transform is defined as:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$x(n) = \frac{1}{N}\sum_{n=0}^{N-1} X(k)W_N^{-nk}, \; 0 \le k \le N-1 \qquad (1.2.3.4)$$

Thus the Discrete Fourier Transform replaced the Fourier Transform resulting in computationally capable algorithm, which has found itself applicable in wide range of digital signal processing and image processing fields. Another aspect is the size of the memory required for an N-point DFT calculation. Since, each input term in equation (1.2.3.2) needs to be preserved until the last output term has been computed, the minimum memory locations required is $2N$ [2].

1.2.4 Fast Fourier Transform

Direct computation of DFT as seen from the previous section, consumes $O(N^2)$ computations for an N-point operation. Though this method of computation results in the correct output, the efficiency of this method when compared to the one to be discussed in this section is very less. The main reason for the inefficiency of the DFT algorithm is because it does not explore the Symmetry and Periodicity properties of the Twiddle factor W_N^{kn}. The two properties are defined as:

$$\text{Symmetry property: } W_N^{k+N/2} = -W_N^k \qquad (1.2.4.1)$$

$$\text{Periodicity property: } W_N^{k+N} = W_N^k \qquad (1.2.4.2)$$

The Fast Fourier Transform algorithms, utilizes the above two properties thereby reducing the total number of computations from $O(N^2)$ to $O(N \log_2 N)$ for an N-point DFT. Due to huge difference in the computational complexity between direct DFT and that of FFT algorithm calculations, FFT has rapidly replaced DFT as the pragmatic tool currently being used in every area of science and engineering that requires transformation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.2.4.1 Mathematical Calculation for FFT Computational Complexity

This section introduces Fast Fourier Transform algorithm's computational advantage over direct Discrete Fourier Transform calculation. The equations (1.2.3.2) and (1.2.3.3) are reintroduced for derivation purpose.

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{kn}\ ,\ 0\leq k\leq N-1 \qquad (1.2.4.1.1)$$

$$W_N=e^{-i2\pi/N} \qquad (1.2.4.1.2)$$

For derivation purpose, the value of *N* is chosen to be even. Hence *N* can be represented in the form $N=2^a$, where *'a'* is a positive integer. Since *N* is chosen to be even, the entire sequence of *x(n)* can be split into two sequences each of length *N/2* with one of the sequence containing even components of *x* while the second sequence containing the odd components. Thus equation (1.2.4.1.1) can be split into two summations each of length *N/2* and can be rewritten as follows:

$$X(k)=\sum_{n_{even}=0}^{N-2}x(n)W_N^{nk}+\sum_{n_{odd}=1}^{N-1}x(n)W_N^{nk} \qquad (1.2.4.1.3)$$

From [2], if *2a* represents the even components and *(2a+1)* the odd components with *a=0, 1, 2... N/2-1* then equation (1.2.4.1.3) can be written as

$$X(k)=\sum_{a=0}^{N/2-1}x(2a)W_N^{2ak}+\sum_{a=0}^{N/2-1}x(2a+1)W_N^{(2a+1)k} \qquad (1.2.4.1.4)$$

$$X(k)=\sum_{a=0}^{N/2-1}x(2a)(W_N^2)^{ak}+\sum_{a=0}^{N/2-1}x(2a+1)(W_N^2)^{ak}(W_N^k) \qquad (1.2.4.1.5)$$

But W_N^2 can be proved to be equal to $W_{N/2}$. Hence, equation (1.2.4.1.5) becomes

$$X(k)=\sum_{a=0}^{N/2-1}x_{EVEN}(a)(W_{N/2}^{ak})\ +(W_N^k)\sum_{a=0}^{N/2-1}x_{ODD}(a)(W_{N/2}^{ak}) \qquad (1.2.4.1.6)$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with $k=0, 1, 2..., N-1$. We have stated that the DFT of a sequence is periodic in its length, which means that the *(N/2)*-point DFTs of $x_{EVEN}(a)$ and $x_{ODD}(a)$ need to be calculated for only *N/2* of the *N* values of *k*. For each value of *k*, the number of addition operations taking place in each of the two summations is *N/2*. Hence total number of addition operations is $2(N/2) = N$. In addition to the *N* addition operations, there are *N* operations of multiplications by (W_N^k). Hence for *N* values of *k*, the total number of operations is N^2 additions and N^2 multiplications for a total of $2N^2$ operations or can be just represented as $O(N^2)$ as the index value 2 has comparatively lesser value. This represents the computational complexity when direct DFT method is employed. For large values of *N*, direct DFT computation becomes tedious and not practical to be implemented by digital systems. For $N = 8$, equation (1.2.4.1.6) can be diagrammatically represented using butterfly structures which depict the computation of $X(k)$ at the output from *N* values of $x(n)$ at the input.

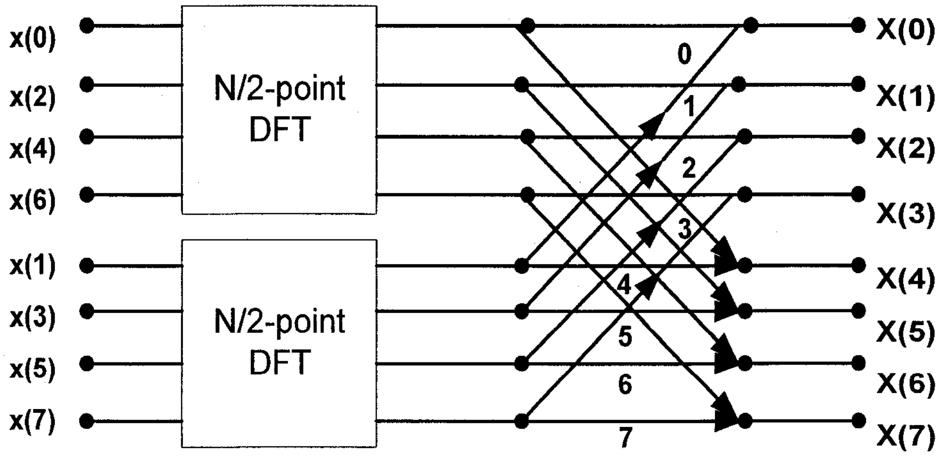

Figure1.1: Data Flow graph of an 8-point DFT calculated by splitting the *N* point input into two *N/2* parts containing even and odd components. The dots represent points wherein addition operations take place. The integers next to the large arrow-marks represent multiplications that take place due to (W_N^k) [2]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Using the property of periodicity, the FFT algorithms calculate DFT of an *N*-point input by splitting it into even and odd components just as direct DFT is computed, just that instead of calculating *k* value from *0* till *N-1,* they only calculate *k* value from *0* until *N/2-1* and the calculated values are re-used for *k=N/2, N/2+1, ... , N-1*. Hence, the total number of operations get reduced from $O(N^2)$ for direct DFT computation to $O(N^2/2)$ resulting in 50% lesser computation. Additional computational reduction can be brought using the Twiddle factor (W_N^k). Using the property of Symmetry, $W_N^{k+N/2} = -W_N^k$. Hence, instead of calculating (W_N^k) for *k* values from *0* until *N-1*, it is enough if we calculate for values of *k* from *0 till N/2-1* and the remaining values of *k* will be symmetric to the previous values. As a result, for an *N*-point input, only *N/2* values of twiddle factors have to be calculated. Thus figure1.1 can be redrawn as

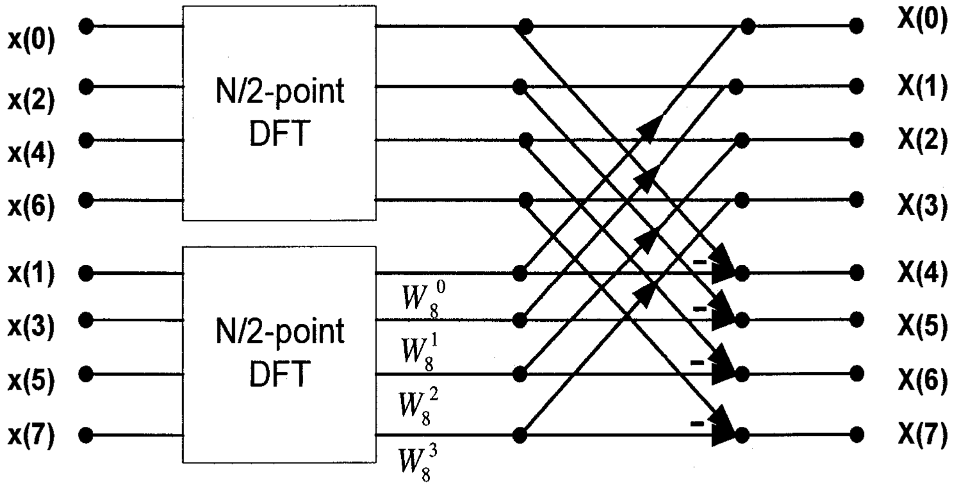

Figure1.2: Data Flow graph of an 8-point DFT calculated by splitting the *N* point input into two *N/2* parts containing even and odd components and using only *N/2* twiddle factors. The dots represent points wherein addition operations take place. When compared to the figure1.1, only four, that is only *N/2* number of twiddle factors are calculated while the remaining *N/2* are just represented as '-' sign due to symmetry property.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The initial process was to divide *N*-point DFT into two *N/2* point DFTs of even and odd components. The same process can be repeated thereby breaking down *N/2* points further down to two *N/4* points of even and odd DFTs and the process of division can be repeated until, it results in 2-point DFTs, which signals the end of the splitting process as calculating a 2-point DFT is very simple. The number of stages of such decomposition for an *N*-point input is $\log_2 N$. The FFT decomposition is shown diagrammatically as:

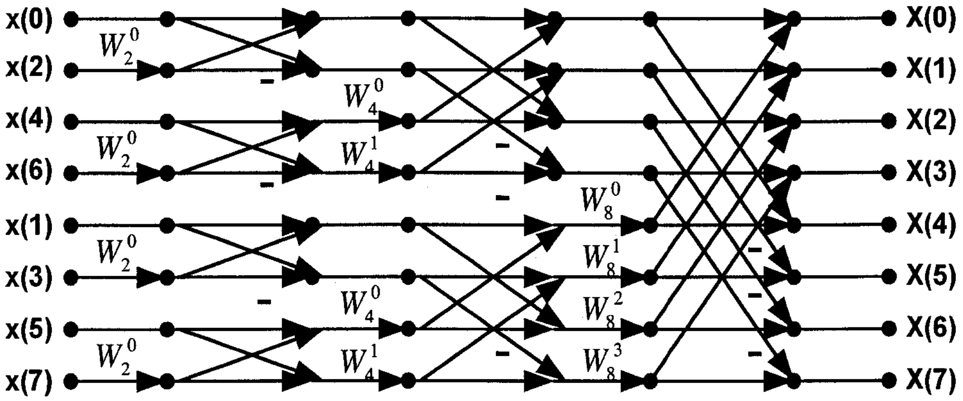

Figure1.3: Data Flow graph of an 8-point FFT, showing the entire decomposition until 2-point DFT stage is reached. The stage with W_2^0 is the last stage of decomposition with all butterflies being 2-point DFTs.

Each of the *N/2* 2-point FFT's requires one addition, one subtraction and there are *N/2* twiddle factor multiplications per stage. One the whole, every stage requires *O(N)* operations per stage. Since there are $\log_2 N$ stages on the whole, the total number of operations for an *N*-point FFT becomes $O(N\log_2 N)$. Thus, when compared to direct DFT computation that has a operational complexity of $O(N^2)$, FFT algorithms only have $O(N\log_2 N)$ which results in 50% and higher savings in computation making FFT one of the fastest and computationally most efficient form of transformation algorithms

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

developed with the significance clearly felt in applications requiring large values of *N* computation. FFT algorithms have virtually overthrown computation through direct DFT method and have found usage in a wide spectrum of fields ranging from sonar/radar detection, cellular communication, digital signal processing applications, image processing, designing High Definition Television (HDTV), medical imaging to name a few. Thus FFT algorithms have become an integral part of many scientific and engineering applications and hence there is an ever-increasing necessity to design more efficient FFT algorithms as well as architectures that map these efficient algorithms onto realizable hardware components.

1.2.5 Power Aware Design

With the semiconductor industry well into the deep sub-micron era, limitations on physical dimensions have led to unprecedented challenges in terms of behavioral aspects of devices. Greater challenges in terms of power dissipation, which posed trivial challenges in the earlier stages, are now posing immense constraints on devices being designed. Heat dissipation from such devices has become a field of research by itself, with the problem seeming to worsen as the device sizes shrink. The advent of mobile computing and communication devices has taken system design to a new high. In addition to designing and improving circuits for increased computational ability with an increased processing capability, higher operational clock frequency, higher throughput and increased packaging density, hardware circuits in deep sub-micron era need to be designed for lower power consumption to improve the life of battery on which these mobile devices solely depend, thereby reducing constraints on heat dissipation causing circuit breakdowns. Power reduction techniques are implemented at every level of design

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

abstraction starting from device modeling all the way until circuit design at the gate and transistor level and layout. Special techniques have also been developed at the fabrication level. With so much importance for design and implementation of power aware architectures and circuits, we take up the task of designing and simulating architecture that maps a unique FFT algorithm mainly aimed at memory access reduction, which could ultimately lead to lesser power consumption at the architectural level.

1.3 Organization of the Thesis Write-up

The entire report is organized into five chapters with the first chapter giving a basic in-depth into the necessity for this work and few of the technical startups needed to understand the underlying concepts behind our work.

Chapter 2 gives a comprehensive description about various Fourier algorithms and architectures based on these algorithms with specific focus on Fast Fourier Transform algorithms. Detailed explanation along with supporting mathematical equations has been provided for clarity purpose. Chapter 2 also gives a comprehensive coverage of the Twiddle Factor Based FFT algorithm, which forms the background of our architectural design. A clear insight on how the decimation is performed is also given.

Chapter 3 explains in detail the architectural technicalities involved in our design. The various logic blocks designed and simulated are shown pictorially and a good insight on the computational complexities of various blocks along with equations has been demonstrated. Memory reduction techniques have also been sighted to enhance the effectiveness of our work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4 deals with simulation output waveforms of various important blocks that have been designed and explained in chapter 3. In addition, various plots that depict the advantage our architecture and the algorithm it is based on possess over other architectures are plotted. The chapter also summarizes in mathematical units the effective power savings and memory access reduction achieved.

Chapter 5 concludes our work with a brief summary of our results and some suggestions for future designers depicting the scope for improvements that can be extended to our work to enhance its effectiveness.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2

FAST FOURIER TRANSFORM ALGORITHMS AND ARCHITECTURES

2.1 Introduction

As seen in chapter 1, the Fast Fourier Transform is computationally the most efficient way of computing the Fourier transformation. From Section 1.2.3, the calculation of direct form of DFT requires $O(N^2)$ and from Section 1.2.4.1, the calculation of DFT through FFT algorithm requires only $O(N\log_2 N)$ computations for an N-point input. For small values of N, this difference is not significant. But for large values of N, the FFT is orders of magnitude more efficient than direct DFT calculation. Table 2.1 below gives a comparative analysis of the computational advantage of FFT over direct DFT calculation for various values of N. A number of FFT algorithm variants have been designed over the years each having inherent computational advantages and disadvantages. FFT algorithms on the broader spectrum are divided into two main types namely Decimation-In-Time (DIT) and Decimation-In-Frequency (DIF). Either of the types involves splitting the input data points into odd and even components, perform DFT operations and then recombine the calculated values to form the output transformed data points. The DIT form of FFT algorithm is formed by splitting $x(n)$ which represents the time domain, into even and odd components of N/r data sequence and then continuing the decomposition or splitting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

operation till r-point DFT sequence is reached, where r is specified in terms of N as $N = r^k$, with k being a positive integer.

Table2.1: Comparative tabulation of DFT and FFT computational efficiency

Transform length *(N)*	DFT operations	FFT operations	DFT operations $\div$FFT operations
16	256	64	4
128	16,400	896	18
1024	1.05×10^6	10,240	102
1,048,576	1.1×10^{12}	2.1×10^7	52,429

The total number of decomposition stages is given by log_rN, with the log_rN th stage having all its computation as r-point DFT operations. Hence if $r=2$, then decomposition of $x(n)$ would proceed until the stage wherein all DFTs operations or so called butterfly structure have just 2-points each for computation forming a *radix-r (with r=2 in this case)* DIT-FFT algorithm. On the other hand, the DIF form of FFT algorithms are formed by splitting $X(k)$ which represents the frequency domain, into even and odd components of N/r data sequence as in case of DIT explained above. Hence, this type of algorithm is termed as *radix-r* DIF FFT algorithm. On a comparative basis, DIT and DIF are computationally same, thus enabling using either of the two forms of algorithms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 Decimation-In-Time (DIT) FFT Algorithm

As we saw in the previous section, Decimation-In-Time FFT algorithm for $N = r^k$, is derived by splitting N-point input sequence into N/r equal sequences of even and odd components of the input data. For example, if $r=2$, then N-point input data sequence given by $x(n)$ is divided into two N/2 sequences one containing even and the other containing odd components of $x(n)$. The equation for output $X(k)$ is given by equation (1.2.4.1.6) which is restated below for convenience.

$$X(k) = \sum_{a=0}^{N/2-1} x_{EVEN}(a)(W_{N/2}^{ak}) + (W_N^k) \sum_{a=0}^{N/2-1} x_{ODD}(a)(W_{N/2}^{ak}) \qquad (2.2.1)$$

Equation (2.2.1) can be generalized as

$$X(k) = F_1(k) + W_N^k F_2(k) \qquad (2.2.2)$$

Splitting $X(k)$ into two $N/2$ components results in the following equations

$$X(k) = F_1(k) + W_N^k F_2(k), \quad k = 0, 1, 2\ldots, N/2-1 \qquad (2.2.3)$$

$$X(k+N/2) = F_1(k) - W_N^k F_2(k), \quad k = 0, 1, 2\ldots, N/2-1 \qquad (2.2.4)$$

Equation (2.2.4) has a negative sign as compared to (2.2.3) because of the fact that $W_N^{k+N/2} = -W_N^k$. A butterfly is a structure that diagrammatically represents equations (2.2.3) and (2.2.4). Using butterflies to draw flow graphs simplifies the diagrams and makes them easier to read.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

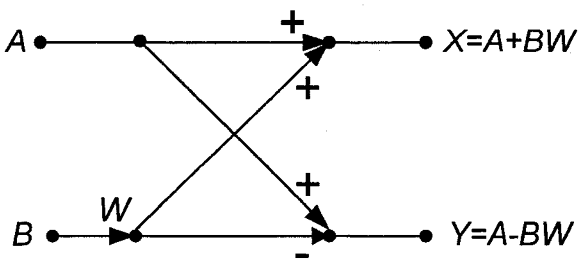

Figure2.1: A radix-2 Decimation-In-Time (DIT) butterfly structure

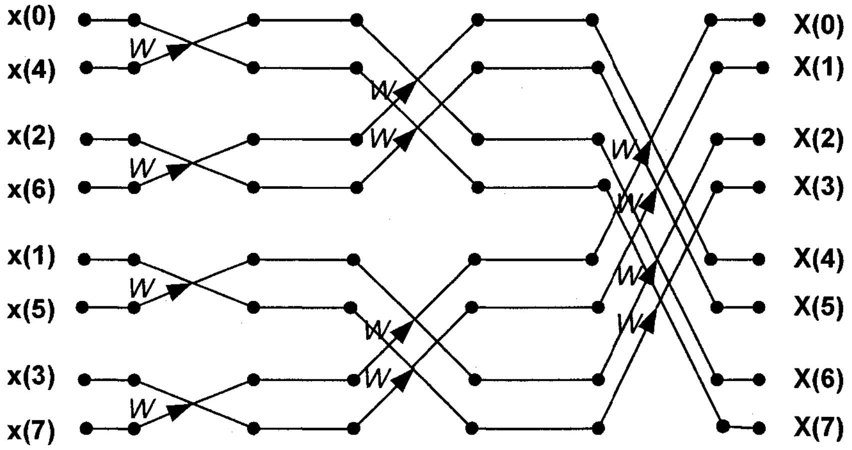

Figure2.2: Flow graph of an 8-point DIT FFT structure using butterflies

2.3 Decimation-In-Frequency (DIF) FFT Algorithm

As in the case of DIT, Decimation-In-Frequency is obtained by splitting the input sequence into *N/r* sequences. If *r=2* (say), then the *N*-point input *x(n)* is split into two sequence each of *N/2* points. Unlike in DIT wherein input data is split into even and odd terms, the DIF just involves splitting the input sequence into *N/r* sequence. To derive the algorithm, we begin by splitting the DFT formula into two summations (since *r=2*), one of which involves the sum over the first *N/2* data points and the second sum involves the last *N/2* data points [8]. Thus from [4] [5] [11] [12] we can write

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$X(k) = \sum_{n=0}^{(N/2-1)} x(n)W_N^{nk} + \sum_{n=N/2}^{(N-1)} x(n)W_N^{nk} \tag{2.3.1}$$

or

$$X(k) = \sum_{n=0}^{(N/2-1)} x(n)W_N^{nk} + W_N^{(N/2)k} \sum_{n=0}^{(N/2-1)} x(n+\frac{N}{2})W_N^{nk} \tag{2.3.2}$$

Combining the two summations in equation (2.3.2) and using the fact that $W_N^{(N/2)k}$ = $(-1)^k$, we obtain

$$X(k) = \sum_{n=0}^{(N/2-1)} [x(n) + (-1)^k x(n+\frac{N}{2})]W_N^{nk} \tag{2.3.3}$$

Considering the even and odd components of k and representing them with $X(2a)$ and $X(2a+1)$ respectively, so that

$$X(2a) = \sum_{n=0}^{(N/2-1)} [x(n) + x(n+\frac{N}{2})]W_N^{2na} \tag{2.3.4}$$

$$X(2a+1) = \sum_{n=0}^{(N/2-1)} [x(n) - x(n+\frac{N}{2})]W_N^{2na}W_N^{n} ,$$

$$a = 0, 1..., (N/2-1) \tag{2.3.5}$$

Thus DIF equations can be generalized for $r=2$ as

$$X(k) = F_1(k) + F_2(k) \tag{2.3.6}$$

$$X(k+\frac{N}{2}) =(F_1(k) - F_2(k))W_N^k ,$$

$$k =0, 1, 2... (N/2-1) \tag{2.3.7}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

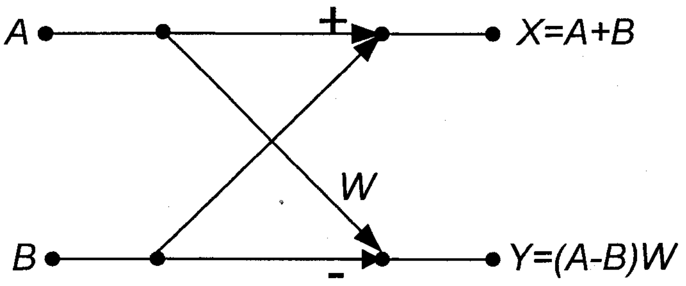

Figure2.3: A radix-2 Decimation-In-Frequency (DIF) butterfly structure

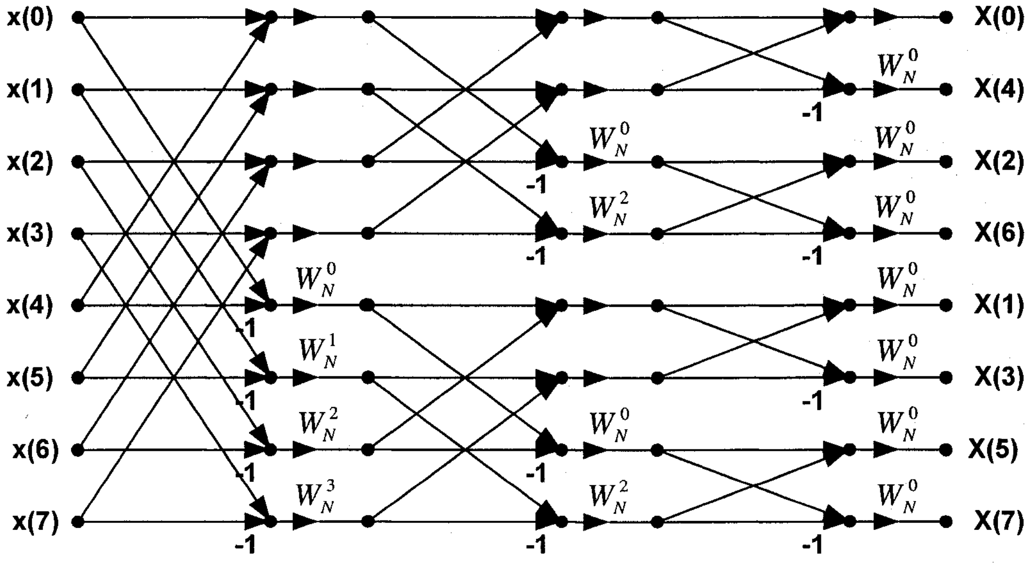

Figure2.4: Flow graph of an 8-point DIF FFT structure using butterflies

2.4 Decimation Based on Twiddle Factors

In microprocessor-based system, memory access is expensive mainly due to larger latency and higher power consumption [1] [10]. From figure 2.4, it can be seen that the various twiddle factors used for the DFT operations depicted by the butterfly structure are repeated at different stages and even within each stage. For example, the twiddle factor W_N^2 gets used in stage1 as well as stage2. Even within stage2, W_N^2 is used twice.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Hence in this 8-point DIF FFT, W_N^2 is seen to be used for a total of three times. In terms of computation, it involves accessing memory thrice to bring in the same twiddle factor to perform DFT computation. Thus, the redundancy in twiddle factor memory access is obvious. For larger values of N, the redundant memory access becomes substantial leading to higher power consumption. A unique Twiddle-Factor-Based FFT algorithm [1], is designed to reduce the frequency of memory access as well as multiplication operations. The algorithm is mainly divided into two sections based on the Twiddle factors that are present. The first section named as Super Stage (SS), computes the butterflies involving twiddle factors W_N^j $(j \neq 0)$ through a computation scheme similar to Hoffman coding [1]. In this section, all butterflies that use the same twiddle factor are clustered together and computed, thereby having to load that twiddle factor only once to compute all the butterflies that use it, instead of accessing the same twiddle factor each time a butterfly that uses it needs to be computed, resulting in substantial memory access reduction. In the second section, $(N-1)$ butterflies involving the twiddle factor W_N^0 are computed using a top-down tree structure. Simulations proved a 20% reduction in clock cycles and an average of 30% reduction in memory access for a 32-point FFT using Twiddle-Factor-Based FFT algorithm when compared with the conventional DIF FFT algorithm [1] [10].

Hence, using the Twiddle-factor-Based FFT algorithm, if a twiddle factor gets loaded from the memory, it gets utilized until there is no further need for it in any further computations. This in terms of number of memory access is only $(N/2-1)$ for an N-point input for Twiddle Factor Based FFT as compared to $(N-1)$ for conventional FFT algorithms. The power saving can be significant using the Twiddle-Factor-Based FFT

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithm especially for large values of N. One main advantage this algorithm has apart from having lesser memory access is that the second stage dealing with W_N^0 does not involve any multiplication as $W_N^0=1$. Hence we also have extra power savings through non-usage of multipliers that are power hungry and computationally intensive.

2.4.1 Decimation Procedure

In case of Twiddle-Factor-Based FFT algorithm described in [1] [10], the butterflies that are computed at each stage are spread across $\log_2 N$ stages of a conventional decimation. The Twiddle-factor-Based FFT decomposition is shown in figure 2.5. As can be seen from figure 2.5, butterflies with the same twiddle factor (represented by $W[x]$, with x varying from 0 till $(N/2-1)$ and this representation is analogous to W_N^x), that were computed at different stages in a conventional algorithm, now gets computed within a single stage thereby avoiding the necessity to load the same twiddle factor numerous times as compared to just once in [1] [10]. The decomposition of the algorithm proceeds in the following fashion. For an N-point FFT, the binary index of a data sample resembles $(A_k A_{k-1} \ldots A_0)$, with $k=log_2N-1$.

(1) At the first stage of decomposition, all data samples of the form $(A_k A_{k-1} \ldots 1)$ are computed and any two data samples of the form $(A_k A_{k-1} \ldots 1)$ and $(\bar{A}_k A_{k-1} \ldots 1)$ can pair together to form a butterfly [1].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

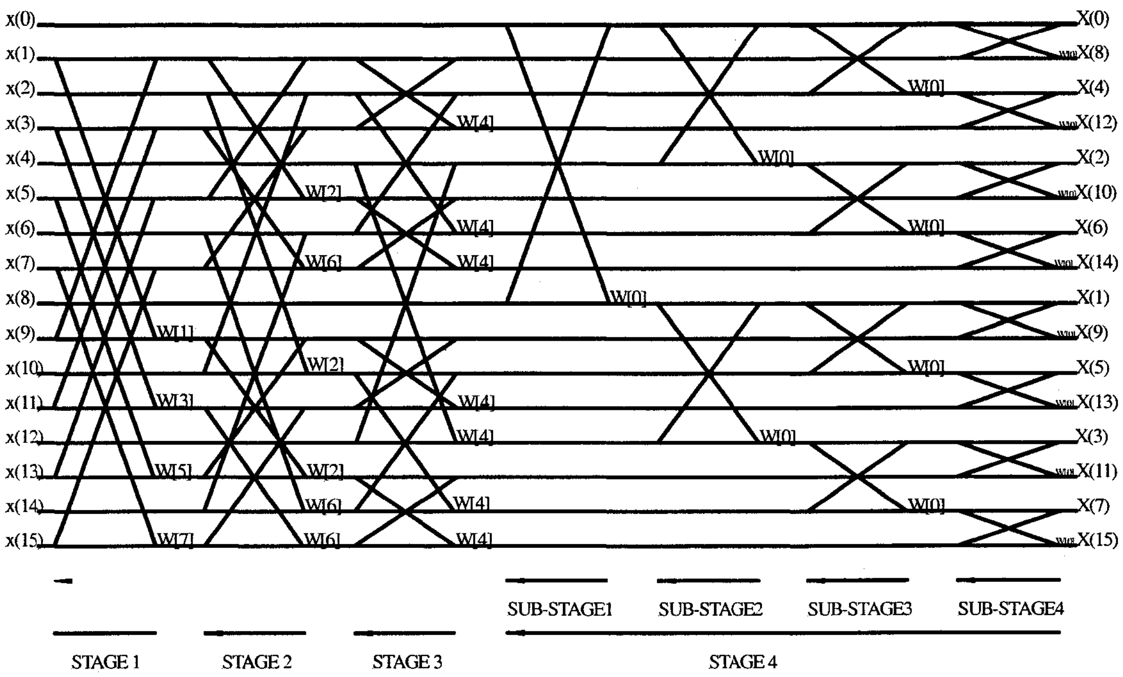

Figure 2.5: Flow graph of a 16-point FFT structure based on Twiddle-Factor-Based algorithm

The twiddle factor that corresponds to this butterfly is given as W_N^j where j corresponds to the decimal value of the binary sequence given by

$(0 A_{k-1} \ldots A_2 A_1 1)$.

(2) At the second stage of decomposition, all the data samples with binary sequence $(A_k A_{k-1} \ldots A_2 10)$ and $(A_k A_{k-1} \ldots A_2 A_1 1)$ are computed. Any two data samples with binary sequence $(A_k A_{k-1} \ldots A_2 10)$ and $(\bar{A}_k A_{k-1} \ldots A_2 10)$, or $(A_k A_{k-1} \ldots A_2 A_1 1)$ and $(A_k \bar{A}_{k-1} \ldots A_2 A_1 1)$ can pair together to form a butterfly. The twiddle factor corresponding to this butterfly is W_N^j where j corresponds to the decimal value of the binary sequence given by $(0 A_{k-1} \ldots A_2 10)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(3) Within log_2N-1 stages of decomposition, all data samples with twiddle factor other than W_N^0 are calculated.

(4) The log_2N^{th} stage involves butterflies whose corresponding twiddle factor value is $(000...1)$.

The obvious advantage that can be seen from the above form of FFT decomposition is the number of memory access that is made with regards to accessing the various twiddle factors. Thus we only need $(N/2-1)$ memory access as against $(N-1)$ required by conventional algorithms [1] [10]. Another interesting aspect in addition to reducing the memory access is that the log_2N^{th} stage of decomposition involves W_N^0 as its twiddle factor and $W_N^0 = 1$. Thus there is no multiplication involved with butterflies using this twiddle factor and hence the final stage decomposition involving $(N-1)$ butterflies does not involve any multiplication thereby helping us save valuable multiplication operations.

Thus, the Twiddle factor based FFT algorithm seems to be a more appropriate algorithm that caters to the need for power aware FFT systems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 3

ARCHITECTURAL DESIGN AND IMPLEMENTATION

3.1 Overview

In chapter 2, we had a comprehensive explanation about the Twiddle-Factor-Based FFT algorithm and also the computational advantages it had over other conventional FFT algorithms was explicitly shown. Designing architecture to map this computationally challenging but intensely less memory access-involving algorithm is the main aspect of our work.

Until the early to mid-90s, low power electronics were for the major part considered only for a very few applications largely comprising of small personal battery-powered devices. But the mid 90s saw the remarkable development of CMOS sub-micron technology, which subsequently led to the advent of deep sub-micron CMOS era. Another radical change that revolutionized the electronics market was the unprecedented demand and subsequent development of portable communication and computational devices that mainly depended on battery power for their operation. Unfortunately, the battery industry could not keep pace with the developments in the semiconductor industry. As a result, high-end electronic portable devices needed constant battery recharging, which made them less user friendly. This resulted in extensive research done towards design and implementation of newer VLSI algorithms, architectures and circuit

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

techniques that would utilize the minimum possible power without sacrificing any of the other parameters such as bandwidth, clock speed, area and throughput.

Our architectural design, based on a power reduced FFT algorithm is a small step in this direction.

3.2 Design Target

The FFT algorithms have a wide range of signal processing and communication transmission applications. In recent times, one important area where FFT has found extensive application is High Definition Television (HDTV). According to the standard of European Digital Broadcasting, FFT/IFFT must execute 8192 points in 896 microseconds. In addition, we also target our architecture towards Orthogonal Frequency Division Multiplexing (OFDM) transceiver whose IEEE 802.11g standard requires it to execute *1024*-point FFT in 51 microseconds.

3.3 Algorithm Setup

We recall the Twiddle Factor Based FFT decomposition structure shown in figure 2.5. From the structure, the algorithm can be broadly classified into three distinct divisions namely Input, Processing and Output stages.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

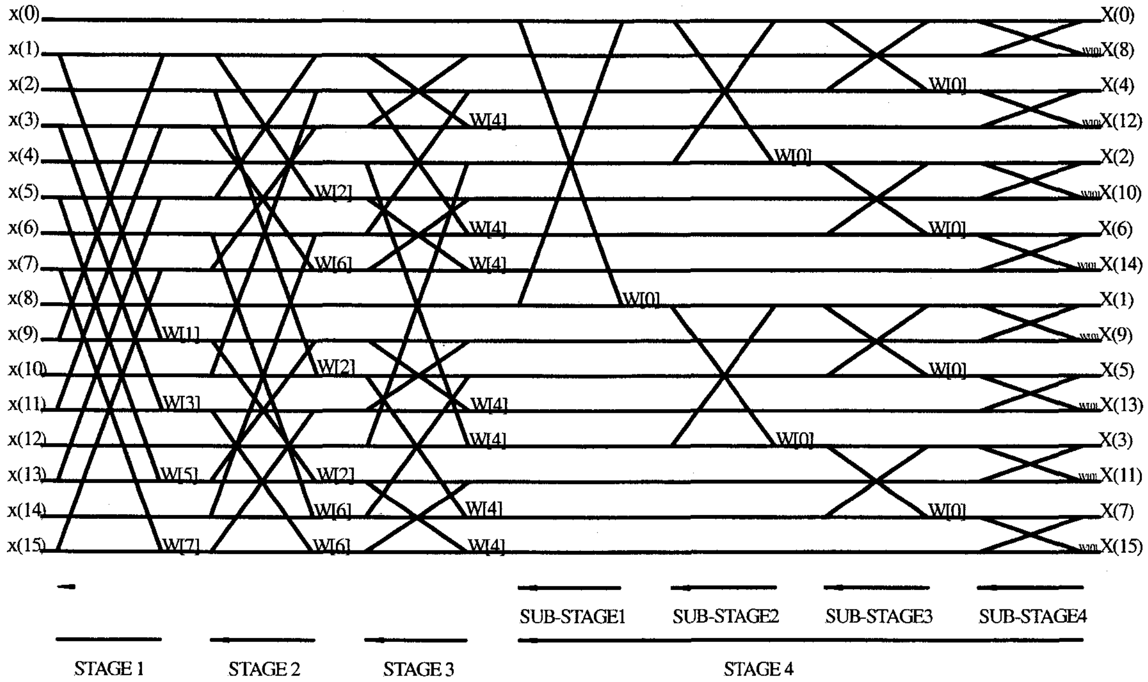

Figure 3.1: Flow graph of a 16-point FFT structure based on Twiddle-Factor-Based algorithm

At the Input stage, complex form data values corresponding to the *N* unprocessed, in other words time domain signals *x(n)* where *n=0,1,2,... ,(N-1)* are input into the memory device from external source (generally other blocks whose output need FFT processing). The address location where the input data values would get stored depends on values of *n*. That is, if data value corresponding to *x(n)* needs to be stored, it gets stored at a location whose address is given by binary equivalent of *(n+1)* , that is if data corresponding to *x(0)* needs to be stored, it gets stored in a memory location whose address is binary equivalent of *(n+1)* or *(0+1=1)*. Similarly for *x(4)*, the address is binary equivalent of *(4+1=5)*. The number of binary bits to represent memory address depends on the size of the memory device as shall be seen in the coming sections. Thus, data is fed into memory starting from *x(0)* all the way up to *x(N-1)*, in subsequent locations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The processing division is where the DFT butterfly operation takes place. This division is equivalent to the Central Processing Unit of any computer system. This block transforms input time domain signals $x(n)$ into its corresponding frequency domain signal $X(k)$.

The transformed $X(k)$ output values are written back to the same locations in the memory device from where they were initially accessed. Thus after all operations, the same memory locations which contained time domain signals would now contain frequency domain signal output. The output division accomplishes this process. For any given N, the total number of decomposition stages is given as $\log_r N$, where $N=r^k$, k being any positive integer. Hence the above stated process is repeated $\log_r N$ times resulting in the final values of frequency domain $X(k)$ signals.

3.4 Random Access Memory (RAM) Address Generation Block Design

The functionality of the RAM address generator is to determine

(1) Total number of stages of decomposition

(2) Total number of DFT butterfly operational groups within each stage based on twiddle factors

(3) Determine the memory address locations of the data needed for each of the DFT butterfly operation.

The RAM address generator is one of the few complex operational blocks designed in our architecture. Based on our observation of the Twiddle-Factor-Based decimation FFT algorithm, we can make the following conclusions assuming $N=2^k$, k is any positive integer.

(1) Total number of stages of decomposition for an N-point input = $\log_2 N$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(2) Only the butterflies in decomposition stages *1* till (log_2N-2) involve complex multiplications.

(3) Butterflies with twiddle factors $W_{\frac{N}{2^{(LOGN-2)}}}^{1,3,5,\ldots,(N/2^{LOGn-1})}$ form the $(log_2N-1)^{th}$ stage and involves only multiplication with imaginary term *j* which can be effectively implemented by switching or swapping real and imaginary terms of the output and inverting the sign term between them, thus avoiding use of complex multiplier.

For the purpose of RAM address generation, we came up with a pseudo code in C++ which could generate addresses of all butterfly DFTs based on Twiddle-Factor-Based FFT algorithm decomposition described in [1] [10]. The Pseudo code consists of two parts with the first describing the decomposition for the first (log_2N-1) stages and the second describing the $(log_2N)^{Th}$ stage. The first Pseudo code depicting address generation of the first (log_2N-1) stages consists of four nested loops.

(1) Loop 1: corresponds to the current stage number (between 1 and (log_2N-1))

(2) Loop 2: corresponds to the total number of different twiddle factors that are used within a given stage

(3) Loop 3: for a given twiddle factor, we have butterfly DFT structures that vary in size. Loop3 computes total number of different sizes of butterflies that utilize a given twiddle factor.

(4) Loop 4: Computer total number of butterflies for a given size and a twiddle factor

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.4.1 Formulae of Computational Complexity for the Loop Structures for *(log_2N-1)* Stages of Decomposition

Loop 1 Computation – The loop 1 as seen from section 3.4, computes the total number of stages of decomposition excluding the last stage, which is generally computed separately as this stage does not involve any twiddle factor multiplication. Excluding the last stage of decomposition, the total number of stages is $(\log_2 N-1)$ and hence loop1 index varies from *1* to $(\log_2 N-1)$.

Loop 2 Computation – The generalized formula for the calculation of Twiddle factors that are used in each of the $(\log_2 N-1)$ stages, where $N=r^k$, *k* being any positive integer is given as

$$W^{x}_{\frac{N}{r^{(CS-1)}}}\ \text{, where } x=1, 3, 5, 7\ldots [(N/r^{cs})-1] \tag{3.4.1.1}$$

where *cs* represents the current stage number

The current stage value *(cs)* varies from *1* to $(\log_2 N-1)$. For our architectural design we assumed $r=2$. Hence, (3.4.1.1) becomes

$$W^{x}_{\frac{N}{2^{(CS-1)}}}\ \text{, where } x=1, 3, 5, 7\ldots [(N/2^{cs})-1] \tag{3.4.1.2}$$

$$= W_N^{x2^{(CS-1)}}\ \text{, where } x=1, 3, 5, 7\ldots [(N/2^{cs})-1] \tag{3.4.1.3}$$

The total number of butterflies in each stage is given as

$$\frac{(2^{CS}-1)N}{2^{(CS+1)}} \tag{3.4.1.4}$$

Equation (3.4.1.4) gives the total number of butterfly operations per stage. Thus, for $(\log_2 N-1)$ stages, the total number of butterflies is given as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\sum_{CS=1}^{\log_2 N-1}\left(\frac{N}{2^{CS+1}}\right)(2^{CS}-1) \qquad (3.4.1.5)$$

Using equation (3.4.1.3) and the values *N=16* and *r=2*, we can work out an example to calculate the twiddle factors used at various stages of decomposition using the Twiddle-Factor-based FFT algorithm. Number of stages with twiddle factor multiplications is given as *(log₂N-1) = (log₂16-1) =3.* Thus the stage index value or *CS* varies from 1 to 3.

Stage#1:

Twiddle factors accessed when *CS*=1 is given as

$$W_N^{x2^{(CS-1)}}, \text{ where } x=1, 3, 5, 7\ldots [(N/2^{CS})-1]$$

$$=W_{16}^{x2^{(1-1)}}, \text{ where } x=1, 3, 5 \text{ and } 7$$

Therefore the twiddle factors that are used when *CS*=1 are W_{16}^1 W_{16}^3 W_{16}^5 and W_{16}^7

Stage#2:

Twiddle factors accessed when *CS*=2 is given as

$$W_N^{x2^{(CS-1)}}, \text{ where } x=1, 3, 5, 7\ldots [(N/2^{CS})-1]$$

$$=W_{16}^{x2^{(2-1)}}, \text{ where } x=1 \text{and } 3$$

Therefore the twiddle factors that are used when *CS*=2 are W_{16}^2 and W_{16}^6

Twiddle factors accessed when *CS*=3 is given as

$$W_N^{x2^{(CS-1)}}, \text{ where } x=1, 3, 5, 7\ldots [(N/2^{CS})-1]$$

$$=W_{16}^{x2^{(3-1)}}, \text{ where } x=1$$

Therefore the twiddle factors that are used when *CS*=1 is W_{16}^4

Loop 3 Computation – For a given twiddle factor within any given stage, there can be DFT butterfly structures of varying sizes, where size of a butterfly with reference to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

conventional FFT algorithm can be said to be $(N/2^{CS})$ where CS corresponds to current index value of stage number which varies from 1 to $\log_2 N$ for a given value of N. In case of conventional FFT algorithms, all butterflies that were computed within a single stage were of the same size. On the other hand, Twiddle-factor-Based FFT algorithm, computes butterflies based on their twiddle values. Hence, butterflies that are computed within a single stage need not necessarily contain butterflies of the same size. The total numbers of different sized butterfly structures that utilize a single twiddle factor at any given stage of decomposition equals the value of the stage of decomposition where the butterfly is currently present. For example, W_{16}^{1} twiddle factor that is present in stage 1, would have butterflies of only one size using that twiddle factor as it lies in the first stage. Similarly, W_{16}^{4} which lies in stage 3, would have butterflies of three different sizes. The different sizes might also be specified in terms of levels, with each level being occupied by butterflies of one particular size. Hence three different sizes for a single twiddle factor means there are three levels for that twiddle factor. The size of butterflies varies starting from $N/2$ for the first level and successive steps having half the size of the butterflies in the previous level. For example, we saw that W_{16}^{4} had three different sizes of butterflies. Hence,

Size of butterflies present in level 1 = $N/2$

Size of butterflies present in level 2 = $\frac{1}{2}(N/2) = N/4$

Size of butterflies present in level 1 = $\frac{1}{2}(N/4) = N/8$

Loop 4 Computation – Having formulated the number of stages of decomposition, total numbers of twiddle factors per stage and the total number of levels of butterflies per

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

twiddle factor. We are left with calculating the total number of butterflies to be computed in each of the levels. The formula for calculating the total number of butterflies present per level is given as $2^{(CS-1)}$. For example, again considering twiddle factor W_{16}^{4}, we can thus calculate the number of butterflies in each of the three levels. Thus

Total number of butterflies present in level 1 = $2^{(1-1)} = 1$

Total number of butterflies present in level 2 = $2^{(2-1)} = 2$

Total number of butterflies present in level 3 = $2^{(3-1)} = 4$

An important aspect of the butterfly decimation is that within a given level, the butterflies can be computed in any order. Similarly for a given twiddle factor within a stage, the various levels can be computed in any order and so can be done with different twiddle factors within a given stage. Thus the parallelism can be exploited at various levels of decomposition. This feature makes the Fast Fourier Transform the most sought after transformation algorithm in numerous signal processing and communication applications.

3.4.2 Pseudo Code Implementation

Based on the four loop structure calculated for the first $(\log_2 N-1)$ decomposition stages based on Twiddle-Factor-Based FFT algorithm, a C-like pseudo code to implement the DFT butterfly operations over $(\log_2 N-1)$ stages is shown below in figure (3.2).

```
bf_size = N/2
LOOP 1:
i = 1 to ($\log_2 N-1$), i++
a1 = (2^ (i-1))

LOOP 2:
j = 1 to ($N/2^{(i+1)}$), j++
bf_size = N/2
a = a1
a = a + 2(j-1)* a1
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
LOOP 3:
K = (i-1) to 0, k- -

    LOOP 4:
    m = 1 to (N/(bf_size*2))
        b = a + bf_size
        c[m] = a
        a = a + (2*bf_size)
        DFT Butterfly operation to be performed

    LOOP 4 ENDS
    a = c[1]/2


    bf_size = (bf_size/2)

LOOP 3 ENDS
LOOP 2 ENDS
LOOP 1 ENDS
```

Figure 3.2: Pseudo code for the *(log_2N-1)* stages decomposition

3.4.3 Architectural Blocks Design and Implementation

For the pseudo code designed in figure (3.2), a logic design at the behavioral level is done using Very large-scale integrated circuits Hardware Description Language (VHDL) tool. Aldec Inc., license version of Active HDL version 6.3 was used entirely to design and simulate the various architectural blocks. The architectural implementation of the *(log_2N-1)* stages of decomposition based on Twiddle factors is shown in figure (3.3). On comparing the pseudo code with the architectural blocks, we can find that the block *i_block* corresponds to loop1, while *j_block* and *k_block* correspond to loop2 and loop3 respectively. Due to logical implementation complexity, loop4 is implemented using three blocks namely *m_block, m1_block* and *m2_block.* The main functionality of the RAM address generator as stated earlier is to generate address of data elements present in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the memory unit that are needed for all the DFT butterfly operations over the entire decomposition stages. For every butterfly operation, two data elements are required, which implies that two addresses are required to be generated by the RAM address generator for every butterfly operations that gets computed. From the architectural design, q_a and q_b represent the two binary address values that get generated. Each of the two outputs is a binary sequence of width determined by the size of the memory unit it accesses as can be seen in the upcoming sections. All the blocks are of non-pipelined nature wherein only one block remains operational at any instant. The blocks gets initiated one after the other in the order required automatically by triggering signals which present within one block trigger the next after the completion of execution by current block. There are six blocks in total implementing the four nested for loops. The value of q_a is generated at multiple locations along the six blocks. The correct value of q_a that corresponds to any particular butterfly operation is chosen among the different q_a values based on a resolution function block. The alphabet V represents a signal that toggles for every value of loop1 index and in turn triggers *j_block* which then gets executed loop2 index times and V' represents the situation wherein *j_block* completes executing loop2 index times and hence control is transferred back to *i_block* for further processing. Similarly, W and W' represent the computational triggering signal between *j_block* and *k_block.* On the same ground, we can explain the functionality of X, X', Y, Z and Z'. The complete VHDL implementation of the architecture described in figure (3.3) is shown in APPENDIX. The other important signals that form a part of our design are the *clear*, *enable* and *clock.* The *clock* is basically a synchronizing signal whose complete functionality is explained in section 4.7. The *clear* signal is used as an erasing signal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which when *active high* or *1* erases all the values at the output ports and assigns them to high impedance.

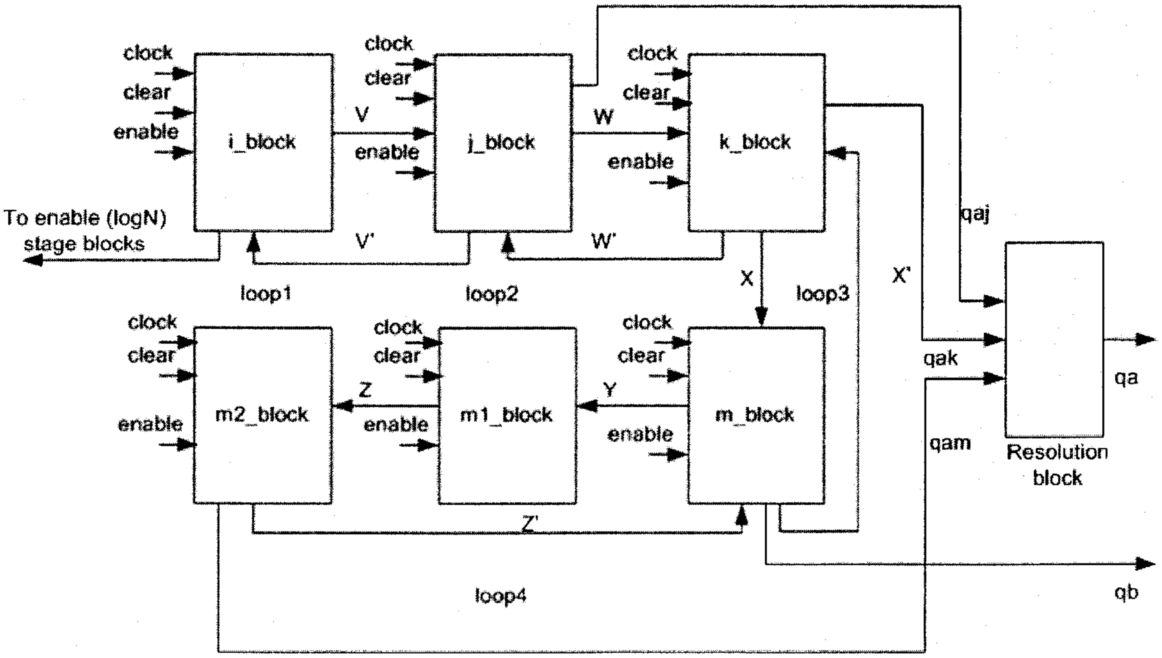

Figure 3.3: Block design representation of *(log_2N-1)* stages based on Twiddle factor decomposition algorithm

The *enable* signal is basically to determine if a block needs to be operational at any given instant. Any block proceeds with its execution only if *enable* signal remains *active low* or *0*, failing which, the previous output of that block is retained irrespective of changes in its input.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.5 Formulae of Computational Complexity for the Loop Structures for log_2N^{th} Stage of Decomposition

In section 3.4, we saw the RAM Address generation block design for the first (log_2N-1) stages of decomposition based on twiddle factors. The final stage or log_2N^{th} stage of decomposition is probably unique when compared to the other stages mainly in terms of usage of multipliers. The log_2N^{th} stage is classified in such a way that all the butterflies in this stage irrespective of their levels or sizes use only the twiddle factor W_N^0. The value of W_N^0 is always equal to 1, irrespective of the value of N. DFT butterfly calculations based on equations (2.3.6) and (2.3.7) reveal that using W_N^0 is equivalent to multiplying equation (2.3.7) by 1. The twiddle factors used for multiplication in the first (log_2N-1) stages are complex in nature (containing real and imaginary terms). Hence when these complex twiddles get into equation (2.3.7), they result in complex multiplications, which are computationally complex and intensive to design and implement. Hence the absence of complex multiplications in the (log_2N^{th}) stage makes this stage computationally less intensive and results in large savings in terms of clock cycles for computation and also in terms of power dissipation. The total number of butterflies that are computed in this stage is given as $(N-1)$. Since there is no multiplication by twiddle factor involved in this stage, we can disable all the multipliers that are designed to handle multiplications. This procedure is more complex in other conventional FFT algorithms. Though W_N^0 factor is present even in other FFT algorithms, they get utilized at different stages of decomposition, unlike in Twiddle factor based algorithm [1] wherein W_N^0 gets utilized only in one stage. Hence in conventional algorithms, disabling the multipliers at different time intervals is more difficult. This is one obvious advantage Twiddle-factor-based FFT

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithm has over other FFT algorithms. As a result, we can disable the multipliers for clock cycles equivalent to implementing *(N-1)* complex multiplications. This results in power savings as digital circuits like memory units and multipliers are power hungry. Though power efficient multipliers are being designed, it is more suitable to reduce the usage of multipliers.

3.5.1 Memory Reduction Technique

The Twiddle-Factor-Based FFT algorithm [1] [10], describes a methodology by which memory access can be reduced even at the $(log_2N)^{th}$ stage. From figure 3.1, butterflies within the $(log_2N)^{th}$ stage is decomposed into (log_2N) further sub-stages as depicted in figure 3.4. From figure 3.4 it can be seen that in sub-stage1 (S1), the two outputs of butterfly are represented as *A* and *B*. The output value *A* serves as input for *C* and that of *B* used as input for *D* present in sub-stage2 (S2), while input values for points *E* and *F* do not depend on their previous stages. Hence input to these points has to be accessed directly from memory units. It becomes redundant if *A* and *B* get stored in the relatively slow memory units and then *C* and *D* accessing those values back from the same memory unit. Instead, *A* and *B* can be stored in temporary registers from where *C* and *D* can access them.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

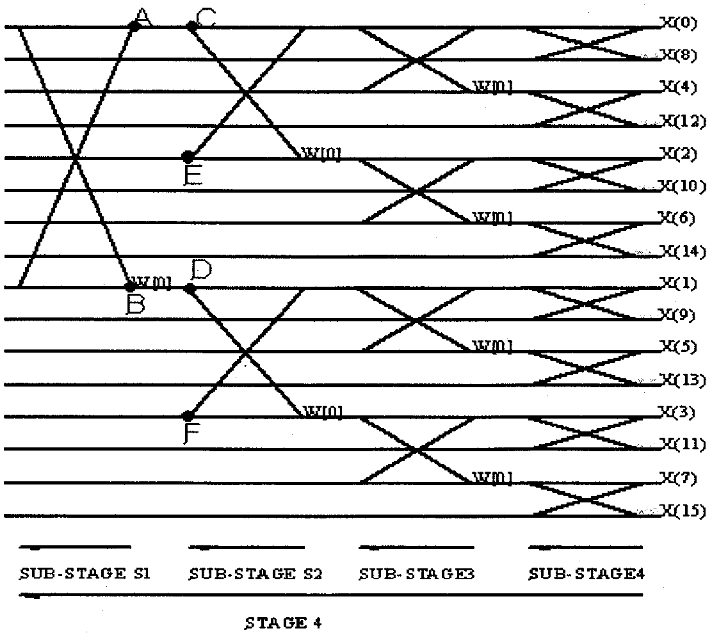

Figure 3.4: $(log_2N)^{th}$ stage decomposition structure for $N = 16$

While one of the inputs to every butterfly can thus be accessed from temporary registers where the previous stage outputs get stored, the second input to the butterflies is only accessed from the main memory units. As a result of this, main memory accessing gets reduced by as much as 50%. Accessing data from main memory units is more power consuming than accessing them from temporary registers simply because, temporary registers can be built according to user needs and can be placed closer to the processing unit. Moreover the main memory structure is designed in a more complex way when compared to the temporary register bank. The decoding for temporary register location is also simple when compared to the main memory. Based on our evaluation, the total number of temporary registers required for an *N*-point input is *N/2*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.5.2 Pseudo Code Implementation

A C-like pseudo code for implementation of $(log_2N)^{th}$ stage is given in figure (3.5). It is a two for loop structure generating the addresses q_a and q_b that represent the binary equivalent of the addresses of the two memory locations from where data for computing the butterfly is accessed. This pseudo code only deals with address generation and not with memory reduction concept.

```
bf_size = N/2
LOOP 1:
k = 1 to (log2N), k++
a1 = 0; a= 0

LOOP 2:
m = 1 to (N/(bf_size*2)), m++
a = a+a1
b = a + bf_size
c = a;
d = b;
c = (a + b)
d = (a – b)
a1 = (bf_size)*2


LOOP 2 ENDS
bf_size = (bf_size/2)



LOOP 1 ENDS
```

Figure 3.5: Pseudo code for the $(log_2N\text{-}1)$ stages decomposition

3.5.3 Architectural Blocks Design and Implementation

The two-loop pseudo code is logically designed using VHDL at the behavioral level. The blocks are designed in such a way that of the two data values that are required for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

computation of every butterfly, one data comes from the temporary register while the other comes from main memory unit. On one hand, the outputs from the butterflies that are computed in the *(log_2N-1)* sub-stages are stored back into temporary registers while on the other hand, the outputs of butterflies computed in the *(log_2N)th* sub-stage get stored into the main memory unit rather than in the temporary registers mainly because, the *(log_2N)th* sub-stage forms the final stage of computation and hence it forms the output stage and the output values need to be written back onto main memory from where they would be accessed by other systems. Figure (3.6) shows the architectural blocks of the *(log_2N)th* stage implementation. As explained in section 3.4.3, the functionality of *A*, *A'*, *B*, *B'*, *C*, C', *D* and *D'*, is to trigger the successive blocks to which they are respectively connected. The *clear*, *enable* and *clock* signals perform the same functionality as that explained in section 3.4.3. The *q_temp_reg_addr* is the address corresponding to the temporary register location from wherein data needs to be accessed. The *qb_logn* represents the second address needed for performing the butterfly operation, which is fetched from RAM. The VHDL block implementation of the architecture explained in figure (3.6) is given in APPENDIX. On the design front, we plan on designing a *1024*-point FFT architecture targeting HDTV application. For this purpose, we need *512* temporary registers each of 32-bits wide. The *512*, 32-bit temporary registers are implemented as a tree structure. Initially, one 32-bit temporary register is designed. It is then duplicated or in VHDL terms port mapped to form the second temporary register.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

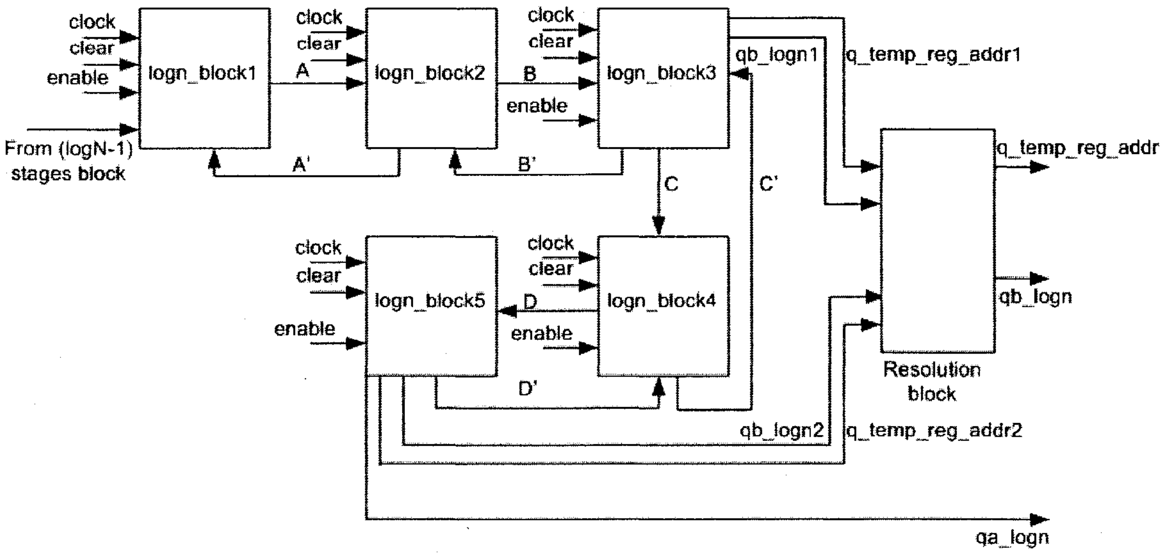

Figure 3.6: VHDL Block implementation of $(log_2N)^{th}$ stage based on Twiddle factor decomposition algorithm

Now, the two identical registers are considered to be a single entity and this is then port-mapped to form two more identical registers, resulting in four registers. Proceeding this way, we get to design *512*, 32-bit temporary register bank. During the different stages of algorithm execution, data is constantly written onto and read out from the temporary registers. Hence, there is a necessity to generate the address of the temporary register where data is to be currently written or to be read. The logic block implemented as shown in figure 3.6 generates the address corresponding to the right temporary register. This address is decoded in stages by a decoder. Each stage of temporary register design has its own decoder, which decided the exact location of the temporary register.

The number of address bits needed to specify a location in the temporary register bank is determined using the following expression

Number of address bits = log_2 *(total number of register locations)* (3.5.3.1)

Hence, in case of temporary register bank, we have *N/2* or *512* register locations. Substituting in equation (3.5.3.1), the number of address bits is *9*. Thus we need *9*-bit

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

binary sequence to represent each of the temporary register locations. In the previous section, we saw that in the $(log_2N)^{th}$ stage, data is accessed both from main memory unit as well as from temporary registers. To achieve this, the design incorporates multiplexers to choose between the two data source. Each temporary register has a *Read* and *Write* signal. If the *Read* signal is set to *high* or *1*, data is read out of temporary register, while making *Write* signal *high* on the other hand, enables us to write data into the registers.

The complete block architecture for address generation for both $(log_2N\text{-}1)$ stages and $(log_2N)^{th}$ stage is shown in figure (3.7).

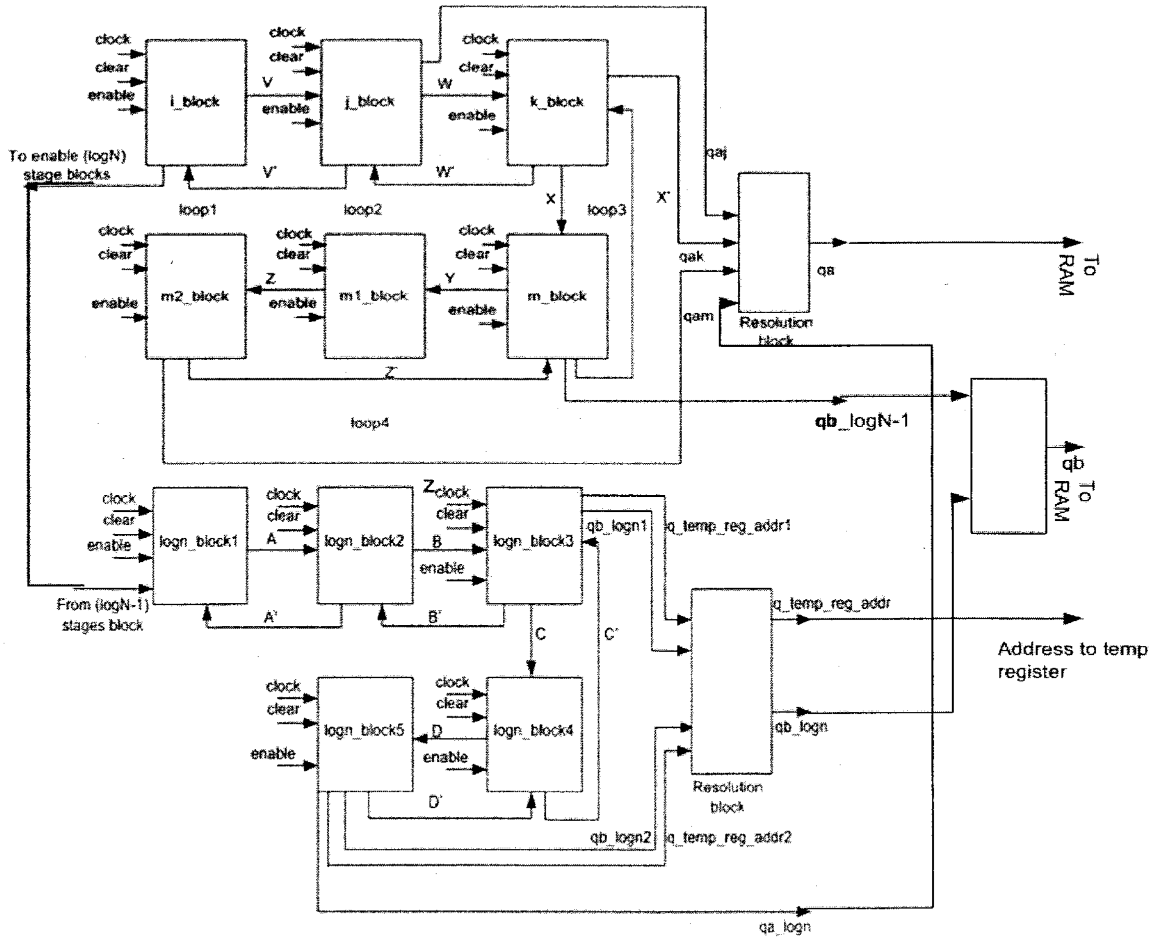

Figure 3.7: Block implementation of $(log_2N\text{-}1)$ *stages and* $(log_2N)^{th}$ stage based on Twiddle factor decomposition algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.6 Random Access Memory Design

The Random Access Memory (RAM) is the main memory system designed and simulated in our architecture structure. Our FFT architecture is designed for *1024* points input system. Hence we have *1024*-points time domain input signals that need transformation. For this purpose, the input data needs to be stored in a memory unit from where they can be accessed whenever necessary. The design and implementation of the Random Access Memory (RAM) tends to serve this purpose.

The biggest advantage that FFT algorithms possesses that other transform algorithms do not is that of *In-place* computation. The FFT algorithms as seen get computed in stages. In conventional FFT algorithms, input data as well as output data are accesses in and out of RAM, which serves as the primary memory unit. The input data that is used in the computation at each stage is only needed for that stage. Once the output to that stage gets generated, the inputs that this stage used are no more needed and can be replaced by the output data obtained. This process of replacing or re-using the same set of memory locations that stored the input data to store output data values is termed as *In-place* computation. This concept results in minimum requirement and usage of memory unit size, which makes it optimal for digital systems. Hence, for an N-point FFT implementation,

The number of Main memory (RAM) locations needed = N (3.6.1)

If each memory location is represented by the term *word*, then, for a *1024*-point FFT implementation, we need *1024 words* or locations in the RAM to store all the input and output values. The address locations for the memory locations start from binary

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

equivalent of *0* and continue all the way up to *1023*, thus specifying all the *1024* locations. Each location can store data of width *32*-bits. From equation (3.5.3.1), the number of bits requires to address each location can be calculated to be *10*. Hence, the address sequence starts from *0000000000* all the way until *1111111111* with increment of *1*. Two distinct but similar RAM blocks having the same set of address sequence is designed to store both the real and imaginary part of the complex data input of the form *(a+jb)* with *a* and *b* being any real number. Thus, accessing the exact memory locations from both the blocks simultaneously results in accessing both the real and imaginary parts of an input data. The basic structure of a *1*-word RAM is shown in figure (3.8). This structure is common for both the real and imaginary parts of RAM design block.

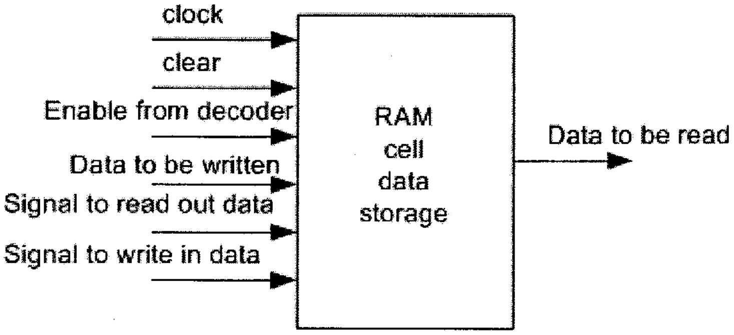

Figure 3.8: Block level representation of 1-word RAM cell for storing *32*-bit data

From figure (3.8), the basic building block of a RAM word consists of a *Read*, *Write*, *Clear* and *enable from decoder* as its signals. The *Read* and *Write* signals enable us to input data into and output data respectively from any memory location. The *Clear* signal if *high* or *1* erases the contents of the memory location and replaces the output as well as the content of the memory to high impedance. Only if the *enable from decoder* is made

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

active low or *0*, will it be possible to access data from or into the location. Otherwise, whatever was the previous output continues to remain at the output port and no changes get reflected. The structural design methodology followed for RAM is similar to the way the temporary register bank was designed in section 3.5.3. The VHDL implementation of the RAM cell is shown in APPENDIX.

In order to access the exact location in the RAM block, we need to decode the address bit sequence generated by the RAM address generator. The decoder block implements this procedure. The decoder is basically a de-multiplexer, which chooses one among its many outputs based on the value of the input. The basic structure of a decoder is shown in figure (3.9).

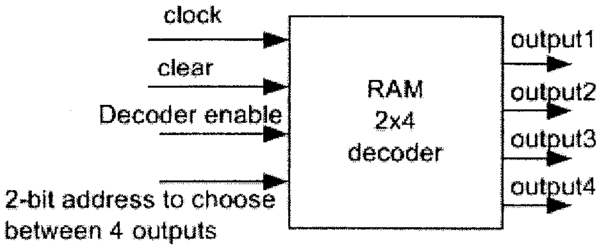

Figure 3.9: Block representation of a RAM decoder

In our design, we implement one decoder to decode *2* address bits thus enabling us to locate four RAM word cells. In other words, one decoder can help us operate four different memory locations. Thus for a *1024*-points RAM cell locations, we need *341* decoders in total. This can be made clear by figure (3.10) and the explanation given below. We know that we need *1* decoder to decode 4 RAM cells and are shown below. Let this combination be termed as a *group*. Hence, a *group* contains 4 RAM cells and *1*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

decoder. Based on the previous calculation, if we have four separate *groups*, each having 4 RAM cells and *1* decoder as in figure (3.10a), then we got to a total of 16 RAM cells and *4* decoders decoding these 16 RAM cells.

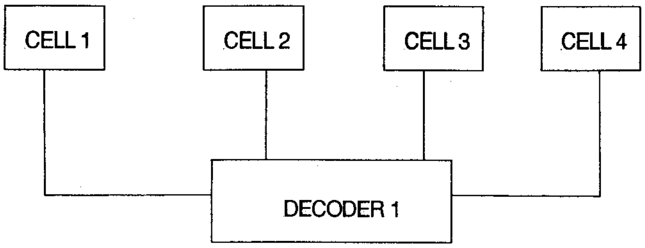

Figure 3.10a: Basic group formation of RAM cells using decoders

This is depicted in figure (3.10b). To choose one *group* among the four available, we need a decoder. Thus, for a total of 16 RAM cells decoding, we get to use *5* decoders. Let us term this as *high group*.

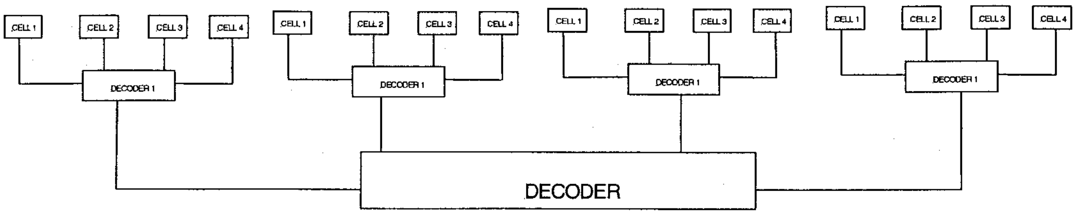

Figure 3.10b: Hierarchical group formation of RAM cells using decoders

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proceeding on a similar ground, we can see that for four such *high groups* to get decoded, we need *20* decoders, thus decoding 64 RAM cells. In order to one among the four *high groups*, we need a decoder. Hence for decoding 64 RAM cells, we need a total of *21* decoders. The calculation can thus be extended to higher levels of grouping and thus for a *1024* cells RAM structure, we would need a total of *341* decoders. The VHDL block level design of such a hierarchical RAM and decoder structure for 64-words RAM is shown in APPENDIX. The disadvantage of such huge memory units is obvious from our previous discussions. The humongous hardware requirements hamper the performance of such memory units in terms of operational speed, data retrieval and storage time, and area and power consumption. Hence in recent times, high-end research is devoted to designing and fabricating huge memory devices with minimum hardware. As a result, high-speed memories such as *cache* and *flash* memories have been designed so as to reduce the operational delays and also power consumption. These high-end memories are designed to be very small and hence are area efficient as well. All these benefits have created a tremendous scope for such memories and are widely being utilized in many system designs especially mobile devices where area and power savings are of primary importance.

3.6.1 Read Only Memory (ROM)

In the previous section, we saw the design and implementation of Random Access Memory (RAM). The RAM is a memory unit wherein data can be written, read and erased. Thus RAM can be termed as an erasable memory. On the other hand, ROM is a Read only option memory wherein data to be stored is pre-determined during its fabrication and is hardwired. Thus the data once hardwired cannot be changed and as a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

result, no new data can be written on to a ROM cell. Hence there is only a *Read* option and no *Write* option. The ROM is thus used only if there are stored values that do not change during the course of execution. In any FFT algorithm, the twiddle factors W_N^k, *k =0, 1... ((N/2)-1)* do not change their values once computed. Hence for a given *N*-point FFT, the entire *N/2* number of complex twiddle factors is stored in the ROM, whose size is determined as *N/2*. Thus for our *1024*-point architecture, we need *512* locations in the ROM to store the various twiddle factor values with each location being *32*-bits wide. Since, there are *512* ROM locations; we need *10*-bit binary sequence to address every ROM location. The address to the ROM blocks are generated by ROM address generation block, which keeps generating consecutive address locations of twiddle factors when ever it gets triggered by a signal from the RAM address generation block. Every time the second for loop in the RAM address generation block gets executed, a new twiddle factor needs to be accessed from ROM. Hence, a trigger signal is sent to the ROM address generation block indicating that address for the new twiddle factor be generated by it and sent to the ROM blocks which will then output the twiddle factor values to the location of the butterfly operation. Figure (3.11) below shows the ROM blocks designed in our architecture. VHDL block representation of ROM is shown in APPENDIX.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

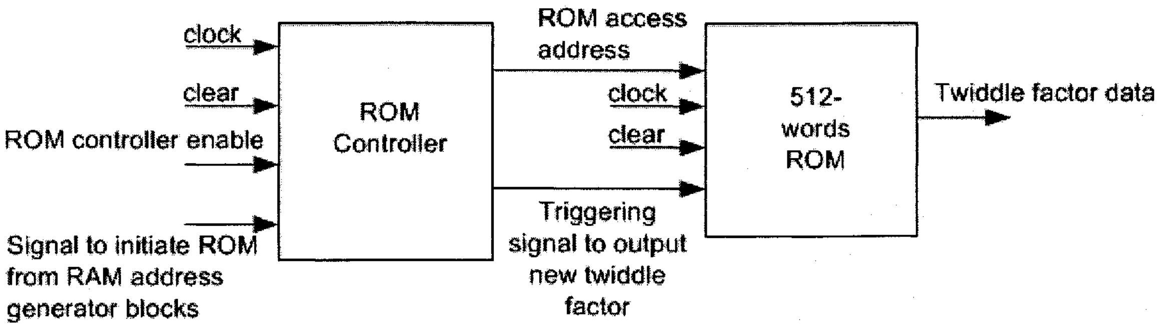

Figure 3.11: Block representation of Read Only Memory (ROM) and its controller

3.7 Data-Path Design

With the address generation block, Random Access Memory and Read Only Memory design complete, the architecture can now generate the addresses of the locations from where time domain signals stored in the Random Access Memory and twiddle factors from the Read Only Memory can be accessed. Once the time domain signal and corresponding twiddle factor data from the memory units get accessed, they must be transformed into frequency domain output signal. In other words, DFT butterfly operation needs to be performed on the accessed input data. For this purpose, we designed the data-path block. It is in this block that arithmetic operations such as addition, subtraction and multiplication are performed.

The formulation for each of the DFT butterfly operations in the Twiddle-Factor-Based FFT algorithm [1] is based on equations (2.3.6) and (2.3.7) and may be recalled for clarity.

$$X(k) = F_1(k) + F_2(k) \tag{3.7.1}$$

$$X(k+\frac{N}{2}) = (F_1(k) - F_2(k))W_N^k,$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$k = 0, 1, 2 \ldots (N/2-1) \qquad (3.7.2)$$

Equations (3.7.1) and (3.7.2) involve one addition, one subtraction and one multiplication. The $(\log_2 N)$ stages Twiddle-Factor FFT algorithm [1] can be divided into three distinct groups based on twiddle factors.

Group1: consists of butterflies from stages 1 till $(\log_2 N-2)$. The twiddle factors that are utilized in any of these stages are complex in nature, which is of the form $(a+jb)$ with a, b being any real number. The input data $F_1(k)$ and $F_2(k)$ stored in the memory units are also complex in nature as the twiddle factors itself. Hence, when equations (3.7.1) and (3.7.2) get computed, we need to perform one addition, one subtraction and one complex multiplication which are computationally more intensive than a real multiplication.

Group2: consists of butterflies in the $(\log_2 N-1)^{st}$ stage. The twiddle factor that gets utilized in this stage is of value $j=\sqrt{-1}$. Hence, when equations (3.7.1) and (3.7.2) get computed, we need to perform one addition, one subtraction and one trivial multiplication with just the imaginary term j. We term this multiplication trivial because multiplication with j can be easily performed by just swapping the input real and imaginary terms and invert the sign between the two terms. Thus, there are no multiplication operations actually involved. Hence all multipliers designed can be disabled.

Group3: consists of butterflies in the $(\log_2 N)^{th}$ stage. The twiddle factor that gets utilized in this stage is W_N^0 whose value is always 1 irrespective of the value of N. Hence, when equations (3.7.1) and (3.7.2) get computed, we need to perform just one addition and one

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

subtraction and there are no multiplications involved in this group. Hence all multipliers designed can be disabled.

From the above three groups, we can find the addition and subtraction operations to be common, while in case of multiplication, only group1 involves complex multiplication and the other two groups do not involve any multiplication. Hence, exception for addition and subtraction operations, we need three separate methods, by which we can choose between performing complex multiplications, swapping operations and no multiplications.

3.7.1 Complex Multiplier Implementation

As seen from section 3.7, butterflies computed in group1 have complex multiplications. Each of the butterfly operation thus involves one complex multiplication. Complex multiplications are more complicated when compared to real multiplications as they contain two terms real and imaginary. Hence complex multiplication involves more computations than an ordinary real multiplication. There are two methods for implementing complex multiplications in digital system. From equation (3.7.2) complex multiplication takes place between $F_2(k)$ and W_N^k, where $k=0, 1..., ((N/2)-1)$. If complex $F_2(k)$ is considered to be of the form $(a+jb)$ and the complex term W_N^k is considered to be of the form $(c+jd)$ with a, b, c, d be any real numbers. The complex multiplication now becomes

$$(a+jb)*(c+jd) \tag{3.7.1.1}$$

$$= ac+jad+jbc-bd \tag{3.7.1.2}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Equation (3.7.1.2) can be implemented in two methods.

Method1: This method is a straightforward implementation of equation (3.7.1.2). Equation (3.7.1.2) has two addition, one subtraction and four real multiplication operations. The terms *ac* and *bd* represent the real term of complex multiplication while *ad* and *bc* represent the imaginary part. Pictorial representation of the multiplication method1 is given below in figure (3.12).

Method2: This method implements equation (3.7.1.2) using three additions, two subtractions and three real multiplications [17]. In this method, the number of real multiplications gets reduced at the cost of increase in the number of additions and subtractions. Pictorial representation of the multiplication method2 is given below in figure (3.13).

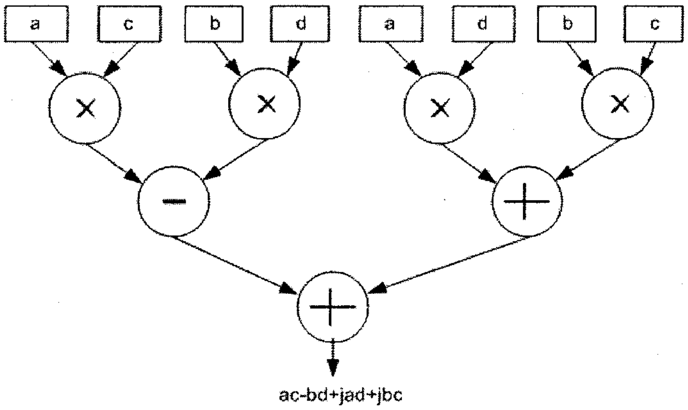

Figure 3.12: Complex multiplication implementation using method 1

Of the two above discussed methods of complex multiplication implementation, method2 has an obvious advantage in terms of number of real multiplications. But the trade-off is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

an increase in the number of addition and subtraction operations. In general digital circuit design terms, a multiplier design and layout is on the higher side in terms of area and power consumption. It consumes more clock cycles for its execution when compared to an adder or subtraction circuitry. Taking these factors into consideration, method2 of implementing complex multiplication is better when compared to method1. Hence our design incorporates method2 for implementing complex multipliers.

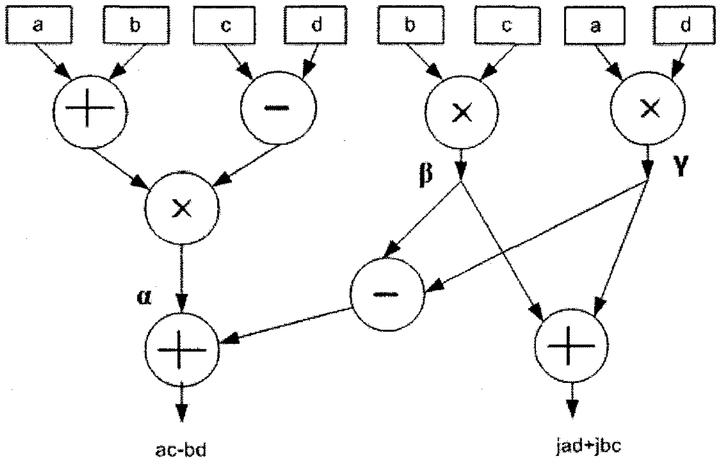

Figure 3.13: Complex multiplication implementation using method 2

From [9], it is seen that the implementation of a real multiplier involves three major steps namely Booth encoding, Partial Product reduction and Carry propagate addition. The purpose of these three steps in order specified is to reduce the number of computations on the multiplication operations. The partial product reduction is based on Wallace tree structure. Complete description about the three steps is mentioned in detail in [9]. Assuming each of the three steps takes one clock cycle to execute, it takes three complete

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

clock cycles for implementing one real multiplication. This is depicted in figure (3.14). The complex multiplication using method2 can be implemented using *3* real multiplications, *3* real additions and *2* real subtractions. The complete block level implementation is shown in figure (3.15). The implementation is seen to consist of *5* stages, with each stage consuming one clock cycle.

Booth Encoding	Partial Product Reduction	Carry Propagate Addition

Figure 3.14: Real multiplication implementation stages.

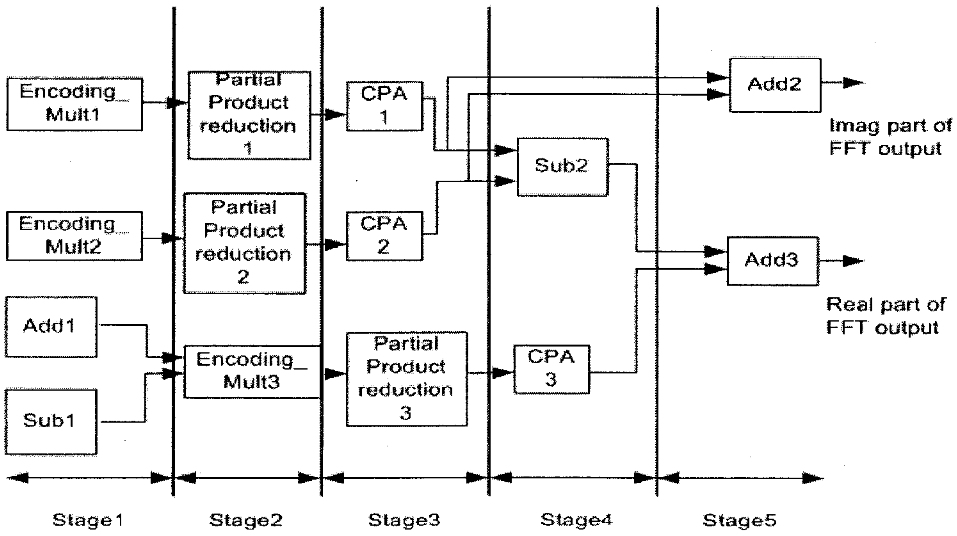

Figure 3.15: Complex multiplication implementation using method 2

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.7.2 Group1 Design and Implementation

In the previous section, we saw two ways by which complex multiplication could be implemented. We also considered one of the two methods to be more suitable for our design. Having described the logic implementation methodology, we need to now see the VHDL implementation of the various blocks that make up data-path for group1 butterflies described in section 3.7. Based on equations (3.7.1) and (3.7.2) we can design our data-path. Equation (3.7.1) involves addition of the two input data $F_1(k)$ and $F_2(k)$. Since $F_1(k)$ and $F_2(k)$ are both complex in nature, we need to add the real and imaginary parts of the two data separately to satisfy equation (3.7.1). Hence we need two real adders to implement equation (3.7.1). Equation (3.7.2) can be split into two parts with the first being $(F_1(k)-F_2(k))$ and the second being complex multiplication with W_N^k. Hence, the second equation involves one subtraction and one complex multiplication. Due to the complex nature of data, we need to have two separate subtractors one each for real and imaginary term. The block diagram depicting the data-path for group1 as designed in our architecture is shown in figure (3.16).

3.7.3 Group2 Design and Implementation

We know that the twiddle factor that gets utilized in this group is of value j. Based on equations (3.7.1) and (3.7.2) we can design our data-path for group2. Equation (3.7.1) is same as that for group1 butterflies. Hence we can retain the same adders and subtractors used for group1 implementation. Equation (3.7.2) can be split into two parts with the first being $(F_1(k)-F_2(k))$ and the second being multiplication with j. As discussed earlier, just swapping the real and imaginary terms and inverting the sign between the swapped terms can achieve multiplying with imaginary term j. Hence, the multipliers are disabled

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

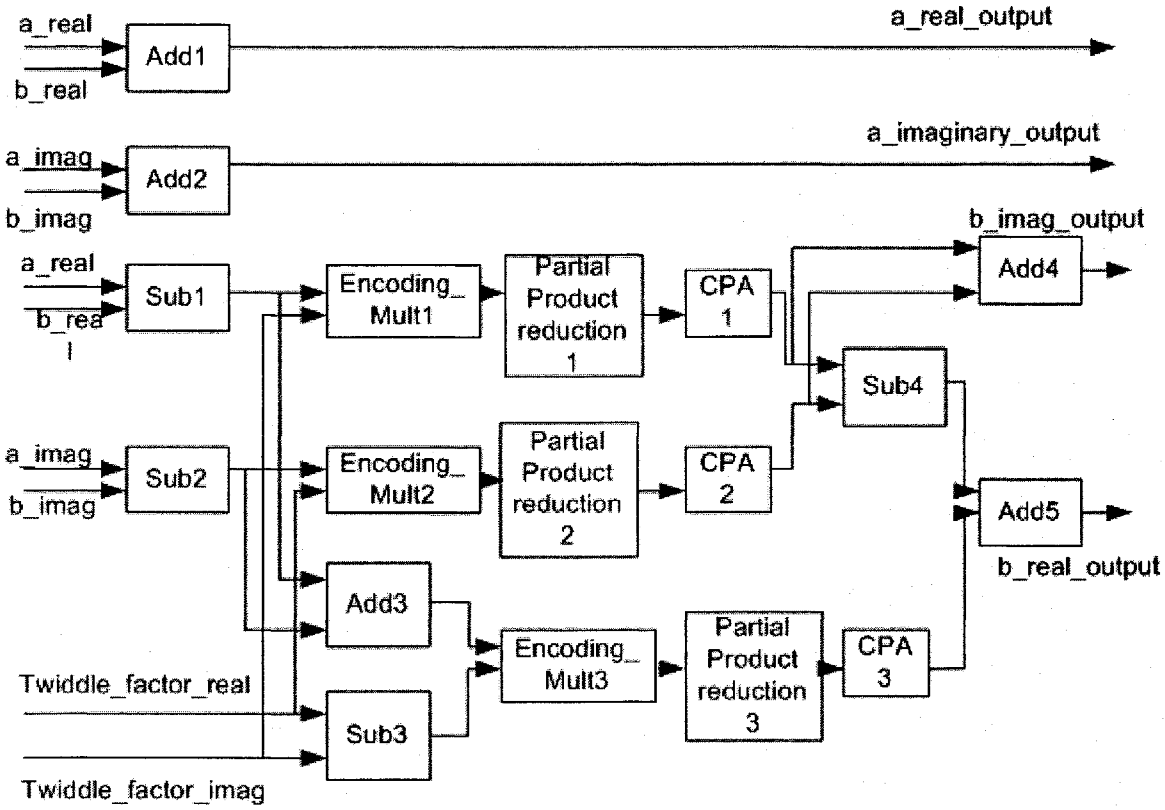

Figure 3.16: Block level Data-path for group1 for twiddle factor based architecture

whenever butterflies of this group get executed. Of the 32-bits of data, the first bit represents the sign bit and is *1* for negative numbers and *0* for positive numbers. Once subtraction operation takes place, we use two registers to swap the real and the imaginary terms. The two registers get activated only when group2 butterflies get computed. Thus data-path for group2 utilizes the same adders and subtractors that were used for group1 computation. It only requires two new registers to swap real and imaginary data. The block diagram depicting the data-path for group2 as designed in our architecture is shown in figure (3.17)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

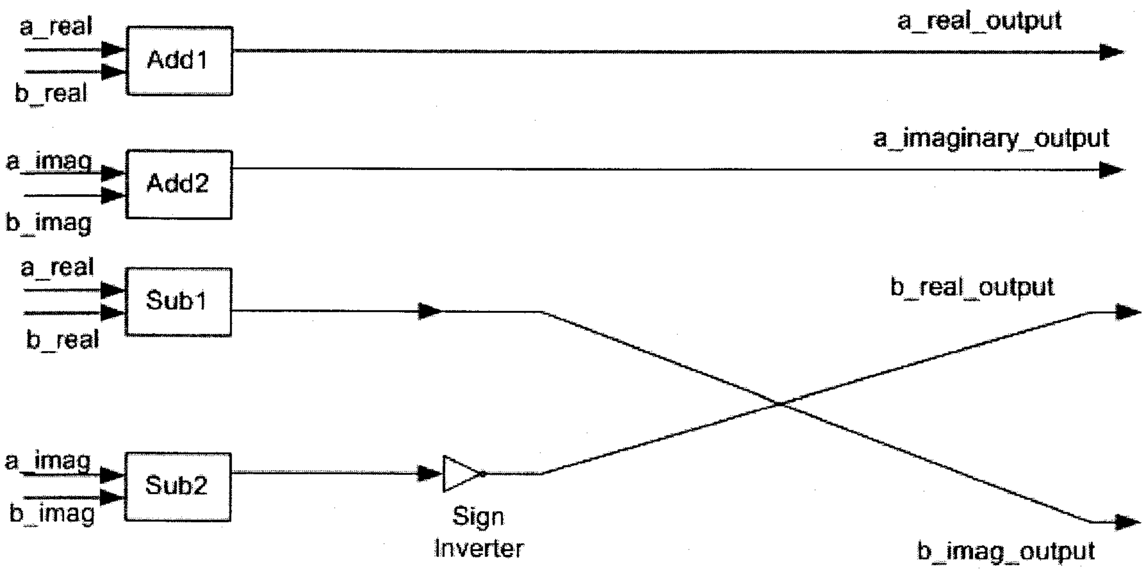

Figure 3.17: Block level Data-path for group2 for twiddle factor based architecture

3.7.4 Group3 Design and Implementation

We know that the twiddle factor that gets utilized in this group is of value *1*. Based on equations (3.7.1) and (3.7.2) we can design our data-path for group 3. Equation (3.7.1) is same as that for group1 and group2 butterflies. Hence we can retain the same adders and subtractors used for group1and 2 implementation. Equation (3.7.2) can be split into two parts with the first being $(F_1(k)-F_2(k))$ and the second being multiplication with *1*. As discussed earlier, multiplying with *1* can be just ignored. Hence equation (3.7.2) just involves two real subtractions one each for real and imaginary term. Hence for group3 butterfly computations, we do not need to design any further blocks. We only disable the multipliers used in group1 and the two swapping registers used in group2. The block

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

diagram depicting the data-path for group3 as designed in our architecture is shown in figure (3.18)

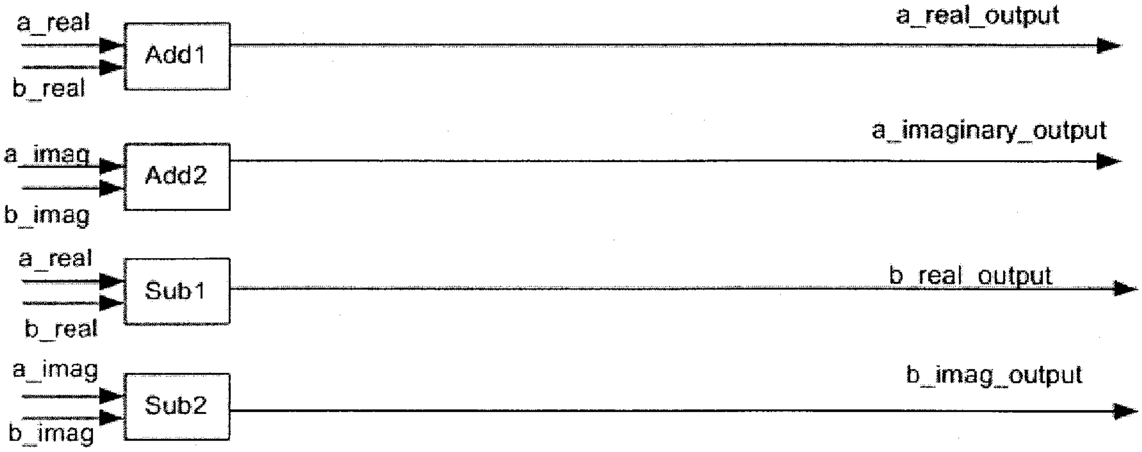

Figure 3.18: Block level Data-path for group3 for twiddle factor based architecture

3.8 Summary

In this chapter, we summarize the various aspects of design of the main blocks of the architecture we designed based on Twiddle-Factor-Based FFT algorithm [1]. The main blocks include the Random Access Memory Address generator, Read Only Memory Address generator, the *1024*-words Random Access Memory (RAM), *512*-words Read Only Memory (ROM), Address decoders, Temporary register bank and Data-path. A few multiplexers are also designed and used in the architecture mainly used to regulate the flow of various signals across the blocks. Signal clarifications are done using Resolution function blocks that are designed at necessary locations. This is done mainly because few of the signals have either multiple sources or destination thereby necessitating resolution functional blocks to resolve the signal conflicts. The resolution functions subsequently

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

increase the complexity of the architecture which can be seen from the operational speed of the design.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 4

ARCHITECTURAL SIMULATION RESULTS AND DISCUSSION

4.1 Output Simulation Results

From chapter 3, we can obtain the architectural design of the various blocks that are used in our proposed FFT processor. All the blocks are designed using VHDL. In this chapter, we get to simulate the various blocks that were previously designed. By simulation, we obtain the output waveforms for the blocks for different input signals under varying control signal environments. We shall also determine numerical values of some important design parameters such as clock frequency, number of arithmetic operations and hence number of arithmetic operators required to meet the target time constraint and the operational speed of the designed processor. During this discussion we shall come across some of the main advantages as well as drawbacks the design has and also some important challenges that require special attention for future designers.

4.2 RAM Address Generation Block Parameters

This is the logic VHDL block designed to generate bit sequences that represent addresses of memory locations in the Random Access Memory (RAM) from where complex data values needed to perform DFT butterfly operations are accessed. The logic for the address generation is based on [1] [10] and the pseudo codes for both the (log_2N-1) stages and $(log_2N)^{th}$ stage are given in sections (3.4.2) and (3.5.2) respectively. In blocks that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

represent the $(log_2N\text{-}1)$ stages, q_a and q_b represent the two output ports through which the address values are output. Both ports output *10*-bit binary sequence. For the $(log_2N)^{th}$ stage computation, q_{addr} and q_b form the two ports outputting address values to the temporary registers and RAM respectively. While q_{addr} outputs *9*-bit address, q_b outputs a *10*-bit address sequence. The difference in the number of bits between the two ports is mainly because of the difference in the size of the two different memory units. The *enable signal* is used to enable or disable the block. Disabling the block (*enable signal* ='1') retains the previous values of the output ports. The other signals that affect the functioning of this block are the *clear*. The *clear* signals represent the clear function which when made *1*, erases the values at the output ports by assigning them to high impedance value *z*. Once the $(log_2N\text{-}1)$ stages are executed, the blocks corresponding to $(log_2N)^{th}$ stage get activated. Each of the blocks designed get activated one after the other with one block triggering the successive block. As a result, the inherent delay within the RAM address generation block for generating successive address is *2 (20nsec)* to *4 (40 nsec)* clock cycles (with *2* being the dominant delay factor) depending on which loop the control is currently in and in which loop the next address is to be generated. The inherent delay is mainly attributed to the non-pipelined nature of the design. As a result, each loop within the address generation unit gets initiated by its preceding loop, which then remains idle till all the succeeding loops complete their execution. As a result of this idle nature, we encounter more delay than in a pipelined structure wherein every block operates at every clock cycle.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3 ROM Address Generation Block Parameters

The block signifies address generation for accessing twiddle factors needed for DFT butterfly operations. The output port generates the *9*-bit address needed to access the twiddle factors from the various *512* ROM locations. Once the address is output, a triggering signal initiates the ROM block to generate the data value corresponding to the address from the ROM address generation block. The real and the imaginary data values are output from separate ROM address blocks. The *clear* represents erasing signal and has the same functionality as described in section 4.2. The ROM block is designed to generate address without any delay and hence it can generate address every clock cycle as per the requirement. There is no inherent delay in this block.

4.4 Random Access Memory Parameters

The RAM as described is a *32*-bit, *1024*-words block. The *10*-bit address generated for data access from RAM is decoded by the *ram_decoder*, to find the exact location from where data is to be read out or stored. Once the address gets decoded, data from that corresponding location gets read out or data gets stored in depending on whether *read* or *write* operation is specified. The *read* and *write* signal get activated automatically by the controller design, which activates the *read* signal as soon as address gets generated by the RAM address generator. While the *read* signal is *high* or *1*, *write* signal remains active *low* or *0* and vice-versa. There is no inherent delay in this block and hence once address is placed for decoding, data from RAM can be accessed within the same clock cycle.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.5 Read Only Memory Block Parameters

The ROM, which stores the twiddle factors, is a *32*-bit, *512*-locations block. The input to the ROM is a *9*-bit address sequence from the ROM address generator block. Once the address is present at the input port, the data stored in that address location is loaded on to the output port. The output is a *32*-bit twiddle factor data. As soon as the address from the ROM memory controller gets generated, the twiddle factor value from the corresponding location is loaded onto the output port. There is no inherent delay in this block and hence data can be accessed within one clock cycle (10nsec).

4.6 Data-Path Block Parameters

The data-path serves as the arithmetic back bone of the processor under design. From sections 3.7.2, 3.7.3 and 3.7.4, data-path seems to vary between the three sections with some components being common among the three. The addition and subtraction operations in equations (3.7.1) and (3.7.2) respectively are common while the varying component is the multiplication with the twiddle factor. Figure (3.13) clearly specifies the various blocks that are designed to minimize the number of multiplication operations.

Assuming the arithmetic blocks getting executed one after the other, one block at a time, it takes *5* clock cycles (50 nsec) excluding the operation of loading the data into the data-path from memory unit and writing them back into memory unit after data-path operation, which takes one clock cycle each to implement any DFT butterfly arithmetic operation in group 1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Group 2 data-path does not involve any multiplication as only swapping between real and imaginary terms are necessary. Hence on the whole it takes *2* clock cycles (20nsec) excluding the operation of loading the data into the data-path from memory unit and writing them back into memory unit after data-path operation, which takes one clock cycle each to implement any DFT butterfly arithmetic operation in group 2.

Group3 data-path is probably the simplest among the three as it only involves multiplication with *1*, which can just be neglected. Hence other than the addition and subtraction operations specified in equations (3.7.1) and (3.7.2), no other arithmetic operations are required. Hence the number of clock cycle's butterflies in this group take to complete the arithmetic operations is just *1* (10 nsec) excluding the operation of loading the data into the data-path from memory unit and writing them back into memory unit after data-path operation, which takes one clock cycle each to implement any DFT butterfly arithmetic operation in group 2.

4.7 Clock Generation and Frequency Calculation

As seen in section 3.2, our architecture is designed to target two main applications namely HDTV and OFDM transceiver. While HDTV FFT necessitates *8192*-points in *896* microseconds, OFDM transceiver requires *1024*-points execution within *51* microseconds. Of the two target timings, OFDM transceiver's *51*micro-seconds is of the shortest duration and hence we need to design our architecture to satisfy *1024*-point execution within *51* micro-seconds which if satisfied would also help achieve *8192*-points FFT execution in *896* micro-seconds. In order to achieve the timing target, we

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

need to first determine the clock operational frequency. The clock is basically used to synchronize the various blocks that are designed and used in the architecture at any given instant. For example, if two distinct blocks need to get executed at the same instant, we need to ensure that they start and end their executions exactly at the same instant and not at differing time intervals. To ensure this, we need a signal that can synchronize all the blocks of the architecture. To ensure synchronization, we need to make sure that the blocks gets enabled or disabled either at the rising or falling edge of the clock pulse. The clock frequency thus determines how frequently the clock signal rises or falls which ultimately determines the number of calculations that can be performed within the required time frame. On the other hand, we can also determine the clock frequency based on the number of operations if known. In our design, we determine the clock frequency based on our target timing of *51* microseconds to perform *1024* points FFT operations. We thus need to determine the number of operations that are involved in *1024* points FFT calculations. To determine the clock frequency, we do the following calculations.

(1) The $(log_2N)^{th}$ stage of operation based on Twiddle Factor decomposition involves no twiddle factor multiplications. Assuming a pipelined structure wherein all the blocks of a given design have the ability to operate simultaneously and no block needs to be idle and wait for data from its previous block. This enables us to generate an output every clock cycle. This assumes one clock cycle period for every butterfly operation through the data-path as described in section 3.7.4.

Total number of butterflies involved in the $(log_2N)^{th}$ stage = $(N-1)$. (4.7.1)

Hence, total number of clock cycles needed assuming one clock cycle for every butterfly operation = $(N-1)$ (4.7.2)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(2) The $(\log_2 N-1)^{th}$ stage involves butterflies with imaginary term '*j*' twiddle factor multiplication.

Total number of butterflies involved in the $(\log_2 N-1)^{th}$ stage = $((N/2)-1)$. Hence, total number of clock cycles needed assuming one clock cycle for every butterfly operation = $((N/2)-1)$ (4.7.3)

(3) The first $(\log_2 N-2)$ stages involve complex multiplications.

Total number of butterflies involved in the $(\log_2 N-2)$ stages

is $\sum_{i=1}^{(\log_2 N-2)} (N/2^{(i+1)})(2^i-1)$ (4.7.4)

Hence, total number of clock cycles needed assuming one clock cycle for every

butterfly operation = $\sum_{i=1}^{(\log_2 N-2)} (N/2^{(i+1)})(2^i-1)$ (4.7.5)

Substituting $N=1024$ in (1), (2) and (3), the total number of butterfly operations involved = *5120*. Hence assuming one clock cycle for every butterfly operation, the total number of clock cycles necessary for $N=1024$ is *5120*. Since our target timing is *51* microseconds, to implement *5120* clock cycles within *51* microseconds, each clock pulse should be of time width *0.02* microseconds. Hence the corresponding clock frequency becomes *100* Mega-hertz.

4.8 Memory Access Feature

The architecture under design is based on Twiddle-factor FFT algorithm [1] [10], which claims to reduce the number of memory access when compared to conventional FFT algorithms. Since our architecture maps the twiddle factor based FFT algorithm, it is also

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

expected to reduce the memory access when compared to other architectures targeting conventional FFT algorithms.

While describing the decomposition procedure for Twiddle-Factor Based FFT algorithm, we showed that once a twiddle factor gets loaded from ROM on to the data-path, no other twiddle factor is loaded until all the butterflies that use this particular twiddle factor get computed. In other words, every twiddle factor gets loaded only once during the entire architecture implementation. The number of twiddle factors that are present in the decomposition of an *N*-point FFT is $(N/2)$. Hence based on the above explanation, the total number of times the twiddle factors would be accessed from the ROM where they are stored is $(N/2)$.

In case of conventional FFT algorithms namely DIF or DIT FFT algorithms, a twiddle factor gets loaded from ROM every time a butterfly gets accessed. The total number of butterfly DFT operations that get computed for an *N*-point input is $(N/2)\ log_2N$. Hence for any conventional FFT algorithms, the total number of times the twiddle factors would be accessed from the ROM where they are stored is $(N/2)\ log_2N$.

In terms of our architectural design, we can see that from loop 2 execution given in section (3.4.1), the total number of times loop 2 gets executed is given by

$$(N/2^{(loop1\ index\ +1)}) \tag{4.8.1}$$

where loop1 varies from 1 till (log_2N-1)

In addition to (4.8.1), the final $(log_2N)^{th}$ stage, utilizes only W_N^0 factor.

Hence the total number of ROM memory access in our architecture is given as

$$(N/2^{(loop1\ index\ +1)}) + 1 \tag{4.8.1}$$

where loop1 varies from 1 till (log_2N-1)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Hence in terms of ROM access, our architecture seems to comply with the memory access reduction claim of [1] [10]. Thus for various values of N, the variation in ROM memory access of our architecture based on [1] [10] as compared to other architectures based on conventional FFT algorithms like DIT and DIF are in a graphical form in figure (4.1). In addition to the memory savings obtained through Read Only Memory (ROM), the twiddle factor based architecture can obtain additional memory savings through lesser Random Memory Access (RAM) accessing.

Computation of every butterfly DFT structure necessitates *4* RAM accesses two of which are for reading out the input data from RAM and the other two for storing back the computed result back on to the same locations in the RAM. Hence for any given N-point input conventional FFT algorithm and hence any architecture based on it, there are $2N\log_2 N$ RAM accesses since there are a total of $\frac{N}{2}\log_2 N$ number of butterfly operations each requiring *4* RAM accesses.

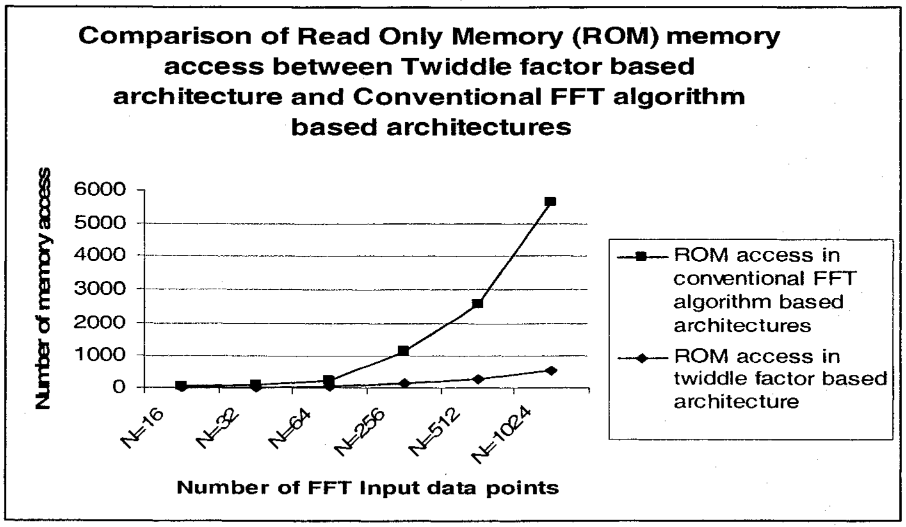

Figure 4.1: Graphical comparison for ROM access between twiddle factor based architecture and conventional DIT or DIF based architectures

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In case of twiddle factor based architecture, the final stage of decomposition involves butterflies utilizing W_N^0 as the only twiddle factor. There are *(N-1)* butterflies in the final stage of which the first $(\frac{N}{2}-1)$ butterflies have only *1* RAM access as all other three accessing is done from the temporary registers. The last $(\frac{N}{2})$ butterflies on the other hand have only *1* input accessing from RAM while both data at the output of the butterfly gets stored back on to the RAM instead of the temporary registers. This is because these set of butterflies form the last stage of operation and hence the final FFT data result needs to be accessed from RAM. Hence these $(\frac{N}{2})$ butterflies each have *3* RAM accessing. Hence the total RAM accessing from the last stage of decomposition is $(\frac{N}{2}-1)+\frac{3N}{2}=(2N-1)$. The butterflies in the remaining stages have *4* RAM access as usual. Hence, total number of RAM memory access for all stages is given as $4((\frac{N}{2}\log_2 N)-(N-1))+2N-1=(2N\log_2 N-2N+3)$. Hence, when compared to the *(2Nlog₂ N)* RAM access obtained from conventional DIT or DIF based architectures, $(2N\log_2 N-2N+3)$ seems to be a significant reduction of as much as *25%*. This directly has an impact on power consumption as it can be further reduced. Though temporary register accessing is still present, it is much faster when compared to the slow RAM accessing mainly due to its small sizing and its positioning closer to the data-path than the RAM. This reduction in RAM memory access is shown pictorially in the graph depicted in figure (4.2).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

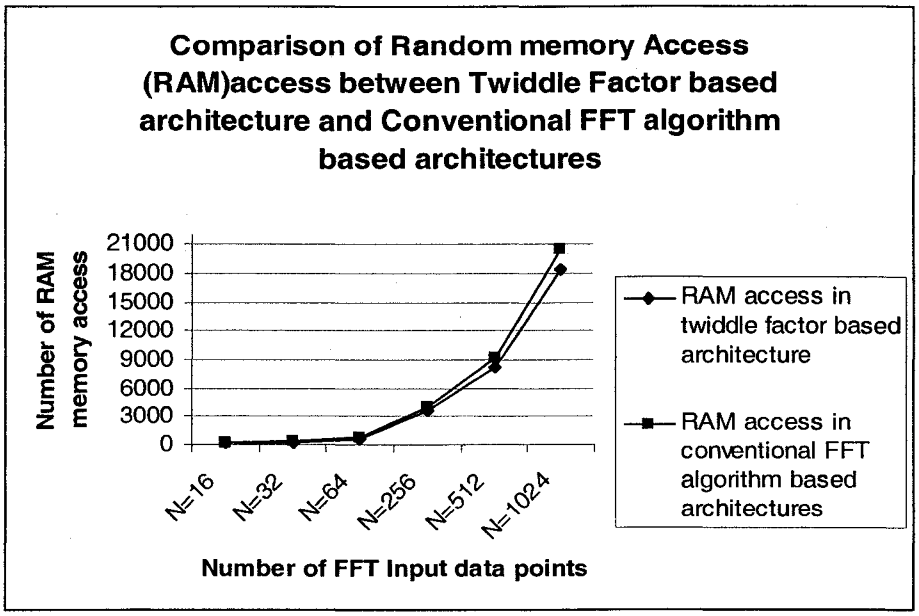

Figure 4.2: Graphical comparison for RAM access between twiddle factor based architecture and conventional DIT or DIF based architectures

4.9 Multiplier Operational Savings Using the Twiddle Factor Based Architecture

As seen from sections (3.7.2), (3.7.3) and (3.7.4), the entire decomposition operation of FFT based on Twiddle factors can be classified into three groups, two of which namely group2 and group3 do not involve any multiplication operations. As a result, the multipliers designed and used in the data-path can be disabled while butterflies from these two groups get computed. In conventional FFT algorithms, butterflies requiring no twiddle factor multiplications are available in every stage of decomposition along with other butterflies that do involve multiplications. Hence disabling the multipliers at different time instances becomes very complex and hence most architectures do not disable the multipliers at any instant [22], [23]. As a result, irrespective of whether a butterfly involves multiplication operation or not, the multiplier used in the data-path remains enabled thus consuming unnecessary clock cycles as well as power. In our

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

architecture, the butterflies involving no twiddle factor multiplications are classified into separate stages and hence it is easy to disable the multiplier across those stages.

Total number of butterfly operations where multiplier remains enabled in conventional FFT algorithm based architectures $= \frac{N}{2}\log_2 N$ (4.9.1)

Total number of butterfly operations where multiplier remains enabled in Twiddle factor based FFT architecture $= (\frac{N}{2}\log_2 N) - [(\frac{N}{2} - 1) + (N-1)]$

$$= (\frac{N}{2}\log_2 N - \frac{3N}{2} - 2) \qquad (4.9.2)$$

Equations (4.9.1) and (4.9.2) give a comparative analysis of the savings we attain in terms of number of times the multipliers get utilized effectively. A graphical representation is shown in figure (4.3).

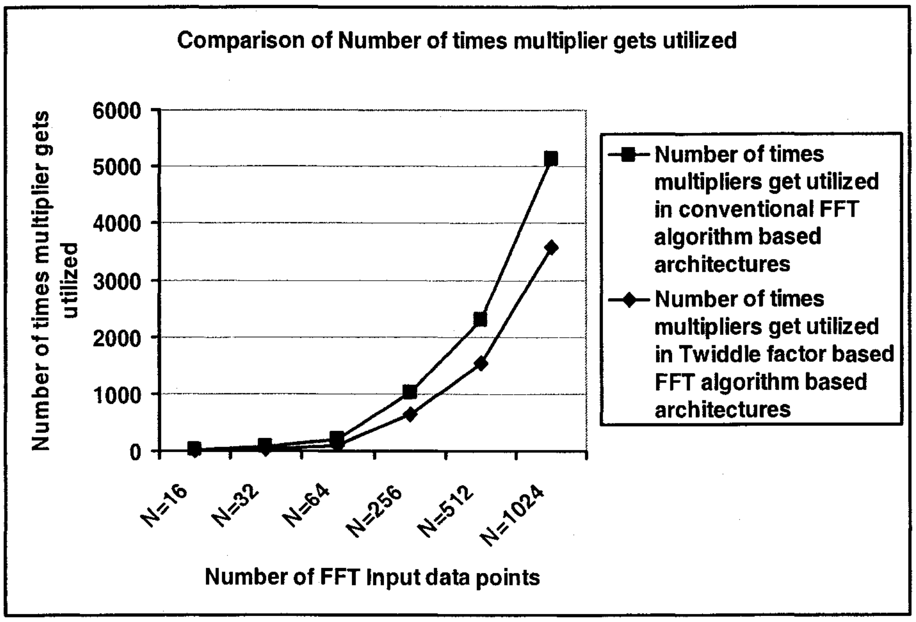

Figure 4.3: Graphical comparison for multiplier operation usage between twiddle factor based architecture and conventional DIT or DIF based architectures

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.10 Design Issues and Challenges

The previous sections described the computational complexity, operational speed in terms of number of clock cycles and also the inherent delays present in each of the architecture blocks. We observed that pipelining a structure could result in a shorter execution time than a non-pipelined structure as pipelining involves every block operating at every clock cycle (when one block computes on one set of data, its preceding block computes on the next set of data while the succeeding block works on the previous set of data that was earlier fetched. That is if block *(x+1)* works on data *(y+1)*, then at the same clock cycle, block *x*, which precedes block *(x+1)* works on data *(y+2)* while the succeeding block *(x+2)* works on data *y*). The non-pipelined structure on the other hand, involves enabling only one block at a time while the succeeding as well as the preceding blocks remains idle till the current block completes its execution.

In case of our architectural design, we have managed to pipeline every block except for the RAM Address generation block. The main challenge involved in pipelining this block is the presence of nested loop structure (a loop within another loop). The presence of nested loops in the address generation logic design makes pipelining a non-pragmatic and extremely challenging task. Since nested loops involve transfer of control back from the outermost loop to the inner most and then all the way back to the outer loop, keeping track of the current values of registers and variables in each loop becomes tedious. Consequently, we were not able to pipeline RAM Address generation block while all other blocks could be pipelined. Non-pipelining of this block resulted in its inherent delay being apparent while FFT butterflies get computed. From section 4.2, we know that the delay caused by RAM Address generation block is *2* to *4* clock cycles of which *2* is the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

most dominant delay factor. We also saw that all the other blocks designed and implemented in our architecture do not involve any delay is the most dominant delay factor. We also saw that all the other blocks designed and implemented in our architecture do not involve any delay. Hence, we encounter *2*-clock pulse delay for every butterfly computed.

Total number of butterflies for an N-point FFT = $\frac{N}{2}\log_2 N$ (4.10.1)

Hence for $N = 1024$, total number of butterflies computed = 5120 (4.10.2)

From section 4.7, the clock frequency utilized in our architecture is found to be *100* MHz. Hence the timing width of each pulse obtained by taking the inverse value of the clock frequency = *10* nanoseconds. (4.10.3)

Hence assuming *2*-clock cycle in calculating every butterfly, it takes *10240* clock cycles to calculate all the 5120 butterflies. Since each clock pulse is of *10* nanoseconds width, it takes a total of *102400* nanoseconds or *102.40* microseconds to compute *1024* input FFT using twiddle factor-based algorithm. Since the RAM Address generation block cannot generate address every clock cycle, the data-path that basically requires data from those corresponding address locations to compute the butterflies will not be able to do so. Since the RAM Address generation block could not be pipelined, the data-path follows suit though it is simple to pipeline this block. Hence in addition to the *2*-clock cycle we encounter due to RAM Address generation block delay, we also have data-path delay included.

The data-path delay as discussed earlier is determined based on whether complex multiplication is involved or not. If complex multiplication is involved, then based on figure (3.16), we have 5 stages of computation and hence the delay is *5* clock cycles. In

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

case of no complex multiplication involvement, then the delay is only *2* clock cycles. In addition to these delays, there are also delays due to resolution functions that add up to additional *2* clock cycles for every butterfly computation. The total clock cycles required computing a butterfly involving no complex multiplication is given as *6* clock cycles that includes the RAM address generation delay, data-path delay and additional delays due to resolution functions.

The total number of butterflies for N = 1024 which involve complex multiplication is *1534*.

Hence, the total number of clock cycles required computing 3586 butterflies

$$= 1534 * 6 = 9204 \qquad (4.10.4)$$

Similarly, the total clock cycles required computing a butterfly involving complex multiplication is given as *9* clock cycles that includes the RAM address generation delay, data-path delay and additional delays due to resolution functions.

The total number of butterflies for N = 1024 which involve complex multiplication is *3586*.

Hence, the total number of clock cycles required computing 3586 butterflies

$$= 3586 * 9 = 32274 \qquad (4.10.5)$$

Combining equations (4.10.4) and (4.10.5), we can determine the total number of clock cycles involved in computing *1024* input FFT based on twiddle factor architecture is *41478*.

With a *10* nanoseconds clock pulse width, the total time taken to execute *1024* points input FFT based on twiddle factor architecture is calculated to be *414.78* microseconds.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.11 Architectural Salient Features and Drawbacks

The biggest advantage the twiddle factor based FFT architecture has over other conventional DIT or DIF based architectures is in terms of memory access which ultimately results in power savings.

(1) Our architectural design has *10* times lesser Read Only Memory (ROM) access when compared to DIT or DIF based architectures.

(2) The Random Access Memory (RAM) access is reduced by as much as *25%* in our design as compared to other previous architectures.

(3) Our design has *1.45* times lesser number of operations taking into account the total number of memory accesses and multiplication computations.

(4) Clock gating where unused blocks are disabled in order to reduce unwanted power consumption while the blocks remain idle is utilized in our design with the help of resolution blocks that enable and disable different blocks.

(5) As a result of the reduction in the number of operations performed, for a *1024* point FFT based on twiddle factor design, power reduction up to *31.25%* in terms of memory access and arithmetic blocks usage is expected.

The main drawback that this design suffers is in the time duration taken to compute *1024*-point operations. This is mainly attributed to the non-pipelined nature of our design. When compared to our initial target application of OFDM transceiver which necessitates a *1024*-point FFT to get computed within *51* microseconds. Hence we can observe that our architectural design despite its advantages in terms of computational complexity and power savings, is *8* times more time consuming in computing *1024*-point FFT as required by the standardized OFDM transceiver.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 5

CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

5.1 Summary of Work

Our entire work focuses on one-to-one mapping of the Twiddle factor based FFT algorithm described in [1] [10] on to hardware blocks so as to extract the maximum benefits derived from the algorithm. Based on the algorithm, various logic blocks were designed and simulated using VHDL. Memory savings in terms of memory access have been mapped on to the blocks from the algorithm. The architecture makes minimum utilization of arithmetic hardware blocks, especially the multipliers. Though this architecture is advantageous in terms of power savings when compared to other previous architectures, the major drawback it faces is in terms of the time it takes to complete its required execution. This is attributed to the non-pipelined nature of its design. Thus *1024*-points FFT can be computed using the Twiddle factor based FFT architecture in *414.78* microseconds with up to *1.45* times lesser number of operations resulting in power savings of up to *31.25%* is expected. All necessary blocks have been designed and simulated in Active-HDL 6.3.

5.2 Suggestions for the Future

As seen in Chapter 4, the main factor that delimits the efficient usage of the twiddle factor based architecture is the execution time. Consequently, we have explained that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pipelining the RAM Address generation block, which automatically enables pipelining of other blocks like the data-path is the most important way of reducing the timing problem. An efficient way of pipelining nested loop structures need to be developed in order to reduce the large execution time for processors.

It is a well-known fact that pipelining make architecture attain their full efficiency and help achieve a higher throughput though with some initial latency. One other important method to reduce the delay of operation is to utilize high-speed hardware blocks, especially the arithmetic blocks like the adders and multipliers. The type of complex multiplier used for our data-path is shown in figure (3.13). It can be seen that this type of complex multiplier design uses more addition operations than real multiplications. Hence usage of fast adders becomes a necessity in-order to maximize efficiency and increase throughput. From [9], it can be inferred that for arithmetic adders involving bit widths of *24* or higher, Carry Look Ahead (CLA) adders are probably the fastest when compared to Ripple Carry Adders (RCA) or Carry Save Adders (CSA) and many others. Hence, it is highly recommended to design CLA adders while seeking improvement in the current design. For implementing real multipliers, Array Based multipliers that are widely and most commonly used over various logic designs can be utilized.

Power analysis to estimate the power efficiency of the twiddle factor based architecture with all the above said improvements implemented should be carried out using power estimation CAD tools. This would give an accurate picture of the effectiveness of the twiddle factor based algorithm [1] [10] over conventional DIF or DIT algorithms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Further savings in terms of memory access, which is the main highlight of our design, can be effectively obtained by using smart memories like Cache and flash memories. These memory units, unlike the Random Access Memory are small in size and are comparatively much more efficient in terms of accessing time. As these memories are smaller in size and are placed closer to the processor than the Ram, which consequently enables them to be much quicker than conventional large-scale memories. These memories help in reducing frequent RAM or ROM access and in turn increase the effective power saving capacity of our design.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX

VHDL DESIGN OF VARIOUS ARCHITECTURAL BLOCKS

The VHDL design and implementation of various Twiddle factor based FFT architectural blocks described in Chapters 3 and 4 are pictorially represented in this Appendix.

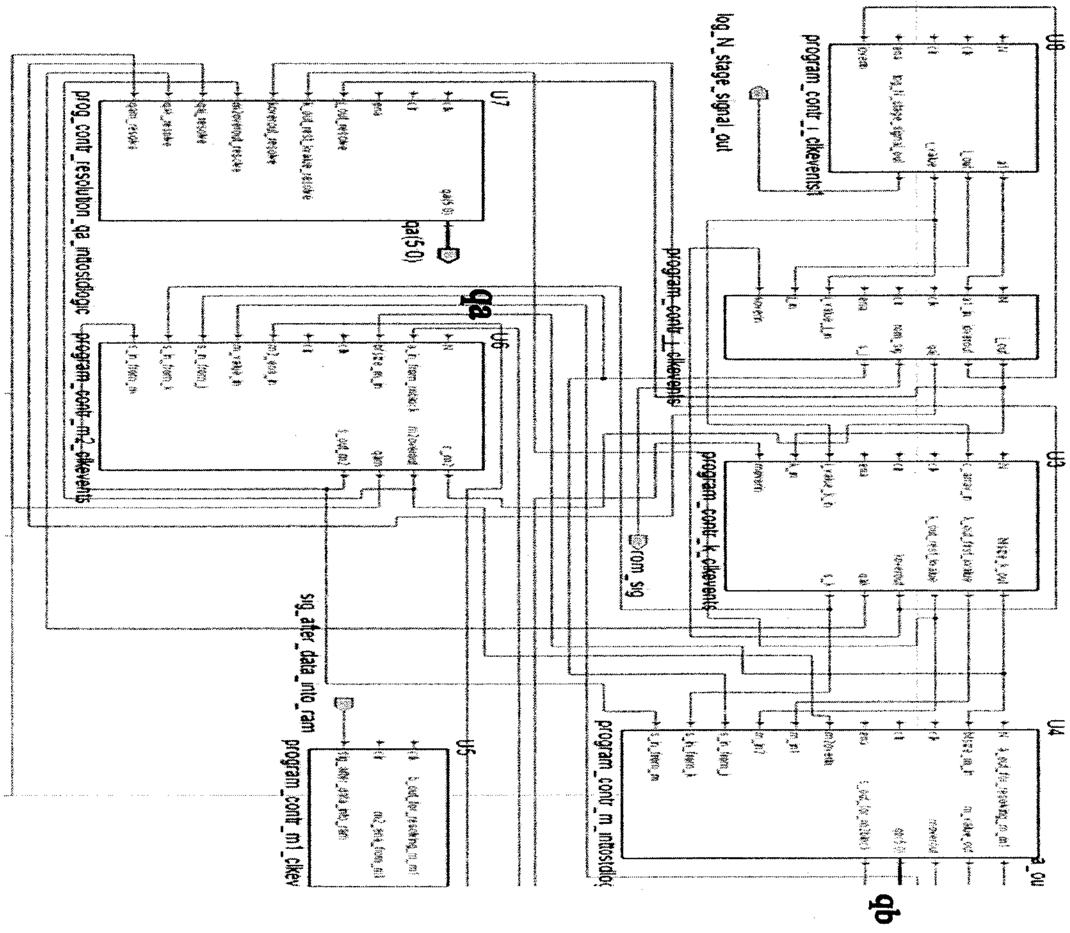
U8
log_N_stage_signal_out
program_contr_i_clkevents
U7
qa(5:0)
qa
prog_contr_resoluton_qa_mtostdlogic
program_contr_m2_clkevents
U6
program_contr_j_clkevents
U3
program_contr_k_clkevents
rom_sig
U4
program_contr_m_mtostdlogic
qb
U5
sig_after_data_into_ram
program_contr_m1_clkev

Figure A-I1: VHDL block representation of $(log_2N\text{-}1)$ stages RAM Address generation block based on Twiddle factor decomposition algorithm generating *10*-bit RAM address sequences qa and qb

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

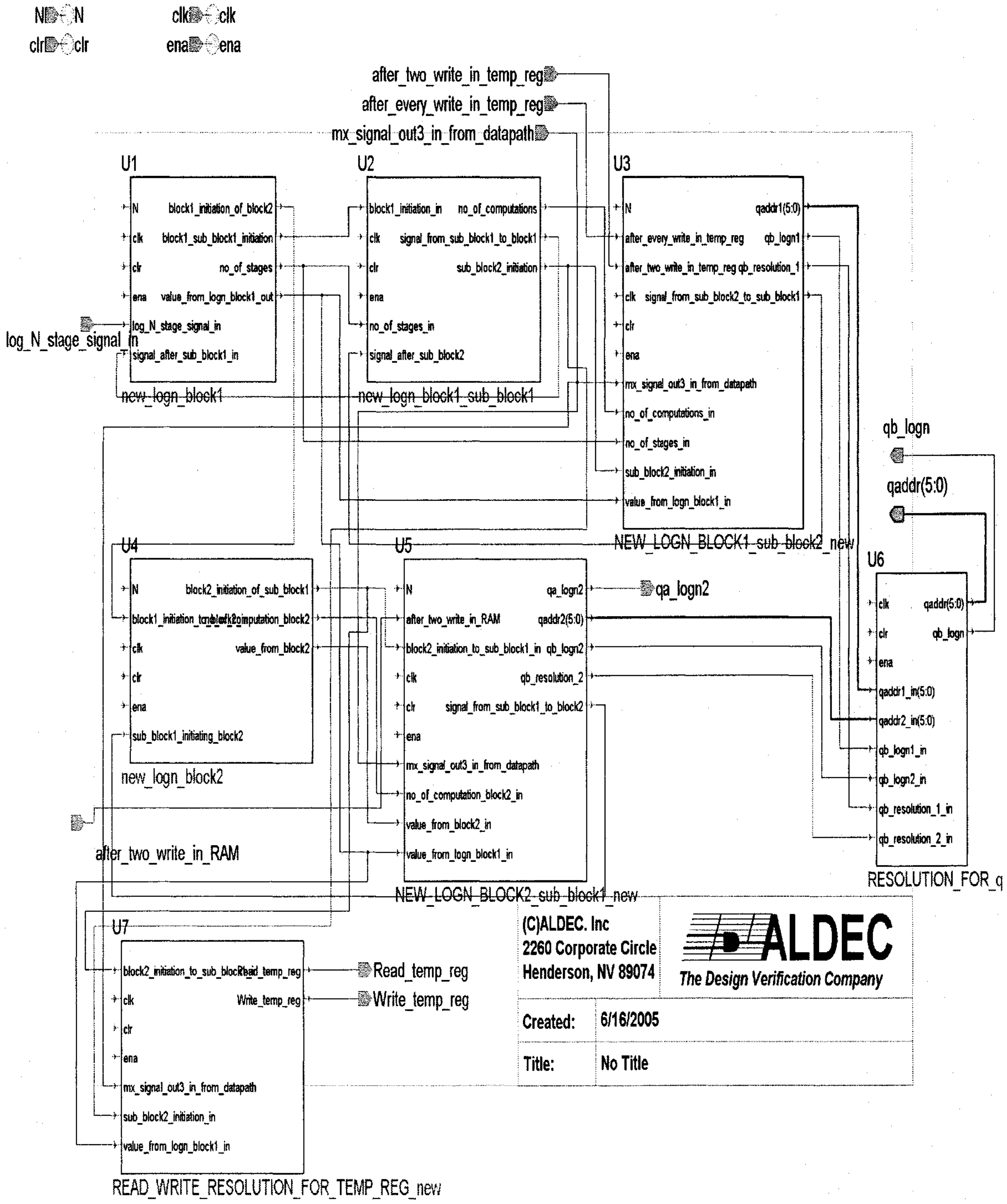

Figure A-I2: VHDL block representation of the $(log_2N)^{th}$ stage RAM Address generation block based on Twiddle factor decomposition algorithm generating *10*-bit RAM address sequences qa_logn2 and qb_logn and *6*-bit Temporary register bank address qaddr

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

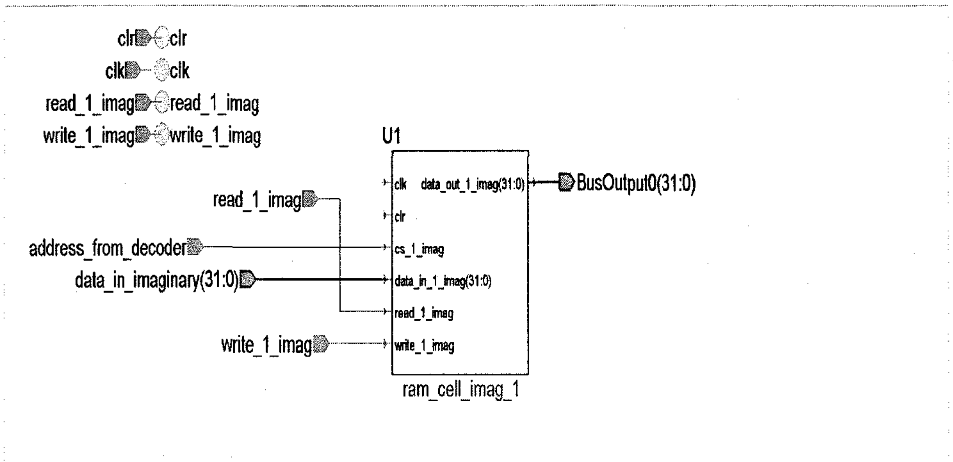

Figure A-I3: VHDL design of a *1*-word RAM cell for storing imaginary part of data.

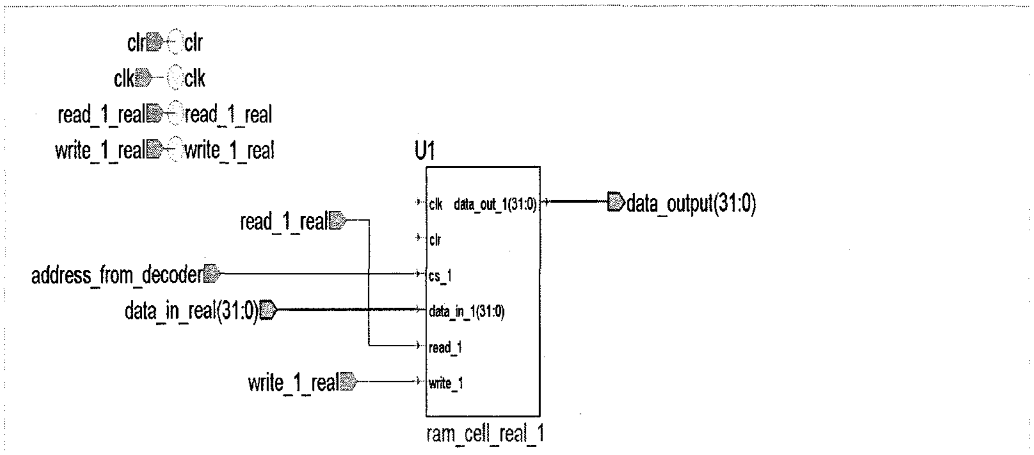

Figure A-I4: VHDL design of a *1*-word RAM cell for storing real part of data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

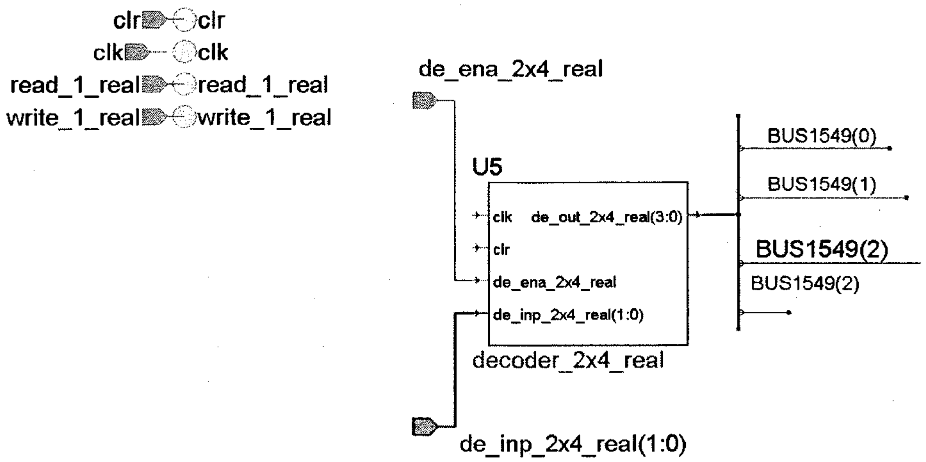

Figure A-I5: VHDL design of a 2x4 Decoder block choosing one among four outputs based on a *2*-bit input. The input port is given as *de_inp_2x4_real(1:0)* and output ports are represented by *de_out_2x4_real(3:0)*

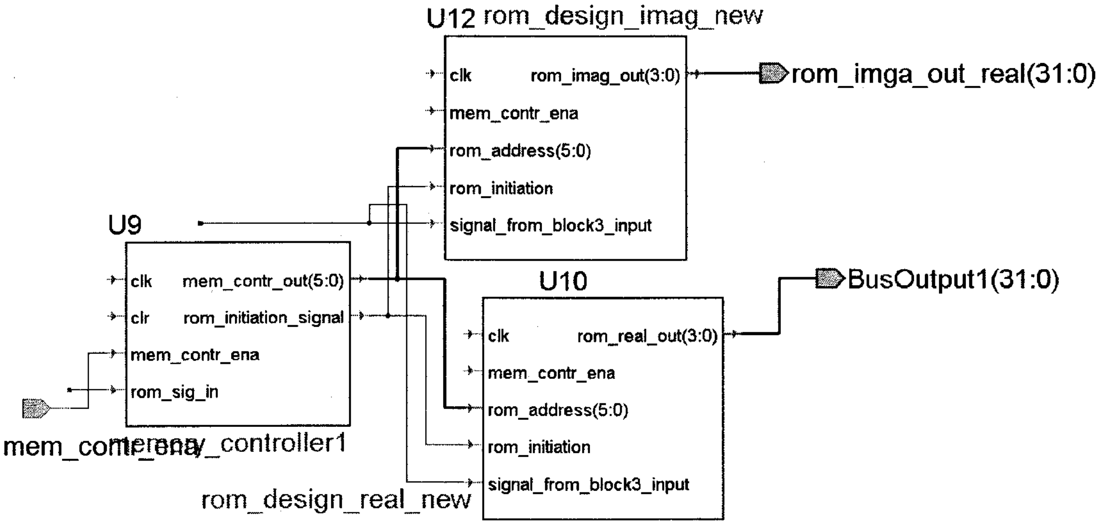

Figure A-I6: VHDL design of a *64*-words ROM and a decoder along with interconnects outputting *32*-bits real and imaginary twiddle factors

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

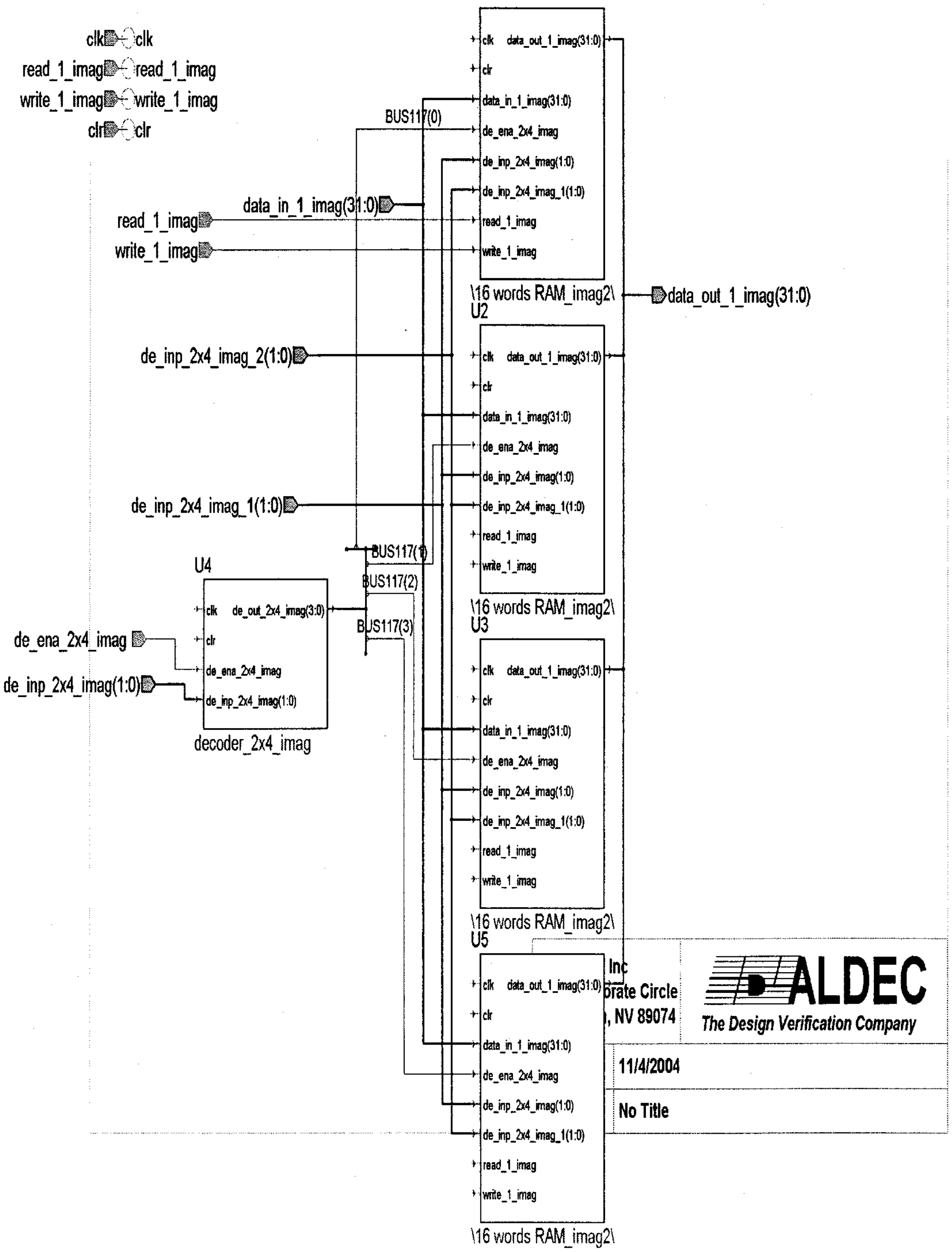

Figure A-I7: VHDL design of 64-words RAM using four *16*-words RAM cells and a decoder along with interconnects outputting *32*-bits real and imaginary data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

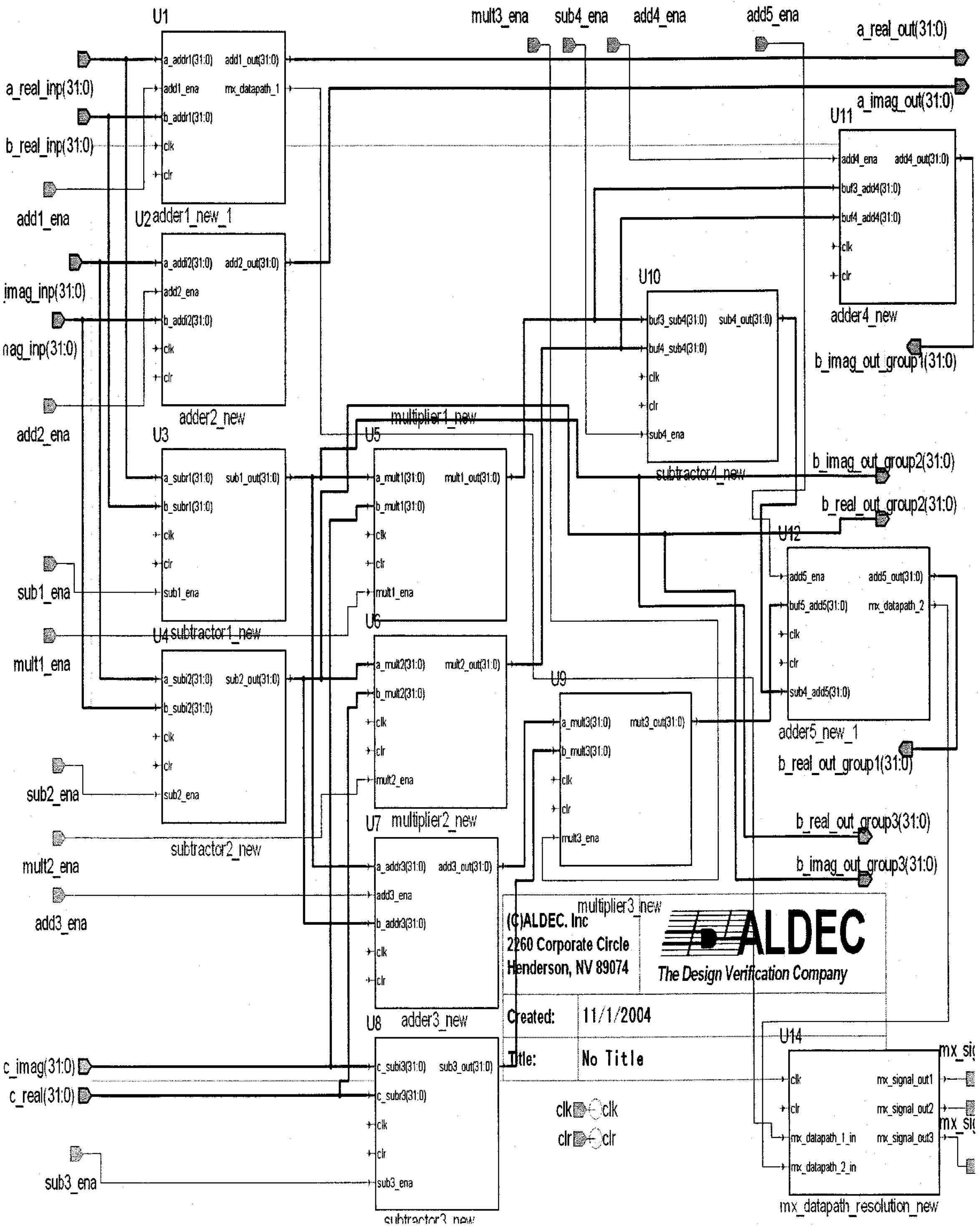

Figure A-I8: VHDL design block showing the data-paths for group1, group2 and group3. The output signals for each group are represented by *b_real_out_groupX* and *b_imaginary_out_groupX* with *X* representing individual group numbers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

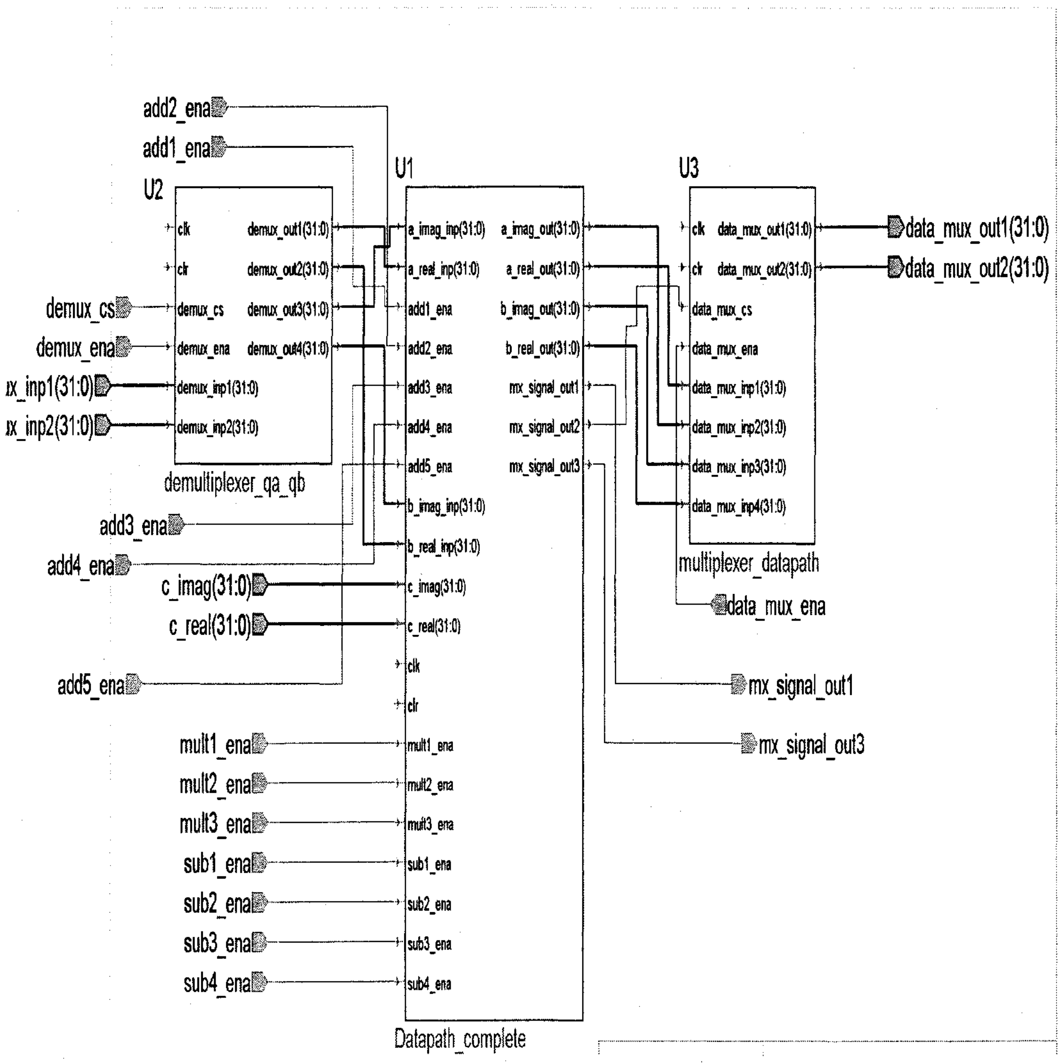

Figure A-I9: VHDL design block showing the resolution function designed for the data-path to choose between the three data-path groups and also for determining the *Read* and *Write* operations of the RAM block and other resolution blocks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

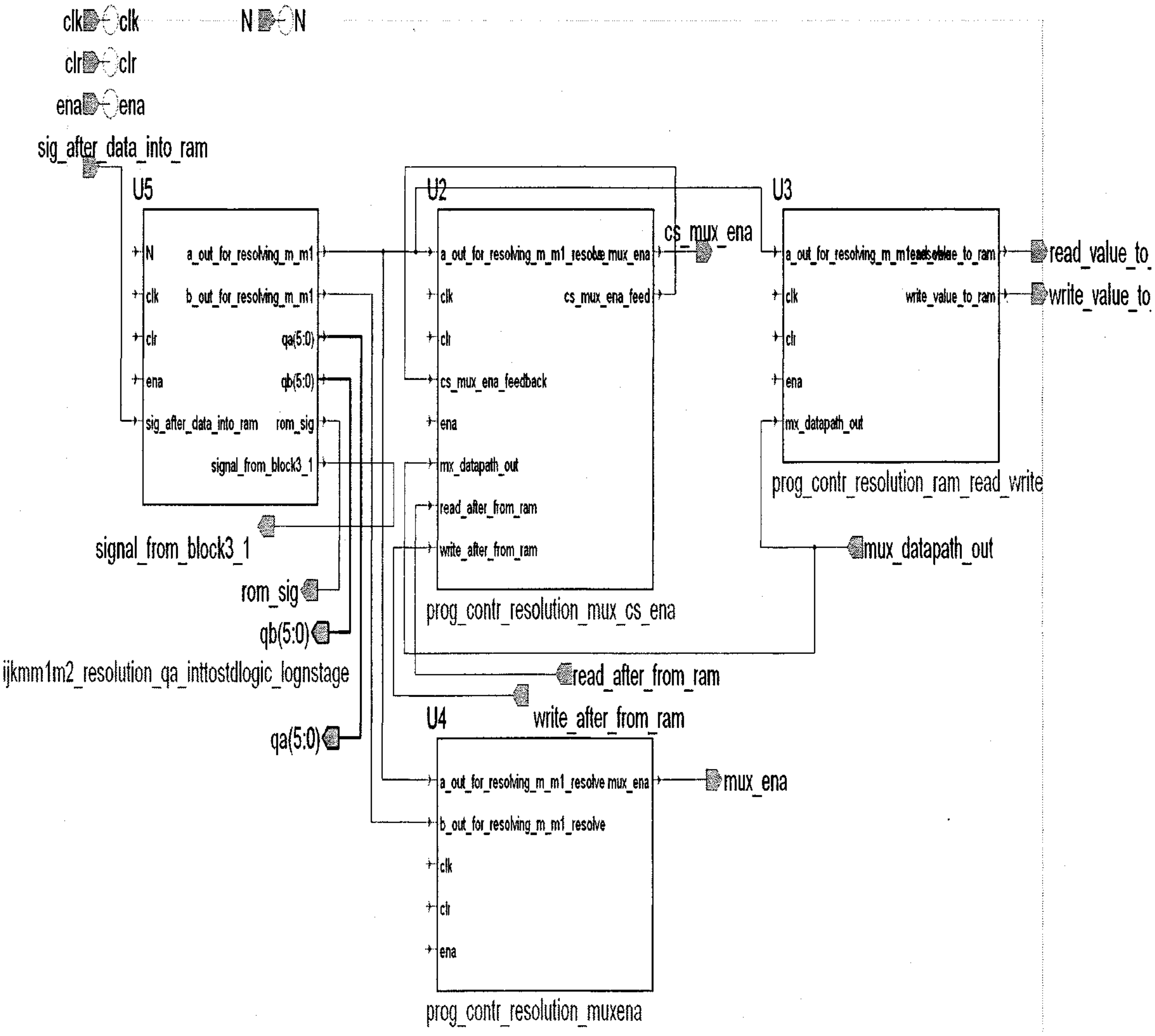

Figure A-I10: VHDL design block showing RAM Address generation block with resolution functions that aid in enabling and disabling various signals at different instances of time

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

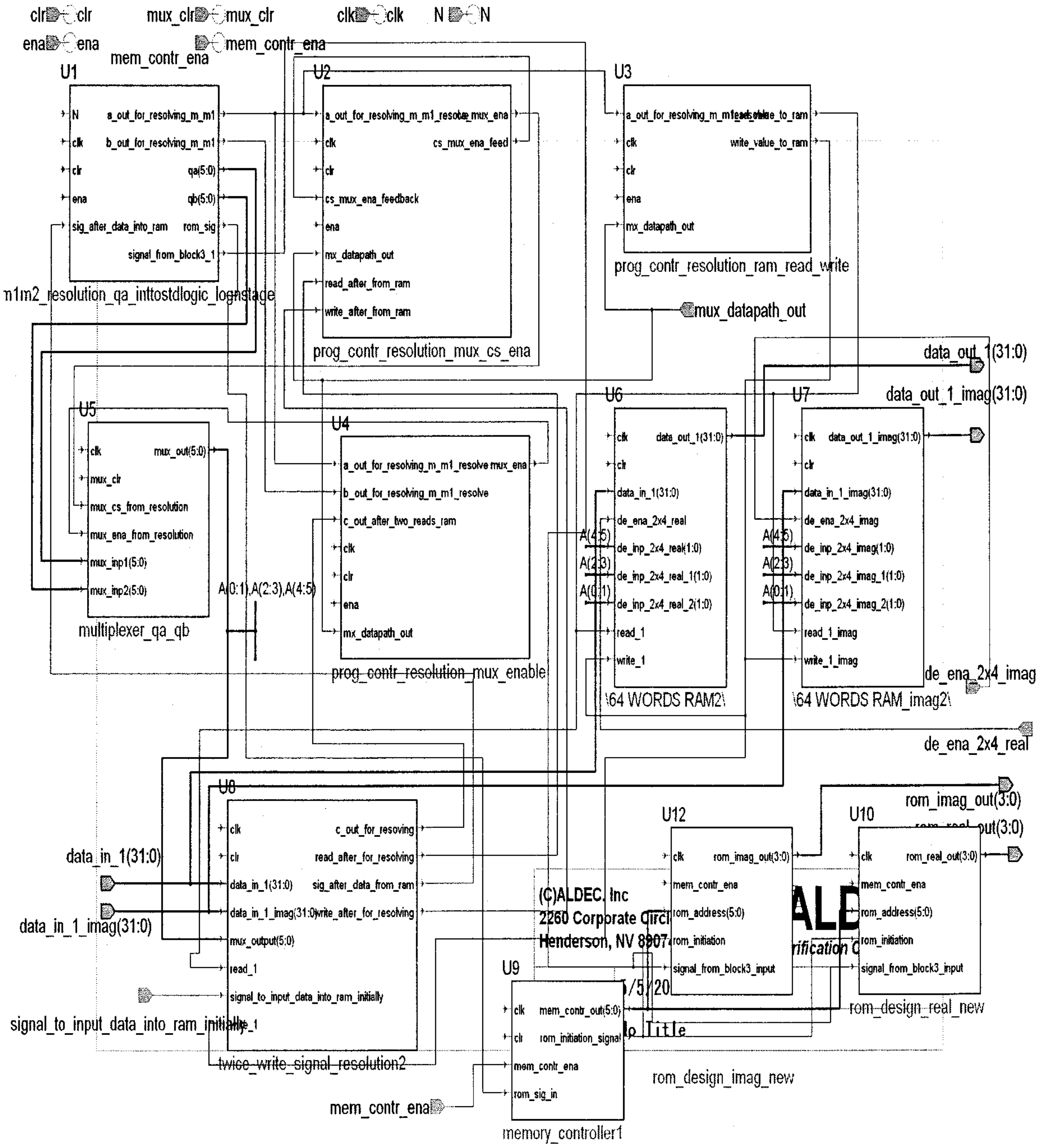

Figure A-I11: VHDL design block showing RAM Address generation block, the RAM blocks, the ROM memory controller block, the ROM blocks and all the other interconnects

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

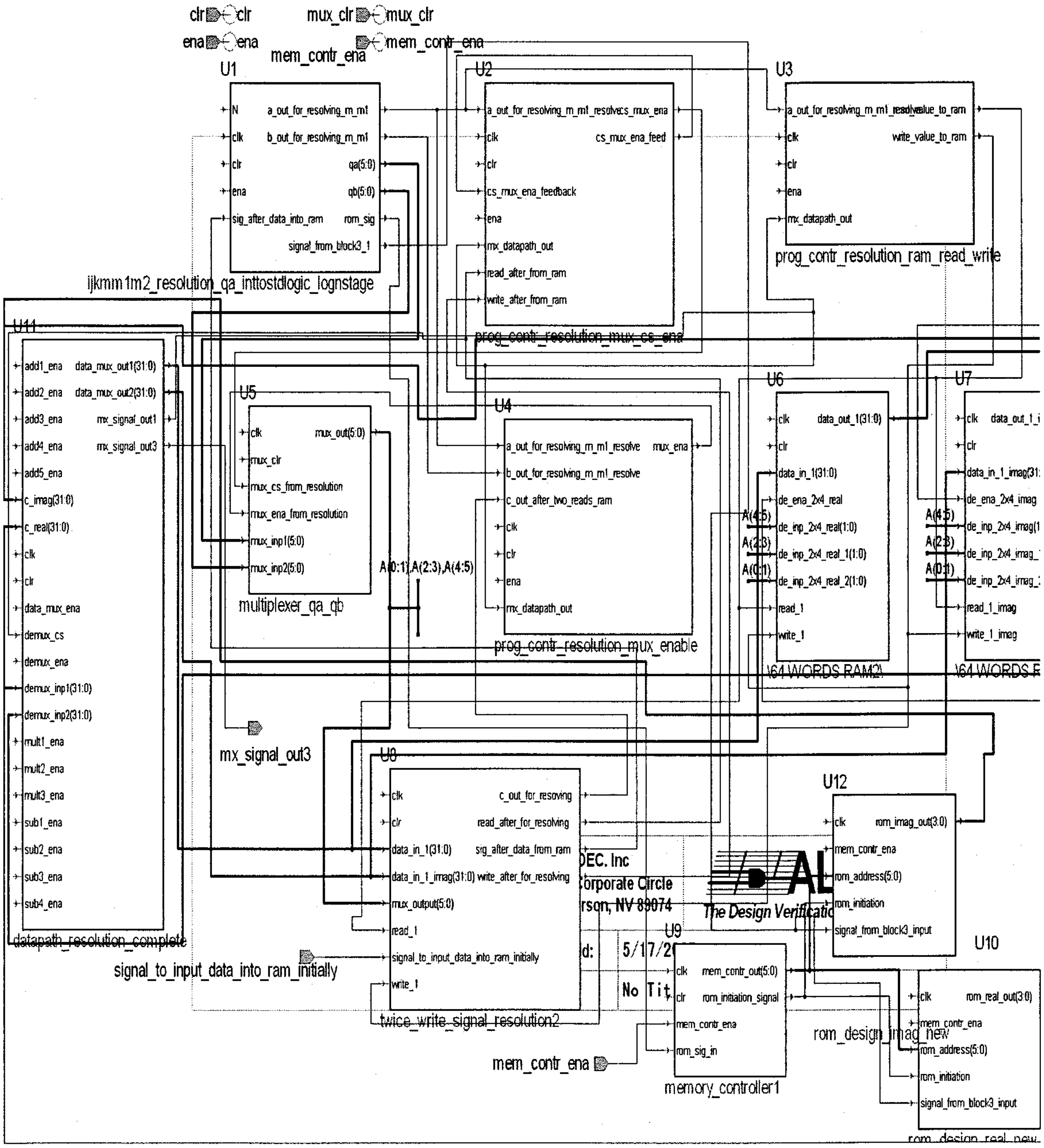

Figure A-I12: VHDL design block partially showing interconnection between all the blocks of the Twiddle factor based FFT architecture.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

[1] Yingtao Jiang, Ting Zhou, Yiyan Tang and Yuke Wang , "Twiddle-Factor-Based FFT Algorithm with Reduced Memory Access", In the *Proceedings of the International Parallel and Distributed Processing Symposium*, IEEE, 2002.

[2] Bevan M.Baas, "An Approach to Low-Power, High-Performance, Fast Fourier Transform Processor Design", Ph.D. dissertation, Stanford University, Stanford, CA, February 1999.

[3] Smith, Steven W., "The Scientist and Engineer's Guide to Digital Signal Processing", 2nd edition. San Diego: California Technical Publishing, 1999. ISBN 0-9660176-3-3.

[4] Alan V.Oppenheim and Ronald W.Sschafer, "Digital Signal Processing", October 2000. ISBN-81-203-0532-9

[5] Brown, J. W. and Churchill, R. V., "Fourier Series and Boundary Value Problems", *5th ed.* New York: McGraw-Hill, 1993.

[6] Brigham E.O., "The Fast Fourier Transform and Its Applications", Prentice-Hall, Englewood Cliffs, NJ, 1988.

[7] Julius O.Smith III, "Mathematics of the Discrete Fourier Transform (DFT), with Music and Audio Applications", W3K Publishing, 2003. ISBN 0-9745607-0-7.

[8] John Proakis, Dimitris Manolakis, "Digital Signal Processing - Principles, Algorithms and Applications", Pearson, ISBN 0133942899.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[9] Jan M.Rabaey, Anantha Chandrakasan and Borivoje Nikolic, “Digital Integrated Circuits-A Design Perspective”, Prentice Hall, Second Edition, 2003. ISBN 0-13-090996-3.

[10] Yiyan Tang, Yingtao Jiang and Yuke Wang, “Reduce FFT Memory reference for low power applications”. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2002, Volume 3, 13-17 May 2002 Pages(s):III-3204- III-3207 vol.3.

[11] Byerly, W. E., “An Elementary Treatise on Fourier's Series, and Spherical, Cylindrical, and Ellipsoidal Harmonics, with Applications to Problems in Mathematical Physics”. New York: Dover, 1959.

[12] Carslaw, H. S., “Introduction to the Theory of Fourier's Series and Integrals”, 3rd ed., rev. and enl. New York: Dover, 1950.

[13] Davis, H. F., “Fourier Series and Orthogonal Functions”, New York: Dover, 1963.

[14] Dym, H. and McKean, H. P. “Fourier Series and Integrals”, New York: Academic Press, 1972.

[15] Folland, G. B., “Fourier Analysis and Its Applications”, Pacific Grove, CA: Brooks/Cole, 1992.

[16] Groemer, H., “Geometric Applications of Fourier Series and Spherical Harmonics”, New York: Cambridge University Press, 1996.

[17] Oppenheim A.V., Schafer R.W., and Buck J.R., “Discrete-Time Signal Processing”, Prentice-Hall, 1999.

[18] Smith S.W., “The Scientist and Engineer's Guide to Digital Signal Processing”, (*http://www.dspguide.com/pdfbook.htm*), California Technical Publishing, San Diego, 2nd edition, 1999.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[19] Retrieved from “http://en.wikipedia.org/wiki/Discrete_Fourier_transform”.

[20] Bevan M. Baas, "A Low-Power, High-Performance, 1024-point FFT Processor." *IEEE Journal of Solid-State Circuits (JSSC),* pp. 380-387, March 1999.

[21] Bevan M. Baas, "A 9.5mW, 330μsec, 1024-point FFT Processor," *Proceedings of the 1998 Custom Integrated Circuits Conference (CICC)*, Santa Clara, CA, USA, 11-14 May 1998.

[22] Wen-Chang Yeh, and Chein-Wei Jen, “High-Speed and Low-Power Split-Radix FFT”. In IEEE *International Conference on Signal Processing*, Vol., 51, No.3, March 2003.

[23] He .S. and Torkelson, “Designing pipeline FFT processor for OFDM (de)Modulation,” in Proc. IEEE *URSI International Symposium of Signals, System and Electron*, 1998, pp.257-262.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA

Graduate College
University of Nevada, Las Vegas

Bhaarath Kumar

Home Address:

1600, E. Rochelle Ave., Apt 42
Las Vegas, NV-89119.

Degrees:

Bachelor of Engineering, Electronics and Communication Engineering, 2002, University of Madras, India

Thesis Title:

Design and Implementation of a Fast Fourier Transform Architecture using Twiddle Factor Based Decomposition Algorithm.

Thesis Examination Committee:

Chairperson, Dr. Yingtao Jiang, Ph. D.
Committee Member, Dr. Emma Regentova, Ph. D.
Committee Member, Dr. Eugene McGaugh, Ph. D.
Graduate College Representative, Dr. Ajit K.Roy, Ph. D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.