The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

# A Parallel Processing Framework for Spectral Based Computations

University of Cape Town

Andrew van der Byl

University of Cape Town

A thesis submitted to the Department of Electrical Engineering, University of Cape Town, in fulfilment of the requirements for the degree of

*Doctor of Philosophy*

December 2012

# Declaration

I declare that this thesis is my own, unaided work. It is being submitted for the degree of Doctor of Philosophy at the University of Cape Town, and I grant the University free licence to reproduce this work in whole, or in part, for the purposes of research. It has not been submitted before for any degree or examination at any other University.

Signature of Author:

Date:

University of Cape Town

One's mind, once stretched by a new idea, never regains its original dimensions

OLIVER WENDELL HOLMES

University of Cape Town

# Acknowledgements

I was once told that the only section of a dissertation that truly belongs to the author is the acknowledgements, and it is a section I gladly lay claim to. Performing this study, and compiling this document can best be described as a labor of love, and is the culmination of many years of hard work, and persistent prayer. First and foremost, all praise, honour and glory I give to my personal Lord and Saviour, Jesus Christ. You have been the light in the darkness, the calm in the chaos, and my strength in times of weakness. No amount of thanks could express my gratitude.

I would like to especially thank my two academic advisers, Professor Michael Inggs (UCT) and Professor Richardt H. Wilkinson (CPUT) who provided exceptional guidance throughout this journey. Your encouragement kept me focused, and your wisdom cleared up many an unclear horizon, and kept me going in the right direction. Your advice in both personal and professional matters has been invaluable, and I feel truly indebted. It has been a privilege to study and grow under your leadership, and I look forward to many future opportunities to work with you both.

In addition, I would like to thank Professor Gerhard De Jager for his time, wisdom, and willingness to provide guidance. Your vast knowledge and advice is also greatly appreciated.

University of Cape Town

I would like to thank my family, both local and abroad, for their consistent support and understanding. To my parents, thank you for the support and push in the right direction. You always had faith in my abilities, and believed in me. To my sister, Julie, thank you for showing me from an early age to pursue excellence in everything I do. You have been my rock in an inconsistent time.

I have been blessed with an exceptional group of friends - all of whom have played, and continue to play a very special part in my life. You all have been a constant source of encouragement and support, and a useful periodic reminder that life does exist beyond the covers of a PhD dissertation.

I would also like to thank Shaun Kaplan at CPUT, and all the guys at the ACELab for the interesting conversations, and learning experiences we all shared.

Last, but by no means least, I would like to thank Vikram Bindal, an exceptional friend whom I have had the privilege of knowing, and studying with for many years. My academic path would not have been nearly as meaningful if it had not been for your friendship. I will carry the many funny and memorable moments, and interesting study experiences with me for the rest of my life. Coca-Cola and euclidean distance take on a whole new meaning :)

Right, 2008 to 2012 has come and gone.....let's showcase this work......

University of Cape Town

# Abstract

Over the past 40 years, the computing industry has benefited from persistent growth largely attributed to improvements in processor performance. Many significant advances have been made in areas such as processor architecture, on-chip and off-chip memory, as well as data transfer interfaces. The ability to transfer data through high-speed interfaces, and access memory within a few tens of cycles has contributed to the overall improvement of computing platforms, but it is the throughput of the processing architectures, when presented a set of instructions, which plays the most significant role in the overall performance of the system.

Traditionally, application code has been programmed and compiled with the underlying architecture and memory system in mind. Masking memory latencies, predicting cache misses, and leveraging specific vector instructions became the norm for squeezing performance from a given system. In the days before wide-scale parallelism, where sequential processors dominated, much reliance existed on the hardware vendors to continue to scale performance as transistor densities improved. The ability to obtain a speedup of legacy software by simply waiting 18 months for newer faster hardware to emerge became the 'free lunch' of the computing industry, but also its Achilles heel. The lack of preparation from the software community to adopt and define parallel software coding practices and standards has seen the development of compilers and supporting languages for parallelism lagging behind.

University of Cape Town

Today, great advances have been made; however the tenet of *'design first, figure out how to program later'* still lingers in the corridors of Silicon Valley. The focus of this study is however not on making a contribution to compilers or software development, nor on determining an efficient generic parallel processing architecture for all classes of computing.

Instead, this study adopts a different design approach, where a class of computing is first selected and analyzed, before determining a suitable hardware structure which can be tailored to the class being considered. The class of computing under investigation in this work is *Spectral Methods*, which by its very nature, has its own processing and data communication requirements. The purpose of this study is to investigate the processing and data handling requirements of the *Spectral Methods* class, and to design a suitable framework to support this class. The approach is different from past traditions - the hardware framework is based on software requirements, and in a sense is designed for the processing required, rather that the other way around.

The underlying computation in *Spectral Methods* is the Fourier transform, which in turn can be analyzed to reveal the underpinning of complex arithmetic. Further consideration of the class shows that other operations possible in Fourier space are also reliant on complex arithmetic.

The goal of achieving a high-throughput system through parallelism with minimal data dependencies steered the study to identify a parallel technique to compute the Fourier transform, while making it possible to use the underlying hardware to support additional Fourier space operations. Investigations revealed that parallel scalable algorithms to compute the Fourier transform do exist, and with suitable modifications could be made to perform both forward and reverse

University of Cape Town

Fourier transforms with error correction abilities for finite wordlength arithmetic.

To perform the Fourier transform this study implemented a recursive technique which allowed the updating of the output on a sample-by-sample basis. Techniques such as the Fast Fourier Transform (FFT) by contrast, operate in a block-data fashion where a set of samples are first required before the computation can be performed. The ability to compute a Fourier update sample-by-sample opens up new opportunity for increased time-frequency resolution, while still maintaining the ability to compute either a forward or reverse transform on blocks of data if required. The sample-by-sample updating requires the recursion of results which in turn introduced the negative side-effect of arithmetic error accumulation for finite-bit arithmetic.

To bound the error growth, this study further develops an error computation and correction technique, making it possible to determine the error at any time instant for use in correcting the respective Fourier output. The results showed that it is possible to error correct and limit the error in any output for finite-bit arithmetic, and achieve bounded square error results in the order of $10^{-6}$ for 36-bit word lengths using 26-bit precision (root mean square error of the order $10^{-3}$ after $\approx 10^6$ iterations). Furthermore, performance results indicate that giga-operations per second with throughputs exceeding 29GBPS are possible for a DFT of length 64 operating at 82MHz on Virtex 5 FPGA hardware. This design included automatic error computation and correction to ensure bounded errors, and provides suitable competition for FFT implementations on multi-core CPU's and multi-threaded GPU's.

University of Cape Town

On reviewing the study objectives, hypotheses, and research questions, it can be concluded that it is possible to obtain scalable performance using a many processing element framework for the fundamental computations of the *Spectral Method* computing class. This was shown in both the software and hardware simulations and tests, and was made possible through the adoption and implementation of algorithms which support concurrency.

It also became clear throughout the study that checks and balances are required, as the algorithm selected due to its advantages of sample-by-sample updating and ease of implementation carried an error cost which needed to be rectified. Jointly considering the research questions, a number of operations were identified for the class (Fourier transform being the dominant one), and through careful evaluation and selection of the algorithm, it was shown that operations such as the Fourier transform can be executed with scalable linear performance using many architecturally simple processing elements. This study has been successful in the objectives stated, and has showed that the design paradigm of '*assess first, design later*' in fact can be beneficial for at least one class of computing.

University of Cape Town

# Contents

**Contents** ix

**List of Figures** xiii

**List of Tables** xvi

**Nomenclature** xix

**1 Introduction** 1

1.1 Problem Description . . . 1

1.1.1 The State of Computing . . . 1

1.1.2 Past Approaches to Parallelism . . . 4

1.1.3 A New Approach . . . 7

1.2 Spectral Methods . . . 12

1.3 Computational and Communication Patterns . . . 13

1.4 Hypothesis and Research Questions . . . 14

1.5 Thesis Objectives and Delineation . . . 15

1.6 Research Approach . . . 18

1.7 Statement of Originality . . . 19

1.8 Publications . . . 20

**2 Spectral Computations - The World of Fourier** 21

2.1 Introduction . . . 21

2.2 Contributions . . . 22

2.3 Spectral Method Computational Patterns . . . 23

2.4 The Discrete Fourier Transform . . . 27

University of Cape Town

2.5 The Goertzel Algorithm . . . 29
2.6 Arithmetic Fourier Transform . . . 31
2.7 The Fast Fourier Transform . . . 33
2.8 Recursive Discrete Fourier Transform . . . 37
2.9 Windowing Data . . . 40
2.10 Technique Selection . . . 41

**3 Spectral Computations: A Recursive Fourier Approach 43**

3.1 Introduction . . . 43
3.2 Contributions . . . 44
3.3 Recursive Fourier Transform . . . 45
3.4 Recursive DFT: Forward Transform . . . 46
3.4.1 Initial Computations . . . 50
3.5 Recursive DFT: Reverse Transform . . . 51
3.6 Quantization . . . 55
3.6.1 Error Sources . . . 58
3.7 Error Correction . . . 64
3.7.1 Error Modelling . . . 64
3.7.2 Error Tolerance . . . 67
3.7.2.1 Rate-Distortion Simulations . . . 70
3.7.2.2 RDFT Component Analysis . . . 81
3.7.3 Analytical Error Model . . . 98
3.7.3.1 Model Simulation . . . 107
3.8 Noise-to-Signal Ratio . . . 111
3.9 Dynamic Range . . . 118
3.10 Simulation . . . 120
3.10.1 Continuous Channel Assessment . . . 120
3.10.1.1 Block FFT . . . 121
3.10.1.2 RDFT . . . 124
3.10.1.3 Simulated Results . . . 126
3.10.1.4 Discussion of Results . . . 129
3.11 Chapter Summary . . . 131

University of Cape Town

**4 Parallel Framework: A Many Element Approach 133**
4.1 Introduction . . . 133
4.2 Contributions . . . 134
4.3 System Framework . . . 134
4.4 System Overview . . . 138
4.5 System Specifications . . . 140
4.6 System Components . . . 142
4.6.1 Finite State Machine Control Unit . . . 142
4.6.1.1 Cold Start . . . 144
4.6.1.2 Reset . . . 144
4.6.1.3 Reset Done . . . 144
4.6.1.4 Transfer Coefficients: Real and Imaginary . . . 144
4.6.1.5 Read Input Data . . . 145
4.6.1.6 Idle . . . 145
4.6.2 Input Source Control . . . 146
4.6.3 Memory System . . . 147
4.6.3.1 Input Buffer, Subtraction and Division . . . 147
4.6.4 Bus Arbitration . . . 152
4.6.4.1 Computational Coefficients . . . 152
4.6.5 Processing Elements . . . 155
4.6.5.1 DFT Computation . . . 158
4.6.5.2 Error Correction . . . 161
4.6.6 Data Handling . . . 167
4.7 Hardware and Software . . . 170
4.7.1 Hardware . . . 170
4.7.1.1 FPGA Features . . . 171
4.7.1.2 Xilinx ML506 Development Kit . . . 172
4.7.1.3 ROACH Development System . . . 173
4.7.2 Software . . . 174
4.8 Chapter Summary . . . 176

**5 Analysis of Results 178**
5.1 Introduction . . . 178

University of Cape Town

5.2 Contributions . . . . . . . . . . 178
5.3 Hardware Implementation Results . . . . . . . . . . 179
5.3.1 Resource Utilisation . . . . . . . . . . 179
5.3.2 Clock Frequency and Processing Latency . . . . . . . . . . 180
5.3.3 System Accuracy . . . . . . . . . . 182
5.3.4 Power Analysis . . . . . . . . . . 186
5.4 System Benchmarking . . . . . . . . . . 187
5.4.1 FPGA . . . . . . . . . . 188
5.4.1.1 Block Based Processing . . . . . . . . . . 188
5.4.1.2 Sample-by-Sample Updating . . . . . . . . . . 189
5.4.2 Multicore CPU . . . . . . . . . . 190
5.4.3 GPU . . . . . . . . . . 196
5.5 FPGA to ASIC Projection . . . . . . . . . . 199
5.6 Additional Hardware Requirements . . . . . . . . . . 201
5.7 Chapter Summary . . . . . . . . . . 201

**6 Conclusions** **204**
6.1 FutureWork . . . . . . . . . . 212
6.1.1 Reciprocal Scaling Property . . . . . . . . . . 213
6.1.2 Time Shift Property . . . . . . . . . . 215
6.1.3 Frequency Shift Property . . . . . . . . . . 216
6.1.4 Integration Property . . . . . . . . . . 217
6.1.5 Time Domain Differentiation . . . . . . . . . . 219
6.1.6 Convolution Property . . . . . . . . . . 219
6.1.7 Area Property . . . . . . . . . . 219

**References** **220**

University of Cape Town

# List of Figures

1.1 Motif Classes [1, 2, 3] . . . . . . . . . . . . . . . . . . . . . . . . . 9
1.2 Motif Classes Wagon Wheel [1, 2, 3] . . . . . . . . . . . . . . . . . . . 17

3.1 Forward RDFT model without error factors . . . . . . . . . . . . . 59
3.2 Reverse RDFT model without error factors . . . . . . . . . . . . . 60
3.3 Computing model of the forward RDFT including error factors . . 62
3.4 Computing model of the reverse RDFT including error factors . . 63
3.5 Fixed 16-Bits Precision vs Variable Threshold (Real Only) . . . . 71
3.6 Fixed 18-Bits Precision vs Variable Threshold (Real Only) . . . . 72
3.7 Fixed 20-Bits Precision vs Variable Threshold (Real Only) . . . . 73
3.8 Fixed 22-Bits Precision vs Variable Threshold (Real Only) . . . . 74
3.9 Fixed 24-Bits Precision vs Variable Threshold (Real Only) . . . . 74
3.10 Fixed 26-Bits Precision vs Variable Threshold (Real Only) . . . . 75
3.11 Fixed $2^{-8}$ Threshold vs Variable Precision Bit Length (Real Only) 78
3.12 Fixed $2^{-12}$ Threshold vs Variable Precision Bit Length (Real Only) 78
3.13 Fixed $2^{-16}$ Threshold vs Variable Precision Bit Length (Real Only) 79
3.14 Fixed $2^{-20}$ Threshold vs Variable Precision Bit Length (Real Only) 79
3.15 Fixed $2^{-24}$ Threshold vs Variable Precision Bit Length (Real Only) 80
3.16 Error analysis for DFT component 2 using $\upsilon = 2^{-10}$ . . . . . . . . 83
3.17 Error analysis for DFT component 3 using $\upsilon = 2^{-10}$ . . . . . . . . 84
3.18 Error analysis for DFT component 2 using $\upsilon = 2^{-12}$ . . . . . . . . 85
3.19 Error analysis for DFT component 3 using $\upsilon = 2^{-12}$ . . . . . . . . 86
3.20 Error analysis for DFT component 2 using $\upsilon = 2^{-14}$ . . . . . . . . 87
3.21 Error analysis for DFT component 3 using $\upsilon = 2^{-14}$ . . . . . . . . 88
3.22 Allan variance for DFT components 2 and 3 using $\upsilon = 2^{-10}$ . . . . 94

University of Cape Town

3.23 Allan variance for DFT components 2 and 3 using $\upsilon = 2^{-12}$ . . . . 95
3.24 Allan variance for DFT components 2 and 3 using $\upsilon = 2^{-14}$ . . . . 96
3.25 Forward Transform: Mean (Real) comparison of generic error model results . . . 108
3.26 Forward Transform: Mean (Imaginary) comparison of generic error model results . . . 109
3.27 Reverse Transform: Mean (Real) comparison of generic error model results . . . 109
3.28 Reverse Transform: Mean (Imaginary) comparison of generic error model results . . . 110
3.29 Linearly Sweeping Chirp Signal: DC to 10MHz in 100$\mu$S . . . . . 121
3.30 Block FFT Processing: ADC Front-End with Double-Buffering . . 122
3.31 Block FFT Processing: Segmented Data Front-End . . . . . . . . 123
3.32 ADC Front-End Sample-by-Sample Processing using the Recursive Fourier Transform . . . 124
3.33 Data Sequence Sample-by-Sample Processing using the Recursive Fourier Transform . . . 125
3.34 Real Output: Time - Frequency representation of the Chirp input sequence using Block-FFT processing . . . 126
3.35 Real Output: Time - Frequency representation of the Chirp input sequence using RDFT processing . . . 127
3.36 Imaginary Output: Time - Frequency representation of the Chirp input sequence using Block-FFT processing . . . 127
3.37 Imaginary Output: Time - Frequency representation of the Chirp input sequence using RDFT processing . . . 128
3.38 Real and Imaginary RMS error between the FFT and RDFT every 64 iterations . . . 129

4.1 System Framework . . . 137
4.2 High Level System Overview . . . 139
4.3 High Level System Overview . . . 143
4.4 Input Source Multiplexing . . . 148
4.5 Input Data Memory System . . . 149

University of Cape Town

4.6 Circular Buffer . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
4.7 Bus Arbiter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
4.8 Coefficient Memory Sub-System . . . . . . . . . . . . . . . . . . . . 154
4.9 Processing Element Array . . . . . . . . . . . . . . . . . . . . . . . 156
4.10 Processing Element Array . . . . . . . . . . . . . . . . . . . . . . . 157
4.11 Processing Element Architecture . . . . . . . . . . . . . . . . . . . . 159
4.12 Error Correction Engine Architecture . . . . . . . . . . . . . . . . . 162
4.13 Dithering Hardware . . . . . . . . . . . . . . . . . . . . . . . . . . 163
4.14 Error Correction Engine - Error Removal . . . . . . . . . . . . . . . 165
4.15 Xilinx ML506 Development Board [4] . . . . . . . . . . . . . . . . . 172
4.16 ROACH Development Board [5] . . . . . . . . . . . . . . . . . . . . 173

5.1 Processing Latency - ChipScope Analyser . . . . . . . . . . . . . . . 181
5.2 Supporting Data Streaming Front-End . . . . . . . . . . . . . . . . . 183
5.3 Supporting Data Streaming Front-End . . . . . . . . . . . . . . . . . 184
5.4 Intel MKL: Single 1D FFT per Thread [6] . . . . . . . . . . . . . . 191
5.5 Hypothetical Performance for Scaled RDFT . . . . . . . . . . . . . 193
5.6 FFT RMSE for CPU and GPU Implementations . . . . . . . . . . . 196
5.7 Single FFT GPU Results [6] . . . . . . . . . . . . . . . . . . . . . . 197
5.8 Batch FFT GPU Results [6] . . . . . . . . . . . . . . . . . . . . . . 198

6.1 Reciprocal Scaling Property . . . . . . . . . . . . . . . . . . . . . . 214
6.2 Time Shift Property . . . . . . . . . . . . . . . . . . . . . . . . . . 215
6.3 Frequency Shift Property . . . . . . . . . . . . . . . . . . . . . . . . 216
6.4 Integration Property . . . . . . . . . . . . . . . . . . . . . . . . . . 218

University of Cape Town

# List of Tables

2.1 Computational Properties of the DFT . . . . . . . . . . . . . . . . . . 28
2.2 Computational Properties of the Goertzel Algorithm . . . . . . . . 30
2.3 Computational Properties of the Arithmetic Fourier Transform . . 32
2.4 Computational Properties of the FFT . . . . . . . . . . . . . . . . . . 36
2.5 Computational Properties of the Recursive DFT . . . . . . . . . . . 39

3.1 Noise Process and $\alpha$ values . . . . . . . . . . . . . . . . . . . . . . . 93

4.1 System Specifications . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
4.2 Xilinx VSX50T Features [7] . . . . . . . . . . . . . . . . . . . . . . . 171
4.3 Xilinx ML506 Hardware Features [4] . . . . . . . . . . . . . . . . . . 172
4.4 ROACH Hardware Features [5] . . . . . . . . . . . . . . . . . . . . . 173
4.5 Software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175

5.1 Resource Utilization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 180
5.2 Power Analysis (XC5VSX95T) . . . . . . . . . . . . . . . . . . . . . 186
5.3 FFT Results: 64 Point DFT . . . . . . . . . . . . . . . . . . . . . . . 188
5.4 FFT Results: 64 Point DFT . . . . . . . . . . . . . . . . . . . . . . . 190
5.5 Hypothetical Performance for Scaled RDFT . . . . . . . . . . . . . 193
5.6 Projected FPGA to ASIC Conversion . . . . . . . . . . . . . . . . . 200

University of Cape Town

# Nomenclature

ADC - Analog-to-Digital Converter
ALU - Arithmetic Logic Unit
ASIC - Application Specific Integrated Circuit
CPU - Central Processing Unit
CMP - Chip Multi-Processor
DFT - Discrete Fourier Transform
DC - Direct Current
FFT - Fast Fourier Transform
FPGA - Field Programmable Gate Array
GPU - Graphics Processing Unit
GBPS - Giga Bits Per Second
HMA - Horizontal Microcoded Architecture
ILP - Instruction Level Parallelism
MKL - Math Kernal Library
MPE - Many Processing Element
MSE - Mean Square Error
PE - Processing Element
RDFT - Recursive Discrete Fourier Transform
RF - Radio Frequency
RMSE - Root Mean Square Error
ROACH - Reconfigurable Open Architecture Computing Hardware
SIMD - Single Instruction Multiple Data
TTA - Transport Triggered Architecture
VLIW - Very Long Instruction Word
VLSI - Very Large Scale Integration

University of Cape Town

# Chapter 1

# Introduction

University of Cape Town

## 1.1 Problem Description

### 1.1.1 The State of Computing

Over the past 40 years, the computing industry has benefited from persistent growth largely attributed to improvements in processor performance. In 1965, Gordon Moore postulated that the number of transistors per processor would double every year (later revised to every second year) [8], and for the larger part of a 40 year industry, the quantity of transistors per die dictated the type of performance obtained from a particular generation of processor. The transistor count increments were due to fabrication technology improvements, where smaller and faster transistors could be placed on a die, which in turn provided a larger transistor budget for designers to explore. This exploration allowed designers the freedom to add extra features which brought about the introduction of superscalar and vector processors, pipelining, and cache memory [9], while higher frequency scaling was also taking place thanks to improved switching times of the smaller transistors [10].

Through generations of improvements, sequential processors became the dominant hardware (the goal of latency-oriented architectures was to minimise the running time of a sequential program [11]), and the software industry began to rely on sequential processor performance improvements for the reduction in software execution time [10, 12]. The expanding bubble that encompassed the sequential performance improvements eventually burst, and manufacturers realised that both practical and physical constraints such as limited pipeline depth, heat, power, cooling and diminishing returns on exploiting instruction level parallelism, ultimately lead to a brick wall [1, 10, 13, 14, 15, 16].

There is no other obvious route to high performance except through parallelism [17], and as a result, there are significant consequences for compute-intensive signal processing software, as the end of frequency scaling means the end of a free speedup for legacy software [14]. Furthermore, it also introduces an era where writing high-performance code becomes increasingly difficult, since the programmer now has to use multiple threads to tune code to the memory hierarchy [14]. This 'moment of truth' has introduced an exciting but daunting time for the computer industry, as the future of processors is at an inflection point. It is at this point that the future of processing can be defined for the next 40 years.

Currently Moore's law still prevails; however, the focus is not on using the improved transistor density per generation for implementing additional hardware features to a sequential processor design that will only yield a marginal improvement, but rather to provide additional cores per die to promote a parallel hardware approach [1]. This naturally means that software needs to be able to make use of these additional processing units in the form of data, task, and thread level parallelism, or throughput would be no better than if a single core remained.

University of Cape Town

This however does not come without challenges, and although a multi-core approach offers high performance potential, the challenge is how to find and exploit this parallelism [16, 18, 19]. Collaboration from both hardware and software is therefore required, as both hardware and software should complement each other in order to harness and exploit concurrency in a manner that is natural to the application at hand.

Parallel processing since the early 90's has been based on a 'build hardware first, figure out how to program later approach', and care must be taken to ensure that the same does not happen to multicore processors [20, 21]. It is important to first understand the computational and communication requirements of applications, so hardware can be developed which supports those requirements detailed by software [22]. However, in the processor domain, bigger does not always mean better, and it has been strongly suggested that the way forward is not simply duplicating existing complex cores on a silicon die (which is likely to face diminishing returns as 16 and 32 processor systems are realized [1]), but rather to utilize architecturally simpler processing elements [1, 23, 24]. This would result in the implementation of many simpler processing elements per die, rather than a handful of highly complex cores.

This in turn brings performance/power trade offs. However it has been shown that it is more efficient to execute a parallel application on a large number of simple cores than a handful of complex ones using the same surface area and power [25]. It has been shown that it is more energy efficient to utilise hardware architectures that perform many operations slowly in parallel, than architectures which perform a single operation very fast [15, 26]. This does not however mean that all cores need to be identical and, in fact, the future has been touted to be heterogeneous [22]. This in turn depicts a future where processors can be a diverse collection of tuned processing elements (to computation and communications) which work synergistically, or alternatively a collection of processing platforms such as FPGA's, CPU's and GPU's allowing transparent seamless processing migration [22].

University of Cape Town

The computational requirements per application which will execute on the processors can be vastly different. Applications can be classified into various computing classes termed 'motifs'(a term used to describe a pattern of computation and communication - detailed in section 1.1.3) which, when viewed as a whole, encompass the majority of the computing spectrum of applications [1, 10].

These processing and communication differences would naturally require different hardware for optimal execution, and a one-size-fits-all approach is not necessarily best. A better way forward, perhaps, is to adopt a many processing element approach, where the elements in the architectural design are specific to a particular class of computing (motif), encouraging more efficient processing.

### 1.1.2 Past Approaches to Parallelism

The concept of parallel architectures is not new and over the past 60 years various options have been explored to aid the execution of software more timeously (in addition to the gains resulting from frequency scaling). As early as the 1950's, massively parallel architectures were proposed in the field of image processing, however, since image operations incorporate similar data access patterns, relatively few alternate parallelization strategies needed to be considered [27]. Taking a more generic approach, special hardware could be implemented in the form of co-processors or specific functional units as in the case of superscalar processors.

University of Cape Town

Transport Triggered Architecture (TTA) systems are similar in this respect, and TTA's can be compared to Very Large Instruction Word (VLIW) processors, as both exploit compile time Instruction Level Parallelism (ILP). TTA hardware is well structured and organised as a set of functional units and register files, where the functional units typically consist of a collection of Arithmetic Logic Units (ALUs), multipliers, shifters, floating point units, buses and registers [28]. The significant difference lies in the program instructions, as they specify data transports where the destination specifies the kind of operation that will be performed on the data.

An advantage of TTA hardware is scalability and easy implementation of arbitrary functionality to support a range of applications. The idea is however not new, dating back to the fifties, and has since been re-invented on multiple occasions [28].

Another study [29] adopted the parallel resource approach and used two application specific TTA processors for implementation of the Discrete Cosine Transform, Fast Fourier Transform, and Viterbi decoder. Alternative approaches such as those described in [30], have focused on extending an existing Chip Multi-Processor (CMP) for applications such as H.264 encoding instead of implementing a custom ASIC design. This approach showed it is possible to close the power efficiency gap between ASIC and CMPs to within 3x, while achieving comparable performance. This type of approach shows promise that generic multi-processor systems such as CMPs can be tweaked, and thus tailored to execute hundreds of application specific operations concurrently while maintaining an element of generality. This characteristic is important, as a processor with a general 'fits all' property may not be best suited to many application domains.

The many-processing element approach however does not have to be in the form of co-processors or specific functional units, but can be in the form of modified Instruction Set Architectures (ISA) [31, 32]. Alternative approaches such as [33], discuss a soft-core based application specific Horizontal Microcoded Architecture (HMA) for designing programmable high performance processing elements. A custom data path is first generated and synthesized for an application, and later the program code is compiled on the architecture to generate control words. Such approaches have the advantage that even if the program code does change, the program can be re-compiled on the existing data path, and re-synthesis of the hardware is not needed. The primary issue for approaches such as this is the application specific nature to data path and instruction sets.

University of Cape Town

It has been suggested that the way forward is not in implementing complex processing cores, but rather simple processing structures targeting wide-scale parallelism [1, 23, 24]. The approach of using small Processing Elements (PE) has surfaced in a variety of applications, and more recently, the introduction of Graphics Processing Units (GPUs) have provided substantial speed advantages in many computing cases [13, 34]. A GPU is a homogeneous multicore architecture with hundreds of simple in-order PEs, designed primarily for high-performance graphics rendering [13, 24]; however it has evolved into a highly parallel and programmable computing platform [35]. Applications which require identical processing on large datasets can achieve significant speedups when processed using throughput-orientated Single Instruction Multiple Data (SIMD) processors such as a GPU [11].

University of Cape Town

This is not always the case, and many application domains exist where speedups are marginal using a SIMD approach, and where single thread execution speed cannot be sacrificed to optimise aggregate throughput [11]. A slightly different approach combines the flexibility of general purpose processors and the speed and power advantage of custom circuits to yield some novel architectures [36]. Hybrid processors can be augmented with custom instructions to execute on application specific hardware units, where a common link is the use of traditional ALUs in a variety of configurations to exploit parallelism [36, 37, 38, 39, 40, 41]. The applications of interest in these studies all seem to offer exploitable parallelism, and it would be of great interest to explore the true computational and communication patterns of the classes which embed these applications, and tailor hardware to support them.

Performance does not always scale proportionally with the increase in processors, and it is suggested that one of the reasons for this is the degree of mismatch between an algorithm and an architecture. Stated differently, the performance of a parallel algorithm depends on how well its communication and computation patterns match the interconnection topology and processing hardware of the system [42].

For this reason, it would be beneficial to view computing, and its respective requirements, in a different light. The historical road map has shown that a build-first, figure-out-how-to-program-later approach does have its limitations, and after a successful period spanning 40 years, has eventually come to a point where revision is required to move forward for another 40 years. One such approach is to consider the computational and communication requirements needed for a given application domain, and tailor hardware specifically to this application space. It is in this light that this work progresses forward.

### 1.1.3 A New Approach

Throughout all computing, processing can be classified according to the patterns it exhibits for both data computation and data communication. Computational patterns assess factors such as data operations (type of arithmetic - e.g. multiply-add; data dependencies; ordering etc.), and communication patterns are concerned with data movement (cache access and internal data path; inter-processor, and between computing nodes often over network links). In many cases, these patterns can be vastly different, and as a result, have a low correlation. Low correlation between patterns would indicate exclusivity, and can be considered to be unique to a particular group. These individual pattern groups are more colloquially termed 'motifs', and when considering all patterns holistically, they encompass all computing styles [1, 2, 3].

If performance is the ultimate goal, tailoring a particular hardware design to an application usually has a significant speed advantage; however the design remains specific to the application, and not to the underlying motif to which it is categorized. If the characteristics of two classes were compared, it can often be found that very little correlation exists, and therefore each class stands independently of another in terms of both computational and communication requirements.

University of Cape Town

The ability to execute any application relevant to a particular motif class in a parallel manner does not necessarily occur by default, and methods or techniques need to be investigated in order to achieve the most computationally efficient processing with minimal data dependencies whilst offering scalable parallelism. This is not often easily achievable, and is the topic of much research. Figure 1.1 details the currently identified motifs and related applications [2].

In a parallel processing utopia, all applications spawn either multiple tasks, threads, or instructions, which can be performed simultaneously and independently of each other, requiring minimal or no data exchange or reliance on a current or future task before completion. Processing problems which share such attributes are often termed “embarrassingly parallel”, and although there is minimal dependency to slow down a task, initial data access and write-back still poses a bottleneck in the overall system.

Many problems exist which do not fit an “embarrassingly parallel” profile, and often data dependency significantly increases execution time, as certain computations cannot complete without obtaining data first from another task, which may or may not yet be available, especially if it is to be sourced from another process or thread (the hardware executing the thread may not be well matched to the task at hand).

University of Cape Town

| Motif | Embedded | Desktop | Games | Database | Machine Learning | High Performance Computing | Medicine | Music | Speech | Context-Based Image Retrieval | Browser |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Dense Linear Algebra | | | | | | | | | | | |
| Sparse Linear Algebra | | | | | | | | | | | |
| Spectral Methods | | | | | | | | | | | |
| N-Body Methods | | | | | | | | | | | |
| Structured Grids | | | | | | | | | | | |
| Unstructured Grids | | | | | | | | | | | |
| Combinational Logic | | | | | | | | | | | |
| Graph Traversal | | | | | | | | | | | |
| Graphical Models | | | | | | | | | | | |
| Finite State Machines | | | | | | | | | | | |
| Dynamic Programming | | | | | | | | | | | |
| Map Reduce | | | | | | | | | | | |
| Backtrack/Branch and Bound | | | | | | | | | | | |

| Temperature Chart Reference | |
|---|---|
| Cool | |
| Medium | |
| Warm | |
| Hot | |

Figure 1.1: Motif Classes [1, 2, 3]. Applications can be grouped according to the motif which most closely represents the computational and communication patterns of the application. It is possible to collectively group applications together which all require similar patterns (which share a motif). Application domains in some cases transcend the boundaries of multiple motif classes, albeit some at different dependence levels. This mutual similarity between classes signifies the relevance of the classes to an application domain, and not necessarily that the motif classes in question share similar processing or communication patterns. This is possible if an application is dependent on two or more distinct underlying processes or data transfer styles for the application to be realised.

Any one hardware design is not likely to be suited to all tasks, and some tasks will naturally execute more efficiently if the underlying processing and interconnection hardware is well matched. This potential mis-match of hardware and software can play a significant role in the overall performance of many systems, and singles out a caveat of general purpose processing. Identification of the relevant motif class is the first step toward class specific computing (as opposed to application specific), however the algorithms utilised for such computations can vary in efficiency. It is therefore equally important to identify effective techniques capable of performing the required processing.

Successful techniques should aim for the parallel utopia of low to no data transfers (i.e. minimal data dependencies) and "embarrassingly parallel" computations, and promote simple implementations. Often a trade off exists, and data dependencies get traded for lower computational complexities promising lower arithmetic overhead.

The computing road map described in [1, 2] suggests an alternative approach toward computing architectures, and suggests a simple structured approach is the best way forward as we enter a new computing realm. The authors list numerous motif classes and potential application domains, however do not provide fine details toward the best algorithmic techniques per computing class that will offer low complexity, wide-scale parallelism with minimal data dependencies. It is therefore the focus of this study to select a computing class, identify key processing requirements, and select a suitable algorithm that shows promise toward a parallel processing utopia.

In the past, a general 'suitable for all' hardware processing approach has offered support for a wide range of motif classes, however at the cost of processing performance. This trade-off highlights the problem area being focused on in this research, and this work suggests an alternative approach to developing processing hardware which takes these requirements into greater consideration.

University of Cape Town

Developing a parallel hardware framework based on motif requirements is a step away from an overall generalised processor applicable to all motif classes, but in turn creates a processing platform that is specific to a motif, but general to the applications applicable to that motif class. The adoption of this approach means that a truly heterogeneous processing framework is feasible if a large variety of motif architectures are included in processing systems.

It should be noted that there are also non-technical factors that come into play when considering a hardware implementation. A cost-performance trade-off exists and as shown through the number of motifs, there are numerous software domains. Developing hardware to address each one would be expensive, and may not be conducive to a final implementation. It is therefore important to consider the applications linked to the respective classes, as well as the underlying computational requirements of the classes to determine the cost-performance ratio. This study sets out to assess the computational and communication requirements, and investigate possible performance of class specific generic hardware for the spectral methods motif.

University of Cape Town

## 1.2 Spectral Methods

The work of Asanovic et al. [1, 2] highlights numerous motif classes spanning the larger portion of the computing spectra, however this study will focus on only one of these classes, namely spectral methods. The term 'spectral methods' invokes thoughts of numerical analysis [43], and while spectral methods and numerical analysis go hand-in-hand, it is not the purpose of this study to focus on numerical computations. Spectral based computations such as the Fourier transform can be found firmly rooted in many application domains, which include (but are not limited to) [44, 45, 46, 47, 48]:

- RF communications (continuous channel assessment, radio astronomy, antennas and Radar)
- Geology (seismic and volcanic activity)
- Climatology
- Economics
- Remote sensing (e.g. high voltage transformer and motor condition monitoring)
- Civil engineering (structural integrity monitoring)
- Numerical analysis (turbulence modelling, non-linear wave equations, weather prediction, seismic exploration)
- Quantum mechanics
- Crystallography
- Signal processing
- Statistics
- Information theory

University of Cape Town

The applications listed are simply the tip of the iceberg, however it is not the purpose of this work to provide an exhaustive overview of suitable applications, and those listed serve as an indicator to the applicability of spectral computations in a variety of disciplines. A unifying computation in numerical methods (and applicable to the applications listed) is the Fourier transform [44, 49, 50], and it is the purpose of this work to analyse this transformation which underlies spectral computations, and to identify techniques amenable to an implementation in a parallel framework.

## 1.3 Computational and Communication Patterns

It is first required to outline the ideal specifications that would cause a technique to be viable for parallel implementation. These specifications can be used as a benchmark when evaluating a processing technique, and are considered from a general computational complexity and data dependency point of view, and therefore do not favour any particular processing architecture. The following aspects are considered during an evaluation:

- Degree of Available Parallelism
- Data Dependencies
- Network or Inter-Processor Communications
- Operational Latency
- Computational Complexity
- Memory Access

An ideal technique would encourage wide scale parallelism with little or no data dependencies to halt the flow of execution for a given process. Often a technique is scalable to a high degree of parallelism in initial stages of execution, but the requirement to obtain newer computed results from adjacent or networked nodes often arises, thereby hindering a process from completing execution. This in turn impacts the latency and reduces overall throughput, as processing cores

University of Cape Town

often sit spinning, waiting on the arrival of data. Latency is also affected by the computational complexity involved in a given computation, and it is as important to assess the arithmetic operations required to achieve an outcome. A low complexity system requiring few computations, using a simple architecture, is an important factor to consider when evaluating techniques. This study will consider these factors when assessing the spectral methods motif class.

## 1.4 Hypothesis and Research Questions

The hypothesis for this study can be stated as follows:

*A Many Processing Element (MPE) framework, tailored to fundamental spectral method motif class operations, promotes scalable performance for spectral method motif class applications.*

This hypothesis can be broken down into a few key points on which this research is focused. It is hypothesized that the spectral methods class of computing contains a handful of fundamental computations which make up the processing characteristic of the class. The conjecture is that implementing these fundamental computations into a parallel processing framework, comprising of many processing elements, will achieve computing performance which is scalable to the degree of parallelism implemented. Furthermore, a parallel approach to the fundamental computations of a particular class will enable improved performance for all applications encompassed within that class, while not showing bias to any individual application.

Consideration of the hypothesis leads to the following research questions:

- What are the fundamental operations of the spectral methods motif class?
- Can a spectral methods motif be implemented using many simple processing elements?
- Is parallelism scalable? If so, to what degree?

University of Cape Town

- What effect would finite word-widths have on fixed-point arithmetic? Can computational errors be determined and corrected?

- Is performance improvement possible using spectral method based MPE's, as compared to a specific hardware accelerated approach?

## 1.5 Thesis Objectives and Delineation

It has been the focus of this chapter to provide an overview of the state of computing, as well as suggest a new approach to solving the problem of limited performance improvement for application spaces which do not match the underlying hardware on which they execute. The purpose of this work is to develop a parallel hardware framework specific to the spectral methods motif, and one which will be generic to applications which reside within the spectral methods class.

The processing framework is based on the evaluation of the motif class, and the focus is on the design of parallel hardware which supports the computation and communication patterns exhibited by this class, regardless of the application. This approach is different from the conventional approach of 'design-first and figure-out-how-to-program-later'.

The Fourier transform is the dominant operation in the spectral methods class, and is of significant importance as all subsequent computations occurring in Fourier space are reliant on this data transformation. The main objective is therefore to produce a parallel framework which will exploit the underlying computation of this transform. The hardware design will focus on simplicity and efficiency, adopting a 'more is better' approach, where the aim is to exploit and implement as much parallelism as possible, and hence fit as many processing elements to a given fabric space, whilst using the simplest hardware that can be afforded to achieve the required outcome.

University of Cape Town

While the focus of this study is on the development of a generic framework for the spectral methods class, and therefore the Fourier transform and related Fourier space operators, there is still a need to adopt a suitable application to apply the design. A front-end RF or communications system was selected as a suitable application domain as it represented a widely applicable and generic application. The discussions which follow are directed toward this application area, however should not be considered exclusive to it.

To help illustrate the focal point of this work, a wagon wheel representing all defined computing classes (with an expansion for spectral methods) is shown in Figure 1.2 (the blocked area represents the focus of this work). The fundamental computation for spectral methods is in essence a basis function expansion - commonly the Fourier Transform.

Once in Fourier space, a variety of operations are possible (which have been included), and it is the purpose of this study to concentrate on an efficient means to transform data into Fourier space, adopting wide-scale parallelism, using a generic framework. This work will discuss this transformation process in depth, and will present an error correction method applicable for finite-bit arithmetic. Consideration will be given to additional operations, but these will largely be excluded from this text.

University of Cape Town

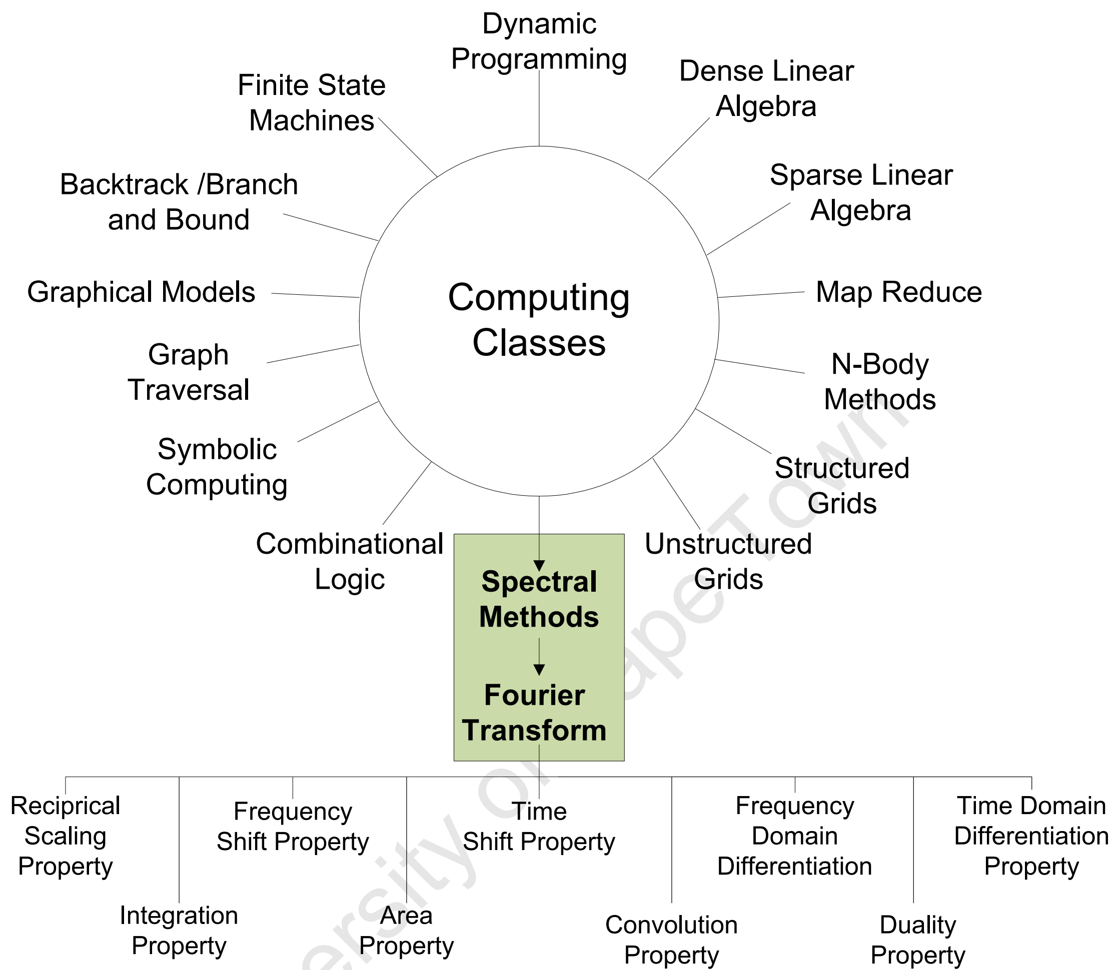



Figure 1.2: Motif Classes Wagon Wheel [1, 2, 3]. This wagon wheel represents all defined computing classes, and is included to aid in illustrating the focal point of this work. The core focus is on the spectral methods class, and has been expanded to highlight the relevant operations of this class. The fundamental computation in the spectral methods class is the Fourier transform, which will be discussed in detail in Chapter 2.

University of Cape Town

## 1.6 Research Approach

This section details the approach taken to meet the objectives of this thesis. It has been outlined in Section 1.5 that the fundamental computation in the spectral methods class is the Fourier transform. To design a parallel processing framework for this class, it is first required to gain a full understanding of the computational complexity behind the Fourier transform and select a suitable technique for its computation. This technique should be efficient in computation and be amenable to simple hardware implementation. A further constraint which should be considered is the communication requirements. High data throughput should be targeted, and data dependencies should be minimal. The framework should not be restricted exclusively to the Fourier transform, but should allow easy adoption of other operators detailed in Chapter 2.

Hardware implementation and data precision is another important consideration. Computational precision in many applications is of great importance, and for this reason cannot be overlooked. Floating-point arithmetic is not considered due to high resource requirements when considering Field Programmable Gate Array (FPGA) fabric, and a more cost-effective approach using fixed-point arithmetic is considered.

The signed fixed-point word length was set to 36-bits for hardware implementation, where 24-bits are allocated to precision, and 11-bits to integer representation (1-bit reserved for sign). This allocation was a fair trade off to allow reasonable representation of fractional numbers (in the order of $10^{-8}$), while allowing integer growth (in the order of $10^3$ before saturating). The hardware implementation was fixed in terms of word width and bit allocation, however simulations for the derived model indicate system performance based on varied word lengths.

In any fixed-point system, the rate-distortion is an important performance metric. The rate-distortion can be defined as the minimal number of bits required to encode an input signal(per sample), and later reconstruct without exceeding a

University of Cape Town

defined distortion level. The technique adopted for the Fourier transform implementation was modeled to determine the rate-distortion inherent in the output (due to the fixed-point arithmetic), as well as to determine the expected latency for the selected technique. This computational model was simulated to assess these factors, and the accuracy was compared with respect to a fixed-point and floating-point computation of the same data in Matlab®. Errors in the output are inherent due to fixed-point representation, and it became necessary to develop an error correction technique to reduce the accumulative effect of recursive arithmetic in fixed-point processing. This error correcting technique was modeled, simulated, and implemented within the processing framework. All results are presented and discussed, highlighting the necessary trade-offs which exist.

## 1.7 Statement of Originality

The candidate's original contributions of this work are listed as follows:

- Extension to the recursive Fourier transform: Previous work by Jacobsen and Lyons [51, 52]; Sherlock and Monro [53]; Brown [54]; and Kamei et. al [55] have shown the merit of this technique for the consistent updating of the output as new input samples become available. Their work however focuses on the forward transform, and gives no consideration to the inverse computation. This work addresses this shortfall as it is important in the processing framework to support the full transformation of data between discrete time and Fourier space. Furthermore, this work produces a finite bit arithmetic model of the recursive discrete Fourier transform and performs analysis to determine the rate-distortion trade off.

- Error correction technique for the correction of fixed-point inherent errors in a recursive implementation: This work extends on a prior model [56], and introduces a technique to compute and correct on-the-fly errors present in the output for fixed-point arithmetic in both complex exponentials and arithmetic roundoff. An analytical error model is produced, and used to compute the worst-case error expected for a given set of system parameters.

University of Cape Town

This model is also used to produce error correction engines in hardware which can be enabled to compute and correct the error in the output.

- Hardware implementation of the forward and reverse discrete Fourier transform using a parallel framework of many processing elements: Building onto the theoretical simulations performed from the model for the recursive forward and inverse Fourier transform, this study developed a hardware implementation targeting FPGA fabric space. FPGA devices are ideal for initial proof-of-concept hardware designs, and as a stepping stone toward ASIC realisation.

- This study proposes extensions to the processing framework to support the inclusion of additional Fourier space operators. Details are provided for the practical implementation of these operators into the existing framework.

## 1.8 Publications

This work has produced the following publications:

- van der Byl, A, Inggs, M, Wilkinson, R.H., "A many processing element framework for the Discrete Fourier Transform," International Conference on Field-Programmable Technology (FPT), 2010, pp.425-428, 8-10 Dec. 2010

- van der Byl, A, Inggs, M, Wilkinson, R.H., "Recursive Fourier Transform Hardware," International RADAR Conference (RADARCON), 2011, pp.23-27, May 2011

University of Cape Town

# Chapter 2

# Spectral Computations - The World of Fourier

University of Cape Town

## 2.1 Introduction

It has been highlighted in Chapter 1 that the road map for computer processing is at an inflection point. Traditionally, general purpose processors have seen increased performance thanks largely to clock frequency scaling, additional functional units (super-scalar processors), and improved instruction-level parallelism.

These techniques saw significant advances in processor performance, however the rate of improvement began to subside as a result of bottlenecks in pipeline depth, heat, power, cooling, as well as diminishing returns on exploiting instruction level parallelism (ILP) [1, 10, 13, 14]. Computers have now entered a multi-processor era, where the way forward is to utilize multi-processor parallelism for performance increase, rather than relying on clock frequency scaling, or the further exploitation of ILP.

The adoption of general purpose multi-processor parallelism yields improvements in application performance in an average sense across all computing classes provided parallelism can be found; however cannot compete in performance with the custom tailored hardware approach found in application specific designs.

What is required is a series of domain or class specific processors, where the hardware is tailored to the fundamental computations in the class, making it applicable to a wide application base, with a clear focus on many, simpler processing cores (many-core) rather than few complex processing cores (multi-core) [1, 23, 24].

Following on this notion, it is suggested that it would be better to develop architectures based on the fundamental computational patterns found within each computing class, rather than adopting the existing approach for general purpose computing of building a one-size-fits-all processor. In essence, looking forward, it would be better to build a processing architecture that is suited to a range of applications classifiable within a particular class, rather than developing an application specific system, or developing a processor that aims to work for all classes.

This approach is a step toward realising a truly heterogeneous many-core processor, where groups of processors are suited to individual classes. It is in this light that this chapter progresses forward, and will discuss the importance of the Fourier transform which is fundamental to the spectral methods computing class. This chapter also considers the underlying processing operations and data communications native to this transform, and investigates the classical approach to the discrete computation of this transform, as well as alternative approaches for parallel implementation.

University of Cape Town

## 2.2 Contributions

In this chapter, the following contributions are made:

- The identification of computational and data communication requirements for the discrete Fourier transform
- The discussion of classical and modern parallel approaches to the implementation of the discrete Fourier transform

## 2.3 Spectral Method Computational Patterns

The solution to creating a parallel framework for any class of computing is multi-faceted. It is important to identify and evaluate all data processing requirements for a given class, and search for common arithmetic, data dependency characteristics, and data transfer requirements.

Analysing the spectral methods motif class, one requirement becomes apparent - the need to operate in Fourier space. Other transformations such as Discrete Sine Transform (DST), Discrete Cosine Transform (DCT), and the Discrete Hartley Transform (DHT) should be duly noted for applications such as transform coding [57, 58], however the more common approach to spectral decomposition, and one widely applicable across multiple disciplines is the Fourier transform.

For any application residing in the spectral methods class, transitioning between time and Fourier space is a vital transformation, and one which has received much attention ever since Jean Baptiste Joseph Fourier suggested it as early as 1807 [46]. The Fourier transform not only enables the decomposition of an arbitrary signal into constituent sinusoidal components, but permits the execution of additional operations on projected data.

In some applications, a transformation into Fourier space is sufficient for analysis, however there are many instances where some form of post-processing is still required before the data is in a final form. Comparing computational costs when considering processing in both time domain or Fourier space, it can be found that performing operations on projected data often leads to computational simplicity, especially when compared to a time domain equivalent. The case of time-domain convolution is a clear example:

University of Cape Town

Convolution is defined as [59]:

$$y[n] = f[n] * h[n] \tag{2.1}$$

$$= \sum_{k=-\infty}^{\infty} f[k]h[n-k] \qquad \text{where} \qquad n, k \in Z \tag{2.2}$$

and if

$$f[n] \Longleftrightarrow F[e^{j\omega}] \tag{2.3}$$

and

$$h[n] \Longleftrightarrow H[e^{j\omega}] \tag{2.4}$$

Then the equivalent result for convolution can be shown in Fourier space:

$$Y[e^{j\omega}] = F[e^{j\omega}]H[e^{j\omega}] \tag{2.5}$$

where:

$$y[n] \Longleftrightarrow Y[e^{j\omega}] \tag{2.6}$$

Scrutinising the arithmetic required for both (2.2) and (2.5), it is possible to see that the same result can be achieved using two completely different approaches. Convolution performed in the time domain has a computational cost at worst proportional to $N^2$, where as a Fourier space equivalent incurs a computational cost of $2N-1$ [59], which is a computational reprieve of the order $N$.

The additional benefit in saving can only truly be appreciated if the processing approach is also taken into consideration. If the above computation were to be executed serially, then the costs stated remain as is. If the computation were to be executed in parallel, a different cost per method is incurred. The convolution in time domain stated in Equation (2.2) can be executed using a data parallel approach to determine $y[n]$, where each thread can represent a unique value for $n$. The cost per thread becomes at worst $2N-1$ of multiply-add arithmetic,

University of Cape Town

requiring memory access patterns for all data points. This is of course based on the assumption that it is possible to spawn $N$ threads simultaneously.

The parallel approach to the Fourier equivalent result in Equation (2.5) is ($O(1)$)(assuming $N$ threads), comprising of a single complex multiplication, requiring memory access per thread to only two complex operands. Comparing the costs attributed to both methods, it can be seen that a Fourier space execution promotes a more cost effective solution on the assumption that the inclusive cost to transform data from a time domain to Fourier space, and to perform the multiplication depicted in Equation (2.5) (as well as the return transformation) does not exceed the overall cost of a time domain convolution. The transformation of data into Fourier space is a crucial computation, and therefore cannot be blithely overlooked, and is the main focus of this work.

The convolution example detailed above is one of many operations which carries a corresponding operation in Fourier space. The ability to operate in such a space while offering computational savings, has gained popularity in application areas such as numerical analysis and digital signal processing [43]. Operations having corresponding Fourier space implementations can be listed as follows:

- Reciprocal Scaling
- Time Shift
- Frequency Shift
- Frequency Domain Differentiation
- Time Domain Differentiation
- Integration
- Convolution
- Area property
- Duality

University of Cape Town

It is not the purpose of this work to analyse each of the Fourier properties listed, and the discussion regarding fundamental operations in Fourier space should be seen as a motivation toward determining a computationally efficient framework which not only supports the transformation of data, but also provides support for additional operations such as convolution.

When determining computational costs, the assumption was made that the overall cost for an operation, including both forward and reverse Fourier transformations should not exceed the cost attributed to the time domain implementation of the the same functional operation. This, by its very nature implies that the computational efficiency of the Fourier transform has to be high. It is therefore understandable that computational efficiency of the Fourier transform has been the focus of much research over the past few decades.

Many functionally equivalent, but computationally simpler techniques such as the Fast Fourier Transform (FFT) have surfaced in this time period. The FFT has played a significant role in the way the transform is used in practice. The Fourier transform is a fundamental tool for many disciplines, and it is the purpose of this work to assess the options available for its computation, and to propose a new processing framework for this transformation, whilst being cognitive of the current state and future trends of computing.

In order to achieve this, it is first required to consider the fundamental algorithm of the discrete Fourier transform, and to consider past approaches in improving computational efficiency. Alternative computational techniques such as the Fast Fourier Transform (FFT) and Goertzel algorithm are considered, however variations of techniques such as these or techniques carrying similar properties are only considered if they have at least one unique characteristic to offer.

University of Cape Town

## 2.4 The Discrete Fourier Transform

To compute the classical Discrete Fourier Transform (DFT), consider a band-limited continuous signal $f(t)$, sampled periodically at some rate $T_s$, which can be represented as the linear sum of complex exponentials by the relation [59]:

$$F[u] = \frac{1}{N} \sum_{n=0}^{N-1} f[n] \exp\left[-j\frac{2\pi nu}{N}\right] \quad \text{for} \quad n \in Z \tag{2.7}$$

where:

$$f[n] = f(nT_s) \qquad \text{and} \qquad n \in Z \tag{2.8}$$

Using the definition for the classical DFT in Equation (2.7), the computational complexity for the transformation, as well as the data communications required to execute this transformation can be assessed.

For any DFT point $u$, the computation to be performed is a series of point-wise multiply-additions. Given a value for $u$, the data sequence denoted by $f[n]$ of length $N$ is first multiplied element for element with a complex exponential of the same length, where the value for the exponential is a function of $u$. This highlights the first computational characteristic of the transform - all multiplications performed require complex arithmetic.

The point-wise multiplication implies that all elements of both vectors need to be accessed per DFT point computation, requiring both temporal and spatial locality in terms of memory access patterns. For each individual point $u$, no computation from past or currently running computations are required resulting in no data dependencies per computation (except for the input data sequence).

University of Cape Town

Since no data dependencies exist no inter-process or inter-processor communications are required. As a whole, the computation of the DFT permits wide-scale parallelism, however the caveat is the requirement for repeated data access per DFT point $u$, imposing a high memory access cost for a large value of $N$.

Furthermore, if a single change were to occur in the input sequence, the entire transform (all points $u$) would need to be recomputed. Another factor to consider is frequency selectability. Each DFT point computed corresponds to a fixed value of $u$. This naturally means that for any application not requiring a full spectral decomposition of all values of $u$, a shorter length DFT of custom $u$ values can be computed. While this may not be applicable in many cases, applications involving filtering for example can leverage this benefit.

Table 2.1 summarises these properties. Sequentially, the DFT in the classical sense would cost $N^2$ operations, and was computationally intractable for most computing systems, even for a moderate size $N$(the computational complexity further increased by an order of $N$ for multi-dimensional transformations).

Table 2.1: Computational Properties of the DFT

| *Property* | *Detail* |
|---|---|
| Available Parallelism | Wide-scale parallelism |
| Arithmetic | Complex |
| Co-efficient Storage | N terms |
| Frequency Selection | Selective |
| Computational Complexity | $N^2$(Multiply-Add) |
| Data Dependencies | None |
| Inter-Process Communications | None |
| Operational Latency | Cycle time: Complex arithmetic |
| Memory Access | Multiple samples<br>Temporal locality<br>Spacial locality |

University of Cape Town

## 2.5 The Goertzel Algorithm

The heart of the computation of the discrete Fourier transform relies on the use of complex exponentials for analysis, and it is possible to utilise the periodic nature of the complex exponentials to reduce the number of computations. The Goertzel algorithm is an example of the use of the periodicity in computing the Fourier transform [59], and enables the computation to be implemented as a matched filter for any frequency $u$ [60]. The Goertzel algorithm is a recursive technique, and the transfer function $H_u[z]$ is expressed in Equation (2.9) [60]:

$$H_u[z] = \frac{1 - \exp\left[j\frac{2\pi u}{N}\right] z^{-1}}{1 - 2\cos(\frac{2\pi u}{N})z^{-1} + z^{-2}} \tag{2.9}$$

The transfer function expressed in (2.9) describes the second-order recursive computation. The second-order computation has the benefit of reducing the number of computations when compared to the direct DFT, but it is still proportional to $N^2$. The Goertzel algorithm does allow selective values for $u$, enabling custom decomposition. Each output can be computed independently, and therefore the computation can be implemented in parallel using multiple concurrent threads.

For each change in the input sequence $f[n]$, the output must be re-computed, which in turn requires the convolution of the entire sequence $f[n]$ with the respective complex exponentials. The memory access pattern therefore exhibits both temporal and spacial locality (assuming the data sequence is stored in a contiguous address space), and requires cache coherence if multiple local caches are supported on the executing hardware.

University of Cape Town

The operational latency is based on the cycle time for the complex arithmetic, and is therefore platform dependent. Ideally the operational latency is kept to a minimum, however the process time will be affected by attributes such as the memory access time. It is the aim of this study to evaluate and identify a technique which will minimise this metric, and promote scalable parallelism. A further benefit of the Goertzel algorithm is the minimal co-efficient storage requirements. The algorithm does require the complex exponential ($\exp\left[j\frac{2\pi u}{N}\right]$) to be computed for multiple values of $u$, however due to the recursive property, only the $2\cos(\frac{2\pi u}{N})$ and $\exp\left[j\frac{2\pi u}{N}\right]$ terms need to be stored.

Table 2.2: Computational Properties of the Goertzel Algorithm

| *Property* | *Detail* |
|---|---|
| Available Parallelism | Wide-scale parallelism |
| Arithmetic | Complex |
| Co-efficient Storage | Two terms |
| Frequency Selection | Selective |
| Computational Complexity | Approx $N^2$ multiplications<br>Approx $2N^2$ additions |
| Data Dependencies | Output from previous iteration |
| Inter-Process Communications | None |
| Operational Latency | Cycle time based on complex arithmetic |
| Memory Access | Multiple samples<br>Temporal locality<br>Spacial locality |

University of Cape Town

## 2.6 Arithmetic Fourier Transform

The Arithmetic Fourier Transform (AFT) is another efficient method for computing the Fourier transform, and offers the ability to compute the Fourier series of a periodic complex function [61, 62]. The method is not recursive by nature (with the exception of the implementation in [63]), and has the advantage of reducing many of the multiplications usually associated with computing discrete Fourier coefficients (with a disadvantage of requiring samples at non-uniformly spaced time values) [61].

Later work [64] developed a method which allows uniform sampling, however the power in the AFT is the relaxed dependence on the use of complex exponentials, and the Fourier coefficients are computed using an average of $f(t)$ taken over any set of $n$ points. The theory of arithmetic transforms is based on Möbius function theorems offering only simple multiplications (-1,0,1) [65]. The non-recursive expression [64] for the AFT is given in Equation (2.10):

$$a_u = \sum_{m=1}^{inf} \mu(m) \left[ \frac{1}{mu} \sum_{r=0}^{mu-1} f\left(-\frac{r}{mu}\right) \right] \tag{2.10}$$

where:

$a_u$ is the Fourier Coefficient $u$

$\mu(m)$ is the Möbius function and

$f()$ represents the input sequence

Assessing equation (2.10), one of the first noticeable points is the lack of complex exponential in the expression. The input Fourier coefficient $a_u$ is computed using an average of the selective set of input sequence samples, which are chosen based on the Möbius function $\mu$. The Möbius function introduces simple multiplication (-1,0,1), and hence simplifies the multiplication requirement.

University of Cape Town

Another interesting point (similar to the Goertzel algorithm) is that the computation (for multiple coefficients $a_u$) can be performed in parallel, requiring no inter-point data dependencies. The computational time is based predominantly on the summative arithmetic, and the division to compute an average.

Table 2.3: Computational Properties of the Arithmetic Fourier Transform

| *Property* | *Detail* |
| --- | --- |
| Available Parallelism | Wide-scale parallelism |
| Arithmetic | Simple |
| Co-efficient Storage | None |
| Frequency Selection | Selective |
| Computational Complexity | Varies, but can be as high as N ($a_u$ dependent) |
| Data Dependencies | None |
| Inter-Process Communications | None |
| Operational Latency | Cycle time based on arithmetic |
| Memory Access | Multiple samples<br>Temporal locality<br>Spacial locality |

University of Cape Town

On the one hand, the AFT offers significant advantages in terms of multiplication reduction and simplicity. However, from a holistic spectral methods perspective, this simplicity comes at the cost of Fourier space functionality, as the absence of multiplication in the AFT would imply no multiplication hardware in the framework. The support of complex multiplication based Fourier operations would later require the inclusion of hardware multiplication, and would not promote the reuse of existing hardware to implement the AFT.

Section 2.3 detailed a list of Fourier space computational requirements, some of which require complex multiplication. The end goal of this work is to create a processing framework which will be tailored to the computational requirements of the spectral methods class. This naturally means any hardware developed should not favour any one characteristic, but offer a general-case hardware platform. Implementing hardware to support the arithmetic Fourier transform method would go against this core aim, as the multiplication required need only be real, not

complex. Features such as parallelism, selective frequency selection, and no data dependencies (with the exception of the time domain samples) are valuable attributes, and will be used to narrow down suitable techniques for this framework.

## 2.7 The Fast Fourier Transform

Transitioning between time domain and Fourier space has been a topic of much research spanning many years. The Fourier transform, and in particular the discrete Fourier transform (if considering discrete time systems), carries a hefty computational price tag of the order of $N^2$(for a single dimension), and was considered computationally intractable for many years for all but the most advanced computing systems. An alternative method to computing this transform was essentially needed to turn a beautiful theory of data decomposition and analysis into a practical realisation.

In 1965, work published by Cooley and Tukey, and aptly named the Fast Fourier Transform (FFT) [66] described a computationally efficient means to compute the discrete Fourier transform in $Nlog_2N$ operations. This revelation spurred a flurry of activity across the many disciplines, as it had become possible to utilise the transform in a practical way on hardware of the day(a more expansive exposition on the topic can be found in [67]).

University of Cape Town

The hardware available 20 years ago is vastly different from the hardware available today, and through significant improvements in sequential processing, the FFT gained widespread popularity as the algorithm of choice for Fourier decomposition. The advent of faster computing permitted more detailed analysis and near realtime performance for mission critical systems.

As detailed previously, sequential computing reached a bottleneck in performance, forcing parallel approaches to be considered more seriously. This in turn meant that the free speedup attributed to consistent sequential processor improvements for legacy code fell away. Not all FFT implementations were software based, and dedicated Application Specific Integrated Circuits (ASIC's) were

developed for specific use, or more flexible hardware platforms such as Field Programmable Gate Arrays (FPGAs) have been used.

The algorithm underpinning the FFT is based on a divide-and-conquer principle. Two approaches are possible, where the DFT is decomposed into successively smaller DFT computations. Decimation-in-time is one such approach where the input sequence $f[n]$ is decomposed into successively smaller subsequences, or, decimation-in-frequency when the output sequence $F[u]$ is decomposed into smaller subsequences [59]. Both approaches are most convenient when $N$ is a integer power-of-2($N = 2^v$ where $v \in \mathbb{N}$).

Concentrating on the former approach (decimation-in-time), the output $F[u]$ is computed by splitting $f[n]$ into two $N/2$ sequences(for even and odd points). In fact, if $N/2$ is even, the $N/2$ sequence can be further separated into two $N/4$ subsequences, with this process repeating until only a two-point transform remains. This will result in $log_2 N$ stages, where the final output $F[u]$ is produced by multiplying and combining the two-point transform with the complex twiddle factors required per stage. Performing this decomposition as far as possible results in a complex multiplication and addition cost of $N log_2 N$ [59]. Taking a snapshot of the mathematics after the first decomposition reveals the general computational trend characteristic of the FFT. This is shown in Equation (2.11):

$$F[u] = H[u] + G[u] \exp\left[-j\frac{2\pi u}{N}\right] \tag{2.11}$$

where:

$$H[u] = \sum_{r=0}^{\frac{N}{2}-1} f[2r] \exp\left[-j\frac{2\pi r u}{\frac{N}{2}}\right] \tag{2.12}$$

$$G[u] = \exp\left[-j\frac{2\pi u}{N}\right] \sum_{r=0}^{\frac{N}{2}-1} f[2r+1] \exp\left[-j\frac{2\pi r u}{\frac{N}{2}}\right] \tag{2.13}$$

University of Cape Town

The decomposition shown in Equations (2.11),(2.12),(2.13) introduces two distinct advantages to the algorithm. Firstly, the reduction in complex arithmetic means significant savings are enjoyed for a large value of $N$. Secondly, the decomposition into two-point DFT's means many individual two-point transforms can be computed concurrently, provided the parallel hardware is available. This however does also lead to a disadvantage when viewed from a parallel perspective. The degree of parallelism reduces per stage when moving toward a final result, as the number of arithmetic steps per stage halves. While this may appear to be advantageous, this decomposition and data aggregation process introduces data dependencies per stage, and future stages cannot begin until previous stages have fully completed.

The data dependencies also imply data movement, and separate threads or processes now require data from previous threads or processes, which often may reside on another processor not sharing the same cache memory. A saving in computational load introduces data dependencies, and data access latencies with a stage depth of $(log_2 N) - 1$, which quantifies the number of stages affected for a given value of $N$.

Shifting focus toward the initial data access, the requirements as described for the DFT in Section 2.4 apply, and for each change in the input sequence, the entire transform for all values of $u$ will need to be recomputed (therefore requiring both temporal and spatial locality memory access, especially if data is stored in a contiguous address space). The values for $u$ are also fixed, removing the flexibility of selective Fourier decomposition. The table listed in 2.4 summarises these properties.

In general from a sequential processing perspective, when compared to a DFT sequential implementation, the FFT algorithm offers substantial computational savings, and as a result, has been widely adopted in practical implementations both in terms of ASIC implementation, as well as multi-processor and FPGA implementations. Before the onset of parallel computing, much effort went into

University of Cape Town

Table 2.4: Computational Properties of the FFT

| *Property* | *Detail* |
|---|---|
| Available Parallelism | Decreasing per stage |
| Arithmetic | Complex |
| Frequency Selection | Fixed |
| Co-efficient Storage | N terms |
| Computational Complexity | $Nlog_2N$(Multiply-Add) |
| Data Dependencies | $(log_2N) - 1$ stages |
| Inter-Process Communications | $(log_2N) - 1$ stages |
| Operational Latency | Cycle time based on complex arithmetic |
| Memory Access | Multiple samples |

University of Cape Town

streamlining FFT algorithms and improving computational efficiency. These improvements came in the form of pipelining [68], improved power efficiency [69] and reduced arithmetic [70].

The introduction of parallel hardware and multi-processing capabilities have resulted in more efficient parallel implementations of the FFT on various platforms [71, 72, 73, 74, 75, 76, 77], as well as automatic library generation for the FFT [78], and hybrid FFT architectures [79]. Many studies have investigated these various techniques to improve on FFT execution, however the underlying limitations are still prevalent in all the modified implementations. A technique which offers benefits of scalable parallelism and custom DFT point computation, as well as reduced data dependencies would provide the necessary flexibility for a universal DFT framework. The following section details such a technique.

## 2.8 Recursive Discrete Fourier Transform

The Recursive Discrete Fourier Transform (RDFT) offers an interesting alternative to computing both the forward and inverse discrete Fourier transform. The one key difference of this technique is the use of recursion in the output computation, where the current output $F[u]$ is used to compute the result in the next iteration.

The output for any given DFT point $u$ is determined iteratively in a sample-by-sample fashion, instead of providing the block of data en-masse (as required for the FFT). For a given DFT length $N$, the technique relies on receiving a single complex data sample as an input, and requires the removal of a single sample located $N$ sample positions previously in the sequence. Stating this more formally:

Let:

$$f_{out} = f[0] \quad \text{be the outgoing sample} \tag{2.14}$$

$$f_{in} = f[N+1] \quad \text{be incoming sample.} \tag{2.15}$$

The new output $F[u]$ can be computed using:

$$F_{new}[u] = \left[ F_{current}[u] + \frac{f_{in} - f_{out}}{N} \right] \exp\left[ j\frac{2\pi u}{N} \right] \tag{2.16}$$

Taking a closer look at the mathematical description in Equation (2.16)(a full derivation can be found in Section 3.4), a number of interesting factors need to be considered. The first point to note is the independence in computation of each DFT point termed $F_{new}[u]$. The computation relies on the current output term, $F_{current}[u]$, as well as the incoming sample ($f_{in}$) and outgoing data ($f_{out}$) samples.

The outgoing sample is simply the sample that is shifted out when a new term is shifted in (offset by $N$). The input data sequence should be seen as a vector of length $N$ used for transformation. This data sequence has not undergone any

University of Cape Town

additional windowing (more on this to follow), however could be understood to have been limited by means of a rectangular function, as no data is used outside this input vector for computation for a given iteration. These terms are simply summed and later multiplied by a single complex exponential which represents the Fourier shifting operator. With the exception of the incoming data sample, no other data needs to be transferred for computation (all other operands can be considered internal to the computation), and no data dependencies exist. In fact, the sample based processing provides additional benefits not inherent to other considered techniques.

The first significant benefit is the reduced overhead of input data transfer. For techniques such as the FFT, the full input data sequence $f[n]$ needs to be transferred to compute the transform, and the computation repeated for every change of input, even if only a single data point is added. From a computational complexity and overhead perspective, a sequential implementation carries the cost of order $O(N^2)$, however a parallel implementation carries the cost of $O(N)$, provided $N$ threads are executed simultaneously.

The ability to resolve a DFT output for a single input sample change highlights another benefit. A single sample change in a sequence implies a re-computation for algorithms such as the DFT and FFT. While this is possible, the overhead associated with these techniques can be prohibitive to this approach (each sample change would incur $N^2$ computations for the DFT, and $Nlog_2N$ for the FFT), and more generally a full new block of $N$ samples is acquired and processed, rather than computing after only a single sample change(assuming the hardware could in fact process $N$ samples before the next sample is captured). This naturally reduces the time-frequency resolution that can be resolved, as the time-frequency resolution is now a function of both sample rate as well as processing latency (before the next set of samples can be gathered).

The sample-by-sample updating offered by the recursive DFT is limited by the processing latency, as the output is updated after every iteration, rather than recomputed for an entire block of data. It should be noted however that sample-

University of Cape Town

based updating is not beneficial to every application, and if data is only ever presented in a block fashion for sample-by-sample processing, the same output after processing the $N^{\text{th}}$ term will be identical to the result obtained as if an FFT were used, except with a lower processing overhead (if *N-wide* concurrency is exploited).

The recursive DFT also permits frequency selectivity, and is flexible and applicable to applications that require only choice decompositions. Table 2.5 details the properties of this technique. From an architectural perspective, the recursive Fourier transform promotes a simple framework. Like the DFT, data dependencies are not an issue, and therefore pipeline stalling and processor idle time is not an issue.

A benefit to the recursive Fourier transform is the minimal data transfer needed for an update - only a single sample (real or complex) is required per update. The processing latency and throughput is a function of the complex addition and multiplication required in the computation, and data such as complex exponential values and current output values can be stored internally.

Table 2.5: Computational Properties of the Recursive DFT

| *Property* | *Detail* |
|---|---|
| Available Parallelism | Wide-scale parallelism |
| Arithmetic | Complex |
| Frequency Selection | Selective |
| Co-efficient Storage | $\frac{N}{4}$ terms |
| Computational Complexity | 1(Multiply-Add) |
| Data Dependencies | Output from previous iteration |
| Inter-Process Communications | None |
| Operational Latency | Cycle time based on complex arithmetic |
| Memory Access | Single input sample |

University of Cape Town

## 2.9 Windowing Data

The discussion to this point has largely excluded data windowing prior to Fourier transformation. The Fourier transform makes the assumption that the data sequence under transformation is in fact periodic, which in practice is not always the case. To overcome this shortfall, a non-periodic sequence can be manipulated to appear periodic by applying a window function which approaches zero in the beginning, and at the end of the sequence.

Altering the data sequence in this manner carries the benefit of improved side lobe suppression in Fourier space - an important factor to consider when assessing transformed data. Windowing data in the time domain involves the multiplication of a data sequence with a window of choice (of the same length), or alternatively in Fourier space, requires convolution. There are a wide variety of window functions available, each carrying its own unique benefits in Fourier space. The choice of window function is application dependent, and for this reason is not considered to be a fixed choice when deciding on suitable Fourier implementations.

The system described in this work has taken a generic approach. The data used is effectively windowed with a rectangular function initially, and transformed from an unaltered state into Fourier space. It has been mentioned that the equivalent to time domain multiplication in Fourier space is convolution. This transformation relationship can be leveraged to apply a window of choice once the transformation has completed, and this approach induces flexibility into the overall system, and loosens computational overhead in the initial Fourier transformation. This does however imply initial conditions for the parallel framework in which this system will operate - the framework should support the Fourier properties listed in Section 2.3.

University of Cape Town

## 2.10 Technique Selection

It is the purpose of this chapter to provide an overview of possible transformation techniques that can be used to steer the design of a processing framework for spectral based computations. The details provided for the techniques outlined in this chapter serve to provide a brief overview of the techniques available for consideration.

The assessment criteria was given in Section 1.3, where the factors considered were the degree of available parallelism; data dependencies; data transfers; operational latency; computational complexity and memory access patterns. The body of literature covering Fourier implementations is vast, and the techniques presented represent those that are classical approaches, as well as techniques that offer unique combinations of benefits such as sample-by-sample updating with scalable parallelism.

The assessment criteria are used to refine the selection, and narrow down possible candidate solutions for implementation in a parallel framework. All aspects of the assessment criteria are important, and if not considered, can lead to significant bottlenecks in performance. Starting at the roots, the Discrete Fourier Transform (DFT) was the first to be considered. This classic approach permits wide-scale parallelism, and enables custom frequency computation. A significant advantage is the lack of data dependencies as well as inter-process communications. The disadvantage of the DFT is the high computational price of $N^2$ operations. A similar disadvantage was found in the Goertzel algorithm, except it did require fewer coefficients to be stored, leading to a memory saving.

University of Cape Town

The arithmetic Fourier transform offers a similar bouquet of features, including a lower computational complexity (can be a high as $N$). The advantage of reduced complexity is overshone by the lack of support for complex arithmetic, which is an important property if generality of processing for the spectral methods class is to be achieved. The study went on to evaluate the FFT, and while this is a widely accepted and popular adoption, the features fell short in terms of data dependencies, inter-process communications, and fixed frequency selection.

A final consideration was the Recursive Discrete Fourier Transform (RDFT). The RDFT promotes the same wide-scale parallel features, and requires only the previous results as a data dependency. Computational complexity can be reduced through parallel implementation, and coefficient memory storage need not be larger than $\frac{N}{4}$. A further benefit is frequency selectivity.

It is undeniable that each technique considered has advantages and features valuable to a particular group of applications. The first aim of this study is to identify a technique most adaptable to the entire class. It is of the opinion of the author that a technique offering a unique combination of benefits, and one promoting a simple architectural approach would be best suited for further analysis and investigation.

Reviewing the properties detailed for each technique, it was decided that the technique which best met assessment criteria is the recursive discrete Fourier transform, as it offers wide scale parallelism, permits sample-by-sample output updating, requires minimal data transfers, and supports complex arithmetic for additional Fourier space operations. The chapter which follows provides a more detailed analysis of this technique, and presents simulated performance results as well as a mathematical model to describe noise for fixed-point arithmetic.

University of Cape Town

# Chapter 3

# Spectral Computations: A Recursive Fourier Approach

University of Cape Town

## 3.1 Introduction

The Recursive Discrete Fourier Transform (RDFT) offers an interesting alternative to computing both the forward and inverse discrete Fourier transform outlined earlier. Chapter 2 discussed a number of possible Fourier implementations for consideration, and highlighted important features which would be required for a spectral processing framework. The chapter concluded by recommending one particular technique for practical consideration, namely the recursive Fourier transform.

This chapter continues where Chapter 2 left off, and discusses the recursive Fourier transform in greater detail, and includes characteristics such as arithmetic accuracy, quantization effects and noise modelling, and introduces an error correction and detection algorithm to bound error growth for finite bit arithmetic. The chapter concludes with a simulated system used to highlight the associated benefits of the technique.

## 3.2 Contributions

- Extension to the recursive Fourier transform. Previous work [51, 52, 53, 54, 55] have shown the merit of this technique for the consistent updating of the output as new input samples become available. The work in [51, 52, 53, 54, 55] however focuses on the forward transform, and gives no consideration to the inverse computation. This work addresses this shortfall, as it is important in the processing framework to support the full transformation of data between discrete time and Fourier space. Furthermore, this work produces a finite bit arithmetic model of the recursive discrete Fourier transform and performs analysis to determine the rate-distortion trade off.

- Error correction technique for the correction of fixed-point inherent errors in a recursive implementation. This work extends a prior model [56], and introduces a technique to compute and correct on-the-fly the errors present in the output for fixed-point arithmetic caused by both complex exponential and arithmetic roundoff. An analytical error model is produced, and used to compute the worst-case error expected for a given set of system parameters. This model is also used to produce error correction engines in hardware which can be enabled to compute in real-time the error in the output.

University of Cape Town

## 3.3 Recursive Fourier Transform

It has been previously discussed that when computing the forward transform, algorithms such as the FFT or native DFT can be used to transform a sampled dataset into Fourier space. The caveat to techniques such as these is the requirement that all input data for a given input sequence $x[n]$ needs to be present to compute the transform, and the computation repeated for every change of input, even if only a single data point is added.

To improve the output update rate, the processing latency incurred would need to be reduced, so sampling can resume for the next block of $N$ samples. The computational overhead for algorithms such as the native DFT and FFT is un-affected when adopting this block processing style, as the addition of a single new sample, or even the acquisition of an entire new block of data requires the same processing to obtain an output.

Analysing the computational dependencies of the FFT and DFT, it can be seen that each successive butterfly stage for the FFT is dependent on the output of the previous stage, whilst the DFT requires no inter-point communications. If the input sequence of size $N$ changes, i.e. a new data sample to position $N+1$ were added for example, the entire transform would need to be re-computed.

An alternative to this approach is the recursive DFT. The recursive DFT is based on the principle of updating a current output $F[u]$ as new data is added to the input sequence. The addition of new data points does not imply that the input sequence has to grow in size, but rather imposes the constraint that a fixed window of size $N$ is required, which shifts to include the new sample. If the output is known a priori, an update can be computed by utilizing the Fourier shift theorem combined with multiplying the same complex exponential with the difference of the incoming and outgoing sample.

University of Cape Town

The computational cost for the recursive DFT technique is far lower than the FFT ($O(nlog_2n)$) [51, 59], and is shown to improve by $\log_2 n$ [53]. Various windowing options exist [53], however the discussion following immediately will focus on the use of a rectangular window of length $N$.

## 3.4 Recursive DFT: Forward Transform

The recursive technique is based on the work of [51, 52, 53, 54, 55, 80] and computes an updated output *F[u]* by removing the contribution of the outgoing sample and adding the contribution of the incoming sample. The work performed by Sherlock and Monro [53] provides a full derivation for the recursive Fourier transform which will be discussed in depth through Equations 3.1 to 3.15.

Consider a data sequence $f[n] = f[0], f[1], ..., f[N-1]$ where the resulting $N$-point DFT is $F[u] = F[0], F[1], ..., F[N-1]$. If a periodic data sequence $f[n]$ is shifted by $s$, the Fourier shift relationship applies to the DFT output $F[u]$ by:

$$f[n] \Longleftrightarrow F[u] \tag{3.1}$$

then

$$f[n-s] \Longleftrightarrow F[u]W_N^{su} \tag{3.2}$$

where the complex exponential is defined in a more compact form:

$$\exp\left[j\frac{2\pi su}{N}\right] = W_N^{su} \tag{3.3}$$

The definition in (3.3) allows for a more compact notation, and will be used extensively in this text. Assessing (3.2), it can be seen that a shift by $s$ in discrete time requires the existing output to be multiplied by a complex exponential which

University of Cape Town

is a function of the shifting variable $s$. If a window function of length $N$ were used on the previously defined data sequence, and the window was shifted by 1 ($s$ would be 1 in this case), the new data sequence can be defined as:

$$f_{new}[n] = \left\{ \begin{array}{lll} f[n+1], & \text{for} & n = 0, 1, ..., N-2; \\ f_{in}, & \text{for} & n = N-1; \end{array} \right\} \tag{3.4}$$

where $f_{in}$ represents the new incoming data sample.

The new data sequence $f_{new}[n]$ now consists of the previous sequence shifted by one sample, except at the boundaries of the sequence, where the last sample entry has been replaced with a new sample $f_{in}$ and the first entry in the sequence ($f[0]$) has been shifted out. The window is considered to be fixed, and therefore all data movements are internal within the window, except at the boundaries.

In this case, the outgoing sample will always be the entry located at $f[0]$ prior to being shifted out, and the incoming sample technically at location $f[N]$ (one sample after the window - before the shift). For the general case, it is simpler to define new labels for the incoming and outgoing samples.

Let:

$$f_{out} = f[0] \quad \text{be the outgoing sample} \tag{3.5}$$

$$f_{in} = f[N] \quad \text{be the incoming sample} \tag{3.6}$$

The expression of Equation (3.4) describes the new shifted data sequence $f_{new}[n]$, and while this is correctly stated, it does not take into account the outgoing sample $f_{out}$. Taking this sample into account is an important factor when computing an updated DFT output for a window of size $N$, as the discrete time sequence shifts out an existing sample for which the output was previously com-

University of Cape Town

puted, and shifts in a new sample for which it has not been previously considered. It would now be required to compute the DFT based on this new sequence.

Since the previous sequence resulted in $F[u]$, the new shifted DFT output produces $F_{new}[u]$, which can be computed in part using the Fourier shift relationship shown in (3.2), and computing a single point DFT on the difference of the new data. The expression in (3.4) does not include the relationship of the incoming and outgoing sample, and therefore needs to be re-written.

Using the new definitions stated in (3.5) and (3.6), (3.4) becomes:

$$f_{new}[n] = \left\{ \begin{array}{lll} f[n+1], & \text{for} & n = 0, 1, ..., N-2; \\ f_{in}, & \text{for} & n = N-1; \end{array} \right\} \tag{3.7}$$

$$= \left\{ \begin{array}{lll} f[n+1], & \text{for} & n = 0, 1, ..., N-2; \\ f_{out}, & \text{for} & n = N-1; \end{array} \right\} + \delta_{n,N-1}[f_{in} - f_{out}] \tag{3.8}$$

The Equation in (3.8) shows that the new time sequence is shifted to the left by one sample, and the new sample entry can be represented as the difference between the incoming and outgoing samples ($f_{in}$ and $f_{out}$ respectively) added with $f_{out}$ for $n = N-1$. The data sequence has in effect been circularly shifted, which has the effect of multiplying by $W_N^{su}$ in Fourier space [81]. (It should be noted that the term can be approximated using a CORDIC computation for finite-bit arithmetic as discussed in [82], however if large enough word-widths are possible, computational savings can be achieved by storing a pre-computed value in memory).

University of Cape Town

Applying Fourier shift and taking a single point DFT to Equation (3.8) gives:

$$F_{new}[u] = F[u]\exp\left[j\frac{2\pi u}{N}\right] + \frac{1}{N}\sum_{n=0}^{N-1}\exp\left[-j\frac{2\pi un}{N}\right]\delta_{n,N-1}[f_{in} - f_{out}] \tag{3.9}$$

Using Equation (3.9), computing the updated DFT output requires the previous DFT output $F[u]$ to be multiplied by a complex exponential, and a single DFT point computation using the difference between $f_{in} - f_{out}$ for $n = N - 1$.

Therefore Equation (3.9) can be simplified to:

$$F_{new}[u] = F[u]\exp\left[j\frac{2\pi u}{N}\right] + \frac{1}{N}\exp\left[-j\frac{2\pi u(N-1)}{N}\right][f_{in} - f_{out}] \tag{3.10}$$

$$= F[u]\exp\left[j\frac{2\pi u}{N}\right] + \frac{1}{N}\exp\left[-j\frac{2\pi uN}{N}\right]\exp\left[j\frac{2\pi u}{N}\right][f_{in} - f_{out}] \tag{3.11}$$

$$= F[u]\exp\left[j\frac{2\pi u}{N}\right] + \frac{1}{N}\exp\left[-j2\pi u\right]\exp\left[j\frac{2\pi u}{N}\right][f_{in} - f_{out}] \tag{3.12}$$

$$= F[u]\exp\left[j\frac{2\pi u}{N}\right] + \frac{1}{N}\exp\left[j\frac{2\pi u}{N}\right][f_{in} - f_{out}] \tag{3.13}$$

$$\text{for } \; u = 0, ..., N-1 \tag{3.14}$$

University of Cape Town

Grouping like terms the new output can be computed using:

$$F_{new}[u] = \left[F[u] + \frac{f_{in} - f_{out}}{N}\right] \exp\left[j\frac{2\pi u}{N}\right] \tag{3.15}$$

Using (3.15), an updated output can be computed for each DFT point based on the difference between the incoming and outgoing samples. A further advantage of this technique is each DFT point can compute an update independently, and much like the direct implementation of the DFT [59], select frequencies of interest can be computed if desired. Furthermore, only minimal data (new samples) need to be transferred to the processing elements used for each DFT point. No data dependencies exist (except for the incoming data point which is common to all processing points), and the ability to compute select frequencies of interest makes this algorithm an ideal choice for implementing on a spectral processor.

Characteristics such as common data inputs permit shared data buses and memory, and flexible computation opportunities such as selecting frequencies of interest for computation allows for run-time configurability. The parallel implementation of this algorithm is discussed in detail in a following chapter, however it is important to realise the significance of such characteristics for the system.

### 3.4.1 Initial Computations

Of course the initial output needs to be computed (typically using an FFT) before the recursive DFT can be used [51, 53, 54]. It can be argued that for a given data sequence $f[n]$, the output is already known at time $t = 0$ ($F[u] = 0 \ \ \forall u$) [51]. Prior to any data entering the system, it can be assumed that the DFT output $F[u]$ is initially zero, and therefore can be used as the initial state for the recursive DFT. Using this initial state, the data sequence can be shifted in, and the resulting output updated as the samples are inserted. After $N$ updates, the resulting output matches the DFT output $F[u]$ for window of length $N$. Adding further samples from this point on results in an updated DFT output for each

University of Cape Town

new sample added. Realising that the same output can be achieved without the use of other algorithms such as the FFT permits a resource saving when implementing the system on silicon.

An FFT could have been implemented for the initial computation, however it has been shown that the recursive DFT can achieve the same result therefore alleviating the need to implement hardware for a once-off computation. It is also important to consider the computational cost at this stage. If the recursive DFT were implemented sequentially, the cost would be in the order of $O(N^2)$.

If concurrency were exploited through many smaller processing elements allowing multiple DFT points to be computed concurrently, the cost reduces to $O(N)$(DFT length $N$), if $N$ processing elements are used. A trade-off exists for the initial cost, however the added advantage is no additional hardware is required to support an FFT for a once-off computation.

University of Cape Town

## 3.5 Recursive DFT: Reverse Transform

The work shown in Section 3.4 describes the forward Fourier transform realised using an algorithm which updates the previous DFT output by applying the Fourier shift theorem and computing a single point DFT computation on the difference of the incoming and outgoing samples when shifted in discrete time. The same technique can be applied when computing the inverse DFT, as a reciprocal relationship exists between time and frequency domain when applying the shift theorem.

From a streamed data perspective the inverse recursive Fourier transform is theoretically possible, but lacks practical usefulness. This is primarily due to the nature and behaviour of Fourier components, as they change from iteration to iteration. It is however beneficial to include inverse transformation for a dataset or a subset of a dataset which is the same length as the transformation.

If a DFT of length $N = 64$ were computed and followed by Fourier space operations such as multiplication (the equivalent of time domain convolution), it is useful to be able to inverse transform the new Fourier components to retrieve a time domain representation.

The inverse recursive Fourier transform is included for completeness, and is aimed to promote the applicability of the recursive approach as a base for a parallel processing framework for the spectral methods class. It is important to note that the computational complexity for the inverse transform is of the order $N^2$ if a sequential processing approach is adopted, however, if concurrency is exploited through many processing elements, the cost reduces to *O(N)* (which is the same as the forward transform). The following section derives the inverse recursive Fourier transform, and makes use of the same approach as used by Sherlock and Monro [53].

If

$$f[n] \Longleftrightarrow F[u] \tag{3.16}$$

then

$$f[n] \exp\left[-j\frac{2\pi sn}{N}\right] \Longleftrightarrow F[u-s] \tag{3.17}$$

Applying the same technique as described in Section 3.4, it is necessary to shift in a set of Fourier coefficients of length $N$.

$$F_{new}[u] = \left\{ \begin{array}{lll} F[u+1], & \text{for} & u = 0, 1, ..., N-2; \\ F_{in}, & \text{for} & u = N-1; \end{array} \right\} \tag{3.18}$$

Here, $F_{new}[u]$ represents a shifted DFT output. The same definition as in (3.5) and (3.6) can be applied.

University of Cape Town

Let:

$$F_{out} = F[0] \quad \text{be the outgoing DFT sample} \tag{3.19}$$

$$F_{in} = F[N] \quad \text{be the incoming DFT sample} \tag{3.20}$$

The technique so far is identical to that described previously, except for the change of computed variable which is $F[u]$ in this case. It is now necessary to include the both the outgoing and incoming sample as before:

$$F_{new}[u] = \left\{ \begin{array}{lll} F[u+1], & \text{for} & u = 0, 1, ..., N-2; \\ F_{in}, & \text{for} & u = N-1; \end{array} \right\} \tag{3.21}$$

$$= \left\{ \begin{array}{lll} F[u+1], & \text{for} & u = 0, 1, ..., N-2; \\ F_{out}, & \text{for} & u = N-1; \end{array} \right\} + \delta_{u,N-1}[F_{in} - F_{out}] \tag{3.22}$$

University of Cape Town

The equation in (3.22) shows that the new DFT sequence is shifted to the left by one point, and the new point entry can be represented as the difference between the incoming and outgoing DFT points ($F_{in}$ and $F_{out}$ respectively). Applying the reciprocal Fourier shift property (3.17) and taking a single point inverse DFT of the final term in (3.22) gives:

$$f_{new}[n] = f[n] \exp\left[-j\frac{2\pi sn}{N}\right] + \sum_{u=0}^{N-1} \exp\left[j\frac{2\pi un}{N}\right] \delta_{u,N-1}[F_{in} - F_{out}] \tag{3.23}$$

Using Equation (3.23), the updated time sequence output can be computed. This requires the previous time sequence value $f[n]$ to be multiplied by a complex exponential (where $s = 1$), and a single inverse DFT point computation using the difference between $F_{in} - F_{out}$ for $u = N-1$. Therefore Equation (3.23) can be simplified to:

$$f_{new}[n] = f[n]\exp\left[-j\frac{2\pi n}{N}\right] + \exp\left[j\frac{2\pi n(N-1)}{N}\right][F_{in} - F_{out}] \tag{3.24}$$

$$= f[n]\exp\left[-j\frac{2\pi n}{N}\right] + \exp\left[j\frac{2\pi nN}{N}\right]\exp\left[-j\frac{2\pi n}{N}\right][F_{in} - F_{out}] \tag{3.25}$$

$$= f[n]\exp\left[-j\frac{2\pi n}{N}\right] + \exp[j2\pi n]\exp\left[-j\frac{2\pi n}{N}\right][F_{in} - F_{out}] \tag{3.26}$$

$$= f[n]\exp\left[-j\frac{2\pi n}{N}\right] + \exp\left[-j\frac{2\pi n}{N}\right][F_{in} - F_{out}] \tag{3.27}$$

$$\text{for} \;\; n = 0, ..., N-1 \;\; \text{and} \;\; n \in \mathbb{Z} \tag{3.28}$$

Grouping like terms the new time sequence output can be computed using:

$$f_{new}[n] = [f[n] + F_{in} - F_{out}]\exp\left[-j\frac{2\pi n}{N}\right] \tag{3.29}$$

It should be noted in Equation (3.29) that the term $\frac{1}{N}$ is not present. The term $\frac{1}{N}$ is present as a result of the complex vector exponentials being orthogonal, but not orthonormal, and is therefore required when computing either the forward or inverse DFT. In this case, $\frac{1}{N}$ was applied in theory to the forward transform, and therefore is absent in the inverse transform. The expression in Equation (3.29) is in the same form as (3.15), with the exception of the $\frac{1}{N}$ term, and the negative sign in the exponential. This makes the implementation of both the forward and reverse transforms straight forward as the system structure can remain the same and promotes hardware reuse. The only hardware difference required would be the choice whether to apply $\frac{1}{N}$, and whether to take the complex conjugate of the exponential. This decision would arise when a set of Fourier

University of Cape Town

components are required to be routed to the input of the system to perform the inverse Fourier transform. A case example would be if the forward transform were applied initially to a dataset and the framework later utilised for further Fourier space operations such as multiplication or differentiation prior to inverse transformation. All hardware considerations will be discussed in chapter 4.

## 3.6 Quantization

The expressions discussed for both the forward and reverse RDFT are valid when all components listed in (3.15) and (3.29) are real or complex numbers. Implementing these algorithms in a high precision system capable of double-floating point arithmetic showed no difference when compared to a commonly used DFT algorithm such as the FFT (using the same precision).

University of Cape Town

However, if algorithms (such as the recursive DFT) are implemented using fixed-point arithmetic of a finite bit length, arithmetic and coefficient quantization errors are unavoidable in the output as a result of using a quantization process with fewer quantization levels therefore producing a more coarse approximation. The accuracy of the quantized result ($\hat{x}$) can be measured by comparison to the original ($x$), and defined using a distortion measure $d(x, \hat{x})$. The distortion measure quantifies the cost or distortion when representing $x$ as $\hat{x}$. The squared error is a common method to compute the distortion measure which is defined as [83]:

$$d(x, \hat{x}) = |x - \hat{x}|^2 \tag{3.30}$$

It is naturally desirable to aim to minimise the distortion measure which is achievable when the quantization step width $\Delta$ is small, resulting in numerous quantization levels for a given region defined for the input space. To define this relationship given an input $x$, the number of bits required to represent $x$ as a binary vector is related to the number of quantization levels $Q$ used in the input space to encode and represent the input data as $\hat{x}$. If there are $Q$ quantization levels (and the number of bits to represent all binary codewords are equal in

length), the binary vectors will need $\log_2 Q$(or next largest integer) number of bits. The number of bits or *rate* $R$ of the code in bits per input data sample is defined as [83]:

$$R(q) = \log_2 Q \tag{3.31}$$

It is now important to determine the trade-off in the relationship of distortion $d$ and rate $R(q)$ for this system as until now, no fixed-rate boundaries have been specified. Furthermore, it is important to produce an analytical model which will determine the expected rate-distortion for a given fixed-point implementation specification.

To achieve this, it is necessary to analyse the computation from a generic perspective (i.e. input independent) and break down the expressions into the constituent parts and determine the rate-distortion contribution from each part at any given iteration of the system. It is desirable to determine an expression in which parameters such as rate can be adjusted to determine the effect on overall distortion. It is first required to assess the components which make up the expressions for both the forward and reverse transform (Equations (3.15) and (3.29) respectively), and determine the influencing error sources. Recalling the expressions for the forward and reverse transforms, it can be shown:

Forward RDFT:

$$F_{new}[u] = \left[F[u] + \frac{f_{in} - f_{out}}{N}\right] \exp\left[j\frac{2\pi u}{N}\right] \tag{3.32}$$

Reverse RDFT:

$$f_{new}[n] = [f[n] + F_{in} - F_{out}] \exp\left[-j\frac{2\pi n}{N}\right] \tag{3.33}$$

Assessing Equation (3.32) and (3.33) it can be seen that recursion exists due to the presence of the $F[u]$ and $f[n]$ terms which represent the current DFT output and the current discrete time sequence data respectively. These are used to compute the updated output value for a given index $u$ and $n$. If the computation had no recursion present, the error in the output would be a function of the

University of Cape Town

arithmetic precision and coefficient quantization for that iteration. In a recursive system where the current output is fed back and used to compute the next output for a given iteration, the error which is inherent in the output becomes a function of the error from the previous iteration combined with the arithmetic and coefficient quantization error for the current iteration. In such a system, if left unconstrained, the error will accumulate and grow exponentially and fail to converge. The recursive DFT algorithm can be considered to fall into this category as the algorithm is recursive by nature and will be limited to fixed-point processing (and have finite bit approximations on all arithmetic computations).

Furthermore, with the exception of the $\frac{1}{N}$ term in Equation (3.32), and the negative exponential in Equation (3.33), the two expressions are otherwise identical. Analysing each component separately it is possible to identify the significant contributors to the distortion measure of the output. Considering the separate terms of Equation (3.32) gives:

$$\begin{aligned} F[u] &: \text{Current DFT output. Intially } \; F[u] = 0 \;\; \forall u \\ f_{in} &: \text{Incoming data sample}^* \\ f_{out} &: \text{Outgoing data sample}^* \\ N &: \text{Constant where } \; N = 2^n \; \text{ and } \; N; n \in Z \\ W_N^u &: \text{Complex Exponential} \\ F_{new}[u] &: \text{Updated DFT output} \end{aligned}$$

To progress forward from this point it is necessary to state an assumption which needs to be made and carried through the remainder of the discussion. The two terms tagged with an * (namely the incoming and outgoing data samples for the forward transform in this explanation) are assumed to be error free terms for all calculations. This assumption is justified as the source for the incoming data is considered independent and transparent to the system (recall that the outgoing data is merely the incoming data sample delayed by $N$ sample periods which is the width of the window or number of DFT points being computed). This data could originate from an external analog source having been first sampled

University of Cape Town

and quantized before entering the system, or could have been generated digitally within an existing system.

It is required to analyse the rate-distortion of the forward and inverse RDFT system independently of any external source of distortion, and therefore is not considered as a source of system error. External sources of error which can be inherent in the terms marked with an $^*$ can be taken into account when considering the entire system performance, however will not be considered when computing the rate-distortion for the computational side of this system.

Turning attention to the other terms, the '$N'$ term in the division of $(f_{in} - f_{out})$ is also error free provided it remains an integer within the bit-width of the system (i.e. the value for $N$ is error free however there can be error associated with the result of the division). A further constraint is placed on the $N$ term in the form of values it can take. It is important to keep $N$ a power of two integer as this allows simple hardware division (can be realised as a bit shift operation in logic space). This constraint is on-par with constraints specified for other DFT computing algorithms such as the FFT [59]. The remaining three terms namely $F[u]; F_{new}[u]; W_N^u$ will be affected by finite register length in the system and therefore the effect needs to be considered.

### 3.6.1 Error Sources

The computation described in both the forward and reverse transforms consist of both complex addition and multiplication, and can be represented diagrammatically (forward transform Figure 3.1; reverse transform Figure 3.2). These representations depict the computation as a flow process where for the forward transform case $f_{in}$ and $f_{out}$ are the inputs, and $F_u[l]$ is the current DFT output for point $u$, at iteration $l$. The models shown in Figures 3.1 and 3.2 assume that no error is inherent in the system and can be considered from a theoretical point-of-view.

University of Cape Town

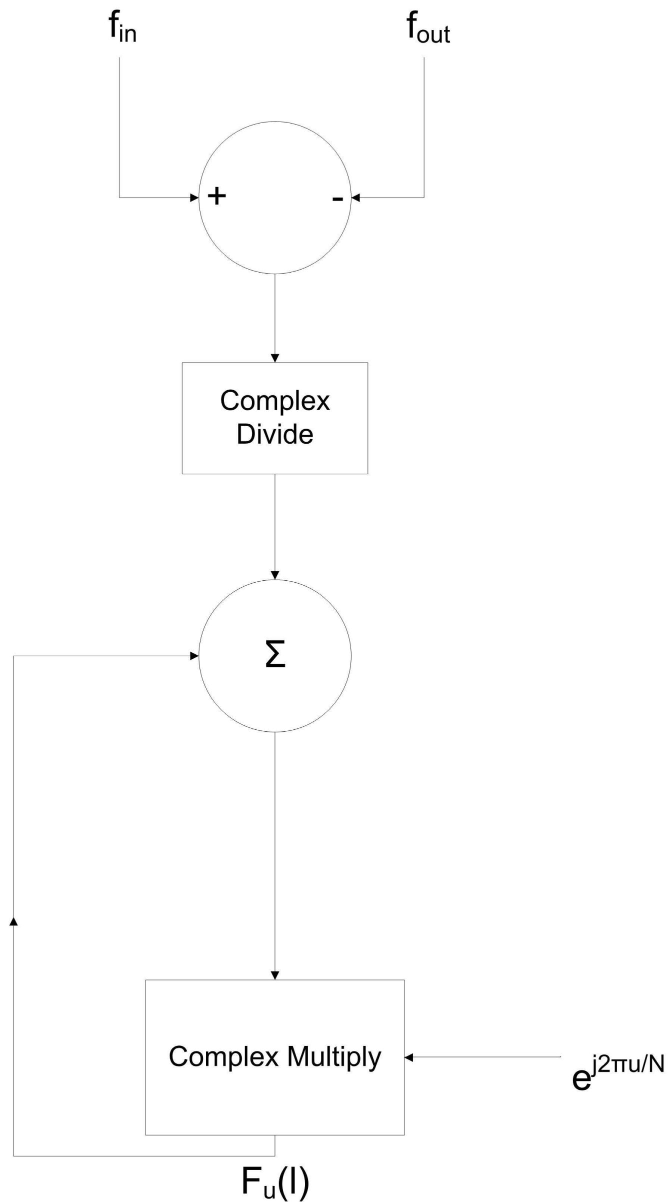



Figure 3.1: Forward RDFT model without error factors. In practice errors will be present in the final output due to coefficient quantization of the complex exponentials, and arithmetic round-off for the complex multiplication. The forward tranform will also include truncation error due to the division operation. The severity of the error is a function of the data representation method such as fixed-point or floating point numbers.

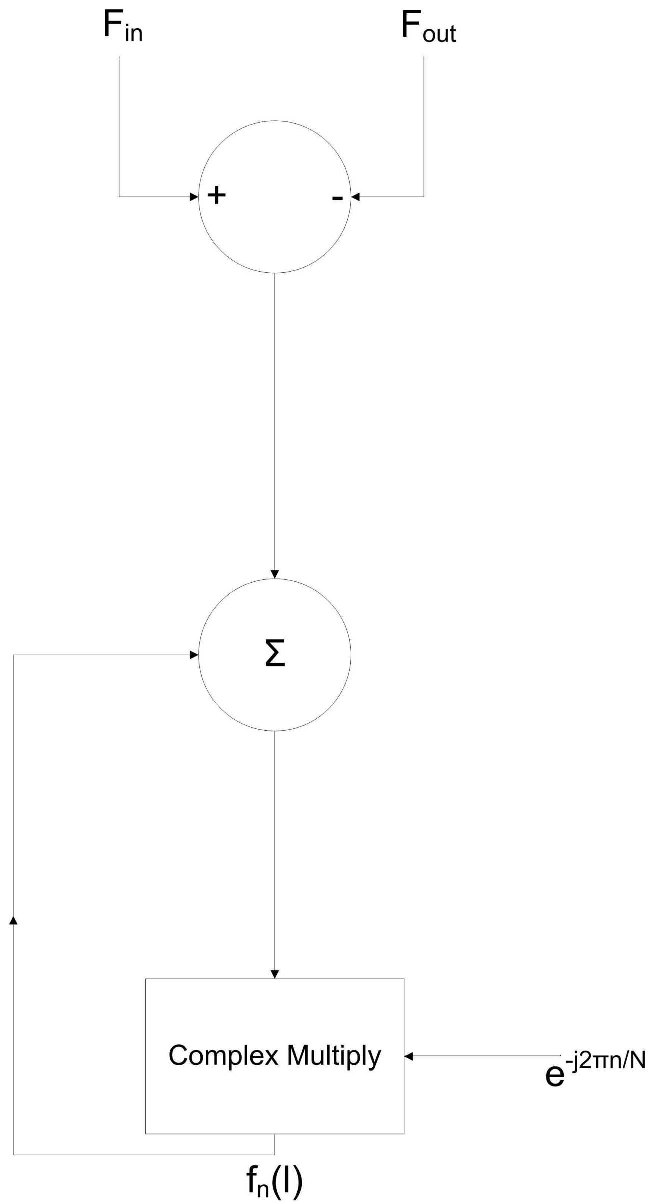



Figure 3.2: Reverse RDFT model without error factors. In practice errors will be present in the final output due to coefficient quantization of the complex exponentials, and arithmetic round-off for the complex multiplication. The severity of the error is a function of the data representation method such as fixed-point or floating point numbers.

To model the system in terms of noise performance, error sources can be modelled as white noise processes [59], and would be contributing factors when determining the overall accuracy of the final computation. The diagram shown in Figure 3.3 includes the contributions from the error factors, and will be used to determine quantization error for the final output for a fixed-point system. Similarly, the diagram in Figure 3.4 shows the error sources inherent in the reverse transform.

In both the forward and reverse transform cases many of the errors are common to both transforms, and so will be discussed jointly (the exception case is for the forward transform where truncation error is inherent due to the division by $N$. This will be discussed separately). The effect of coefficient quantization is inherently non statistical, but a reasonable estimate due to the quantization process can be achieved if basic statistical analysis is performed [59].

To perform this analysis, the error ($\epsilon_C$) (Figure 3.3) is seen as a jitter factor added to the coefficient, so the coefficient value is replaced using the true value plus some value from a white-noise process. Arithmetic round-off error ($\epsilon_A$) is also viewed as a white-noise process, and the arithmetic result is seen as the true value summed with the $\epsilon_A$ jitter factor. It is important to realise that both sources of error ($\epsilon_A$ and $\epsilon_C$) are complex valued sequences where successive values in the sequence are uncorrelated.

The rate-error analysis is performed for the forward transform however is also applicable to the reverse transform where a scaling factor $N$ is omitted. The rate-error analysis result for the reverse transform will follow the description for the forward transform.

University of Cape Town

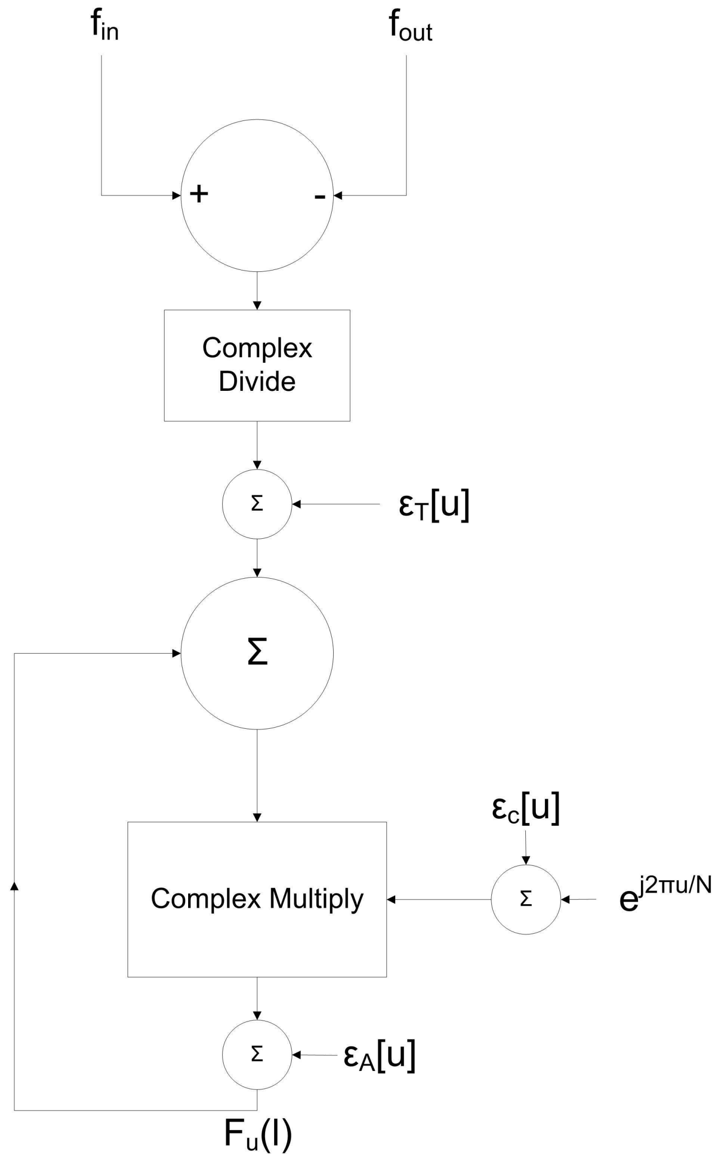



Figure 3.3: The same RDFT model structure is used as shown in Figure 3.1, except the error sources ($\epsilon_A$=arithemtic error;$\epsilon_C$=coefficient quantization error; and $\epsilon_T$=truncation error) are included. The real and imaginary components of the noise sources are also assumed to be uncorrelated, and both $\epsilon_A$ and $\epsilon_C$ are considered to have the same variance $\sigma^2$. The recursive Fourier transform for forward computation can be viewed as a set of parallel processes, where each process represents a DFT point $u$ in the system (a single $u$ is illustrated).

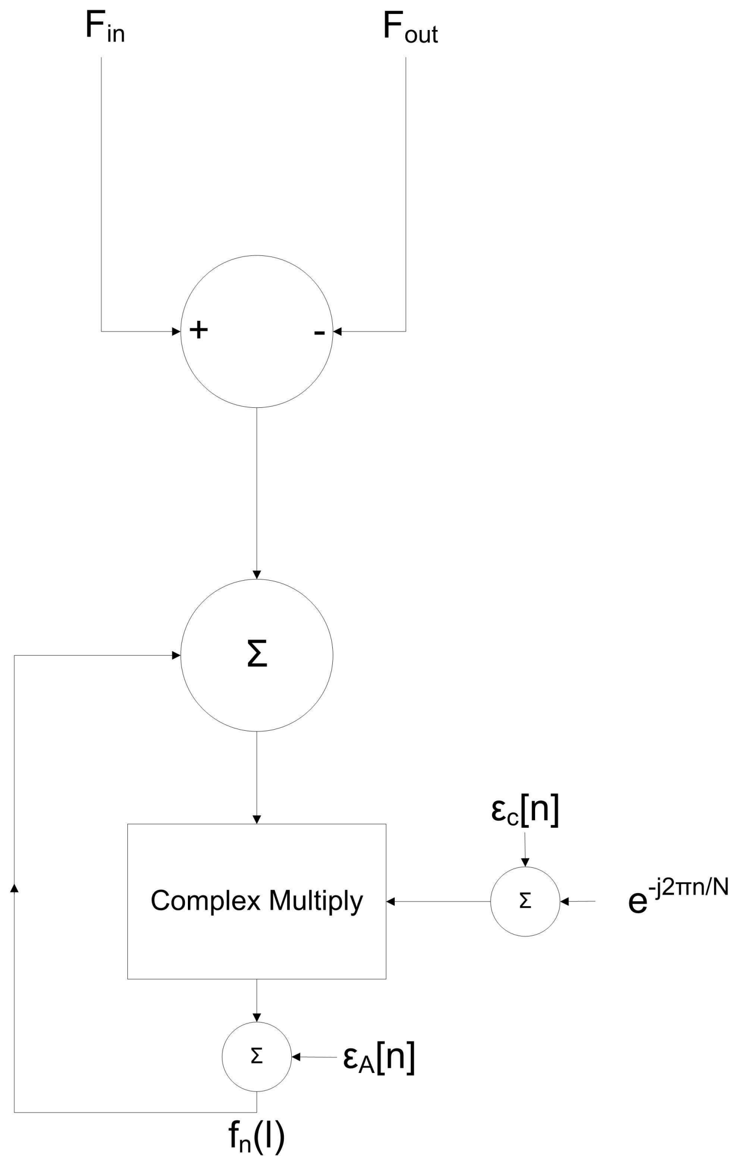



Figure 3.4: The same RDFT model structure is used as shown in Figure 3.2, except the error sources ($\epsilon_A$=arithemtic error;$\epsilon_C$=coefficient quantization error) are included. The real and imaginary components of the noise sources are also assumed to be uncorrelated, and both $\epsilon_A$ and $\epsilon_C$ are considered to have the same variance $\sigma^2$. The recursive Fourier transform for reverse computation can be viewed as a set of parallel processes, where each process represents a DFT point $n$ in the system (a single $n$ is illustrated).

## 3.7 Error Correction

The recursion in the recursive DFT technique for both forward and reverse transforms can be seen when evaluating the expressions in (3.15) and (3.29) where $F_u[l]$ and $f_n[l]$ components represent the two respective outputs for (3.15) and (3.29). In order to compute the next iteration denoted by $l$, the current output is used in conjunction with the incoming and outgoing data samples to compute the next output value. The result is an output which includes the error from the previous iteration which has propagated through, as well as the contributions from the various error sources per iteration. This error is used during every iteration, and as a result, causes a growth in the error present in each output vector. It is therefore necessary to implement a technique capable of determining the state of the error within the system at any iteration, and use this information to adjust the individual outputs in order to correct the final result within a known error tolerance. In this manner the error present at each iteration becomes known and bounded, alleviating the problem of non-convergence.

### 3.7.1 Error Modelling

For the remaining discussion on error correction and modelling, the focus will be specifically on the forward transform. It has been shown that there are only two minor differences between the forward and reverse transforms (the division by $N$, and the negative complex exponential), and therefore the correction method described in this section is equally applicable to both transforms. The issue of non-convergence in error has been experienced previously [54], and various options such as periodic re-computation of the DFT or rather implementation of floating-point arithmetic to alleviate these errors has been suggested.

A further suggestion is to implement double length accumulators or registers to help reduce round-off error in the final output. Unfortunately, the use of double length registers would not assist in reducing the round-off errors, as it is necessary to compute further iterations using the reduced bit length approximation for future iterations [84].

University of Cape Town

However, Kim and Chang[56] suggests a means to investigate the error as it develops, which if modified and used correctly, can allow on-the-fly error correction per point, enabling the reduction of the overall error in the produced output.

Using the notation shown in 3.3, the error at sample time $l+1$ is shown in [56] to be:

$$E_u[l] = \alpha_u W_N^u E_u[l-1] + \delta_u W_N^u [F_u[l-1] + f_{in} - f_{out}] \tag{3.34}$$

where:

$$\delta_u = \frac{[\hat{W}_N^u - W_N^u]}{W_N^u} \tag{3.35}$$

$$\alpha_u = \frac{\hat{W}_N^u}{W_N^u} \tag{3.36}$$

and:

$E_u[l]$ is the computed error for the current iteration

$E_u[l-1]$ is the computed error in the past iteration

$W_N^u$ is the complex twiddle factor and

$\hat{W}_N^u$ is the finite-bit approximation of $W_N^u$

$F_u[l-1]$ is the full precision DFT point output from the past iteration

$f_{in}$ and $f_{out}$ are the incoming sample and outgoing samples

University of Cape Town

It can be seen that the error $E_u[l]$ is defined as a combination of past error computations $E_u[l-1]$; current and past input samples; and the current available DFT point from the previous iteration $F_u[l-1]$ (at full precision), and therefore cannot be assumed to have a normal distribution nor pre-determined mean. The above representation requires that the complex exponential $W_N^u$ is known at full precision, where in a fixed-point implementation, only the finite-bit approximation $\hat{W_N^u}$ is in fact known. This will in turn introduce additional error into the system not catered for in [56].

It is possible to manipulate the expression in (3.34) to reduce the occurrence of $W_N^u$ (which can't be realised in full precision, and therefore reduce the impact of this loss factor), and utilize $\hat{W_N^u}$ where possible (which can be realised in the finite-bit system). Furthermore, the output $F_u[l-1]$ is not known in full precision, so it is necessary to formulate the expression in terms of the output resolution that is known, namely $\hat{F}_u[l-1]$. To do this, it is necessary to start with a theoretical approach of knowing both the full precision output ($F_u[l]$) as well as the finite-bit approximated output ($\hat{F}_u[l]$) for computing $E_u[l]$. Using these two terms, expressing $E_u[l]$ in terms of $\hat{W_N^u}$ and $\hat{F}_u[l-1]$ is shown as follows:

$$E_u[l] = \hat{F}_u[l] - F_u[l] \tag{3.37}$$

where

$$\hat{F}_u[l] = \hat{W_N^u}\left[\hat{F}_u[l-1] + \frac{f_{in} - f_{out}}{N}\right] \tag{3.38}$$

and

$$F_u[l] = W_N^u\left[F_u[l-1] + \frac{f_{in} - f_{out}}{N}\right] \tag{3.39}$$

Substituting (3.38) and (3.39) into (3.37) and letting:

$$F_u[l-1] = \hat{F}_u[l-1] - E_u[l-1] \tag{3.40}$$

University of Cape Town

yields the result:

$$E_u[l] = \sigma_u \left[ \hat{F}_u[l-1] + \frac{f_{in} - f_{out}}{N} \right] + W_N^u E_u[l-1] \tag{3.41}$$

where:

$$\sigma_u = \hat{W}_N^u - W_N^u \tag{3.42}$$

The term $W_N^u$ now only appears once individually, which can be substituted with $\hat{W}_N^u$. This introduces marginal additional error, but the inability to realise a full precision representation of $W_N^u$ has been confined to influence only the feedback error $E_u[l-1]$. A more important benefit is that the error can be computed in terms of the current finite-bit output, $\hat{F}_u[l-1]$. The terms $\sigma_u$ and $E_u$ can be stored in hardware using full precision registers (i.e. no integer allocation), and used to track the error which develops and is embedded in the DFT output.

### 3.7.2 Error Tolerance

The error is computed individually per DFT point, and is later used independently of all other points. Once the error has been computed, it is possible to use this computed result to provide correction to the respective DFT output vectors. It is however not recommended to directly apply the computed correction result on each iteration in which it is computed, but rather to allow the error per DFT vector to accumulate to a tolerable threshold (represented as $\upsilon$) before using the correction vector to rectify the respective outputs.

The purpose of the decision threshold $\upsilon$ is to provide a measure to compare against when assessing the error computed for each DFT output, and the value for $\upsilon$ cannot be less than the smallest bit resolution realised in the DFT output data. Whenever the threshold is equaled or exceeded, the DFT output vector is corrected using the most recent calculated error. It is necessary to accommodate a remainder vector, as there will be a difference between the computed error vector ($E_u[l]$), and the actual vector used in correcting the respective DFT

University of Cape Town

outputs (let's call this $E_u[l]^\star$). The reason for the difference lies in the number of bits allocated to precision when computing $E_u[l]$, and the precision allotted to $F_u$. The computation of $E_u[l]$ requires no integer storage capability, so the full register length is used for precision, whereas the DFT outputs ($F_u$), are required to accommodate integer growth. It is therefore important to realise that the DFT output vectors have fewer bits representing the precision, and so the precision of the DFT output vectors cannot accommodate the full resolution to which the error is computed. A remainder vector (provided to each DFT output in hardware) stores the difference, and is used to seed the next iteration of the error calculation. If the threshold has not been exceeded, the seed for the next iteration becomes the current value computed for $E_u[l]$.

The decision threshold is an important factor when considering convergence of the error. As would be expected, if the error correction were never applied the error would grow exponentially and never converge. This is due to the addition of new arithmetic and coefficient error per iteration in combination with the previous error computed in the prior iteration. If $\upsilon$ were set to the smallest bit resolution possible for the output DFT vector the error correction would be invoked frequently. Issues arise when quantizing the computed error vector ($E_u[l]$) to the maximum precision of the DFT output vector which produces $E_u[l]^\star$ for use in correcting the output (the difference, $E_u[l] - E_u[l]^\star$, is used as the next seed).

At this point only a small error has accumulated which is of large enough magnitude to exceed the threshold and invoke the error correction. The result after quantization of the error correction vector yields the correction vector to be the same value as the threshold. The correction vector therefore contains a quantization error due to the lower precision, and is used to correct the respective output (the remainder is stored and used in the next iteration). The DFT output vector is now a more accurate representation of the true DFT output, and the remainder which is not realisable in the DFT output resolution has not been used to correct the vector.

University of Cape Town

The remainder however is a value somewhere between the maximum resolution realisable in the output vector, and the maximum resolution realisable in the remainder vector (to put into numbers for this system, it would lie somewhere between $2^{-24}$ and $2^{-36}$). This remainder range would be true for all correction cases and the ratio of unusable correction data and usable correction data can at worst be almost 1:1. Stated another way, the amount which is corrected and the amount which is left over as it is not significant enough to exceed the next bit level can be almost equal. The loss of accuracy due to a large remainder to correction ratio is more profound at this level of quantization, and results in under-corrected DFT output vector.

Looking back at the expression for the computation of the DFT output vectors in Equation (3.15), the current DFT output, namely $\hat{F}_u[l]$, is dependent on the past DFT output vector $\hat{F}_u[l-1]$. An under-corrected output therefore impacts the computation of successive output values. If $\hat{F}_u[l]$ were compared to a true output, the difference would consistently grow due to the recursive nature of the algorithm.

If $\upsilon$ is set to a larger value, the error is allowed to accumulate to a larger value prior to correction, making the remainder a far smaller fraction of the overall quantized correction vector value. The remainder is still used as a seed, but is not a large contributor, making the corrected output a more accurate representation with respect to the true value. This comes at the expense of a larger square error magnitude and a longer latency prior to correction of the output vector, but has the benefit of an error factor that converges to a determinable variance with a known mean (the selection of suitable $\upsilon$ values is discussed in the next section).

University of Cape Town

#### 3.7.2.1 Rate-Distortion Simulations

To assess the error factor, simulations were performed using a random test set ($> 10^5$ samples) with a normal distribution. The simulation tests were computed using a Matlab model of the intended system prior to gateware development, and the simulated model of the system applied the expected constraints for fixed point processing. The purpose of the simulation tests was to assess the influence on the total square error of the computation for various threshold values as well as various fixed-point positions. For the first set of illustrations various $\upsilon$ values ranging from $2^{-8}$ to $2^{-24}$ (in 4-bit increments) was used while the fixed-point length was fixed.

The second set of illustrations uses the same computed data but fixes the threshold value while displaying the square error for different fixed-point bit lengths. This allows the relationship of fixed-point bit-lengths as well as the influence of the applied threshold to be determined. In all cases the square error was computed over a total of 64 DFT outputs (and summed - for this system, a 64 point DFT was modelled), with respect to a true DFT output (double floating point precision version). Figures 3.5 to 3.15 illustrate these relationships, and show results for only the *real* component for the complex error. The results are comparable for the *imaginary* component.

Viewing the illustrations shown in Figures 3.5 to 3.15, it is possible to characterise the system in terms of square error performance for various threshold and fixed-point bit length values. All traces in Figures 3.5 to 3.15 are displayed on a single set of axes using a logarithmic scale.

University of Cape Town

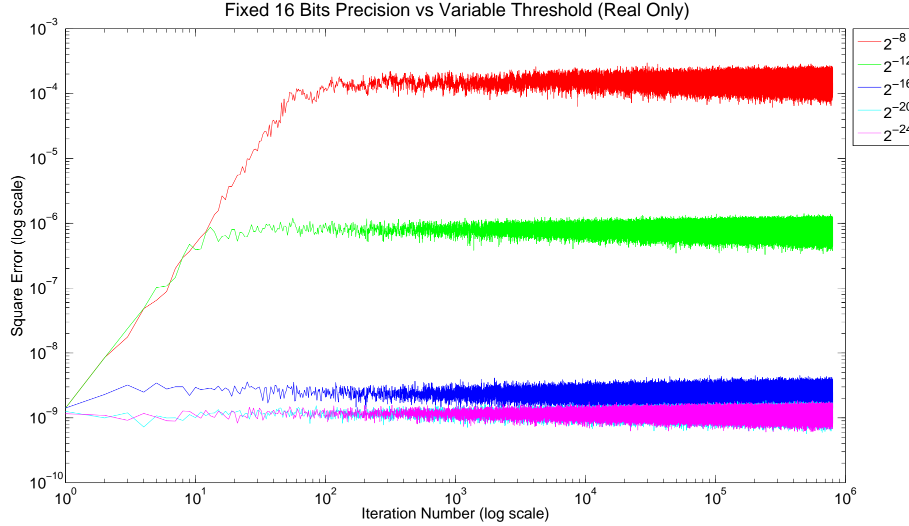



Figure 3.5: Fixed 16-Bits Precision vs Variable Threshold (Real Only). The traces represent the sum of the square errors for each DFT point, and show highly correlative results for threshold values of $\upsilon = 2^{-16}$ to $\upsilon = 2^{-24}$. This indicates that any threshold below $\upsilon = 2^{-16}$ will yield a similar result for a fixed point resolution of 16-bits. It can also be seen that the error correction is invoked after a very short time frame for the threshold values of $2^{-16}$, $2^{-20}$ and $2^{-24}$, resulting in a smaller accumulated, but unbounded error (determined from the consistent broadening of each trace).

University of Cape

Assessing Figure 3.5, the fixed-point bit length is stated to be 16-bits in length, and the threshold is varied from $2^{-8}$ to $2^{-24}$. What is distinctly noticeable at first is the rate of increase in the error within the first $10^2$ sample iterations ($2^{-8}$ and $2^{-12}$ traces). This is due to the total computed error not exceeding the threshold stipulated, and the error is allowed to accumulate after every iteration. The error correction is invoked when the threshold is equaled or exceeded, and the respective output vectors are corrected independently. Each error trace stabilises as the error correction regulates the error in the system. The amount of error tolerated is a function of the stipulated threshold value.

A further important consideration is the issue of convergence in the traces. Examining the traces in Figure 3.5, it can be seen that each trace stabilises to an initial mean for a particular threshold value, however error continues to build as time passes (each trace become broader as the fluctuations are unbounded). Figure 3.5 shows that the traces for the threshold values of $\upsilon = 2^{-20}$ and $\upsilon = 2^{-24}$ are virtually identical (the mean of the trace for $\upsilon = 2^{-16}$ is correlative), indicating that any threshold beyond $\upsilon = 2^{-16}$ will yield a highly correlative result for a fixed point resolution of 16-bits.

Figure 3.5 also indicates that the error correction is invoked after a very short time frame for the threshold values of $2^{-16}$, $2^{-20}$ and $2^{-24}$, resulting in a smaller accumulated error. Whilst this may seem ideal, it was found that as the threshold approaches the fixed point resolution the ratio of unusable correction data to usable correctable data increases and the resulting square error fails to converge.

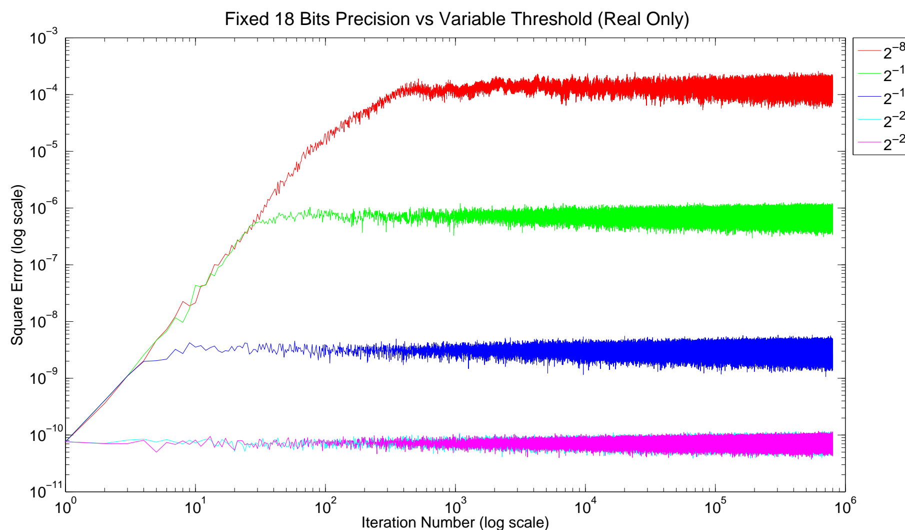



Figure 3.6: Fixed 18-Bits Precision vs Variable Threshold (Real Only). The traces for $\upsilon = 2^{-20}$ and $\upsilon = 2^{-24}$ have shifted by an order of magnitude to the region of $10^{-10}$ due to increased number of bits allocated to precision.

Cape Town

Figure 3.6 illustrates a setup where the fixed point resolution has been fixed to 18-bits and uses the same set of threshold values. The only significant difference between the traces of Figure 3.6 and Figure 3.5 is the traces for $\upsilon = 2^{-20}$ and $\upsilon = 2^{-24}$ have shifted by an order of magnitude to the region of $10^{-10}$. This could have been expected as mean of the square error for small threshold values such as $\upsilon = 2^{-20}$ and $\upsilon = 2^{-24}$ will reduce as a larger number of bits are allocated for precision. This same trend is detectable for Figures 3.7 to 3.10, except that the traces for $\upsilon = 2^{-20}$ and $\upsilon = 2^{-24}$ begin to separate as each threshold value represents a certain error tolerance.

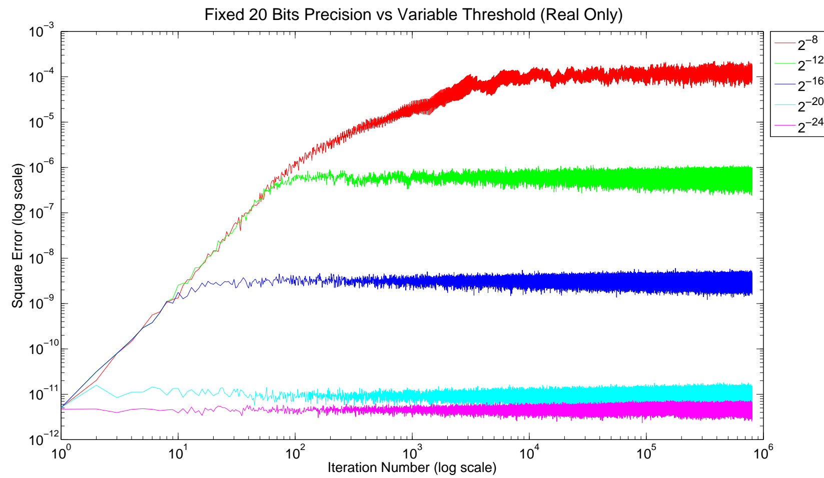



Figure 3.7: Fixed 20-Bits Precision vs Variable Threshold (Real Only). The traces for $\upsilon = 2^{-20}$ and $\upsilon = 2^{-24}$ have shifted by an order of magnitude to the region of $10^{-11}$ and start to separate. The lower error is due to increased number of bits allocated to precision.

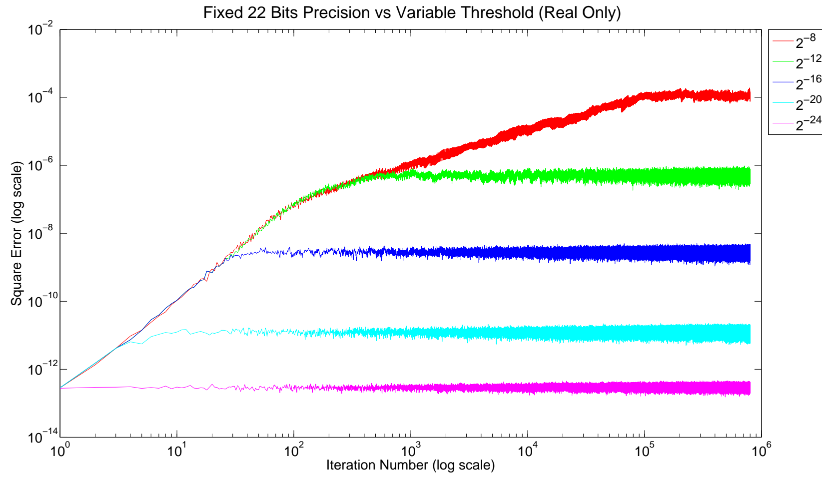



Figure 3.8: Fixed 22-Bits Precision vs Variable Threshold (Real Only). The traces for $\upsilon = 2^{-20}$ and $\upsilon = 2^{-24}$ continue to separate, and the trace for $\upsilon = 2^{-8}$ becomes bounded.

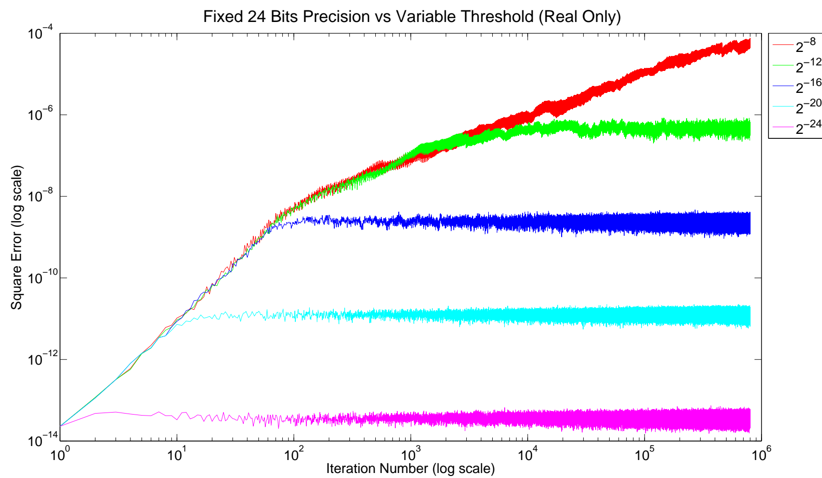



Figure 3.9: Fixed 24-Bits Precision vs Variable Threshold (Real Only). The separation for traces for $\upsilon = 2^{-20}$ and $\upsilon = 2^{-24}$ continue, and the trace for $\upsilon = 2^{-24}$ reaches the lowest bound.

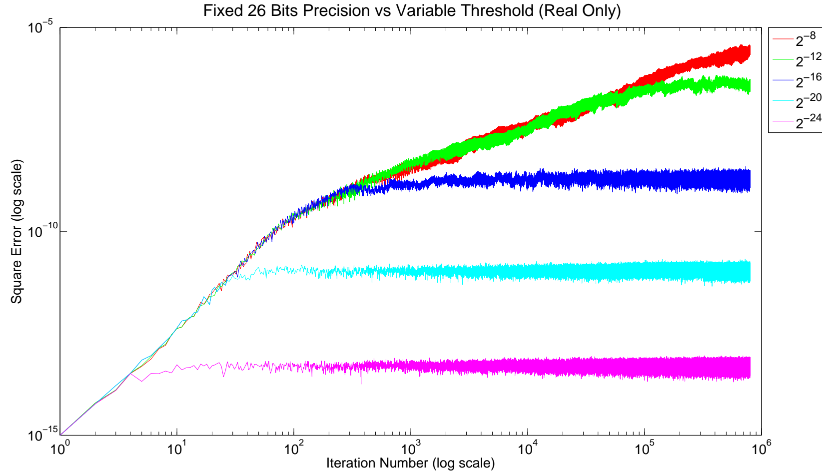



Figure 3.10: Fixed 26-Bits Precision vs Variable Threshold (Real Only). The trace for $\upsilon = 2^{-12}$ stabilises and becomes bounded.

Assessing Figures 3.5 through to 3.10, two additional characteristics can be observed. Firstly, the time required for the initial convergence to a square error mean changes depending on the fixed point resolution. This is understandable as the fixed point resolution plays a role in the overall quantization error, and the fewer the bits allocated to precision, the larger the quantization error. The rate of error increase will therefore be higher for lower fixed point resolutions, and lower for higher fixed point resolutions.

Secondly, it can be noted that from a fixed point resolution of 22-bits (Figure 3.8), the error for $\upsilon = 2^{-8}$ not only converges initially, but converges to a stable mean. This can be observed in Figure 3.8 (red trace). Analysing this trace, it can be seen that while there is fluctuation (due to the error correction system being invoked and the outputs being corrected), the deviation does not grow (become broader) in the same manner as the other traces in the figure. This same situation occurs for a fixed point resolution of 26-bits (Figure 3.10), however extends to both threshold values for $\upsilon = 2^{-8}$ and $\upsilon = 2^{-12}$. The results showed that as a guideline, the projected error only converges for cases when the difference between the fixed-point length and the threshold is greater than or equal to 14-bits.

University of Cape Town

Using a case example, for a fixed-point length of 24-bits, the value of $\upsilon$ needs to be at least equal to $2^{-10}$ (or larger) for the output to converge. A converged output in this instance refers to when the output provided by the error corrected recursive Fourier transform no longer continues to diverge when compared to the true value of the DFT. In this case, the error would stabilise at a particular mean value, providing a constant, but determinable error.

Setting $\upsilon = 2^{-10}$ or larger, results in a converged error with a value dependent on the selection of $\upsilon$. Using a larger $\upsilon$ value results in a larger accumulated error prior to correction per point, and will naturally result in a larger overall square-error value when considering all the DFT outputs. It is important to realise that the correction per point does not necessarily happen across all points simultaneously, but rather independently as each point is computed individually.

The amount of error and the rate of error growth per point will vary as the amount of coefficient quantization depends on the coefficient value. It is worthwhile pointing out that in some cases there will be no coefficient quantization error for coefficient values of unit amplitude. From a theoretical standpoint this is ideal, however this creates an additional problem when implementing this error correction in hardware as $\sigma_u = 0$, therefore nulling the first term when computing $E_u[l]$. This will be discussed later in the hardware design chapter. For this reason, the square error will not approach or become zero as there will be an error remainder due to the quantization process, as well as the fact that not all points are corrected simultaneously. The sum total of the individual corrections per point at any given iteration is detectable in the minor fluctuations seen on any of the traces in the illustrations, and varies according to the current state of each DFT output.

University of Cape Town

The second method to illustrate the simulated datasets is to maintain a fixed threshold value and vary the fixed point resolutions. This has the benefit of grouping all the traces for a given threshold value while displaying the error response as a function of fixed point length. Illustrating the data in this fashion makes it easier to discern the various growth rates which are a function of the number of bits available for fixed point representation. Assessing Figures 3.11 to 3.15, the error traces start at the various orders of magnitude, however the point of interest is the change in growth rate as a function of threshold value.

Focusing on 3.11, it can be seen that all traces have similar gradients and also eventually converge to the same square error magnitude. This is due to the fixed threshold value and could be expected. While it is not clear from the overlapped traces, it can be seen that many traces such as those representing 24 and 26-bits respectively stay bounded in their trace deviation as the trace approaches the converged square error mean value. This is not always the case as can be seen in the remaining Figures 3.12 to 3.15.

The traces in the figures which follow begin to separate and converge to different square error mean values (similarly shown in Figures 3.5 to 3.10). The particular feature of interest in these plots is that for a given error tolerance, the same mean error could be achieved while using fewer bits to represent the data. It is important to consider if in fact the deviation from the mean does remain bounded, and this is clearly possible as shown in these traces.

University of Cape Town

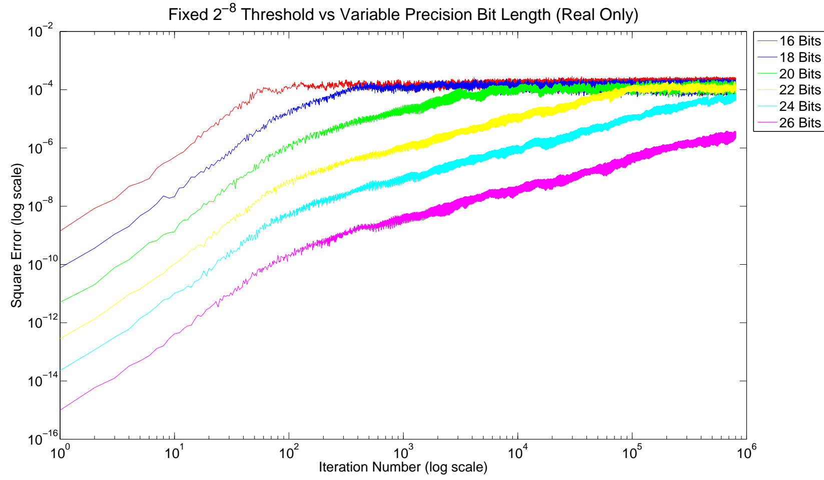



Figure 3.11: Fixed $2^{-8}$ Threshold vs Variable Precision Bit Length (Real Only). All traces have similar gradients and also eventually converge to the same square error magnitude. This is due to the fixed threshold value.

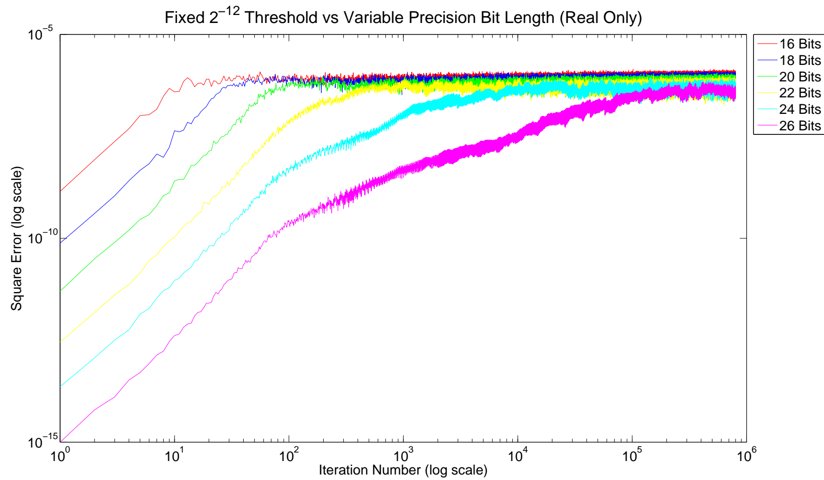



Figure 3.12: Fixed $2^{-12}$ Threshold vs Variable Precision Bit Length (Real Only). Similarly to Figure 3.11, the traces still eventually converge to the same value. The rate of convergence is however higher.

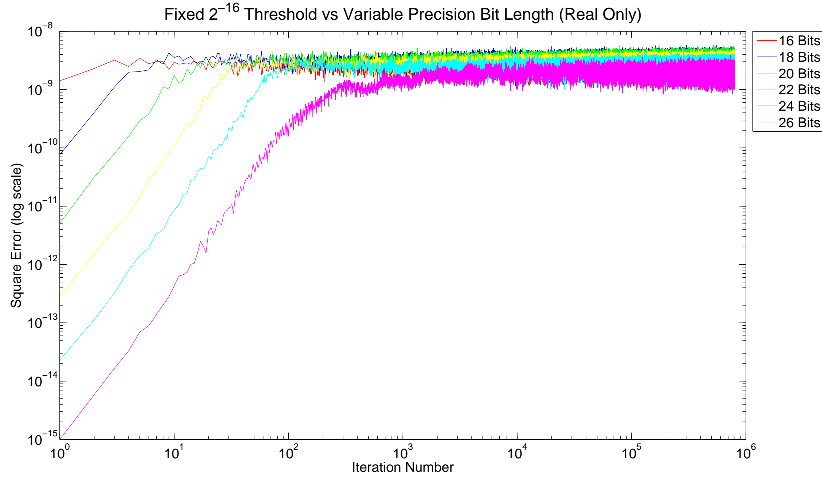



Figure 3.13: Fixed $2^{-16}$ Threshold vs Variable Precision Bit Length (Real Only). In a similar fashion to Figures 3.11 and 3.12, the rate of convergence continues to increase. The unbounded nature of the traces become more apparent as the lower threshold results in a higher rounding error.

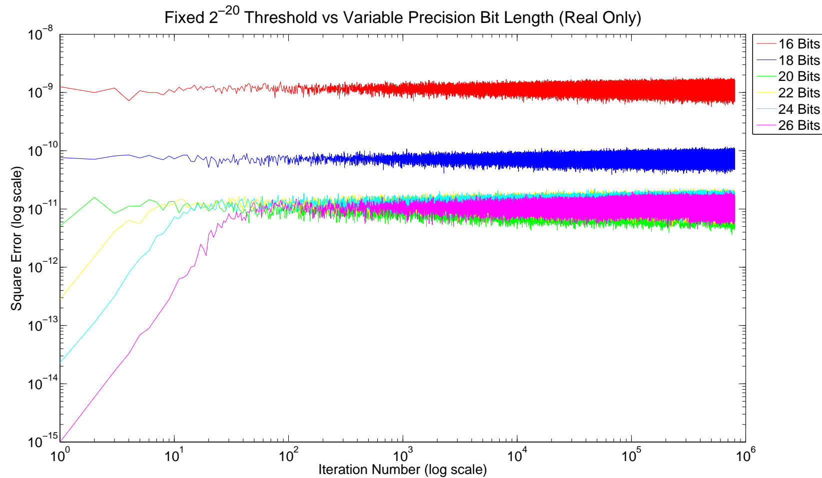



Figure 3.14: Fixed $2^{-20}$ Threshold vs Variable Precision Bit Length (Real Only). The traces begin to separate without all converging to the same mean. The traces for 20, 22, 24, 26-bits however do still approach the same mean, which indicates that additional bit length provides little benefit for $\upsilon = 2^{-20}$.

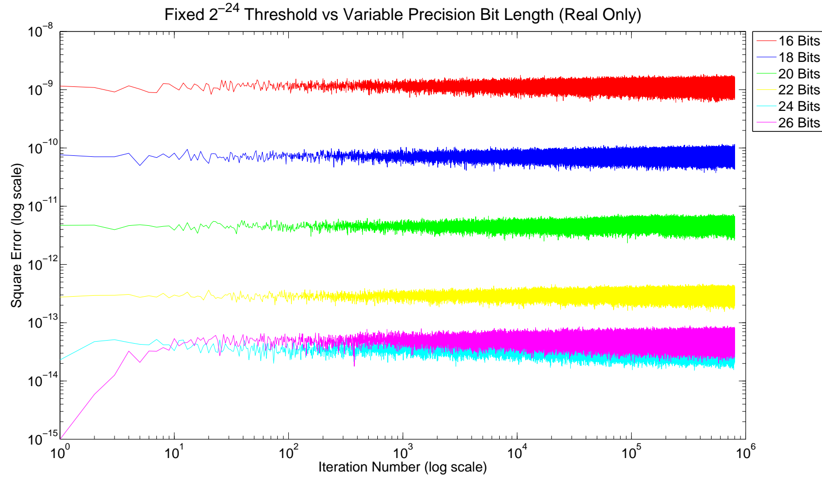



Figure 3.15: Fixed $2^{-24}$ Threshold vs Variable Precision Bit Length (Real Only). Under the condition of $\upsilon = 2^{-24}$, it can be seen that the number of bits allocated to precision plays an important role in the initial converged error value. No trace however remains bounded. To alleviate this issue, it was found that a bit difference of 14-bits should be enforced between the threshold and the number of bits allocated to precision.

All of the Figures (3.5 to 3.15) highlight important characteristics for the system under investigation. The figures illustrate data only for the *Real* component, however the same results apply to the *Imaginary* component as well. The important factor to realise is that implementing the error correction expressed in (3.41) while specifying a threshold $\upsilon$, results in a method to compute and correct a finite-precision recursive DFT computation to high accuracy with a known mean square error.

It is worthwhile to point out that the use of the error correction when computing the recursive Fourier transform provides substantial improvement in the number of iterations that can occur for which the recursive Fourier transform provides useful data. As discussed, the selection of $\upsilon$ determines the overall error, and does play a pivotal role in the rate of error increase and potential convergence. The use of the error correction helps significantly in the containment of

University of Cape Town

the accumulated error, however goes not provide a guarantee of error-reduced operation for an indefinite number of iterations. Over a prolonged period, it could still be beneficial to reset, and allow any unavoidable accumulation to be removed. This cost can be considered minor when compared with re-computing the output vectors every few thousand iterations, or the overhead of implementing floating-point arithmetic. The section which follows focuses on individual DFT component analysis, followed by modeling the error correction method in a generic formulation to determine the expected error and noise-to-signal ratio for any arbitrary input sequence.

#### 3.7.2.2 RDFT Component Analysis

The analysis performed in Section 3.7.2.1 focuses on viewing the combined square error measurements for various threshold values as well as fixed-point positions. This analysis is useful for providing an indication of the relationship of threshold and fixed-point precision and convergence, and if functioning correctly, the trace for a given threshold and fixed-point precision should level off and remain bounded during operation.

It is clear in Figures 3.5 to 3.15 that even in the bounded error cases, the error fluctuates as the error correction operates due to consistent coefficient and arithmetic quantization taking place per iteration. It is shown in the following section (Section 3.7.3) that these error sources can be considered as noise sources and modelled appropriately. It is the intention of this section to analyse the role of the error correction at an individual frequency component level, and provide evidence of correct operation of the error correction algorithm.

University of Cape Town

Each frequency component operates independently and in turn is corrected independently of each other. Noise sources local to each component are modelled similarly, except will each have its own unique value based on coefficient value and the level of quantization used. It is therefore useful to concentrate on individual DFT components, and consider cases of uncorrected and corrected outputs when compared to a known reference such as the FFT (using the same computational precision).

To perform this comparison, the difference between the RDFT output (corrected and uncorrected) is computed with respect to the same fixed-point precision FFT, and the envelope of the error is computed (real data used for illustration). Using the envelope a least-squares linear approximation is made as this enables a trend to be observed in the output error. Ideally in the case of a corrected output, the difference of the DFT component viewed over time when linearly approximated should have a zero gradient, or be linearly decreasing if the observation includes error prior to the error exceeding the threshold. A zero gradient would indicate a consistent error, or a negative gradient an error value which is consistently decreasing. This comparison was performed for the threshold values of $2^{-10}$, $2^{-12}$, $2^{-14}$ for DFT components 2 and 3 (first DFT component omitted as $\sigma_u = 0$, and will not provide a typical computation), and is illustrated in Figures 3.16 to 3.21.

University of Cape Town

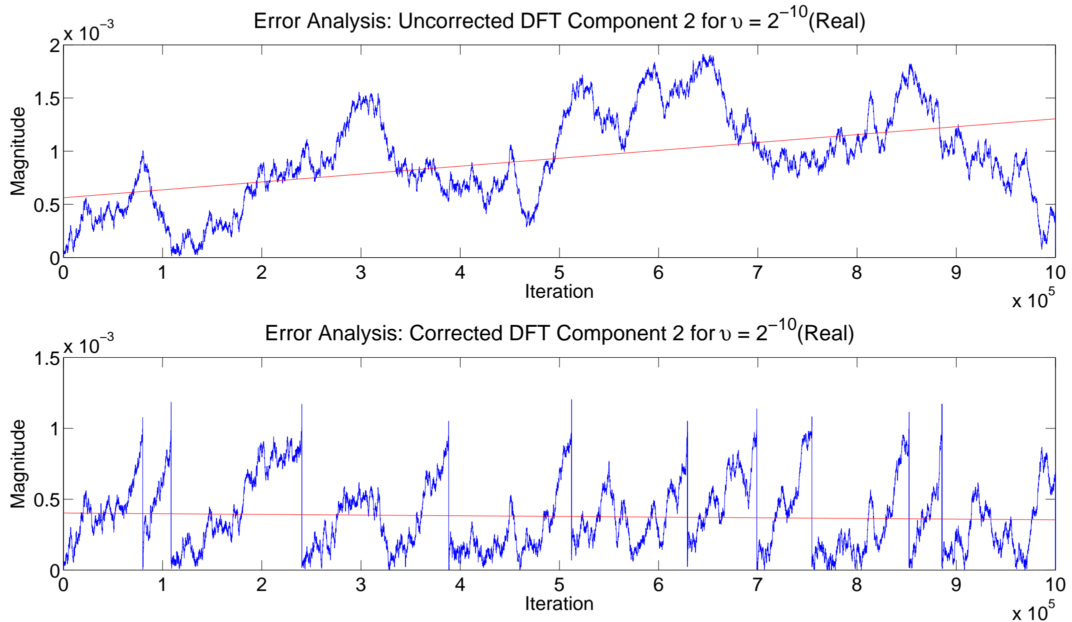



Figure 3.16: Error analysis for DFT component 2 using $\upsilon = 2^{-10}$. The error at any iteration is a function of input data and will therefore fluctuate. However, the recursion present will cause an accumulation of error as shown as a linear fit (top red trace). In this case a large enough error tolerance is allowed to permit sufficient error accumulation and therefore correction without discarding crucial information through the rounding process. The gradient of red trace (corrected component) is $-4.77 \times 10^{-9}$.

University of Cape Town

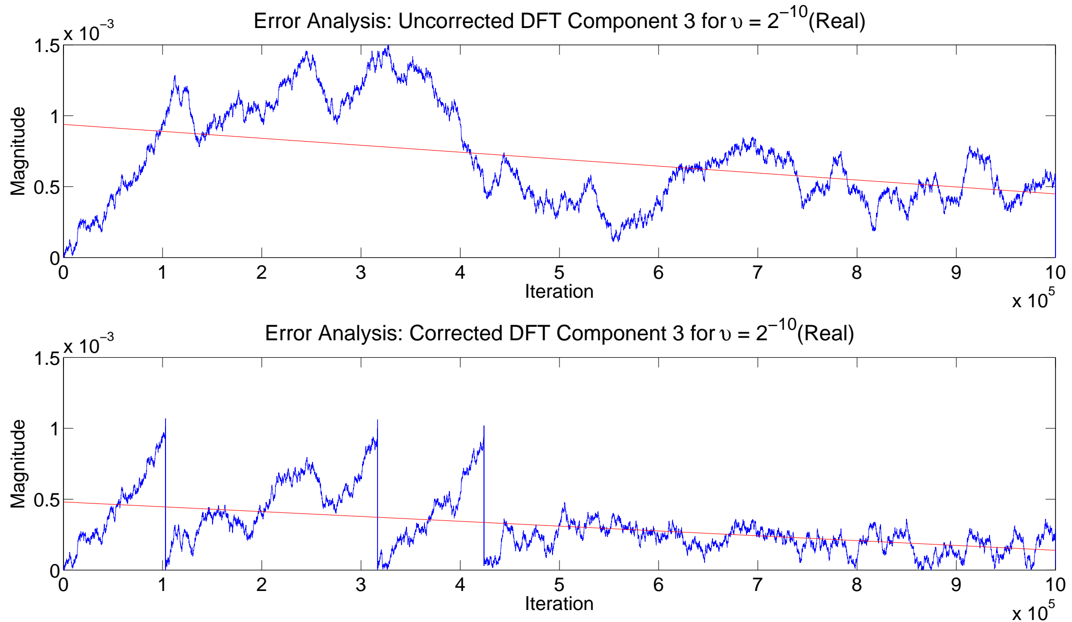



Figure 3.17: Error analysis for DFT component 3 using $\upsilon = 2^{-10}$. The error at any iteration is a function of input data and will therefore fluctuate. However, the recursion present will cause an accumulation of error as shown as a linear fit (top red trace). In this case a large enough error tolerance is allowed to permit sufficient error accumulation and therefore correction without discarding crucial information through the rounding process. The gradient of red trace (corrected component) is $-3.41 \times 10^{-8}$.

University of Ca

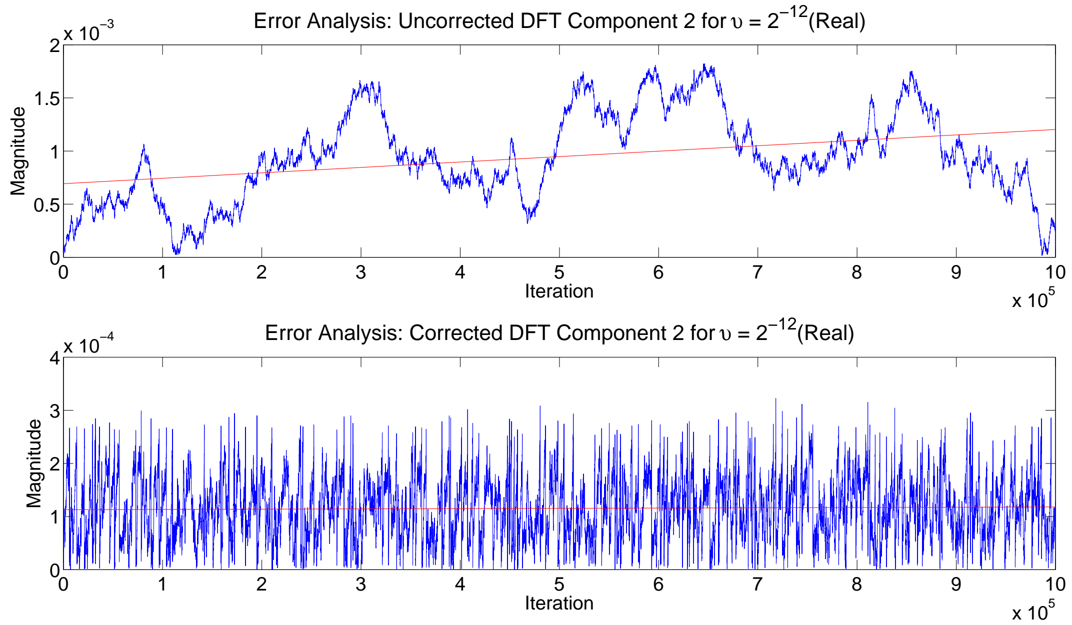



Figure 3.18: Error analysis for DFT component 2 using $\upsilon = 2^{-12}$. A similar situation exists for $\upsilon = 2^{-12}$ compared to $\upsilon = 2^{-10}$. A point to note is the error is an order of magnitude lower than found with $\upsilon = 2^{-10}$. Using $\upsilon = 2^{-12}$ is borderline for error convergence, and begins to show signs of a gradual drift. The gradient of red trace (corrected component) is $5.18 \times 10^{-10}$.

University of Ca

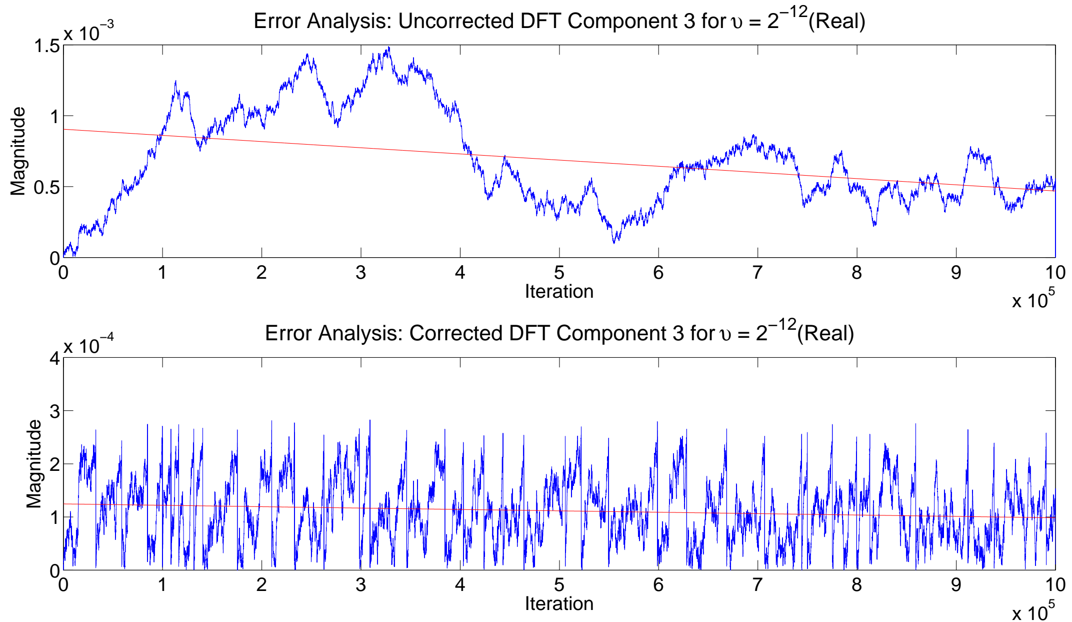



Figure 3.19: Error analysis for DFT component 3 using $\upsilon = 2^{-12}$. A similar situation exists for $\upsilon = 2^{-12}$ compared to $\upsilon = 2^{-10}$. A point to note is the error is an order of magnitude lower than found with $\upsilon = 2^{-10}$. Using $\upsilon = 2^{-12}$ is borderline for error convergence, and begins to show signs of a gradual drift. The decline in the linear fit for the uncorrect output (red trace) is due to input data characteristics and the nature of the arithmetic and coefficient quantization for this DFT component. If a longer observation window were used the error would linearly increase. The gradient of red trace (corrected component) is $-2.61\times10^{-9}$.

University of Ca

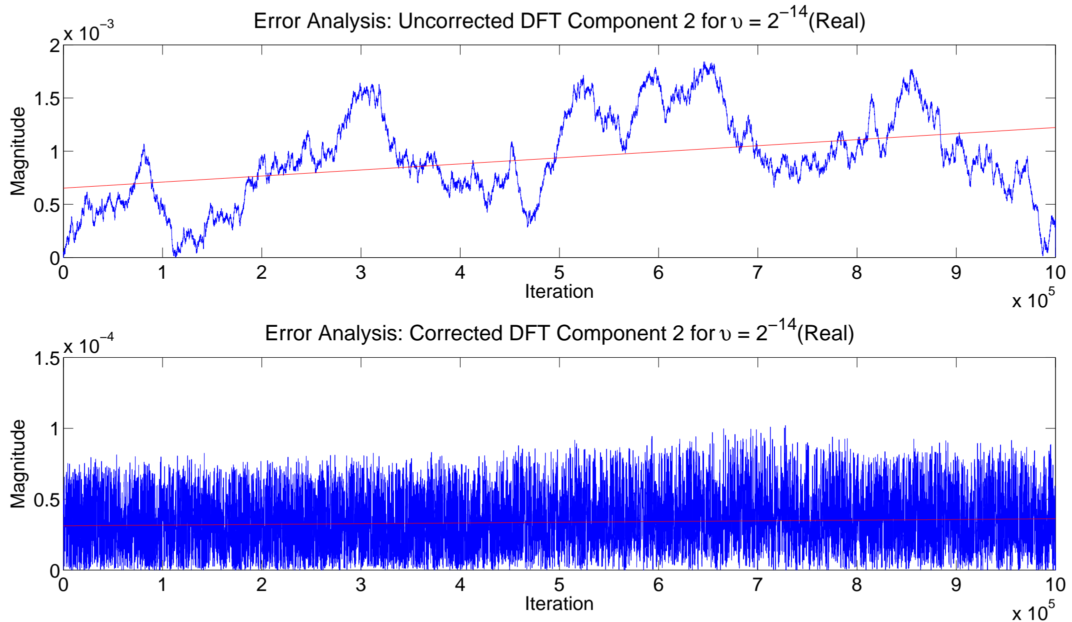



Figure 3.20: Error analysis for DFT component 2 using $\upsilon = 2^{-14}$. Using $\upsilon = 2^{-14}$ does not adhere to the 14-bit difference guideline between fixed-point precision and threshold value. The result is a corrected output, however the error continues to drift gradually through inherent losses which occur in the rounding in the error correction process. A larger threshold will reduce this negative impact. The gradient of red trace (corrected component) is $4.97 \times 10^{-10}$.

University of Ca

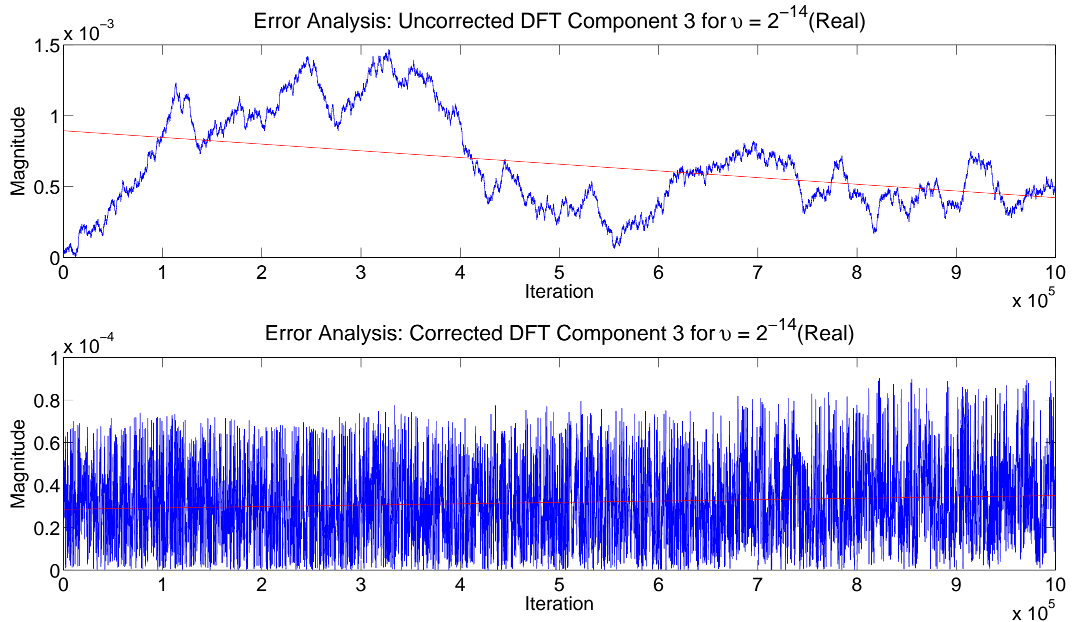



Figure 3.21: Error analysis for DFT component 3 using $\upsilon = 2^{-14}$. Using $\upsilon = 2^{-14}$ does not adhere to the 14-bit difference guideline between fixed-point precision and threshold value. The result is a corrected output, however the error continues to drift gradually through inherent losses which occur in the rounding in the error correction process. A larger threshold will reduce this negative impact. The gradient of red trace (corrected component) is $6.46 \times 10^{-10}$.

University of Cape Town

Figures 3.16 to 3.21 tell an interesting story when analysed. The first set of error results for threshold ($\upsilon = 2^{-10}$) is shown in Figures 3.16 and 3.17 for DFT components 2 and 3 respectively. Viewing DFT component 2, if left uncorrected, the error would have accumulated consistently with brief periods of decrease due to dependence on the input data. By comparison, the uncorrected component of DFT 3 initially starts to grow rapidly but later decreases for a short period. The decrease from the $5 \times 10^5$ iteration contributes to the linear decrease in the least-square linear fit (red trace). Over time the error would start to increase again.

From these two trace sets it is clear that not all errors in the DFT components operate in unison, and in fact, some (coefficient quantization dependent) contribute very little overall error to the system. The corrected trace in Figure 3.16 and 3.17 show a different story for the same input dataset. These two sets of traces represent the corrected error status for the same two DFT components. In the case of DFT 2, the linear fit shows a fractional decrease over time. The threshold of $\upsilon = 2^{-10}$ causes the error correction to be invoked any time the error accumulates to a value over $2^{-10} = 9.77 \times 10^{-4}$. This explains the rapid decrease in the error (blue trace) which is represented by 'spikes'. The error resumes accumulation post-correction, and is corrected again when exceeding the threshold. Performing a linear least-squares fit to the corrected error trace produces the trace shown in red. A similar situation exists for DFT 3. The error in this case requires fewer corrections, but when a linear least-squares fit is applied it reveals a negative gradient. This can be interpreted as when observed over time, the error accumulation does not consistently increase.

The results in Figures 3.18 and 3.19 carry a similar explanation to that for the results in Figures 3.16 and 3.17. The most distinguishable difference is the results shown for the corrected error for DFT components 2 and 3 where the threshold is now $2^{-12} = 2.44 \times 10^{-4}$. This will naturally mean that the error correction will be invoked sooner and more frequently as the tolerance for accumulated error is far lower. The nature of error is irregular, but of an order of magnitude lower than that for a threshold of $\upsilon = 2^{-10}$. The question which arises is whether or not the error will naturally drift over time. This is answered in part now, and concluded in the section which follows on Allan Variance.

A least squares linear fit in both DFT component cases (red trace) shows a gradient (DFT component 2) of $5.18 \times 10^{-10}$ and of (DFT component 3) $-2.61 \times 10^{-9}$. A negative gradient indicates a decreasing error over time, while a positive gradient indicates an error increase. These results show that the guideline criteria of a 14-bit difference between the threshold and maximum precision is borderline. By contrast, the two same gradients for a threshold of $\upsilon = 2^{-10}$ is $-4.77 \times 10^{-9}$ (DFT component 2), and $-3.41 \times 10^{-8}$ (DFT component 3).

University of Cape Town

The results in Figures 3.20 and 3.21 provide a clear representation of what happens when the 14-bit guideline is exceeded. A threshold of $\upsilon = 2^{-14}$ is used in this case, and as can be seen, the error is initially an order of magnitude lower than the equivalent traces for a threshold of $\upsilon = 2^{-10}$. The difference however is the distinguishable drift that becomes inherent as time passes. While observable, the least-squares linear fit confirms this with a gradient of $4.97 \times 10^{-10}$ (DFT component 2) and $6.46 \times 10^{-10}$ (DFT component 3). Any further decrease in the threshold will result in poorer performance for the same fixed-point precision.

The results discussed have concentrated exclusively on two DFT components, but in reality the same interpretation applies to the remaining components in the set. The only significant difference is the value of coefficient quantization which in turns influences the representation of the error.

University of Cape Town

**Allan Variance** Taking the analysis one step further, it would be beneficial to analyse the role of the error correction and the stability of the system when it is employed. To achieve this, it is useful to turn to a measurement metric well established in the field of oscillator design. Characterizing the random variations of a clock source is required for the optimal estimation of both environmental influences, as well as to the design of combining algorithms for the generation of uniform time, and ensuring a stable frequency reference [85].

In oscillator design, a dimensionless quantity termed the fractional frequency provides the measure of the frequency deviation $\nu(t)$ from its nominal value $\nu_0$. Defined accordingly:

$$y(t) = \frac{\nu(t) - \nu_0}{\nu_0} \tag{3.43}$$

Integrating $y(t)$ gives the time deviation $x(t)$ such that:

$$x(t) = \int_0^t y(t')dt' \tag{3.44}$$

Clocks in practice deviate from the ideal for two primary reasons: systematic reasons such as frequency drift and frequency offset (other deviations can also be due to environmental reasons); and the second primary reason is random deviations $\epsilon(t)$ which are not typically deterministic.

Using this concept in signal processing, the deviation of the recursive DFT outputs with respect to an ideal set of outputs can be considered in place of an oscillator deviation. Each DFT component is a rotating phasor with magnitude and phase, and the error in either the real or imaginary data (typically both) is considered a deviation from the ideal. In this case, the ideal is a fixed point FFT of the same precision as the recursive DFT computation, as this will indicate the inclusion of error and performance of the error correction algorithm.

In oscillator analysis, the time deviations (error) are used to compute the fractional frequency. Similarly, the error produced with respect to time as the difference between the recursive DFT and FFT can be used to compute a fractional frequency. If each time interval between samples is $\tau_0$, and a sequence of error samples are produced, the average fractional frequency is [85, 86]:

$$y_i^{\tau_0} = \frac{x_{i+1} - x_i}{\tau_0} \tag{3.45}$$

Using Equation 3.45, a set of discrete frequency values can be computed from the time difference dataset. Standard deviation is known to be divergent and a function of data length, and an IEEE recommended method known as Allan variance, or two-sample variance is often applied [85].

University of Cape Town

The Allan variance (shown in Equation 3.46) is computed as the difference between adjacent fractional frequency measurements each sampled over an interval $\tau$ and is given by:

$$\sigma_y^2(\tau) = \frac{1}{2(M-2n+1)} \sum_{k=1}^{M-2n+1} (y_{k+n}^{\tau} - y_k^{\tau})^2 \tag{3.46}$$

where:

$M$ is the finite number of data samples

$n$ is the number of adjacent values used if averaging is required. If averaging multiple samples, then $\tau = n\tau_0$, where $\tau_0$ is the minimal data spacing in the dataset.

Alternatively, the data from the initial error sequence can be used, and the Allan variance can be computed as:

$$\sigma_y^2(\tau) = \frac{1}{2\tau^2(M-2n+1)} \sum_{k=1}^{M-2n+1} (x_{i+2n} - 2x_{i+n} + x_i)^2 \tag{3.47}$$

There is a relationship which exists between the Allan variance $\sigma_y^2(\tau)$ and power-law spectra. Random frequency deviations in oscillators can be characterised by power-law spectra, where the power-law spectra is $S_y(f) \sim f^{\alpha}$. In this case, $f$ is the Fourier frequency and would take on integer values (-2,-1,0,1,2). As an example, $f^0$ would be the expected result for a white noise process. Various combinations of noise may be present in a signal, and may take place at different time slots. Determining the Allan variance over a wide range of $\tau$, and plotting accordingly, can provide a useful indication of the noise process present, as well as the level of uncertainty in the actual outputs.

To relate the results of Allan variance and power-law spectra, a proportionality applies: $\sigma_y^2(\tau) \sim \tau^{\mu}$, where $\mu$ is usually constant for a value of $\alpha$. The relationship between the spectral density exponent ($\alpha$), and $\mu$ can be shown for two ranges of $\alpha$. If $-3 < \alpha \leq 1$, then $\mu = -\alpha - 1$, and $\mu = -2$ for $\alpha \geq 1$.

University of Cape Town

Table 3.1: Noise Process and $\alpha$ values

| $\alpha$ | *Noise Process* |
|---|---|
| 2 | White-noise phase modulation |
| 1 | Flicker-noise phase modulation |
| 0 | White-noise frequency modulation |
| -1 | Flicker-noise frequency modulation |
| -2 | Random-walk frequency modulation |

There is an ambiguity which exists for $\mu = -2$ (it is not clear if the noise process is flicker-noise phase modulation, or white-noise phase modulation), however can be resolved through variable measurement bandwidths [85].

The following table outlines the noise process types for various $\alpha$ values.

Computing the Allan variance for the three threshold conditions shown in Figures 3.16 to 3.21 produces the following results.

University of Cape Town

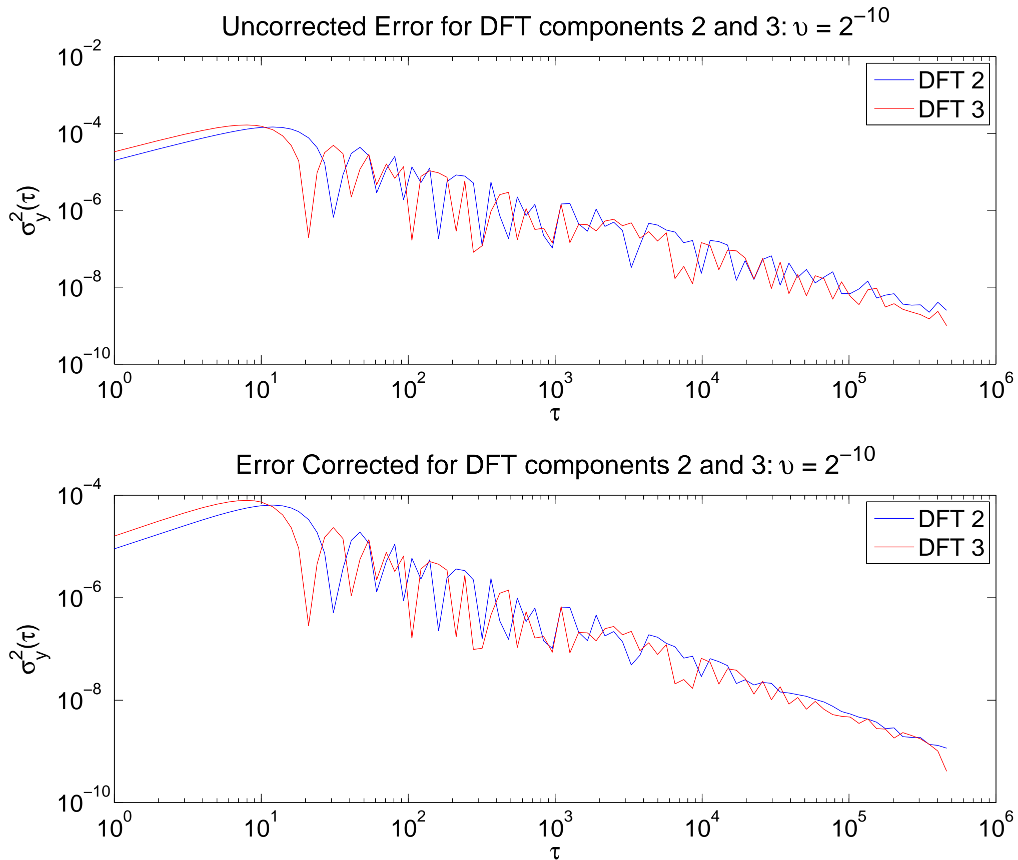



Figure 3.22: Allan variance for DFT components 2 and 3 using $\upsilon = 2^{-10}$. The negative average gradient with reducing ripple effect indicates a white noise process is present (and improving due to the averaging performed through Allan variance computation). Once error correction is enabled, the output uncertainty is reduced, but still fluctuates due to higher error threshold.

Univer

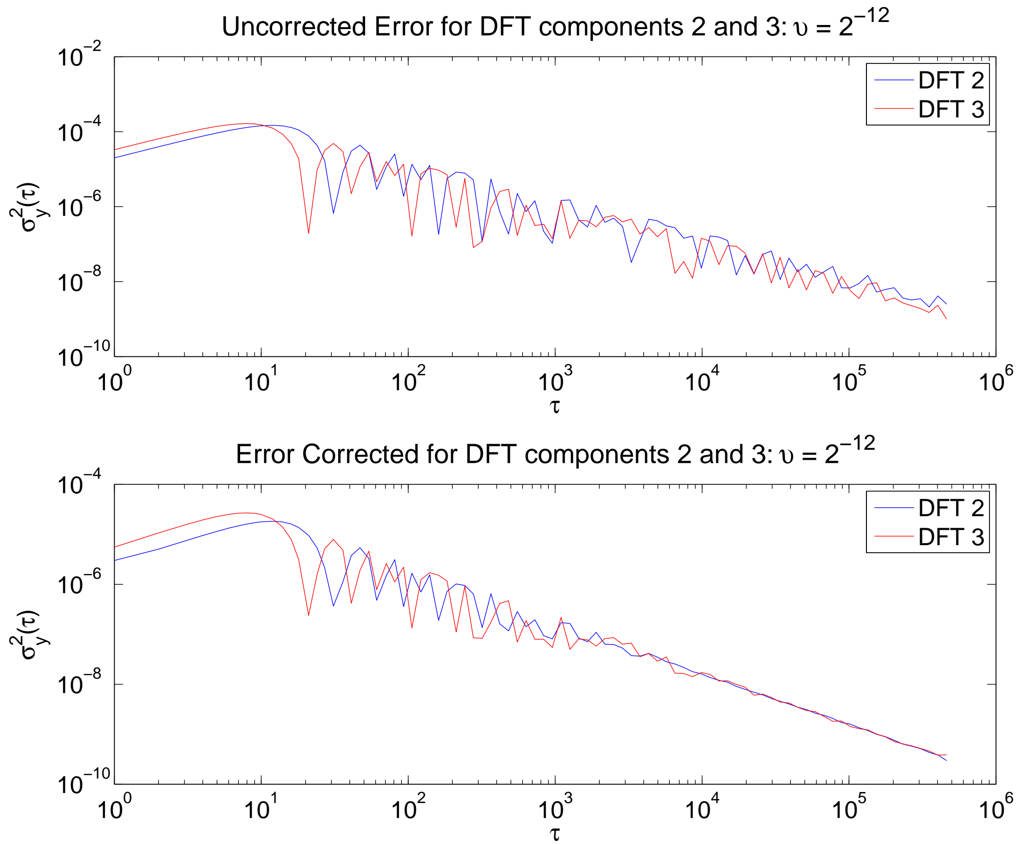



Figure 3.23: Allan variance for DFT components 2 and 3 using $\upsilon = 2^{-12}$. By comparison to $\upsilon = 2^{-10}$, when correction is enabled, the accumulated error is quickly reduced and maintained. This is evident from $\tau \approx 10^4$.

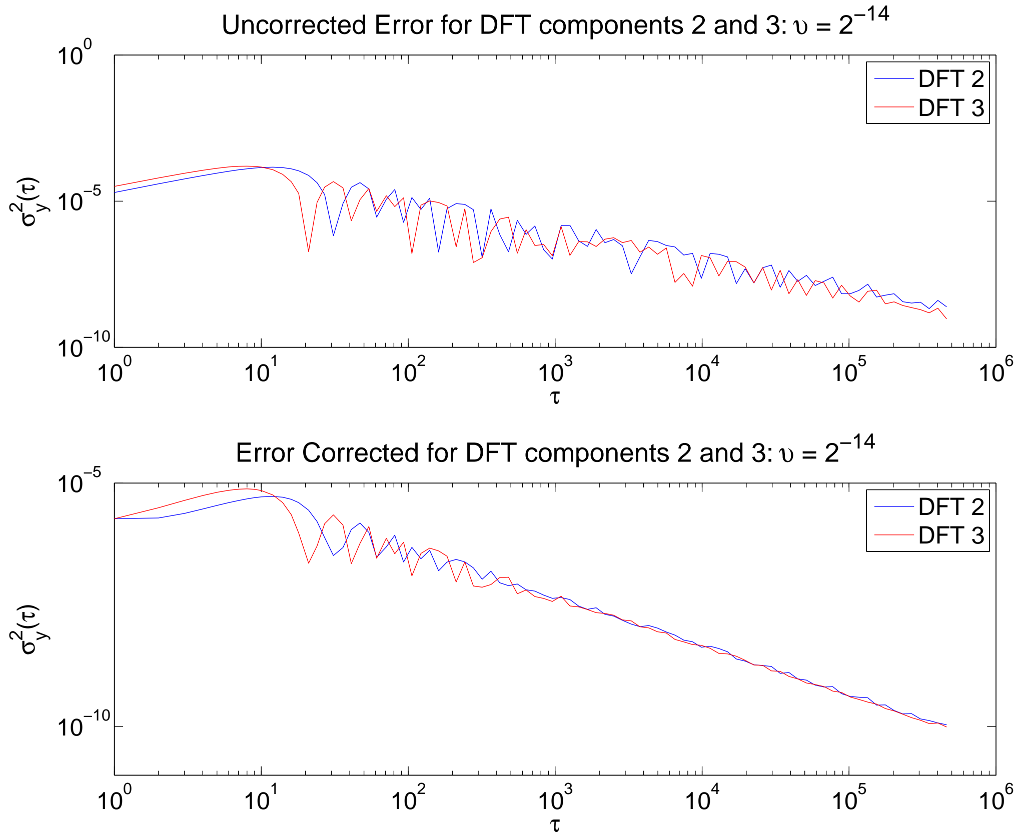



Figure 3.24: Allan variance for DFT components 2 and 3 using $\upsilon = 2^{-14}$. The lower threshold will invoke the error correction earlier in the computation of the sequence, however the loss of information due to rounding in the error correction results in error drift. This is evident in the ripple which remains throughout all values of $\tau$.

Figures 3.22 to 3.24 illustrate the Allan variance computed for the three threshold values of $\upsilon = 2^{-10}, 2^{-12}, 2^{-14}$. Starting with Figure 3.22 ($\upsilon = 2^{-10}$), a number of interesting points stand out. Firstly, in both the corrected and uncorrected cases, the variance starts off with a significant ripple effect. The ripple is an indication of acceleration in frequency, and begins to smooth off as time passes. Secondly, the average slope of the variance is negative, and the ripple reduces as time passes. This indicates that the noise process is white, as averaging

is used in the computation of the Allan variance, and an averaged white sequence will reduce to a mean of zero.

The difference lies in the rate at which the ripple is reduced when comparing the two sets of traces. Using a threshold of $\upsilon = 2^{-10}$ will allow a higher accumulated error to occur before the error correction will be enabled. This can be seen from the point where $\tau \approx 10^4$. From this point, the ripple continues to be reduced, however will still fluctuate due to a higher threshold value.

Taking a closer look at Figure 3.23, a similar picture emerges. The ripple is initially evident, however from the point where $\tau \approx 10^3$, the error correction is invoked. The interesting point here, is that the traces for both DFT components rapidly become smooth. The smoothing of the traces is a good indication that the uncertainty in the output signal is reduced.

Figure 3.24 shows similar results, except has a threshold of $\upsilon = 2^{-14}$. The lower threshold invokes the error correction more frequently, however the 14-bit difference between the threshold and the precision is no longer in place. The drift becomes evident in the traces shown, as the ripple remains even after correction.

An additional point to consider is the gradient of each of the three sets of corrected datasets. The gradient is approximately 1 for all three sets (in the linear region), and when reference to Table 3.1, indicates a noise process present of flicker-noise phase modulation, which lies between the the two white processes of phase and frequency modulation. This by comparison is very different to the uncorrected state at the same points, which indicates that additional noise processes are present. The inclusion of the error correction can be seen as the application of a filter which removes these additional noise processes, and results in an output with only reduced flicker-noise present.

University of Cape Town

### 3.7.3 Analytical Error Model

It has been shown that it is possible to compute and track the error in the final output $F_u$ (forward transform case), without knowing a priori the true value of the output (as is the case in any fixed-point arithmetic system susceptible to finite-bit approximation errors). It was shown in (3.41) that the error $E_u$ for the forward transform can be expressed as:

$$E_u[l] = \sigma_u \left[ \hat{F}_u[l-1] + \frac{f_{in} - f_{out}}{N} \right] + W_N^u E_u[l-1] \tag{3.48}$$

University of Cape Town

Similarly, the error $E_n$ for the reverse transform can be expressed as:

$$E_n[l] = \sigma_n \left[ \hat{f}_n[l-1] + F_{in} - F_{out} \right] + W_N^n E_n[l-1] \tag{3.49}$$

where:

$$\sigma_u = \hat{W}_N^u - W_N^u \tag{3.50}$$

Comparing (3.48) and (3.49), the division by $N$ is the only factor which differs between the two expressions. The term $E_u[l-1]$ is the error from the previous iteration $l-1$, and the term $\sigma_u$ is the coefficient quantization error due to the finite register length used in hardware. For the general case, this error can be approximated by a white noise sequence with a uniform distribution. The magnitude of the error is inversely related to the number of quantization steps, that is, the greater the number of quantization levels, the smaller the resulting error. The error would therefore approach zero as the number of quantization levels increases.

Since the value of the complex exponential being quantized is dependent on $u$, the value of quantization error can vary for different coefficients for a fixed number of quantization steps (i.e. for a fixed-bit length system). It is intuitive to model the coefficient quantization as a white noise process with a uniform distribution of zero mean and standard deviation (anywhere between zero and $\pm\frac{2^{-B}}{2}$), as the error can be any value within this range, and approach or be equal to zero for a $u$ value that will in turn provide an integer result for $u^{th}$ complex exponential (e.g.using $u = 0$, $W_N^u = 1$ (or whenever $u$ is an integer multiple of $N$)).

Assessing arithmetic error (denoted by $\epsilon_A$) for multiplication, a similar approach is required, where the noise model used for arithmetic error is based on a uniform white process. For a small quantization step size ($\Delta$), the magnitude of the error can fall anywhere in the range of $-\frac{\Delta}{2}$ to $\frac{\Delta}{2}$, and it is assumed that successive noise samples are uncorrelated, and that $\epsilon_A$ is uncorrelated with the input sequence [59].

The arithmetic and coefficient quantization errors result from a rounding process, however, the arithmetic error for division should be computed using different limits. The error associated with the division by $N$ is due to truncation rather than a rounding process. The division by $N$ will be discussed in detail in Chapter 4, however it should be noted that for this study, $N$ is assumed to be a power-of-2, and therefore can be implemented using simple bit shifters rather than hardware expensive dividers. As a result of the shifting process, the least significant bit/s are truncated to maintain the desired word length (to help minimise this error, the hardware implementation performed unbiased rounding after shifting). The truncation error can also be approximated using a uniformly distributed random process, however the probability density is from $-\Delta$ to 0 [59].

For the general error model case, the expression for $E_u[l]$ does not represent the approximations for arithmetic rounding or truncation. Working on the forward transform, Equation (3.48) can be modified to express these sources. The math which follows is best expressed after a small simplification.



University of Cape Town

Let:

$$S_u[l] = \hat{F}_u[l-1] + \frac{f_{in} - f_{out}}{N} \quad \text{(Forward Transform)} \tag{3.51}$$

and:

$$S_n[l] = \hat{f}_n[l-1] + F_{in} - F_{out} \quad \text{(Reverse Transform)} \tag{3.52}$$

The arithmetic rounding error term can now be introduced (the coefficient quantization term was inherently present and represented by $\sigma_u$, and for the assessment which follows, it is assumed to be a white noise process of uniform distribution). These terms are jitter terms and are added to the true value as represented in the illustration in Figure 3.3 (the coefficient quantization term is shown as the $\sigma_u$ term). The arithmetic rounding error term is represented by $\epsilon_A$ (same as in Figure 3.3), and is a sample from a white noise process with a uniform distribution. The arithmetic truncation error is denoted by $\epsilon_T$, and should be included in the $S_u[l]$ term for the forward transform expressed in 3.51.

University of Cape Town

Amending the expression of $S_u[l]$ for the forward transform case, (3.51) becomes:

$$S_u[l] = \hat{F}_u[l-1] + \frac{f_{in} - f_{out}}{N} + \epsilon_T \quad \text{(Forward Transform)} \tag{3.53}$$

Using these error terms, the expressions for (3.48) and (3.49) for $l > 1$ can be expressed as:

$$E_u[l] = \sigma_{u(l:1)} S_u[l] + \epsilon_{A_\dagger(l:1)} + \left[W_N^u - \sigma_{u(l:1)}\right] E_u[l-1] + \sum_{i=1}^{2^l - l} \epsilon_{A_\ddagger(l:i)} \tag{3.54}$$

where:

$\sigma_{u(l:1)}$ is the first instance of coefficient quantization for iteration $l$

$\epsilon_{A_\dagger(l:1)}$ is the first instance of arithmetic error for iteration $l$

$\sum_{i=1}^{2^l - l} \epsilon_{A_\ddagger(l:i)}$ represents additonal arithmetic instances 1 through to $(2^l - l)$ for iteration $l$

The expression in (3.54) holds for any iteration $l$ provided $E_u[l-1]$ is known ($l$ must be greater than 1 for $E_u[l-1]$ to exist). The complex term $\sum_{i=1}^{2^l-l} \epsilon_{A_\ddagger(l:i)}$ represents the number of arithmetic rounding errors to be included, and is derived shortly.

It is important to clarify at this stage the difference between $\epsilon_{A_\dagger}$ and $\epsilon_{A_\ddagger}$. It will detailed shortly that the bit ranges differ depending on the source of error. The term $\epsilon_{A_\dagger}$ corresponds to the arithmetic errors associated to the multiplication of $\sigma_u$ are with respect to 24-bit precision, and the term $\epsilon_{A_\ddagger}$ corresponds to the arithmetic errors associated to the feedback error term $E_u$ and are associated to 36-bit precision. It is necessary to make this distinction at an early stage, as it will later be found that the error term $\epsilon_{A_\dagger}$ combined with the coefficient error term $\sigma_u$ are the dominant error sources.

University of Cape Town

In the general case, $E_u[l-1]$ is not necessarily known, so it is useful to describe the accumulative system error independent of $E_u[l-1]$, for any iteration $l$. This can be achieved by induction, by assessing the system from the beginning for $l = 1$. The subscript modification $(l:1)$ for the arithmetic error $(\epsilon_{A_\dagger(l:1)})$ indicates the iteration instance $l$ and the index for the occurrence in that iteration. It will be shown shortly that the number of included error terms grows as $l$ increases. This increase in terms is due to the number of product terms growing as each arithmetic error term represents the error per product. This can be shown as follows, letting the process start at iteration $l = 1$:

Let $l = 1$:

$$E_u[1] = \sigma_{u(1:1)} S_u[1] + \epsilon_{A_\dagger(1:1)} + \left[W_N^u - \sigma_{u(1:1)}\right][0] \tag{3.55}$$

Assessing (3.55), $E_u[l-1]$ is replaced with a zero value as there is no prior error value present in the system when $l = 1$. Since there is one product term $(\sigma_u S_u[1])$, a single instance of arithmetic error $(\epsilon_{A_\dagger(1:1)})$ needs to be included for this iteration. For the next iteration, $E_u[l-1]$ will be replaced with the expression represented in (3.55).

At this point it is worth discussing the difference between the model for the forward transform and reverse transform implementations. It was mentioned previously that the truncation error, $\epsilon_T$, should be included when computing the error model for the forward transform. This error term was included in (3.53), and can be substituted into (3.55) for $S_u[1]$. Performing this substitution, (3.55) becomes:

University of Cape Town

$$E_u[1] = \sigma_{u(1:1)} \left[ \hat{F}_u[l-1] + \frac{f_{in} - f_{out}}{N} + \epsilon_T \right] + \epsilon_{A_\dagger(1:1)} + \left[ W_N^u - \sigma_{u(1:1)} \right][0]$$
$$= \sigma_{u(1:1)} \left[ \hat{F}_u[l-1] + \frac{f_{in} - f_{out}}{N} \right] + \sigma_{u(1:1)}\epsilon_T + \epsilon_{A_\dagger(1:1)} + \left[ W_N^u - \sigma_{u(1:1)} \right][0] \quad (3.56)$$

The expression in (3.56) now carries the product of two error sources, namely $\sigma_{u(1:1)}$ and $\epsilon_T$. This expression can be further simplified if it is assumed that the product of any two error terms is approximately zero. This assumption can be justified as the product of any two negative powers results in a smaller number. In this case, the variance for any uniformly distributed error source is [59]:

$$\sigma_e^2 = \frac{2^{-2B}}{12} \quad (3.57)$$

The product of two of these terms is $\approx 2^{-4B}$, and for any large $B$ value (24-bits in this case), the result can be negated without any significant affect on the overall outcome of the calculation. Applying this premise, (3.56) becomes:

$$E_u[1] = \sigma_{u(1:1)} \left[ \hat{F}_u[l-1] + \frac{f_{in} - f_{out}}{N} \right] + \epsilon_{A_\dagger(1:1)} + \left[ W_N^u - \sigma_{u(1:1)} \right][0]$$
$$= \sigma_{u(1:1)} S_u[1] + \epsilon_{A_\dagger(1:1)} + \left[ W_N^u - \sigma_{u(1:1)} \right][0] \quad (3.58)$$

The result shown in (3.58) shows that in a gross sense, the computation of the total error can negate the error contribution from the truncation error sources, $\epsilon_T$, and still obtain a close representation of the overall error which can be expected. The expression in (3.58) also highlights that in fact the method of error computation remains common for both the forward and reverse transforms. It

University of Cape Town

will be seen later that error performance simulations do support this notion, and it can be seen that in fact the error performance for both the forward and reverse transforms are highly correlative.

Returning to the computation of the total error $E_u$, the second iteration can be considered:

Let $l = 2$:

$$\begin{aligned} E_u[2] &= \sigma_{u(2:1)} S_u[2] + \epsilon_{A_\dagger(2:1)} + \left[W_N^u - \sigma_{u(2:1)}\right] \left[\sigma_{u(1:1)} S_u[1] + \epsilon_{A_\dagger(1:1)}\right] \\ &= \sigma_{u(2:1)} S_u[2] + \epsilon_{A_\dagger(2:1)} \\ &+ \left[W_N^u \sigma_{u(1:1)} S_u[1] + W_N^u \epsilon_{A_\dagger(1:1)} - \sigma_{u(1:1)} \sigma_{u(2:1)} S_u[1] - \sigma_{u(2:1)} \epsilon_{A_\dagger(1:1)}\right] \\ &+ \epsilon_{A_\ddagger(2:1)} + \epsilon_{A_\ddagger(2:2)} \end{aligned} \tag{3.59}$$

The expression given for $l = 2$ in (3.59) now carries the error terms from the previous iteration. Applying the above assumption that the product of two error terms are approximately zero, the expression in (3.59) can be reduced to:

$$E_u[2] = \sigma_{u(2:1)} S_u[2] + \epsilon_{A_\dagger(2:1)} + \left[W_N^u \sigma_{u(1:1)} S_u[1] + W_N^u \epsilon_{A_\dagger(1:1)}\right] + \sum_{i=1}^{2} \epsilon_{A_\ddagger(2:i)} \tag{3.60}$$

It should be noted that the term $\sum_{i=1}^{2} \epsilon_{A_\ddagger(2:i)}$ represents the two arithmetic error terms added due to the product terms (in brackets) in the expression of (3.60). The product in brackets is computed in 36-bit arithmetic as it originates from $E_u[l-1]$ (which is stored using higher precision). Going through one more iteration, let $l = 3$:

University of Cape Town

Let $l = 3$:

$$E_u[3] = \sigma_{u(3:1)} S_u[3] + \epsilon_{A_\dagger(3:1)} \\ + \left[W_N^u - \sigma_{u(3:1)}\right] \left[\sigma_{u(2:1)} S_u[2] + \epsilon_{A_\dagger(2:1)} + W_N^u \sigma_{u(1:1)} S_u[1] + W_N^u \epsilon_{A_\dagger(1:1)} + \sum_{i=1}^{2} \epsilon_{A_\ddagger(2:i)}\right] \tag{3.61}$$

Multiplying out, and applying the same error product reduction technique as before, (3.61) can be reduced to:

$$E_u[3] = \sigma_{u(3:1)} S_u[3] + \epsilon_{A_\dagger(3:1)} \\ + W_N^u \sigma_{u(2:1)} S_u[2] + W_N^u \epsilon_{A_\dagger(2:1)} + W_N^{2u} \sigma_{u(1:1)} S_u[1] + W_N^{2u} \epsilon_{A_\dagger(1:1)} + W_N^u \sum_{i=1}^{2} \epsilon_{A_\ddagger(2:i)} \\ + \sum_{i=1}^{5} \epsilon_{A_\ddagger(3:i)} \tag{3.62}$$

Using multiple iterations, a generic expression can be formulated for any iteration value $l > 1$:

$$E_u[l] = \sigma_{u(l:1)} S_u[l] + \epsilon_{A_\dagger(l:1)} + \sum_{p=1}^{l-1} W_N^{pu} \sigma_{u((l-p):1)} S_u[l-p] \\ + \sum_{p=1}^{l-1} W_N^{pu} \epsilon_{A_\dagger((l-p):1)} + \sum_{i=1}^{l-1} W_N^u \epsilon_{A_\ddagger((l-1):i)} + \sum_{i=1}^{2^l - l} \epsilon_{A_\ddagger(l:i)} \tag{3.63}$$

The expression in (3.63) indicates that the total error for any iteration $l$ consists of multiple summations of arithmetic errors associated with product pairs from previous errors, as well as being a function of coefficient quantization for the current $u$ term combined with the sum of the incoming and outgoing samples



University of Cape Town

with the previous DFT point value $F_u[l-1]$ (recall this is represented by $S_u[l]$).

It is fair to say that recursive effect of the algorithm contributes a large number of past error components to form any output at a given iteration. It can be seen that the growth rate for arithmetic error associated to 36-bit precision is of the order of $2^l$, however the dominant error contributors arise from the 24-bit arithmetic error.

The error sources used in this model, and expressed as $\sigma_u$ and $\epsilon_A$ in (3.63), are random by nature, but do have constraints which depend on the fixed-point length, as well as the total word length of the register to be used to store the remaining computed error that cannot be used to correct the output for a given iteration (it cannot be realised using fixed-point length used in the output). This remainder is used in the next iteration to seed the algorithm and is shown as $E_u[l-1]$ in (3.41).

To compute the worst case unbounded error using this error model, the fixed-point length is first required, as this will determine the magnitude of the error represented by $\sigma_u$, as $\sigma_u$ is the difference between a true representation of the complex exponential and a finite-bit approximation. To model this as an error source requires knowledge of the finite-bit length used, which for this system, was 24-bits. It is therefore necessary to constrain the standard deviation (uniform distribution) to within the typical range of values for $\sigma_u$. This is done by limiting the deviation between $\frac{-2^{-B}}{2}$ to $\frac{+2^{-B}}{2}$, where $B$ is the fixed-point length, and the mean is zero.

A similar constraint exists for the arithmetic error which has the range of $\frac{-2^{-35}}{2}$ to $\frac{+2^{-35}}{2}$(register length is 36-bits, however 1-bit is held for the sign). The difference in ranges is due to the storage capabilities when computing the values for the error sources. The arithmetic error is present due to the finite-bit arithmetic of the products in the computation of $E_u$. This system allowed the operands to be either 24-bits or 36-bits in length (depending on the product under evaluation), however the final $E_u$ computation is stored in a word length of

University of Cape Town

36-bits, resulting in an arithmetic error in the order of $2^{-36}$. The error is therefore uniformly distributed in the range of $\frac{-2^{-35}}{2}$ to $\frac{+2^{-35}}{2}$ to model a range of possible errors for the generic case.

#### 3.7.3.1 Model Simulation

Using the error model expressed in Equation (3.54), it is possible to compute and illustrate the expected error of the system for a particular fixed-point length and threshold. Figures 3.25, 3.26, 3.27, 3.28, compare the results from the error model (red trace), with the mean of the true uncorrected error (actual error between true DFT and uncorrected RDFT)(green trace) and the mean of the error when using error correction in the RDFT (blue trace).

The error traces are calculated using an uncorrelated complex dataset with a normal distribution. The red trace represents the approximated error and was computed using Equation 3.54 where the noise sources were replaced by the statistical approximations described earlier in the chapter ($\sigma_u; \epsilon_A; \epsilon_C$). The green trace is computed as the difference between a DFT (FFT used) and the output of the RDFT (with error correction disabled - ideally, in a gross sense, the two traces should be very similar). The blue trace represents the the same computation performed for the green trace (actual error), except the error correction is enabled. Initially these two traces should be identical, except from the point when the error correction algorithm begins to adjust the output.

Figures 3.25, 3.26, 3.27, 3.28 illustrate these means for both the real and imaginary components, for both possible error models relating to the forward and reverse transform. The minor difference between the error models for the forward (Figures 3.25, 3.26) and reverse (Figures 3.27, 3.28) transform is the addition of the truncation noise component which is introduced due to the division by $N$. In a gross sense, these two results are comparable, and highly correlative. The input dataset ($5 \times 10^5$ samples) was complex valued with a normal distribution, where $\upsilon = 2^{-10}$ and the fixed-point length was 24-bits.

University of Cape Town

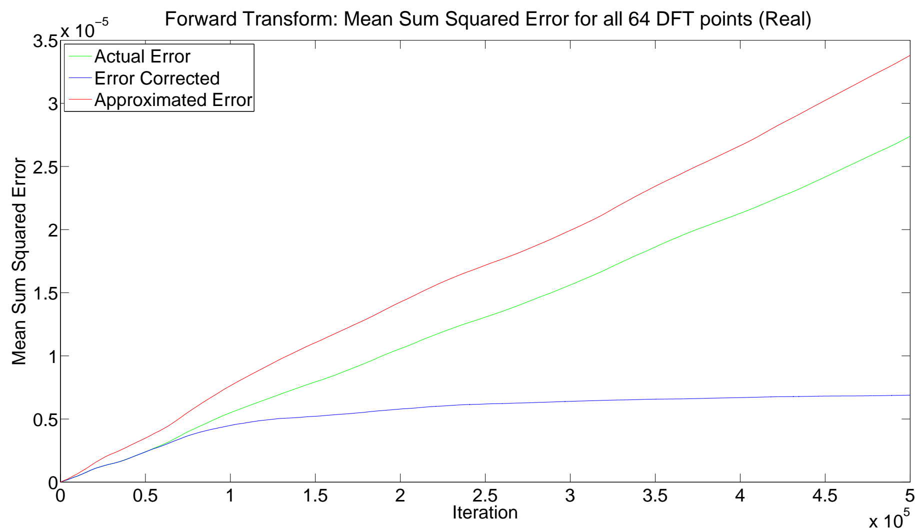



Figure 3.25: Forward Transform: Mean (Real) comparison of generic error model results. The traces shown represent the results from the error model (red trace), with the mean of the true un-constrained uncorrected error (green trace) and the mean of the error when using error correction (blue trace).

University of Cape

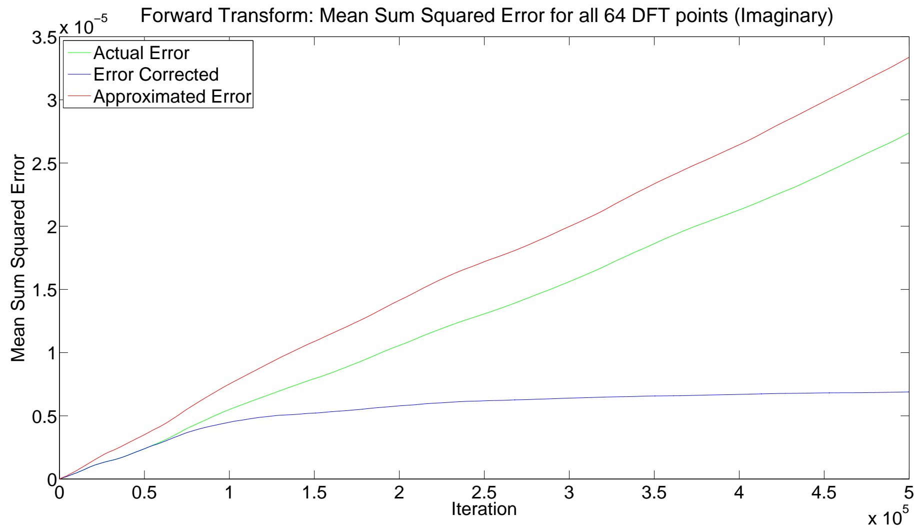



Figure 3.26: Forward Transform: Mean (Imaginary) comparison of generic error model results. The traces shown represent the results from the error model (red trace), with the mean of the true un-constrained uncorrected error (green trace) and the mean of the error when using error correction (blue trace).

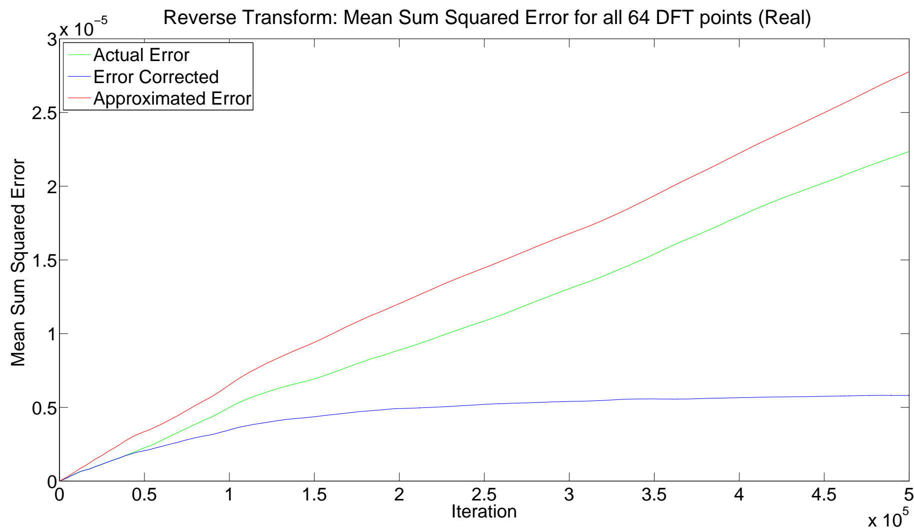



Figure 3.27: Reverse Transform: Mean (Real) comparison of generic error model results. The traces shown represent the results from the error model (red trace), with the mean of the true un-constrained uncorrected error (green trace) and the mean of the error when using error correction (blue trace).

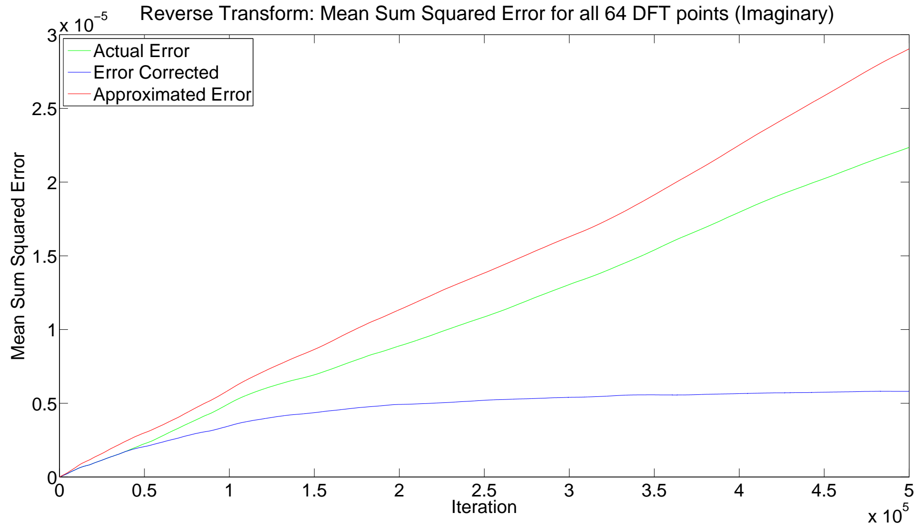



Figure 3.28: Reverse Transform: Mean (Imaginary) comparison of generic error model results. The traces shown represent the results from the error model (red trace), with the mean of the true un-constrained uncorrected error (green trace) and the mean of the error when using error correction (blue trace).

The red trace represents the computed mean error using the generic error model of the system, and the green trace represents the mean of the true error that exists for the given input dataset. The blue trace represents the error correction which occurs for the same input dataset, and therefore begins to converge as the error becomes bounded. The trace shown for the error model will not be identical to the true computed error as the sources of error in the generic model have been represented as noise sources, and therefore will differ from the values used in an actual implementation. The idea of the generic model is to provide an indication of the expected error in a gross sense, and therefore can be considered as a worst-case situation for a given system configuration. Having determined the noise source contributors, a meaningful metric to quantify the impact of the computed error should be determined. A common method to determine the significance of the noise is to compute the noise-to-signal ratio for the system. The section which follows computes this metric.

University of Cape

## 3.8 Noise-to-Signal Ratio

The noise-to-signal ratio is a means to express the impact of unwanted contributions in the output in the form of a power ratio. To determine an explicit expression for the noise-to-signal ratio present at the output of the system, it is necessary to consider the noise contributions in the output with respect to an uncorrupted input signal. It was shown in Section 3.7.3 that the noise model for the forward and reverse transform are comparable, and therefore will share the same noise-to-signal ratio.

The noise-to-signal ratio is expressed as the power ratio of the expected value for noise and the expected value for the input sequence. To compute this metric, consider a complex input sequence where both real and imaginary components of the input are uncorrelated identically distributed values (a white noise input signal). The noise component is the sum of all the noise contributions which are present in the output.

**Noise Component** The expression for the output noise was shown in Equation (3.63), and is repeated in Equation (3.64).

$$E_u[l] = \sigma_{u(l:1)} S_u[l] + \sum_{p=1}^{l-1} W_N^{pu} \sigma_{u((l-p):1)} S_u[l-p] + \sum_{p=1}^{l-1} \sum_{m=1}^{2^{(l-p)}-1} W_N^{pu} \epsilon_{A((l-p):m)} + \sum_{m=1}^{2^l-1} \epsilon_{A(l:m)} \tag{3.64}$$

Each of the error sources in Equation (3.64), namely $\epsilon_A$ and $\sigma_u$, are complex valued errors, and represent the respective arithmetic and coefficient quantization errors. Focusing at first on the arithmetic error source, $\epsilon_A$, it can be shown that $\epsilon_A$ consists of four round-off errors originating from the two real multiplications, and two imaginary multiplications when multiplying two complex terms [59].

University of Cape Town

The $\epsilon_A$ errors are uniformly distributed between $-\frac{1}{2}2^{-B}$ and $+\frac{1}{2}2^{-B}$, where $B$ represents the number of bits in the precision portion of the holding register. The four round-off terms mentioned previously are due to the four multiplications that occur from the product of two complex terms. This can be shown using the product of the two complex terms $a[n]$ and $b[n]$ and shown as follows:

$$a[n]b[n] = \Big[\Re a[n]\Re b[n]\Big] - \Big[\Im a[n]\Im b[n]\Big] + j\Big[\Re a[n]\Im b[n]\Big] + j\Big[\Re b[n]\Im a[n]\Big] \tag{3.65}$$

where:

$$\Re a[n] \;\text{ and }\; \Re b[n] \;\text{ represent the real part of the complex terms a[n] and b[n]} \tag{3.66}$$

$$\Im a[n] \;\text{ and }\; \Im b[n] \;\text{ represent the imaginary part of the complex terms a[n] and b[n]}$$

Arithmetic rounding of the expression in (3.65) due to quantization introduces errors which can be represented as the terms $\epsilon_1$,$\epsilon_2$,$\epsilon_3$ and $\epsilon_4$. The quantized form of the expression in (3.65) is:

$$\begin{aligned} Q\Big[a[n]b[n]\Big] &= \Big[\Re a[n]\Re b[n] + \epsilon_1\Big] - \Big[\Im a[n]\Im b[n] + \epsilon_2\Big] \\ &\quad + j\Big[\Re a[n]\Im b[n] + \epsilon_3\Big] + j\Big[\Re b[n]\Im a[n] + \epsilon_4\Big] \end{aligned} \tag{3.67}$$

The expressions in (3.67) follow the same product grouping as in Equation (3.65), however Equation (3.67) includes the respective multiplications and associated arithmetic error terms. Each of these error terms has the same variance, where the variance of a random variable is defined as the second central moment, which is the expected value of the variable squared after removing the mean. The mean is zero in this case, as round-off is considered, and not truncation [59, 87].

University of Cape Town

Defining variance [87]:

$$\sigma_\epsilon^2 = \mathbf{E}\left\{|\epsilon_x(n) - \mu_\epsilon|^2\right\} = \int_{-\infty}^{\infty} |\epsilon_x(n) - \mu_\epsilon|^2 p_\epsilon(n) dn \tag{3.68}$$

Each of the error sources are uniformly distributed between $-\frac{1}{2}2^{-B}$ and $+\frac{1}{2}2^{-B}$, and have a probability, $p_\epsilon(n)$, of $\frac{1}{q}$. Applying the limits and $p_\epsilon(n)$ gives:

$$\begin{aligned}\sigma_\epsilon^2 &= \int_{-\frac{q}{2}}^{\frac{q}{2}} |\epsilon_x(n)|^2 \frac{1}{q} dn \\ &= \frac{1}{q}\frac{1}{3}\epsilon_x(n)^3 \Big|_{-\frac{q}{2}}^{+\frac{q}{2}} \\ &= \frac{q^2}{12}\end{aligned} \tag{3.69}$$

The term $q$ in the expression in (3.69) is the quantization level for the system, and is between $-\frac{1}{2}2^{-B}$ and $\frac{1}{2}2^{-B}$, or simply $q = 2^{-B}$.

The final expression for the variance of the random variable $\epsilon$ is:

$$\sigma_\epsilon^2 = \frac{2^{-2B}}{12} \tag{3.70}$$

Prior to using the computed variance in (3.70), the squared magnitude of the complex error can be computed by isolating the error terms present in (3.67). The squared magnitude of the complex error, $\epsilon_A$, is:

$$|\epsilon_A|^2 = \Big[\epsilon_1 + \epsilon_2\Big]^2 + \Big[\epsilon_3 + \epsilon_4\Big]^2 \tag{3.71}$$

Using the computed variance in Equation (3.70), the noise terms in Equation (3.71) can be expressed in terms of expected values (variance is the expected value of the second central moment). The average or expected value of $|\epsilon_A|^2$ is

University of Cape Town

then [59]:

$$\begin{aligned}\mathbf{E}\left\{|\epsilon_A|^2\right\} &= 4\frac{2^{-2B}}{12} \\ &= \frac{1}{3}2^{-2B}\end{aligned} \tag{3.72}$$

The average value expressed in (3.72) is the expected value for each instance of complex error that occurs in the DFT computation. In Equation (3.64), the overall error in a DFT output vector is a combination of scaled arithmetic errors, $\epsilon_A$, as well as scaled coefficient quantization errors, $\sigma_u$. The coefficient quantization error term is theoretically non-statistical (as they are fixed for a given DFT length), however, can be approximated as a noise source with a uniform distribution. This therefore has the same distribution as $\epsilon_A$, and is uniformly distributed between $-\frac{1}{2}2^{-C}$ and $+\frac{1}{2}2^{-C}$, where $C$ represents the fractional bit length assigned to the coefficient representation.

Equation 3.50 defines the error to be the difference between the exact root-of-unity coefficient and the quantized coefficient, so the variance will be the sum of the variances for the real and imaginary parts respectively. The expected value for the coefficient quantization error can be shown to be:

$$\begin{aligned}|\sigma_u|^2 &= 2\frac{2^{-2C}}{12} \\ &= \frac{1}{6}2^{-2C}\end{aligned} \tag{3.73}$$

Reverting back to Equation (3.64), the noise sources can be replaced with the expected values computed from (3.72) and (3.73). Using (3.64), the magnitude

University of Cape Town

squared of $E_u[l]$ is:

$$|E_u[l]|^2 = |\sigma_{u(l:1)} S_u[l]|^2 + |\sum_{p=1}^{l-1} W_N^{pu} \sigma_{u((l-p):1)} S_u[l-p]|^2 + |\sum_{p=1}^{l-1} \sum_{m=1}^{2^{(l-p)}-1} W_N^{pu} \epsilon_{A((l-p):m)}|^2 + |\sum_{m=1}^{2^l-1} \epsilon_{A(l:m)}|^2 \quad (3.74)$$

The variance of a variable is defined as the expected value squared of the second central moment of the variable. The terms $|\sigma_u|^2$ and $|\epsilon_A|^2$ can be therefore be replaced with the variances computed previously (3.72 and 3.73). The expression in (3.74) then becomes:

$$\mathbf{E}\left\{|E_u[l]|^2\right\} = \left[\frac{1}{3}2^{-2C}\right]\Big|S_u[l]\Big|^2 + \left[\frac{1}{3}2^{-2C}\right]\Big|lW_N^{pu} S_u[l-p]\Big|^2 + \left[\frac{1}{3}2^{-2B}\right]\Big|l2^{l-p}W_N^{pu}\Big|^2 + \left[\frac{1}{3}2^{-2B}\right]\Big|[2^l]\Big|^2 \quad (3.75)$$

The expression in (3.75) assumes that all DFT points generate coefficient quantization, however not all DFT points will incur coefficient quantization, as some complex exponentials can be unity in value. It is assumed that coefficient quantization does occur at each point, and in this manner it is possible to produce an upper bound on the output noise. To obtain an expression for the noise-to-signal ratio, the output signal needs to be considered. This is considered in the next section.

University of Cape Town

**Output Signal Component** To compute an expression for the calculation of the noise-to-signal ratio at the output of the RDFT, first consider an uncorrelated input sequence(real and imaginary also uncorrelated). The real and imaginary components have an amplitude density which is uniform between $-\frac{1}{\sqrt{2}N}$ and $\frac{1}{\sqrt{2}N}$.

Using these limits, and the process for computing the expected value of a variable, the average squared magnitude of the complex input is:

$$\mathbf{E}\left\{|x[n]|^2\right\} = \frac{1}{3N^2} \tag{3.76}$$

The expression for the squared magnitude of $E_u$ in (3.75) is the noise components at the output of the RDFT. It is therefore necessary to express the input sequence in terms of the output, so the noise-to-signal ratio can be formed. The DFT of input sequence must then be taken. The expectation operator can be placed inside the summation since the cross terms are uncorrelated and therefore average to zero. The expected value of the DFT of the input sequence is:

$$\begin{aligned}\mathbf{E}\left\{|X[u]|^2\right\} &= \sum_{n=0}^{N-1} \mathbf{E}\left\{|x[n]|^2\right\} |W_N^u|^2 \\ &= N\frac{1}{3N^2} \\ &= \frac{1}{3N}\end{aligned} \tag{3.77}$$

University of Cape Town

**Forming the ratio** Combining the expected values for both the noise (for any iteration value $l$) and output sequences in (3.75) and (3.77) yields:

$$\frac{\mathbf{E}\left\{|E_u|^2\right\}}{\mathbf{E}\left\{|X[u]|^2\right\}} = 3N\left[\left[\frac{1}{6}2^{-2C}\right]\left|S_u[l]\right|^2 + \left[\frac{1}{6}2^{-2C}\right]\left|lW_N^{pu}S_u[l-p]\right|^2 + \left[\frac{1}{3}2^{-2B}\right]\left|l2^{l-p}W_N^{pu}\right|^2 + \left[\frac{1}{3}.2^{-2B}\right]\left|[2^l]\right|^2\right] \tag{3.78}$$

Assessing Equation (3.78), the noise-to-signal ratio increases linearly with $N$. Due to the recursion involved in the recursive DFT, the cause of the error growth is not limited to the size of $N$, but the number of bits allocated to the fixed-point precision, as well as the number of iterations performed. This error will grow continuously and eventually hit an upper bound (described in Section 3.9 which follows shortly).

If the error correction algorithm is invoked, the method of noise-to-signal ratio computation does not change, except the error present is reduced every time the error for any of the DFT components equals or exceeds a pre-defined threshold. The error value is highly dependent on the incoming data sequence (as can be seen in Equation (3.41)), and the noise-to-signal ratio which results after the error correction will dependent on the amount of correction which has taken place. The threshold forces an upper-bound on the error, and under the conditions outlined in Section 3.7.2, the error will remain bounded and predictable. The boundaries for the DFT computation should be defined, and are the subject of the following section.

University of Cape Town

## 3.9 Dynamic Range

Dynamic range is another constraint that needs to be taken into consideration. Quantization error focuses on the maximum precision that can be obtained for any computation, where as dynamic range takes the growth of a number into consideration.

Given a register length of $B$-bits, if maximum precision is required, then all bits are allocated to precision with the exception of the sign bit if signed fractions are used. This is advantageous for precision, but limits the output to be less than unity before an overflow occurs. It can be useful to permit integer growth, however this comes at the cost of precision for fixed-point processing.

One solution would be to increase the overall bit range, however this is not always suitable for fixed length processing architecture. It is therefore necessary to quantify the bounds for which numbers can be represented before overflow occurs which will invariably increase distortion. In this system, a subset $B_i$ of the total $B$-bits in the data registers are allocated to integer growth, with the remaining bits ($B_p$) used for precision (1-bit retained for sign). At this point, it is necessary to recall the original definition for the update DFT of point $u$ (expressed in (3.79)).

$$F_{new}[u] = \left[F[u] + \frac{f_{in} - f_{out}}{N}\right] \exp\left[j\frac{2\pi u}{N}\right] \tag{3.79}$$

For no overflow condition to occur, the magnitude of the final output $F_{new}[u]$ cannot exceed $2^{B_i}$. Stated concisely:

$$|F_{new}[u]| \leq 2^{B_i} \tag{3.80}$$

which in turn requires the contributing terms to be bounded as well.

University of Cape Town

Thus:

$$|F[u]| \leq 2^{B_i-1} \tag{3.81}$$

$$\left|\frac{f_{in} - f_{out}}{N}\right| \leq 2^{B_i-1} \tag{3.82}$$

The conditions expressed in (3.81) and (3.82) need to be strictly adhered to for the worst case when $|\exp\left[j\frac{2\pi u}{N}\right]| = 1$. The constraint loosens when the division by $N$ occurs, or when $|\exp\left[j\frac{2\pi u}{N}\right]| < 1$. In the case of division by $N$, the difference between the incoming and outgoing samples, namely,$|f_{in} - f_{out}|$ can be $N$ times greater before overflow will occur.

A similar condition holds for the case of the reverse recursive Fourier transform. Recalling the definition:

$$f_{new}[n] = [f[n] + F_{in} - F_{out}] \exp\left[-j\frac{2\pi n}{N}\right] \tag{3.83}$$

where $|f_{new}[n]| \leq 2^{B_i}$, which can be guaranteed if both $|f[n]| \leq 2^{B_i-1}$ and $|F_{in} - F_{out}| \leq 2^{B_i-1}$.

University of Cape Town

## 3.10 Simulation

The discussion in this chapter has been focused on the theoretical aspects of the recursive DFT, and has shown that the recursive DFT can be computed using finite-bit arithmetic where accumulated error can be corrected and bounded under the right conditions. The ability to compute with finite-bit arithmetic makes the practical implementation feasible on platforms offering gate-level programming (such as the Field Programmable Gate Array(FPGA)).

A useful characteristic of the recursive DFT is the ability to update an existing output with minimal overhead after receiving only one new sample, alleviating the need to gather a block of samples and compute the newer spectral output. This beneficial characteristic can best be illustrated using an example of continuous channel assessment.

University of Cape Town

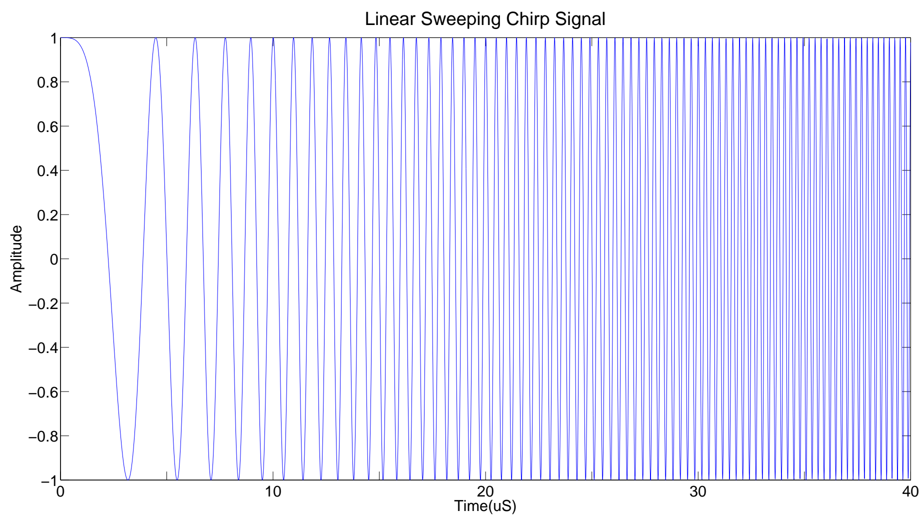
### 3.10.1 Continuous Channel Assessment

Continuous channel assessment is a convenient vehicle to expose the benefits of the recursive DFT in a communications environment. In the field of telecommunications, it is important to monitor and analyse the utilization of Radio Frequency (RF) communication channels. Often channels are multiplexed and hence can have multiple sources, where each source can put energy into the channel in an asynchronous fashion. To prevent collisions, receivers are required to measure total received energy and determine if the channel is in use [88]. Other applications of channel assessment can be in the identification of transmissions in bands of interest; the presence of harmonics in fault diagnosis, as well as frequency tracking.

Traditionally, this channel would be sampled at some rate $T_s$ where $N$ samples would be collected before computation would commence (usually an FFT of some form). As an example of continuous channel assessment, let us compute a 64-point complex valued DFT on an incoming chirp data sequence starting from DC and sweeping linearly to 10MHz using $10^4$ samples over a period of $100\mu$S and discretized by the ADC. The input signal is illustrated in Figure 3.29 for the

first 40$\mu$S.

This example case can be simulated and illustrated using Matlab®, and it is assumed that the input signal is known a priori, but in practice the signal could be any time varying phenomenon. For purposes of comparison, the recursive DFT result will be compared to the block computed result from a 64-point FFT.




University

Figure 3.29: Linearly Sweeping Chirp Signal: DC to 10MHz in 100$\mu$S. The Chirp begins with a discontinuity at time $t = 0$. This discontinuty influences the energy dispersion accross the finite bins in the frequency domain for the first $N = 64$ iterations, and can be seen in Figures 3.34 to 3.37.

#### 3.10.1.1 Block FFT

In this example the goal is to compute and compare a 64-point FFT processed output with a 64-point recursive DFT output using $10^4$ samples. Processing the input chirp sequence using the FFT requires the dataset to be block-processed, meaning the simulated front-end would first obtain 64 samples ($N = 64$) before presenting the data to the input stage of the FFT (Figure 3.30).

Practically, an ADC front-end would capture at some rate $T_s$, and once $N$ samples have been gathered, would present these for further processing. After the first $N$ samples have been captured, the processing (FFT) will commence. In this example, it is assumed that the data sequence has been double buffered, thereby allowing the next $N$ samples to be captured while the previous set is processed(this would incur an initial buffering latency). This naturally implies that if the processing latency is longer than the total sample acquisition time for $N$ samples, then periods of 'black-out' will exist in the time-frequency representation.

Figure 3.30 represents this system diagrammatically, and for conciseness, assumes all RF front-end processing is included in the 'ADC' block. Received data would be double-buffered, and presented to an FFT module. The transfer has been represented using a parallel, $N - wide$ bus, however the method of transfer could also be sequential. While this has significant differences in practical implementation, the emphasis here is that multiple data samples need to be available prior to FFT processing.

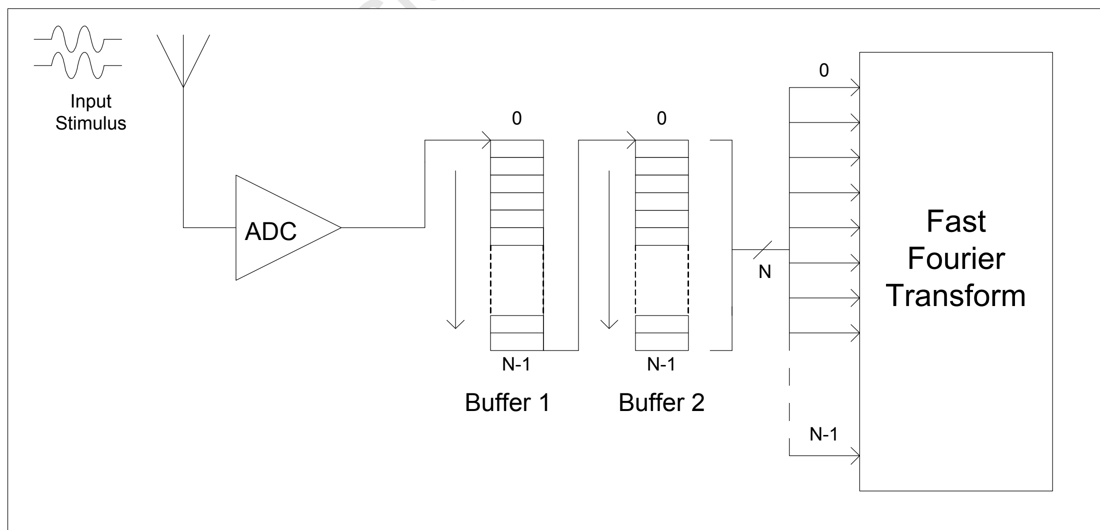



Figure 3.30: Block FFT Processing: ADC Front-End with Double-Buffering

To simulate this system, the ADC front-end and double buffering can be replaced with a segmented data sequence, where the dataset is first segmented into consecutive blocks of data, each containing $N$ samples. These blocks of data are processed sequentially, producing a time-frequency representation. The segmentation-and-compute process is aimed to simulate the sample-and-compute process found in a real hardware implementation. Figure 3.31 illustrates the segmentation-and-compute process.

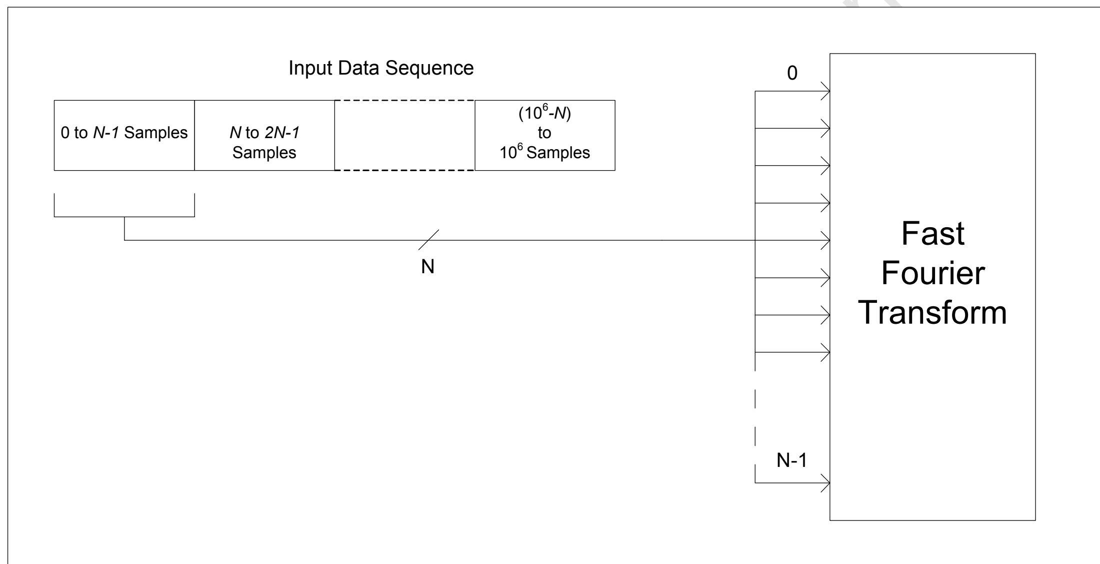



Figure 3.31: Block FFT Processing: Segmented Data Front-End

From the diagram shown in Figure 3.31, the time-frequency resolution produced will be dependent on the size of the FFT (which defines the number of samples segmented per block). Each sample block represents a time period of $T_s \times N$ seconds, and would limit the signal energy captured (and update rate observed).

To improve the update rate, faster processing would be required in the FFT block, as well as quicker sampling rates for the ADC (and associated memory accesses). To generate a time-frequency representation, the respective FFT outputs per segmented input block are collected to produce a complex valued dataset representing frequency decomposition per block with respect to the time resolution per block.

#### 3.10.1.2 RDFT

Processing the chirp signal shown in Figure 3.29 for the recursive Fourier transform differs slightly from the method employed for block-FFT processing. The FFT implementation requires the full $N$ samples to be presented prior to computation, where the RDFT computes on a sample-by-sample basis. From a practical implementation perspective, the bus, memory, and data transfer requirements are far lower, as only a single sample needs to be transferred per data acquisition cycle of the ADC. Figure 3.32 illustrates the simple implementation required for data processing. It should be noted that the *Recursive Fourier Transform* block abstracts away the initial processing required for the recursive DFT (covered in Chapter 4).

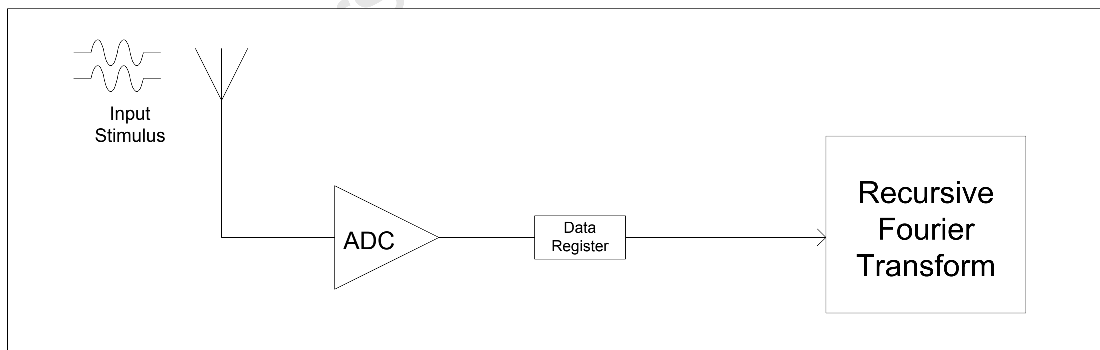



Figure 3.32: ADC Front-End Sample-by-Sample Processing using the Recursive Fourier Transform

University of Cape Town

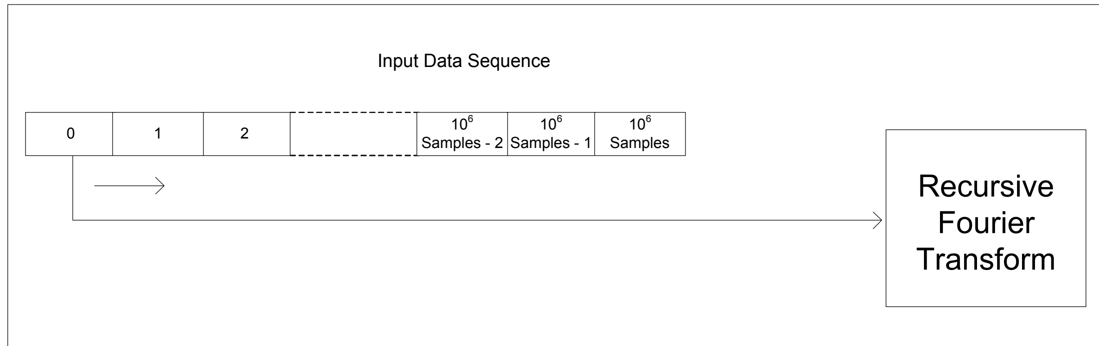



Figure 3.33: Data Sequence Sample-by-Sample Processing using the Recursive Fourier Transform

The equivalent simulation based processing model is illustrated in Figure 3.33, where the ADC front-end and data register have been replaced with a data sequence which iterates through contiguously stored data, providing a single sample per iteration. The single sample per iteration allows the frequency spectra for the incoming signal to be updated per iteration, which in turn has a time resolution linked to the sample time $T_s$. Initially, the first $N$ iterations will provide an incomplete picture of the incoming signal, as the spectrum (and associated input) is assumed to have been zero. After the first $N$ iterations have completed, the output will match the result shown by the initial block computed by the FFT, thereafter the recursive DFT provides a decomposition per sample iteration, where the FFT requires another $N$ samples taking a $T_s \times N$ time period.

University of Cape Town

### 3.10.1.3 Simulated Results

Processing the data sequence shown in Figure 3.29, produces the outputs shown in Figures 3.34, 3.35, 3.36 and 3.37. Figures 3.34 and 3.35 show the real-valued data of the complex DFT output, where the *x-axis* represents the associated block time (FFT) or sample iteration (RDFT), and the *y-axis* represents the decomposition frequencies. The *colourmap* which indicates the magnitude of each DFT point is also included. Similarly, Figures 3.36 and 3.37 shows the imaginary component of the complex DFT output. Finally, Figure 3.38 illustrates the RMS error which exists between the FFT and RDFT when computed every 64 iterations of the RDFT. These results will be discussed in the section which follows.

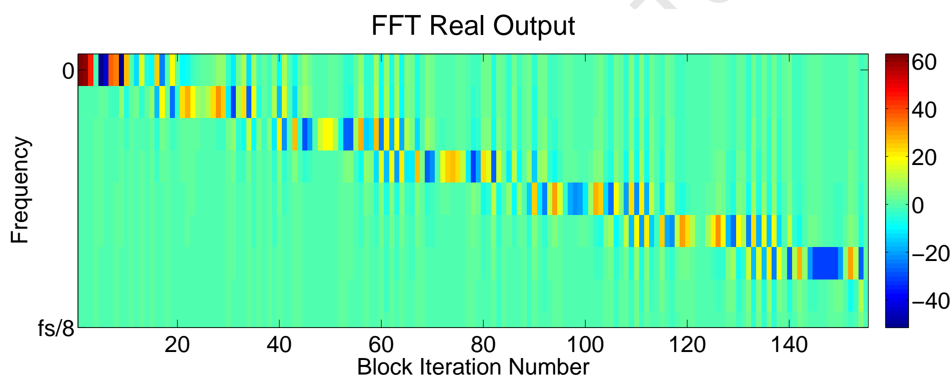



Figure 3.34: Real Output: Time - Frequency representation of the Chirp input sequence using Block-FFT processing (shown to the $\frac{f_s}{8}$ point for clarity as remaining channels are zero). The output is updated every 64 samples creating a block-based appearance in the time-frequency output.[1]

[1]It should be noted that the visual appearance of the illustrations shown in Figures 3.34, 3.35, 3.36 and 3.37 can be misleading due to print resolution and horizontal resolution of the illustrations. Figures 3.34 and 3.36 have a horizontal resolution of 160 data points, and Figures 3.35 and 3.37 have a horizontal resolution of 10000 data points (compressed into the same physical print space). Closer examination (electronically) reveals the similarity every $64^{th}$ sample, and is detailed in Figure 3.38.

University of Cape Town

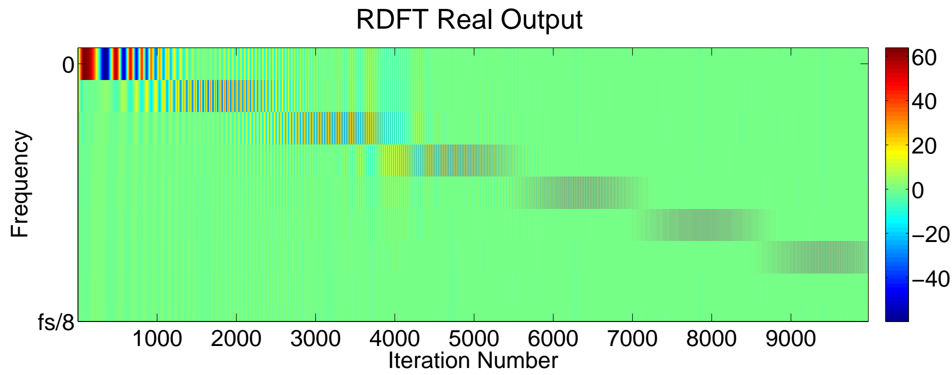



Figure 3.35: Real Output: Time - Frequency representation of the Chirp input sequence using RDFT processing (shown to the $\frac{f_s}{8}$ point for clarity as remaining channels are zero). The same input sequence is applied to block-based processing, except the windowed sequence is processed sample-by-sample.

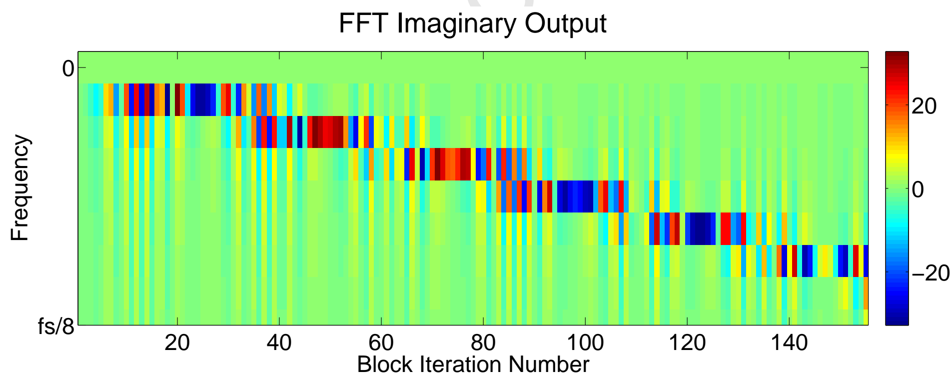



Figure 3.36: Imaginary Output: Time - Frequency representation of the Chirp input sequence using Block-FFT processing (shown to the $\frac{f_s}{8}$ point for clarity as remaining channels are zero). The output is updated every 64 samples creating a block-based appearance in the time-frequency output.

University of Cape Town

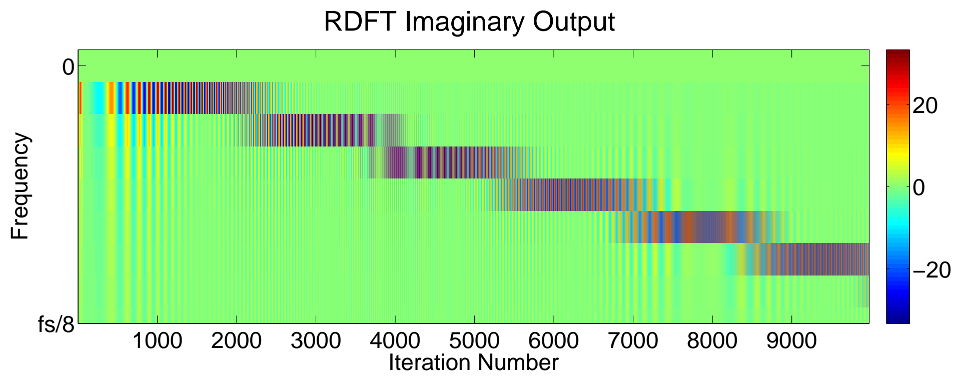



Figure 3.37: Imaginary Output: Time - Frequency representation of the Chirp input sequence using RDFT processing (shown to the $\frac{f_s}{8}$ point for clarity as remaining channels are zero). The same input sequence is applied to block-based processing, except the windowed sequence is processed sample-by-sample.

University of Cape Town

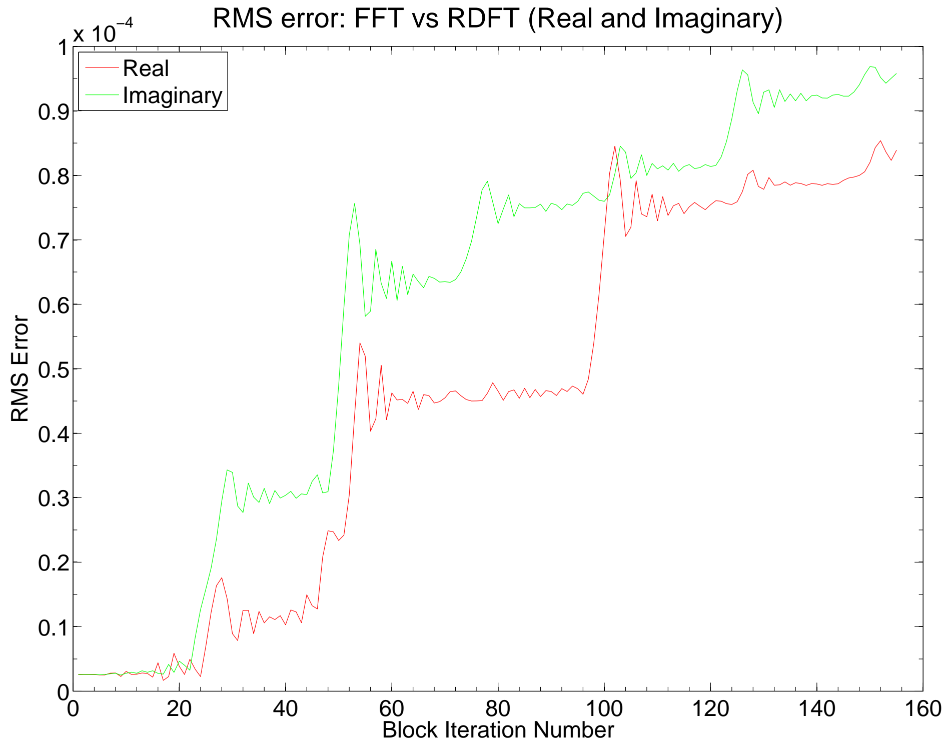



Figure 3.38: Real and Imaginary output for the RMS error between the FFT and the RDFT every 64 iterations. In this example, the error grows rapidly and the rate of increase slowly decreases as the error correction is invoked. The experiment operated with $\upsilon = 2^{-10}$, and worked over a dataset of approximately $10^4$ samples.

#### 3.10.1.4 Discussion of Results

The results shown in Figures 3.34 and 3.36 for the block-processed FFT, and in Figures 3.35 and 3.37 for the recursive DFT, can now be discussed in greater depth. For the block processing approach, the output results are segmented into blocks of data spanning 64 samples per block, and therefore have an output resolution which can only be updated every 64 samples.

Analysing Figures 3.34 and 3.36, it is possible to detect the block-size dependent resolution. The FFT and RDFT outputs (real and imaginary magnitude) are represented by a colourmap alongside each illustration. A single-sided spectrum is shown in all Figures (up to the $\frac{f_s}{8}$ point for clarify, as the remaining channels up to $\frac{f_s}{2}$ are zero).

Comparing the outputs for the FFT and the recursive DFT, an interesting picture emerges. It can be seen that the spreading of energy is prevalent in the initial stages of computation, but decreases significantly as the window is shifted. The spreading which occurs in the beginning stages is due to the initial discontinuity in the input data sequence. The input data sequence is assumed to be zero at $t = 0$, and therefore will have a discontinuity present in the window frame for the first $N$ samples (unless the input sequence rises gradually and has no sharp transients initially). Taking a closer look at the input sequence illustrated in Figure 3.29, the very first sample input to the rectangular window has a value of unity (and will cause a discontinuous jump from zero to 1 after the first iteration).

Comparison of the two outputs can be performed if the RDFT is analyzed with respect to the FFT every 64 iterations. The RMS of the difference between the two algorithm outputs is computed and illustrated in Figure 3.38. As could be expected, the error rapidly increases due to the errors associated to recursive fixed-point arithmetic. The threshold ($\upsilon$) is set to $2^{-10}$ in this example, and results in a slower convergence in expected error for given word length and threshold. It can be seen that the rate of increase in the RMS error starts to decreases as the number of block iterations exceeds 100 (this would be approximately 6400 iterations for the RDFT). The small number of iterations would result in accumulated error less than the stipulated threshold and would therefore permit accumulation. This is seen in the traces in Figure 3.38 where the magnitude of the RMS error for both the real and imaginary components is in the expected range for the word length and threshold as discussed in section 3.7.2.

University of Cape Town

## 3.11 Chapter Summary

This chapter discussed the recursive Fourier transform in finer detail, and included characteristics such as arithmetic accuracy, quantization effects and noise modelling, and introduced an error correction and detection algorithm to bound error growth for finite bit arithmetic. Computational complexity was discussed and showed to be $O(N)$ for a DFT length $N$, if $N$ processing elements are used (forward and inverse transform). The work showed that it is possible to compute both the forward and reverse Fourier transform in an analogous manner, where the computations differ in the presence of the division by $N$ and the negation of the complex exponential.

The chapter also discussed the implications of computing using finite bit arithmetic, and showed the error introduced in the output is a function of both arithmetic round-off, and coefficient quantization (truncation due to the division by $N$ is also included for the forward transform case). If unrestrained, the error accumulates and grows without bound due to the recursion process, and it was shown that it is possible to compute and correct the output to a high degree of accuracy on-the-fly.

University of Cape Town

An error model was developed to investigate the relationship of the finite bit length and induced error, and it was found that a difference of 14-bits should exist between the overall output fixed-point length and the threshold ($\upsilon$), which defines the limit used to activate the error correction. The error model is also extended to a generic perspective, and used to determine the rate of error growth for any arbitrary length DFT (64 points in this case).

The results showed that the predicted error closely matches the true error, and the arithmetic accuracy of the system can be determined with high confidence. Finally, assessing the Noise-to-Signal ratio, it was shown that the noise present scales linearly with $N$, and is also a function of the number of bits allocated to the fixed-point precision, as well as the number of iterations performed.

The chapter finally concludes with a continuous channel assessment example, and compares the results of block processing which is characteristic of an FFT, and the sample-by-sample updating attribute of the recursive Fourier transform which proved to be instrumental in providing a finer time-frequency analysis. Comparison of results showed that the two transformation algorithms (FFT and RDFT) produce correlative results every 64 iterations, and differ only by the error margin induced through fixed point arithmetic. The output resolution of the FFT is governed by the input length ($N = 64$ in this case), however the recursive Fourier transform is able to update on each sample input.

An initial study performed in Chapter 2 traded-off many attributes to refine the selection of a suitable technique for the processing framework. This chapter extended on this evaluation and highlighted the benefits and caveats of the selected technique. The following chapter will now build onto this foundation of the recursive Fourier transform and show how it can be implemented into a processing framework and prototyped on silicon.

University of Cape Town

# Chapter 4

# Parallel Framework: A Many Element Approach

University of Cape Town

## 4.1 Introduction

This chapter discusses the processing framework developed for spectral method based computations. Previous chapters have discussed the key computations considered in the spectral methods class, of which the Fourier transform realised in discrete form is found to be a fundamental building block. Chapter 3 describes the Recursive Discrete Fourier Transform (RDFT) in detail and reveals that the algorithm used to realise the DFT computation is inherently parallel and therefore lends itself to practical implementation in a parallel VLSI system. The aim was to develop a generic spectral processing framework which can adequately support the requirements set out by the RDFT, however, be flexible enough to accommodate the additional operators specific to spectral method computations. This chapter concludes with an overview of the hardware platforms and software tools used in developing and testing the system.

## 4.2 Contributions

- Parallel framework for spectral computations. This chapter introduces a parallel finite-bit arithmetic framework for computing the RDFT, where all DFT points are computed synchronously. The framework exploits redundancy by sharing hardware resources where possible, and provides supporting mechanisms for implementing the additional Fourier space operators identified in Section 2.3.

- Error Correction Engines. The computational hardware of the RDFT is implemented in this study using finite-bit arithmetic, and when combined with the recursive nature of the RDFT, leads to arithmetic error growth which does not converge unaided. Chapter 3 discussed a means to dynamically correct accumulative error on-the-fly by utilising the input data; the quantized output; accumulated past error; and known quantization errors of coefficients. This chapter introduces an error correction engine implemented in hardware to perform this error computation which is used to correct the output vector. This engine is identical for all DFT points, however computes separate error vectors per point, and is therefore embedded in each processing element per point for exclusive use.

University of Cape Town

## 4.3 System Framework

To begin with describing a system framework, it is first necessary to reflect back onto the ideal properties previously outlined for a spectral methods based system. An ideal framework would offer:

- System controller
- Support scalable parallelism
- Generic to spectral methods class
- Provide support for Fourier transform computation
- Support additional Fourier operator implementation

- High-Speed Input/Output (I/O) interfaces
- Flexible memory system
- Shared resources
- Bus arbitration

Using the properties listed, it is the task of this study to produce a framework capable of satisfying these requirements. The results of the selection processes for the Fourier transform algorithm detailed in Chapter 2 revealed that the recursive Fourier transform offered the sought after features required for this framework. It is therefore necessary to consider the processing requirements for this algorithm as well as the requirements listed when developing a framework.

A desired property for this system is scalable parallelism. It was shown in Section 2.8 that the recursive DFT requires no inter-point communications during processing, and shows that each processing element can operate independently. This approach promotes scalable parallelism, and the only two further requirements for processing is initial coefficient loading and an input data stream.

To handle the control logistics for the system, a system controller should be included. The system controller should be tasked with handling processor addressing and coefficient loading into the respective processing elements; memory address generation; as well as bus arbitration control. A further task would include I/O control for the exporting and importing of data (importing data can be from a stored location), however as mentioned in Section 1.5, I/O control and external data transfer will not be addressed in this work.

University of Cape Town

The flexibility property listed refers to the ability of the system to adapt to changing design parameters. If the design is going to be adjusted (e.g. increasing the number of processing elements, or using custom coefficient values), the system must allow for parameterised configuration prior to design synthesis. The memory system and associated processor and memory addressing in the system controller should enable simple parameter entry changes to specify memory and addressing depth.

The proposed system framework (illustrated in Figure 4.1) also includes a memory subsystem. The memory subsystem stores all data coefficients required for transfer before processing commences, and should provide data buffering facilities for real-time input data streams. The memory subsystem also handles any bus arbitration tasks (control comes from the system controller), as the common bus used to supply data to the processing elements should be shared for data coefficient transfers (only valid during initial configuration stage), as well as input data streaming (data processing stage). It should be noted that the bus system, memory system and system controller are all shared resources, as there is no need for individual subsystems in each processor.

To support additional Fourier operators, the internal architecture of each processing element should provide such support. In a handful of cases (such as frequency shift property and reciprocal scaling property), it would be required to transfer data between processors to perform the desired operation once the initial Fourier transform has been computed. The internal architecture of each processing element would otherwise require a combination of accumulators, multiplexers, data registers and control lines from the system controller. The prototype system described in this work will however not discuss an implementation of this additional hardware requirement; however further details are provided in Section 6.1.

University of Cape Town

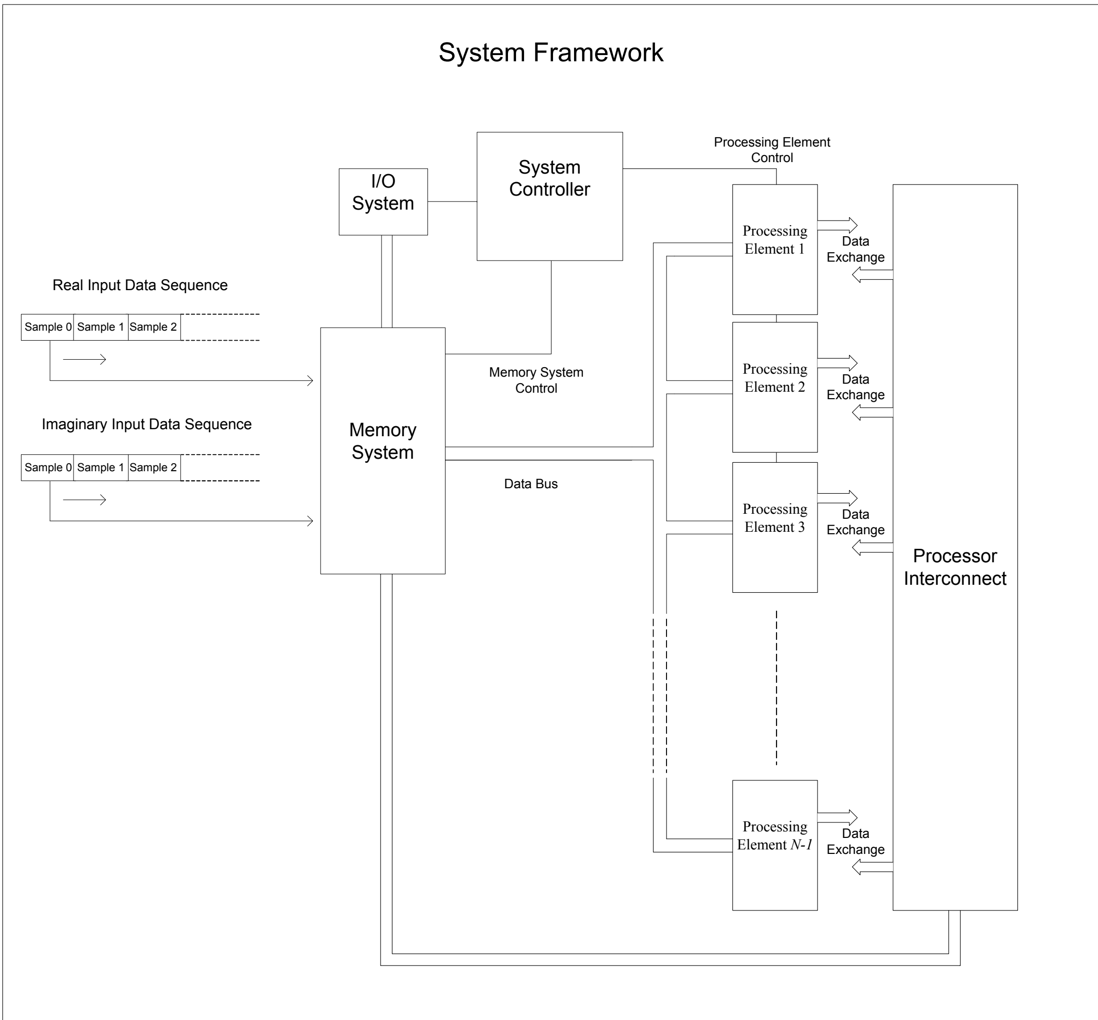



Figure 4.1: System Framework. A common memory system is used to supply all processing elements streamed data through a shared bus. The system controller provides all the necessary control signals for memory, I/O, and processing element control.

## 4.4 System Overview

The initial discussion of this chapter focused on the high-level framework for this system. We will now provide a detailed system overview. A high-level system overview is illustrated in Figure 4.2 (details such as control signals and multiplexed buses are abstracted away). A closer look shows that the high-level view of the system contains a finite state machine (system controller), two memory systems and the processing elements. The state machine handles initialisation (including bus arbitration and control signals for loading of coefficients from memory), and finally input source control.

The memory system has been separated into two distinct parts - memory for input sample buffering (and delay for the $N$-sample delay needed for $f_{out}$ in the computation - see expression 3.15 in Chapter 3), and memory for both complex exponential coefficient storage and $\sigma_u$ coefficient storage. A common bus runs between the memory systems and processing elements, and is arbited by a multiplexer (not shown in the illustration), controlled by the finite state machine. The input sequence allows a complex input, and is latched and stored synchronously with the main clock of the system. The specifications for each of these sub-systems will now be discussed in detail, and operational details will be discussed in following sections.

University of Cape Town

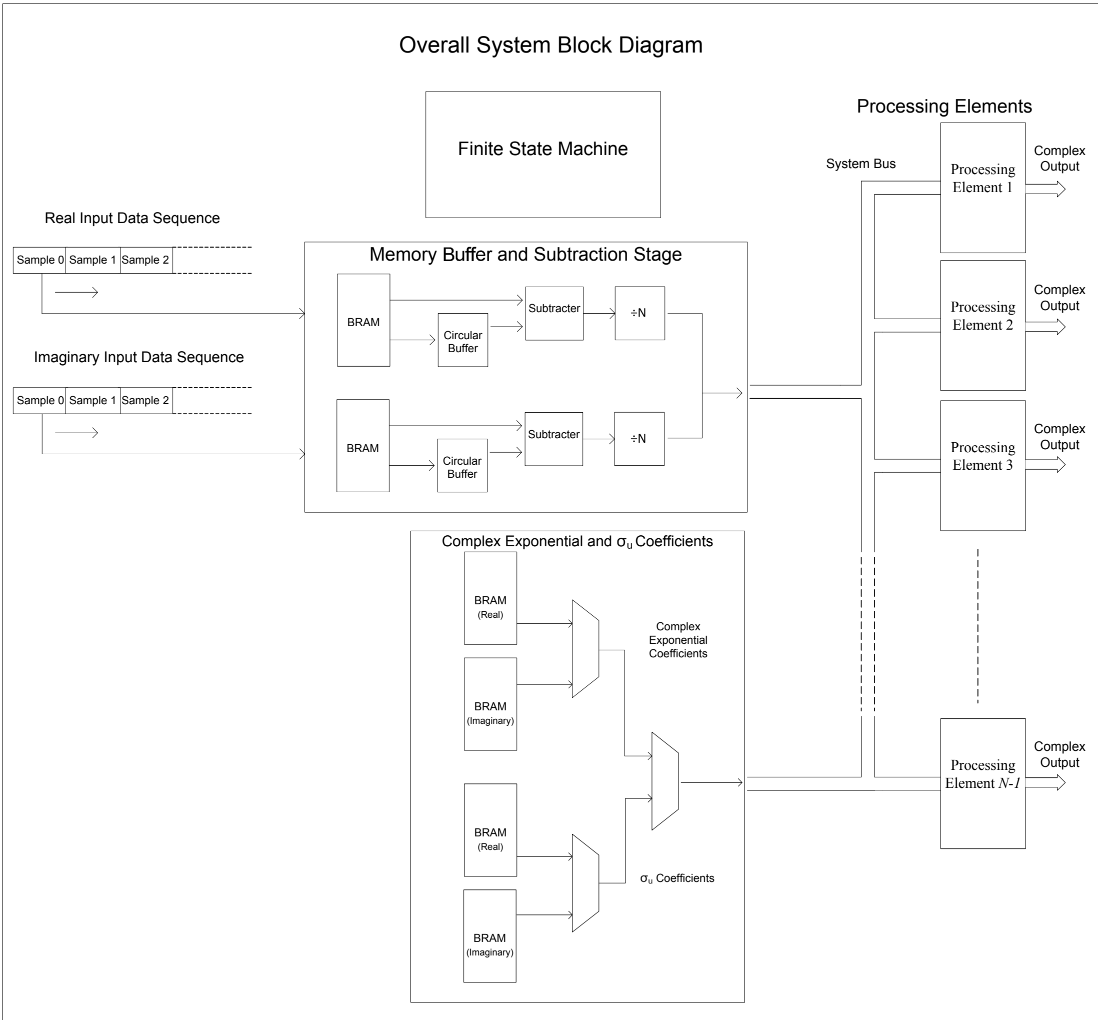



Figure 4.2: High Level System Overview

## 4.5 System Specifications

It was noted earlier that the system described in this study has not been developed for any particular application and remains generic to any application requiring a DFT computation. This creates many interesting design trade-offs, as a hardware implementation is usually optimized for a given application, and system details such as bus widths, arithmetic and coefficient precision, as well as throughput become specific to the context of the application.

The hardware developed in this study forms an initial proof-of-concept design for the theory covered in past chapters. The DFT output resolution has been limited to 64 points, which was chosen based on the largest power-of-2 DFT output width that could fit using the FPGA resources available (initial design did not include any error correction hardware). The choice of DFT width has no theoretical limit, as the algorithm maps comfortably to any arbitrary width.

As will be discussed shortly, it was not possible to place-and-route the final version of this prototype system to the FPGA used for testing, as the inclusion of the error correction engines exceeded the total DSP blocks available on the devices (sub-sectional design implementation and testing was performed on the FPGA hardware instead). The design specifications have been kept as generic as possible, and have been listed in Table 4.1.

The data input to the system consists of a single 36-bit sample (real or complex) per iteration. The input system has been designed to accommodate a preloaded data sequence in BRAM, or to accept real-time input samples streamed from a source such as an Analog-to-Digital Converter (ADC). The bit length of the input samples can be up to 36-bits (signed), and the fixed point precision was set to 24-bits, allowing 11-bits for integer growth (1-bit remains to indicate sign).

Buffer memory (implemented in BRAM) is provided for input data buffering, allowing the input state to change and settle during the processing period, without affecting the final output for the current data sample. The outputs are all

University of Cape Town

Table 4.1: System Specifications

| *Component* | *Specification* |
| --- | --- |
| Input | 36-bit (Complex) |
| Outputs | 64 (Complex) |
| Resolution | 36-bit (Signed) |
| Precision resolution | 24-bit |
| Buffer Memory | Block RAM (BRAM) |
| Complex Exponentials | Pre-Loaded (36-bit) |
| Complex Multipliers | DSP Blocks (36-bit) |
| Bus Width | 36-bit (Shared) |

University of Cape Town

complex, giving a combined total of 128 real and imaginary outputs, each 36-bits wide. The complex exponentials used during computation are pre-loaded into block RAM memory for fast access, and can be changed to allow computation of specific DFT points.

The complex multipliers implemented in both the DFT computation as well as the correction vectors allow 36-bit computation. A shared bus is implemented for transferring both data as well as coefficients (during initialisation) to the respective processing elements. The bus width running to the individual processing elements is 36-bits wide, and originates from the fixed-point resolution that was selected.

This system is aimed to target a wide range of applications, and it was decided to allow for up to 24-bits precision in the computations. It should be noted that if a lower precision can be tolerated, a practical choice could be 18-bits, as 18-bits is a standard size for a Xilinx BRAM primitive, and the latency incurred when performing a complex multiply (DSP blocks) remained constant for bit widths up to 18-bits. Implementing 18-bit hardware can provide additional computational latency savings (fewer cycles required for multiply), as well as permitting a larger-scale design to fit in available FPGA resources (fewer DSP and block memory resources required).

## 4.6 System Components

The processing system framework is designed to be modular, and separates key processing components into different stages. The fragmentation of a system offers benefits such as modular design and debugging, and the ability to replace existing modules with newer and possibly more efficient or higher performing units. The design approach in this system adopted such a practice, and all modules which make up the overall system can be interchanged when required.

The high-level block diagram of the system illustrated in Figure 4.2 consists of a Finite State Machine (FSM), memory, support modules, and a bank of processing elements. The support modules handle common shared resources, and each processing element contains the hardware for DFT point computation, and a separate error correction engine. Each of these components will now be discussed in detail.

### 4.6.1 Finite State Machine Control Unit

The control unit used in this system is a finite state machine which manages the initial start-up procedure of the system, and places the system into a run state once the start-up is complete. The state machine consists of a total of 7 control states, most of which transition in an orderly manner with no deviation caused by external stimulus. The exception state is the final *Read Input Data* state, where the control line used to specify the data source influences the which operations follow. The functional state and flow diagram for this system is illustrated in Figure 4.3.

University of Cape Town

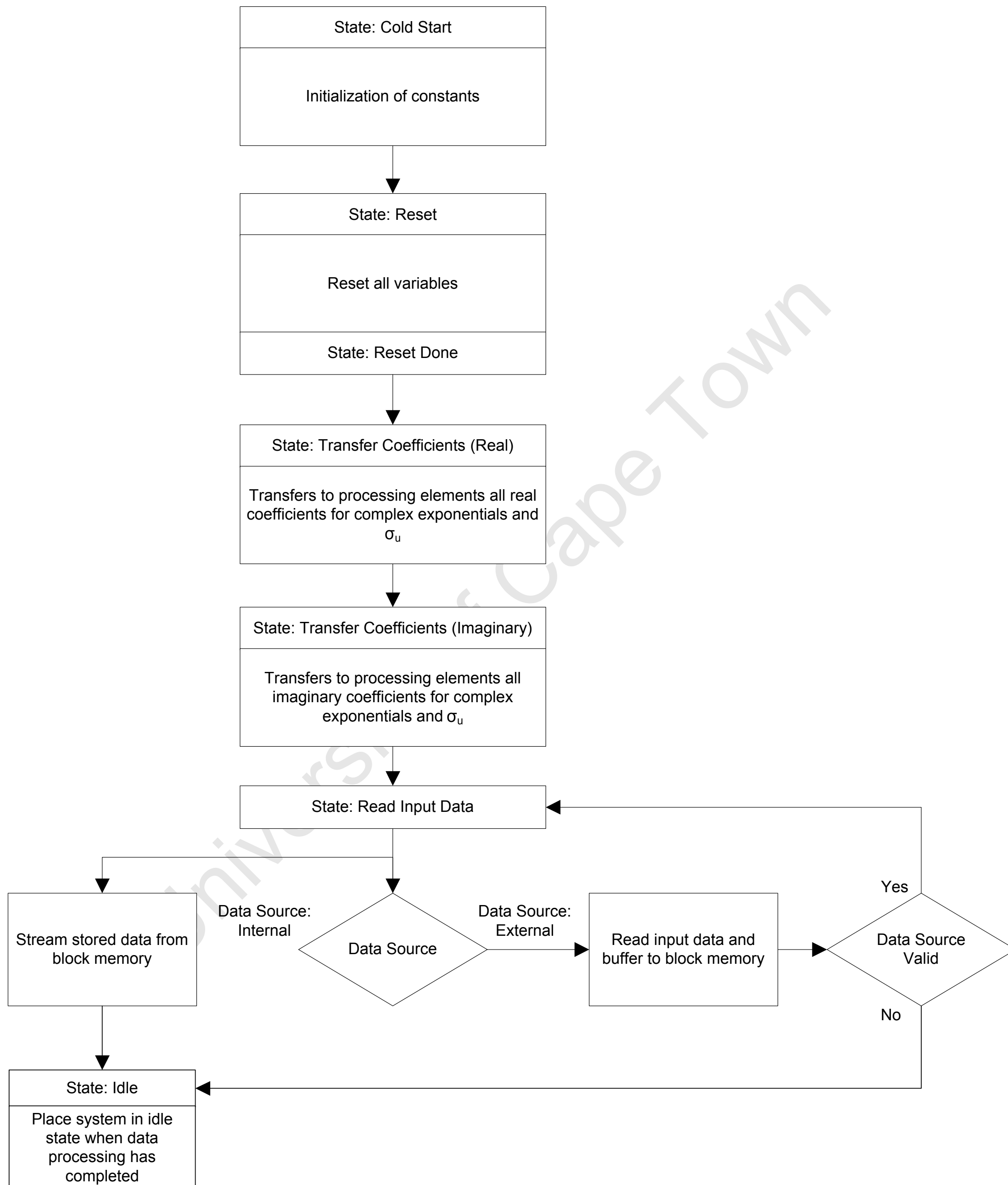



Figure 4.3: High Level System Overview

The state and program flow as illustrated in Figure 4.3 is separated into distinct states, each of which will be discussed individually.

#### 4.6.1.1 Cold Start

The *Cold Start* state is entered when the system is first powered up. All constants used are initialized for first use. These constants include number of processing elements used in the final processing stage, as well as the number of addressable spaces in memory. If the number of processing stages were changed (e.g. the number of DFT points to be computed grew), a single variable would need to be changed to instruct the state machine (in a following state) to address more processing elements.

#### 4.6.1.2 Reset

The *Reset* state is used to reset all run-time variables back to initial conditions. This includes the variables used to incrementally address processing elements, as well as address variables used to specify the required address space in all the memory banks.

#### 4.6.1.3 Reset Done

This state is briefly entered after a successful reset. No additional operations are performed, and this state was implemented to signify the completion of a reset for debug and monitoring purposes.

#### 4.6.1.4 Transfer Coefficients: Real and Imaginary

The transferring of the *Real* and *Imaginary* coefficients (complex exponential coefficients used during DFT computation, as well as the $\sigma_u$ coefficients required by the error correction engines embedded in each processing element) stored in block memory needs to be transferred in two stages across the shared bus. The processing elements each contain data registers to hold the coefficient data assigned during these two states. Each processing element is addressed separately and sequentially (as each is assigned its own unique vector for a given set of DFT

University of Cape Town

computations), for which four 36-bit data transfers are required. A shared bus is used for the transferring this data, and in each of the *Transfer Coefficient* states, the complex exponential data is first transferred, followed by the $\sigma_u$ data.

#### 4.6.1.5 Read Input Data

Once the processing elements have all been addressed and configured with the respective complex exponential and $\sigma_u$ coefficients, the system is ready for processing streamed input data. It is at this point the flow of the states may change. A control line input is supplied to the state machine which defines the source of the data to be processed. Two options are available, namely internal source (from pre-loaded block memory), or external source (such as an ADC).

If the internal source is selected, the necessary addresses are generated and supplied to the memory system to stream the data into the processing elements across the shared bus. When the final address has been reached, the system is placed in the *Idle* state, and will sit idle until new data has been loaded in to the block memory, and the system reset to initialize processing, or the data source is changed, and the system is re-started. If an external source is selected, the system remains in the *Read Input Data* state until changed.

#### 4.6.1.6 Idle

The *Idle* state in itself performs no additional operations, and is used simply as a holding state when the addressing of the internal block memory has completed.

University of Cape Town

### 4.6.2 Input Source Control

The system described in this work is capable of specifying two different sources of incoming data for further processing. The discussion of the possible states for the finite state machine highlighted that it is possible to select between streaming data from an external source, or simply from the embedded block memory. The detail of this hardware selection is omitted from Figure 4.2, and is illustrated in Figure 4.4 and Figure 4.5 which illustrates part of the memory system.

To select between the streaming input source or the pre-loaded internal memory, two multiplexers are used. The first (shown in Figure 4.4) handles the data input feed to the BRAM shown in Figure 4.5. When internal BRAM is selected as a data source, the data inputs for the BRAM modules are set to zero as the *select* line on the multiplexers forces *d0* as the selected input. The *Xil Bus Arbiter* control line comes from the state machine, and serves a dual purpose. It specifies which resource(coefficient memory or input data) can have access to the shared bus (through a separate multiplexer discussed in Section 4.6.4), and is also used in the input source control to zero any input stream to the system when the coefficient transfer state is underway (which is using the shared bus).

University of Cape Town

### 4.6.3 Memory System

#### 4.6.3.1 Input Buffer, Subtraction and Division

The source of data is specified by the *Data Source* control line, and if internal memory source is selected, the complex streaming input is ignored. In this case, the state machine provides the memory access addresses through the *Add A* bus, and disables any write access to the memory. The addressing iterates through the all the memory addresses consecutively, and the system is placed in the *Idle* state once the output is computed for the last data sample.

If the *Data Source* control specifies the external data source, the streaming data is first buffered into BRAM (Figure 4.5), before being placed on the bus to the processing elements. The *Data Source* control line is an input to the state machine, and when selected, write access to the BRAM memory is provided by the state machine.

University of Cape Town

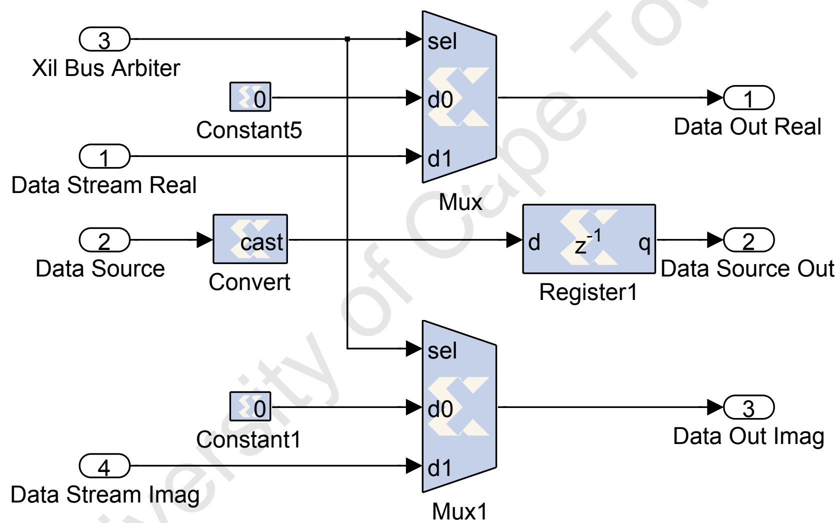



Figure 4.4: Input Source Multiplexing.

University of Cape Town

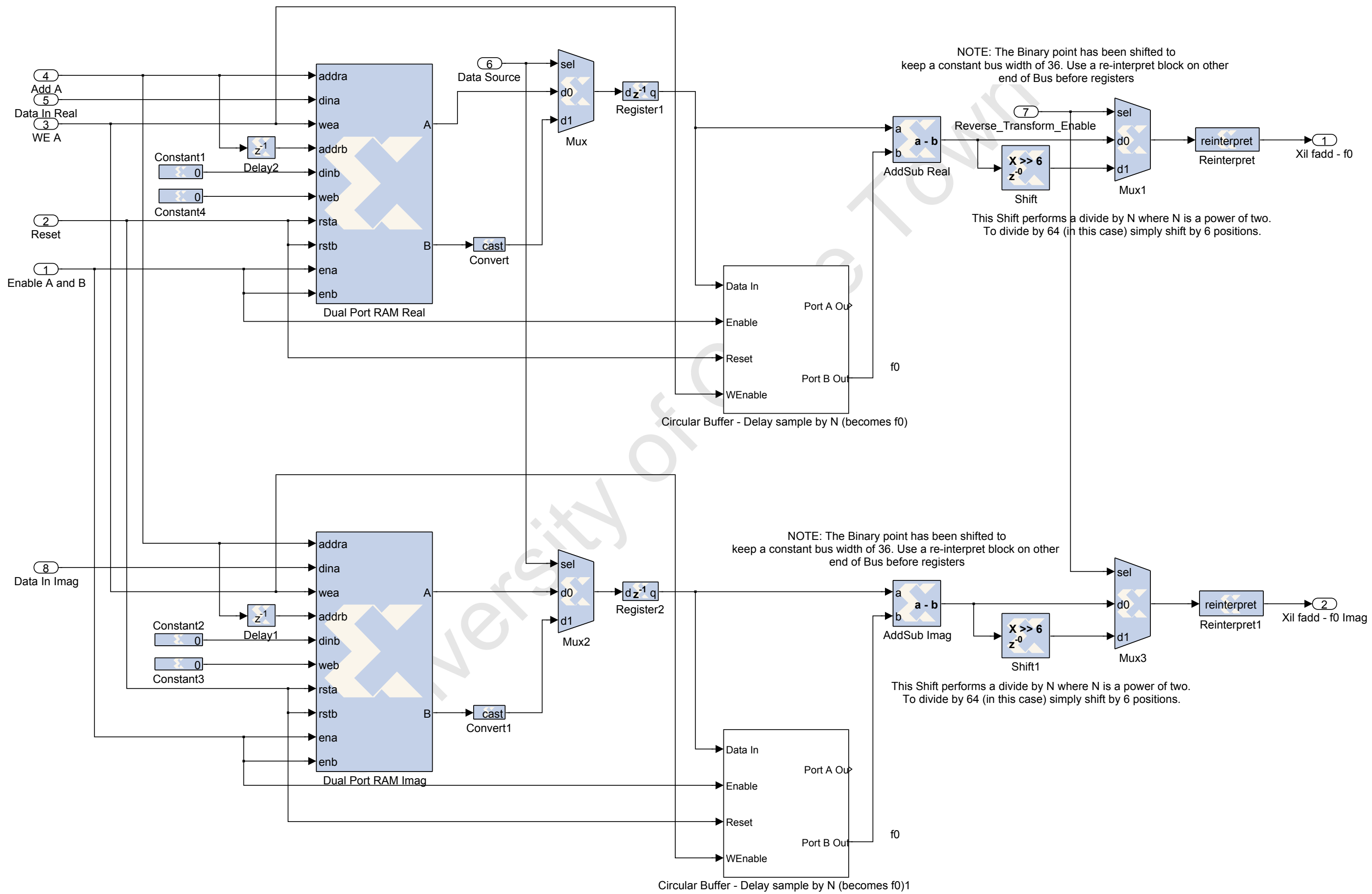



Figure 4.5: Input Data Memory System

The buffering process uses the same internal memory allocated for pre-loading (if applicable), however uses the *Port B* output from the BRAM module. When in the buffer state, the state machine generates the same address sequence (for writing), except the generated address is delayed by one sample period and provided to the *Port B* address input for the read address.

The *Data Source* control line specifies which of the two BRAM outputs to access before feeding to the circular buffer. The buffered samples iterate through the entire address space of the memory, and roll over to the start address when exceeding the maximum address space. The delay by one sample period ensures that the incoming sample never over-writes the current sample, as dual read-write access to the same BRAM address is not permitted. Figure 4.5 illustrates a dual memory system, as it is necessary to accommodate both *real* and *imaginary* data samples for a complex input.

The stage following the data memory and multiplexer is the circular buffer. The circular buffer is required to delay all samples entering the system by $N$ sample periods. This is to permit the difference calculation between the most recent incoming sample and the outgoing samples which occurred $N$ sample periods previously. To implement this circular buffer, a block of dedicated memory was declared and supplied for both *real* and *imaginary* data samples. Figure 4.6 illustrates the circular buffer.

University of Cape Town

The *From A* is a control line is supplied from the processing elements (element 0 in this case) which indicates when the computation has completed for a given input sample. This control flag is used to enable or disable a 6-bit free-running counter (synchronous to the clock) which is used to generate the addresses for the BRAM module (memory depth 64). This address is the write address (the same write enable *WEnable* control is used from the data memory) for the circular buffer memory. The write address is increased by one and supplied to the Port $B$ address. The Port $B$ output is the only output used in this sub-system, and is further delayed by another 24 sample periods to ensure the outgoing sample arrives at the input the subtracter *AddSub* at the correct time with respect to

the most recent incoming sample. The additional delay of 24 was determined based on the inherent delays incurred during writing and reading process (and associated logic) of BRAM memory.

The final stage in this sub-system is division. It was mentioned previously that this system took the approach of implementing a DFT computation that is a power-of-2. This in turn defines $N$ to be a power-of-2, and greatly simplifies the division requirement in the algorithm of the recursive DFT. The division for this system can be implemented using simple bit shifting, alleviating the need of additional hardware division resources. In this concept system, a value of 64 was chosen for $N$ ($N = 2^6$), which in turn requires a bit shift of 6 bit positions. In the implementation, a bit shifter in employed to perform the division by $N$, and the output is supplied to the final multiplexing section of this subsystem.

If a non-power-of-2 $N$ is desired, this block can be replaced using a divider implemented using a DSP slice in the FPGA. The final multiplexer is present to select between a divided or un-divided result. This selection became necessary as the framework described in this work is aimed to perform both the forward and reverse DFT. The mathematics is essentially the same, however the scaling factor of $\frac{1}{N}$ differs depending on the selection of the transform.

The final block following the multiplexer allows for the re-interpretation of the data provided by the multiplexer. The bus system is shared between multiple sources, each having a different arithmetic precision. To overcome the need to run separate buses, all data is interpreted as 36-bit words with a shifted binary point. The receiving modules are equipped to re-interpret this data to ensure data consistency.

University of Cape Town

### 4.6.4 Bus Arbitration

The bus arbiting process is controlled by the state machine which supplies a control line to the main bus multiplexer (shown in Figure 4.7). This multiplexer selects between the two main sub-systems requiring bus access - the coefficient memory and the input data memory system. Each of the these subsystems have their own data multiplexing for bus use(recall that this system can accommodate complex data - both coefficient and input data). Coefficient transfer requires its own internal bus control which will be discussed shortly). During the system initialization phase, the bus arbiter selects the first source (the coefficient memory sub-system), after which the state machine issues control to the data input.

#### 4.6.4.1 Computational Coefficients

The coefficient memory sub-system is illustrated in Figure 4.8, and handles the coefficient storage and transfer for the complex exponentials as well as the $\sigma_u$ terms. The addressing control for the memory comes from the state machine, and is the same address lines used to access the individual processing elements.

When the state machine is in the *Transfer Coefficients* state, the system addresses each processing element and provides the required coefficient data that particular element requires for the desired computation. The time required to transfer all coefficients is dependent on the DFT length ($N$), where the total cycles required for coefficient transfer is $N \times 4$. The multiplication by 4 accounts for the complex coefficients for both the complex exponentials, as well as $\sigma_u$.

It is at this point that any arbitrary DFT point can be defined. Before start-up, the BRAM can be pre-loaded with any complex exponential value necessary, and the transfer takes place during the start-up phase. As mentioned in the discussion on division by $N$(section 4.6.3.1), if a non-power-of-2 value is used, a full divider will need to be implemented requiring additional hardware resources.

University of Cape Town

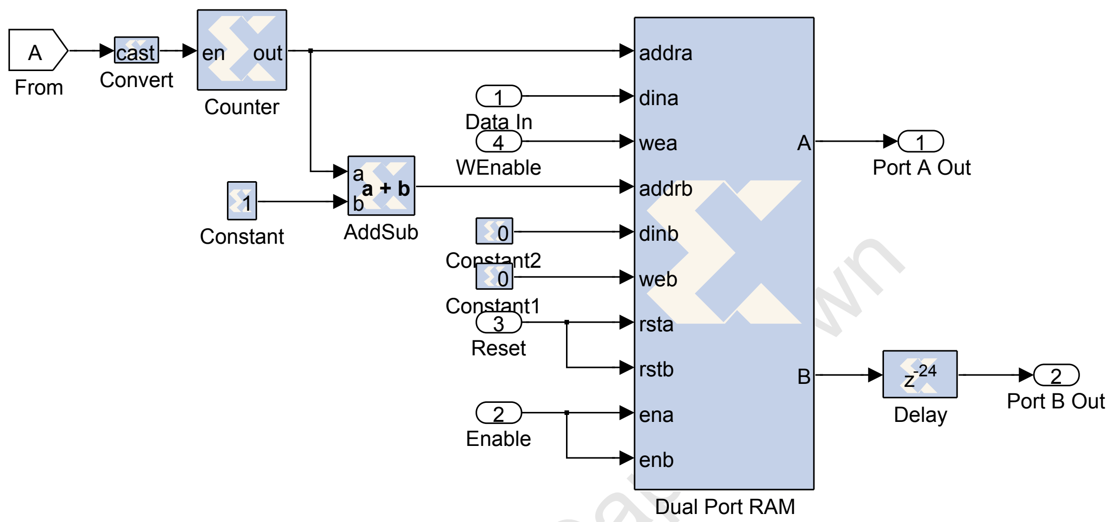



Figure 4.6: Circular Buffer

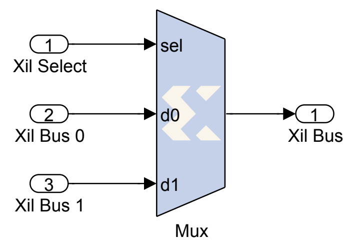
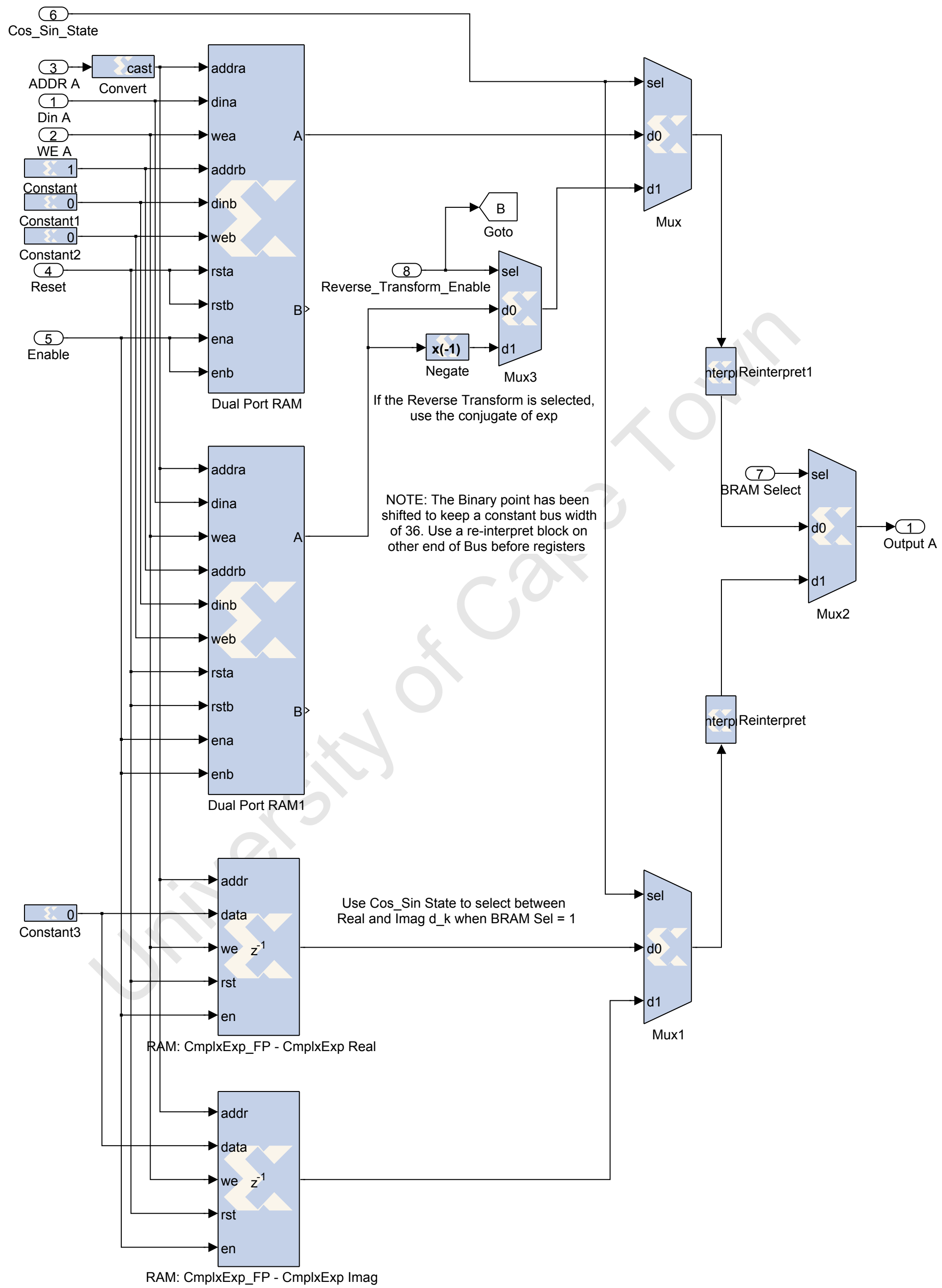



Figure 4.7: Bus Arbiter

University of Cape Town



University of Cape Town

Figure 4.8: Coefficient Memory Sub-System

The $\sigma_u$ coefficients are stored in a separate dedicated memory block and accessed in the same fashion as the complex exponential coefficients. The precise time the coefficients are selected depends on the state of the *Cos Sin State* and *BRAM Select* control lines, dictated by the state machine. In both cases, the data is re-interpreted (binary point shifted to ensure uniformity from all modules accessing the bus) before being placed on the shared bus.

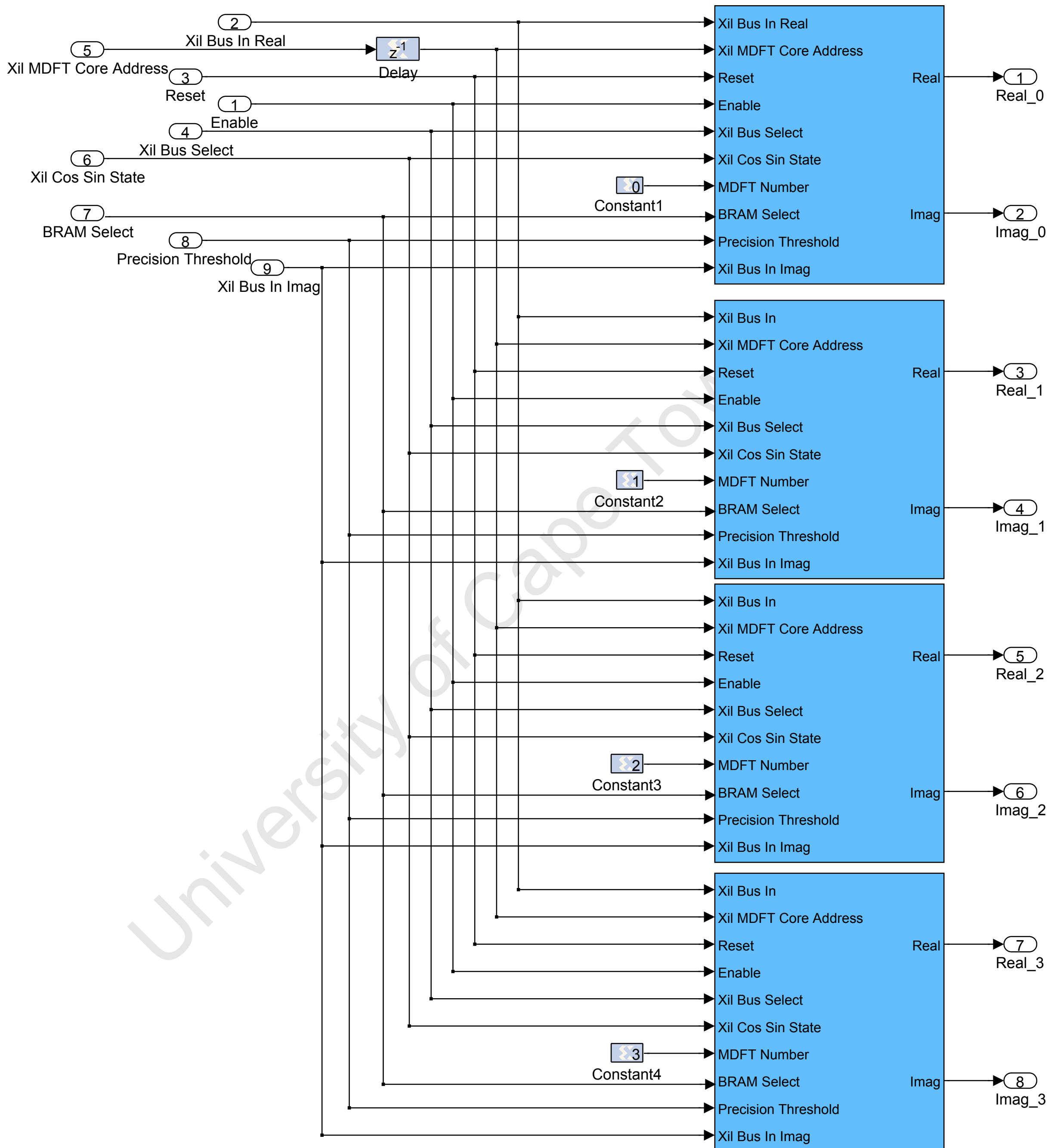
The coefficient memory system also implements an additional multiplexer and *Negate* block. If the reverse DFT is selected, it is necessary to multiply by the complex conjugate of the complex exponentials. This can be achieved by selecting the *imaginary* component in an unchanged state for the forward transform, or by inverting the 'sign' for the reverse transform.

### 4.6.5 Processing Elements

The hardware discussed up to this point is common to all processing elements and can be implemented once and shared. The remaining processing is custom to each of the DFT computations, and while the hardware is identical (with the exception of the first DFT point - this will be discussed shortly), the computational results are not.

The algorithm discussed in Section 2.8 points out the parallel nature of the recursive DFT, and this inherent parallelism is exploited in the hardware implementation. The processing elements are connected in an array and share common bus lines and clock source. Each processing element is given a unique identifier which is used during addressing of each element during coefficient transfer. Figure 4.9 represents four of the 64 processing elements connected in parallel for this system, and Figure 4.10 provides a high level block diagram of an individual element.

University of Cape Town



University of Cape Town

Figure 4.9: Processing Element Array. Only the first four elements are illustrated.

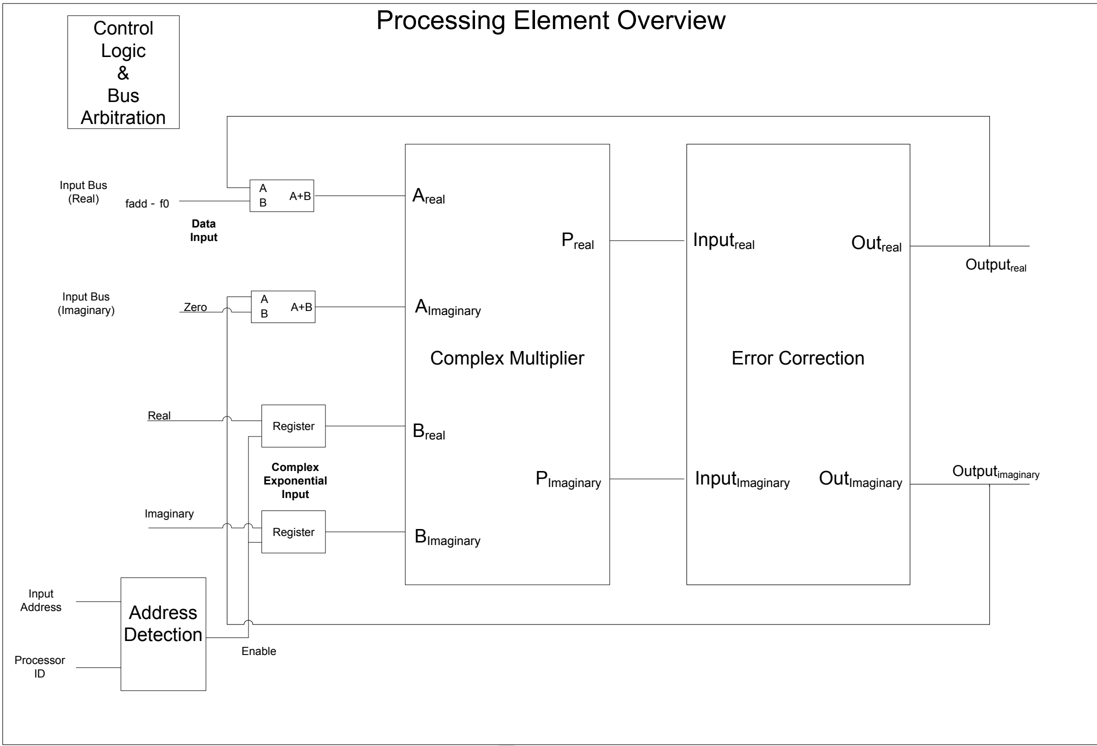



Figure 4.10: Processing Element Array

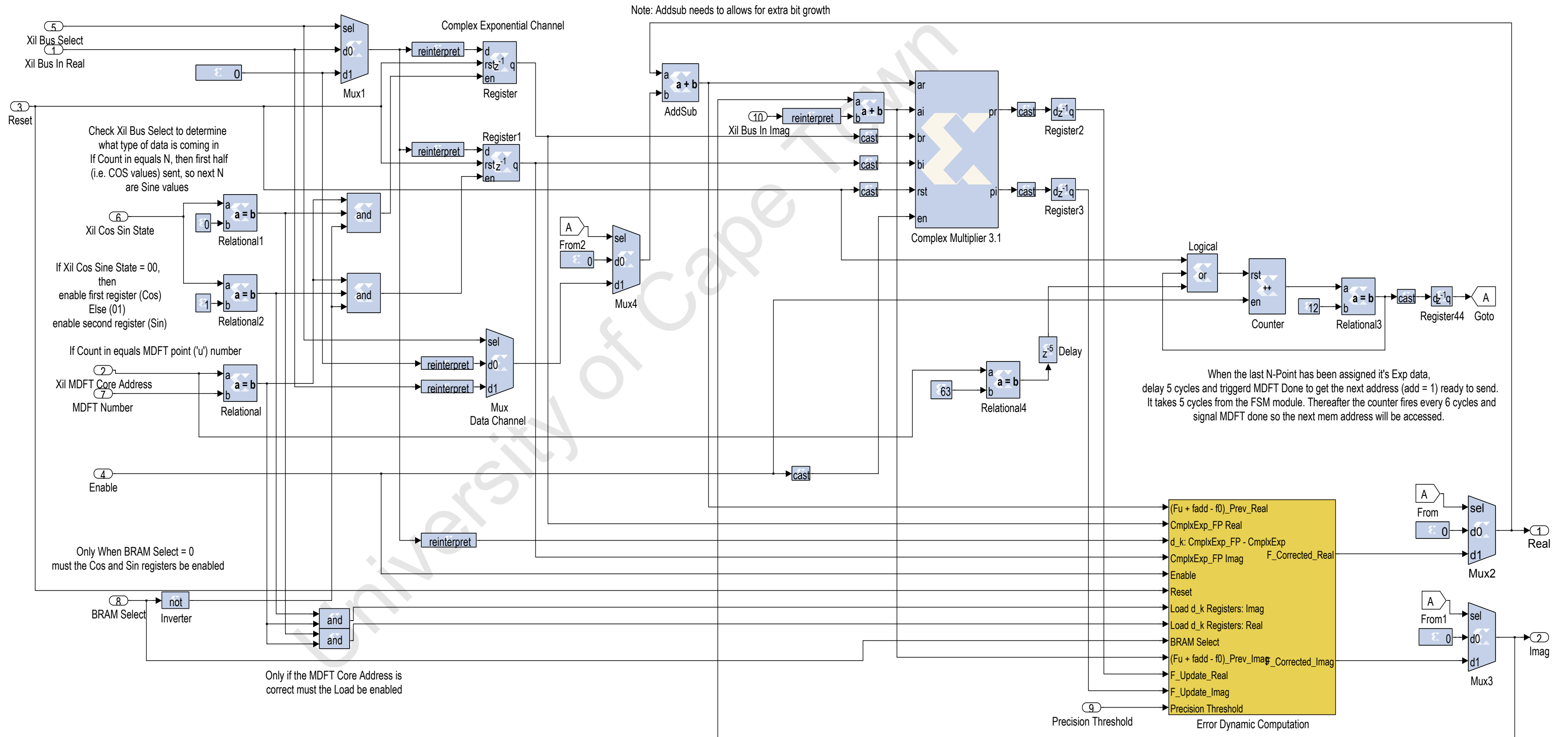
University of

#### 4.6.5.1 DFT Computation

During the coefficient transfer state, each processing element (Figure 4.11) is individually addressed as each has been assigned its own unique identifier. The initial stage in each processing element contains hardware used to identify when data has been addressed to it, as well as hardware to interpret the current state of transfer (i.e. real or imaginary data for both complex exponentials and $\sigma_u$).

The logic implemented determines which registers should be enabled based on the *Xil Bus Select; Xil Cos Sin State; and BRAM Select* control lines as well as the joint states of logic control used to determine if the processing element is currently being addressed. The incoming data is re-interpreted (binary point shifted to the correct position), and stored in dedicated registers.

University of Cape Town



University of Cape Town

Figure 4.11: Processing Element Architecture

After the completion of the coefficient transfer state, the system changes to the processing state, and control from the state machine enables all processing elements to begin computation. Section 4.6.3.1 discusses the common computations such as the difference between the incoming ($f_{in}$) and outgoing ($f_{out}$) samples and (selective) division by $N$ for all elements, and it should be noted that it is this partial computation in complex form which is transferred to each processing element during this computational state. The recursive algorithm (Equation 3.15) requires that the current output for a given DFT point $F_u$ is summed with the partial computation.

The next section discusses the error correction hardware; however it is worthwhile to point out one small difference which exists between the first DFT point and all those which follow. The processing for all DFT points occurs concurrently and synchronously. It is however necessary to signal the state machine when the computation has completed to enable the next sample to be shifted in. All processing elements require the same computational latency, so it is only necessary to signal the state machine from a single processing element. The element chosen in this implementation is the first DFT point.

When the processing elements are enabled (post coefficient transfer), a 4-bit counter is enabled and triggers after 12 cycles have elapsed. The total computational latency required per input sample is 13 cycles, and the final cycle is a latching delay in the storage register. Various conditions are also required to reset the system (and counter), and these sources are monitored and fed to a logical $OR$ gate fed to the reset line.

University of Cape Town

#### 4.6.5.2 Error Correction

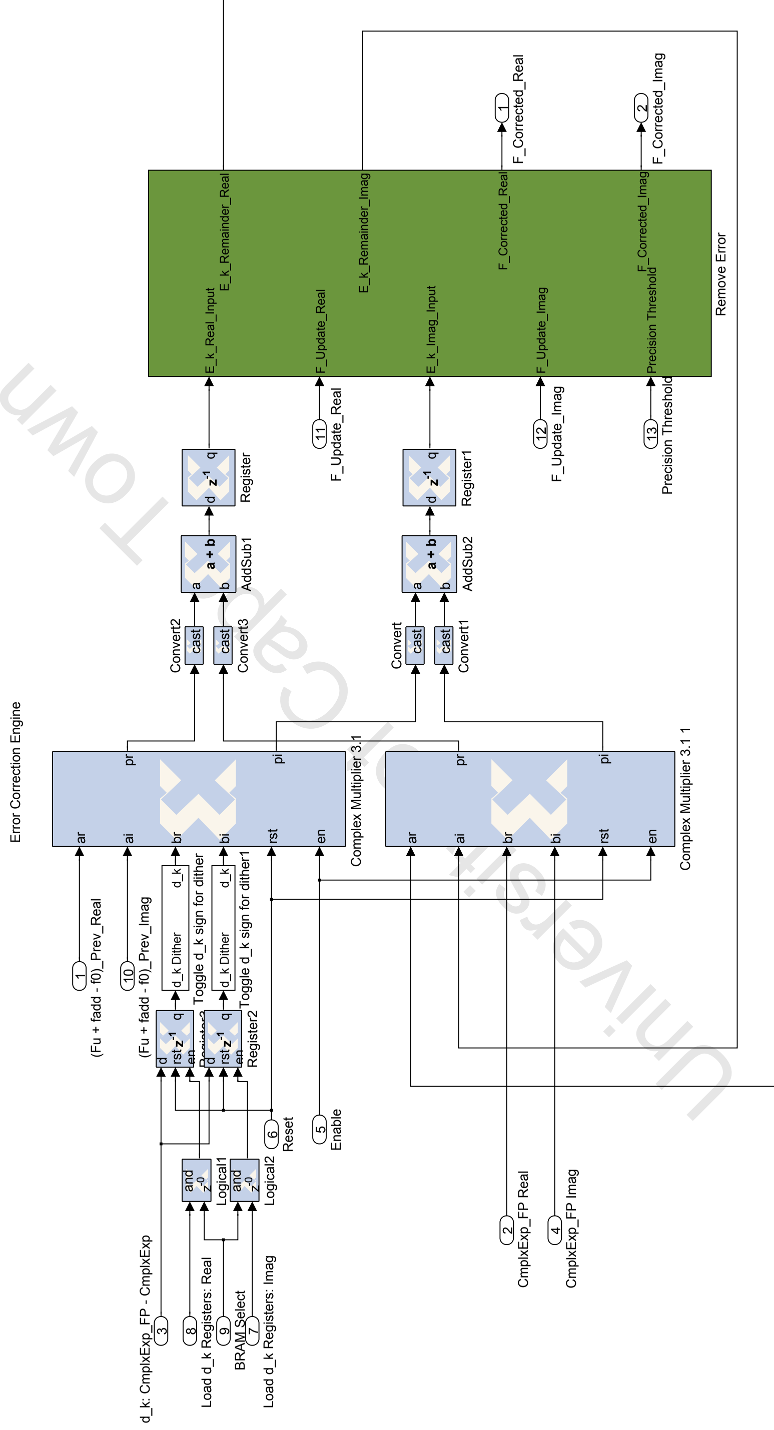
The error correction hardware implemented in this system executes concurrently with the computation of the DFT vector for the given processing element. It was pointed out in Section 3.7.2 that a threshold value needs to be applied to specify the appropriate error tolerance for the system, and that not all DFT points will perform correction at the same point in time (Section 3.7.2.1).

A threshold of $2^{-10}$ is applied in this system, and is common to all error correction engines. The error correction engine in this system (shown in Figure 4.12) contains registers to store the complex $\sigma_u$ coefficients, and includes hardware for coefficient dithering (Figure 4.13).

The dithering hardware plays an important role for any $\sigma_u$ coefficient that is too small to realise in hardware (with the finite bit range allocated), or where complex exponentials used are of unit value (where $\sigma_u = 0$)(Discussed in Section 3.7.2.1). In situations like this, the first term when computing $E_u[l]$ is nulled, which increases the error in the computation. To alleviate this problem, dithering noise is purposely stored for the $\sigma_u$ coefficient, and is set to be the smallest bit resolution realisable for the system ($2^{-36}$ in this case).

Injecting noise overcomes the term nulling problem in the computation of the error, $E_u[l]$, but creates a bias in results, depending on the signed magnitude of the injected noise. To remove the bias, the noise coefficient used for $\sigma_u$ undergoes arithmetic negation for every new sample computed (sign alternates every cycle). The magnitude of the injected noise ($2^{-36}$) does not change, only the sign, resulting in a statistical average of zero mean noise added when viewed over a wide time-span. The hardware used to achieve this dithering is shown in Figure 4.13.

University of Cape Town



University of Cape Town

Figure 4.12: Error Correction Engine Architecture

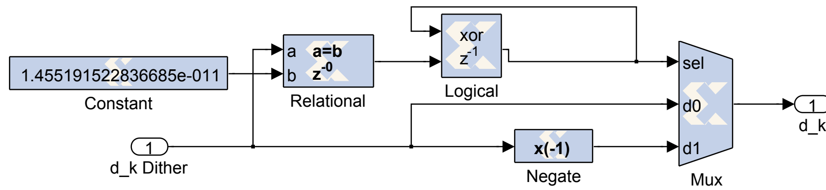



Figure 4.13: Dithering Hardware

To determine if dithering is required for the particular DFT point under computation, the $\sigma_u$ coefficient is compared to a fixed reference ($2^{-36} = 1.455191\text{x}10^{-11}$). The $\sigma_u$ coefficient would have been pre-loaded during the coefficient transfer state, and would have been replaced with a fixed value of $2^{-36}$ if required. Under the conditions that the $\sigma_u$ value matches the fixed reference, the output of the relational operator would be true, providing a logic state of '1' on the input of the exclusive-OR gate.

The exclusive-OR gate output is given an initial state of zero, resulting in the output changing state on the following clock cycle. The output of the exclusive-OR gate controls the select line of a multiplexer which selects between a negated (logic '1') or unchanged (logic '0') $\sigma_u$ value. This dithered result is provided to the input of the complex multiplier used to compute the error correction value $E_u[l]$. If the output of the relational logic is zero (i.e. no match), the output from the exclusive-OR remains constant and the multiplexer remains selecting the unchanged $\sigma_u$ value for the input to the complex multiplier. This hardware is duplicated across all processing elements, and can be used to detect any DFT point requiring dithering in the error computation.

University of Cape Town

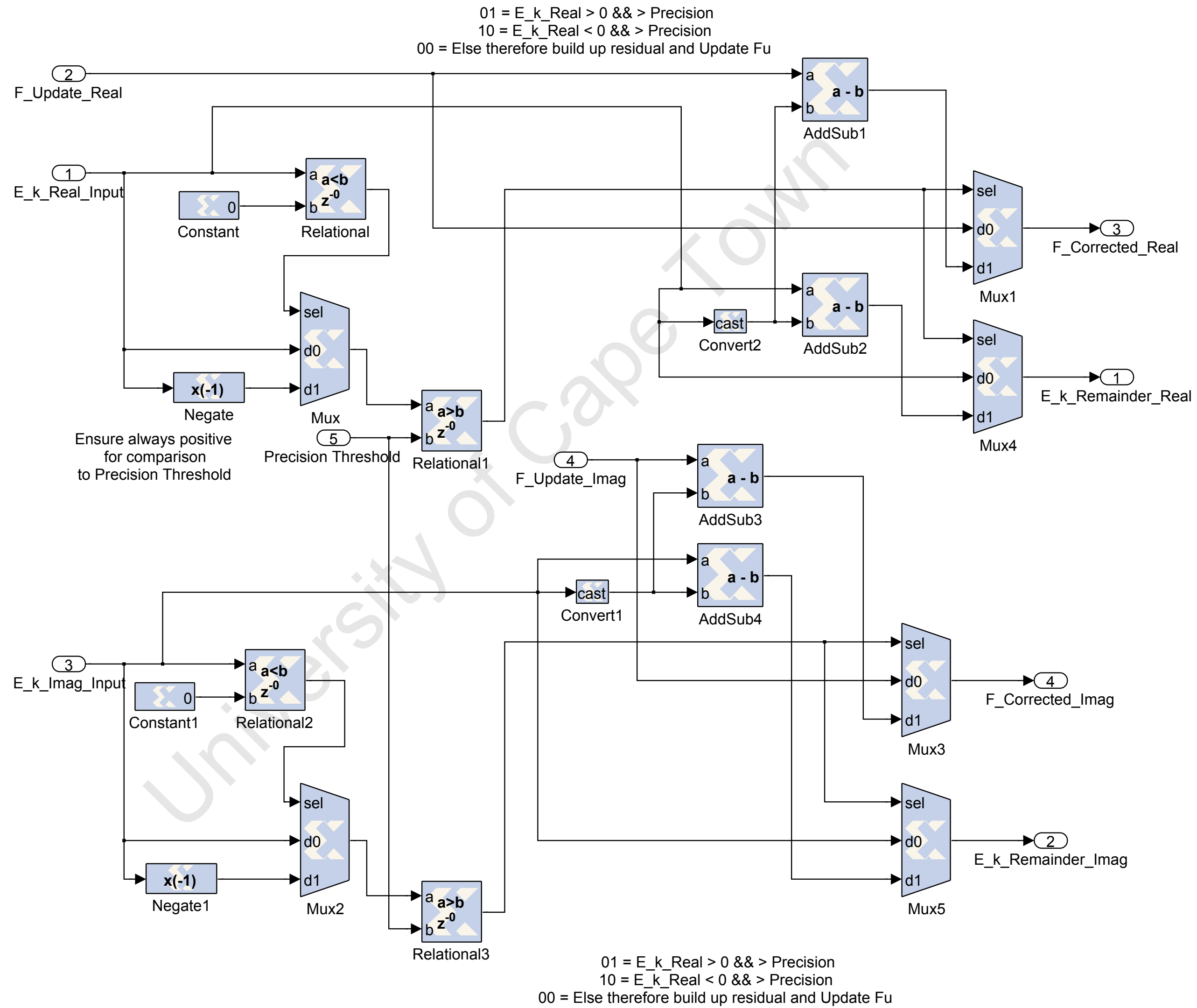
The error computation is discussed in detail in Section 3.7, and the final expression describing the computation is repeated for convenience (Equation 4.1).

$$E_u[l] = \sigma_u \left[ \hat{F}_u[l-1] + \frac{f_{in} - f_{out}}{N} \right] + W_N^u E_u[l-1] \tag{4.1}$$

All of the terms in Equation 4.1 are complex valued, and the two distinct complex multiplications can be computed concurrently to mask some of the processing latency.

The outputs of the two complex multipliers used to perform the computations are truncated to 36-bit word lengths, and later summed (*real* and *imaginary* results respectively). The resulting outputs are stored in data registers, and in turn feed the final computational block which corrects the current DFT vector if the computed error vector is larger that the preset threshold ($\upsilon$)(and computes the remainder value for $E_u[l]$ if necessary).

University of Cape Town




University of Cape Town

Figure 4.14: Error Correction Engine - Error Removal

The hardware used to perform the threshold detection and error removal is illustrated in Figure 4.14. The current computed (uncorrected) DFT vector is input to the sub-system in conjunction with the $E_u[l]$ results. The sign of the $E_u[l]$ value is first determined and negated if necessary(to ensure a magnitude only comparison to the threshold $\upsilon$).

If $E_u[l]$ is less than $\upsilon$, $F_u$ and $E_u[l]$ are fed directly out via a multiplexer (select line controlled by the relational result of the threshold comparison). If the threshold is exceeded, the select line of the multiplexer selects the input which is connected to a subtracter which produces the corrected $F_u$ vector ($E_u[l]$ first is cast to a 36-bit word with arithmetic precision of 24-bits).

A second multiplexer is used to select the difference between $E_u[l]$ and the 24-bit precision quantized version $E_u[l]^\star$. This correction computation is used for both *real* and *imaginary* terms, and the final $E_u[l]$ output is returned to the input of the complex multiplier from the previous stage, and the corrected DFT vector is returned to the summing input stage for the DFT computation.

The hardware illustrated in Figure 4.14 performs $E_u[l] - E_u[l]^\star$ and DFT vector corrections irrespective of the actual value of $E_u[l]$, however due to the use of multiplexers and logic to determine the current $E_u[l]$ state with respect to $\upsilon$, the adjusted values for $E_u[l]$ and $F_u$ are ignored if $\upsilon$ is not exceeded. This is intentionally implemented to help standardize computational time across all processing elements, as not all elements will enable correction synchronously.

University of Cape Town

### 4.6.6 Data Handling

The system has been designed to be expandable and allows additional processing elements to be added with very simple changes needed to the addressing range of the system controller. As more processing elements are invoked however, a greater demand is placed on the system controller to handle output data as it becomes available. Each processing element incurs the same processing latency from valid input data to output, and since the system operates synchronously, all processing is completed in lock-step, and the outputs are available simultaneously.

While this achieves the desired outcome for wide parallelism, a bottleneck is created as the system now has to deal with all the parallel data that has simultaneously become valid. It is important to handle this data as quickly as possible minimising the handling time, so the system can begin processing again. If the data generated is to be transfered directly out of the processing system, the best means to transfer data would be to implement the multi-gigabit transceivers available in most FPGA families. The data overhead is dependent on the number of processing elements; the word lengths used in processing; and the clock rate used by the system.

In this prototype system, a full 64-point system with complex outputs is tested with a clock rate of 82.2MHz. The data throughput in bits per second can be computed as follows:

$$Data\ Throughput = (\frac{Clock\ Rate}{Processing\ Latency}) \times DFT_{Total} \times 2 \times Wordlength \tag{4.2}$$

where:

*Clock Rate* is the clock frequency applied to the system

*Processing Latency* is the computational cycles per output for a new input

$DFT_{Total}$ is the total DFT points computed



University of Cape Town

Multiplication by 2 for complex output

*Wordlength* is the number of bits per output word

Discussion of the clock frequency and processing latency will be handled in Section 5.3.2, but can be stated for purposes of the computing *Data Throughput* computation. Post place-and-route results indicate a synthesizable operating clock frequency of 82.2MHz with a processing latency of 13 clock cycles per output. The total number of DFT points in this prototype system is 64 with a computed word length of 36-bits per sample. Applying these values to Equation 4.2 gives:

$$\begin{aligned} Data\ Throughput &= \left(\frac{82.2 \times 10^6}{13}\right) \times\ 64\ \times\ 2\ \times\ 36 \\ &= 29.136\ Gb/s \end{aligned} \tag{4.3}$$

University of Cape Town

Alternatively, this can be stated in samples per second:

$$\begin{aligned} Data\ Throughput &= \left(\frac{82.2 \times 10^6}{13}\right)\ \times\ 64\ \times\ 2 \\ &= 809\ MSPS \end{aligned} \tag{4.4}$$

Handling data at the full rate would require the use of at least five 6.5Gb/s transceivers. A later extension of this framework suggests this data be used for additional processing in Fourier space. This would introduce additional processing latency which in turn would lower the required data throughput rate for a fixed DFT width. To maximise throughput, the handling time should not exceed the processing latency, or the system would require halting to allow output data handling, before accepting new input data for processing.

Output data handling has not been a core focus in study, as newer generation FPGAs are consistently produced offering improved DSP slice resources as well as transceiver operation. This study utilises Virtex 5 FPGA hardware (discussed in Section 4.7) offering 6.5 Gb/s transceivers, where newer FPGAs on the market offer transceivers exceeding 28 Gb/s [89]. Adopting hardware applicable to current design specifications will allow correct data handling.

University of Cape Town

## 4.7 Hardware and Software

### 4.7.1 Hardware

The initial platform targeted for testing the system design discussed in this work was a Field Programmable Gate Array (FPGA). In theory, hardware testing could have been replaced with a system simulation, however it was the experience of the author (in the early stages of design) that it is possible to get results which differ from simulation and actual implementation. For example, on one occasion a design simulated correctly post HDL synthesis, however during the place and route process some hardware was removed and rendered the final results inaccurate.

In another separate instance, a simulation provided correct results, however the final built implementation difference in latency by one clock cycle. This in turn negatively affected the internal timing, and caused the system to become unstable and errornous after a handful of iterations. It is for this reason it was decided to test and verify a fully compiled and built system with a system simulation. This type of testing best reveals errors of this kind.

The FPGA hardware available during testing was two Xilinx Virtex 5 FPGAs (VSX50T initially, and later a VSX95T). The initial system development was performed using the Virtex 5 VSX50T device on a Xilinx ML506 development board. At the time of the design, this was the largest DSP based FPGA available for use, and supported co-hardware simulation using Matlab Simulink. The ML506 development system was used extensively for system design verification and debugging, and all arithmetic accuracy results were obtained using this platform.

Later on in the study a larger DSP based FPGA (VSX95T) became available (on a Reconfigurable Open Architecture Computing Hardware (ROACH) development system [5]). The larger FPGA was later used for implementing a larger design, and performing timing analysis of the system operating on the hardware.

University of Cape Town

The limited hardware resources available on either FPGAs did not permit a full system place-and-route, and the design was verified using a combination of software emulation and co-hardware simulation (discussed in Section 4.7.2).

The FPGAs were selected as they include DSP blocks within the fabric space allowing the design to harness the computational power available. The design was kept as generic as possible, however did utilize custom IP from Xilinx (such as complex multipliers) to ensure performance. These modules can be replaced with other vendor IP offering the same features if the design was to be ported with minimal effort for use on another manufacturer's FPGA. The two FPGAs used in this study will now be discussed, followed by a brief overview of the two development systems.

#### 4.7.1.1 FPGA Features

It will be useful at this point to assess the differences between the two devices used for testing, and Table 4.2 details some of the key features of the two devices. In this study the most important resource under consideration is the DSP slices which are available per FPGA. Resource utilisation will be discussed in detail in section 5.3.1.

Table 4.2: Xilinx VSX50T Features [7]

| *Resource* | $VSX50T$ | $VSX95T$ |
|---|---|---|
| Slices | 8160 | 14720 |
| Block RAM (Max) | 4752 Kb | 8784 Kb |
| DSP48E Slices | 288 | 640 |
| I/O | 480 | 640 |

University of Cape Town

#### 4.7.1.2 Xilinx ML506 Development Kit

The first FPGA considered for use in this study was the Xilinx ML506 FPGA development board (see Figure 4.15). This development system offers the hardware features listed in Table 4.3. The hardware design in this work considers only internal memory to the FPGA, and uses the JTAG interface for debugging and verification purposes.

Table 4.3: Xilinx ML506 Hardware Features [4]

| *Hardware* | *Specification* |
|---|---|
| FPGA | XC5VSX50T |
| RAM | DDR2 SODIMM (256 MB) |
| Programming Interface | JTAG Programming Interface |
| Ethernet | 10/100/1000 Ethernet |

University of Cape Town

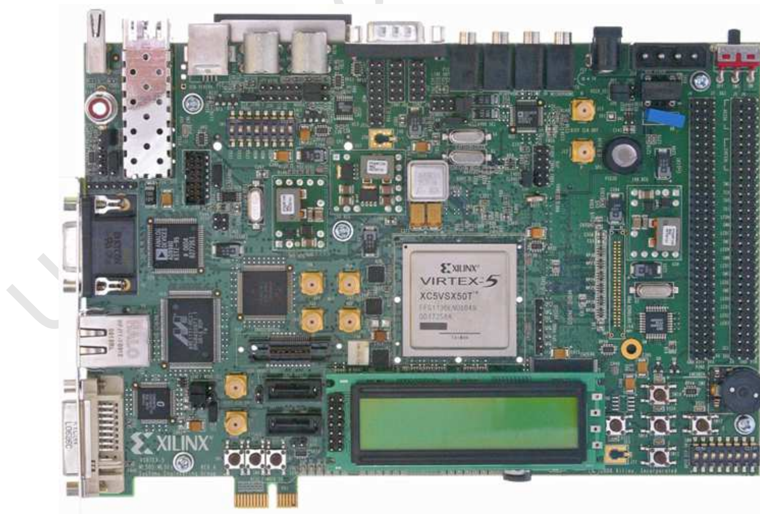



Figure 4.15: Xilinx ML506 Development Board [4]

#### 4.7.1.3 ROACH Development System

Table 4.4: ROACH Hardware Features [5]

| *Hardware* | *Specification* |
| --- | --- |
| FPGA | XC5VSX95T |
| RAM | DDR2 DRAM |
| Onboard Processor | PowerPC 440 Core |
| Programming Interface | JTAG Programming Interface |
| Ethernet | 10/100/1000(PPC), 10 Gigabit Ethernet(FPGA) |

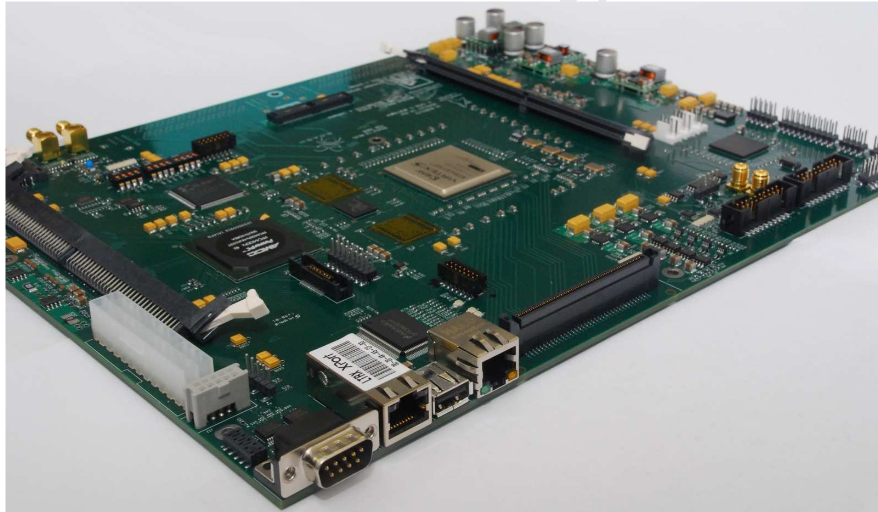


Figure 4.16: ROACH Development Board [5]

The ROACH board was a late inclusion into the project, and was used as the onboard FPGA offered more DSP resources, which therefore allowed a larger design to be mapped to it. The ROACH development system contains a PowerPC 440 core which runs an extended version of the Linux kernel. This front-end handles FPGAs similar to that of CPUs, and allows easy access to the FPGA through the PPC processor. This in turn made it possible to access data registers

within the system running on the FPGA, and was used to verify the processing latency of the system.

### 4.7.2 Software

The software environment for this system consisted of two distinct, but collaborative design environments, each with additional supporting design tools. Matlab and Simulink were the primary design tools utilised in conjunction with the Xilinx ISE toolchain. Matlab Simulink and the Xilinx toolchain link together to allows FPGA gateware development and simulation, where Simulink uses the Xilinx tool set for synthesis and place-and-route. Simulink provided the front-end building interface for the system, and gives access to Xilinx IP and generic primitives.

The combination of these two products also permits an additional means of hardware testing through co-hardware simulation. Co-hardware simulation allows the design built in Simulink to be deployed and tested on FPGA fabric, and transfers results to the Matlab workspace for further use through a JTAG-USB interface. The operational rate is significantly reduced (if running in synchronous mode), however allows for easy access, analysis, and verification of results within Matlab. This was a particularly useful feature as all system and algorithm modelling had been previously performed in Matlab (allowing for comparison of simulated and real world results).

The only module not developed within the Simulink framework was the finite state machine. The Xilinx toolchain (ISE 11.4) included AccelDSP which provides a means to convert Matlab code into either of the two Hardware Descriptive Languages(HDL)(Verilog or VHDL) or, into a Simulink System block for import. The finite state machine used in this system was first written and verified using Matlab, and later exported using AccelDSP into a Simulink system block for importing into the final design.

University of Cape Town

Table 4.5: Software

| *Software* | *Version* |
|---|---|
| Matlab and Simulink | R2009a |
| Xilinx ISE | 11.4 |
| AccelDSP | 11.4 |
| CASPER Toolchain | BWRC Snapshot(July 2011) |
| ChipScope | 11.4 |

An advantage of the Simulink HDL environment is that it allows the use of custom libraries developed for a broad range of applications. A radio astronomy research group (Collaboration for Astronomy Signal Processing and Electronics Research - CASPER) have developed 'gateware' libraries to enable radio astronomers to quickly design and deploy new radio astronomy instruments. These libraries are open-source (freely available online [90]), and seamlessly integrate into the Simulink HDL design environment.

This study used the CASPER toolchain in the final stages of development to enable the design to be built and tested on the ROACH hardware described in 4.7.1.3. The CASPER libraries enabled the access of data stored in data registers within the FPGA to be accessible by the PowerPC core on the ROACH platform. A list of the software packages (and versions) used in this study are detailed in Table 4.5.

University of Cape Town

## 4.8 Chapter Summary

This chapter discussed the processing framework developed for spectral method based computations. The work discussed in Chapter 3 describes the Recursive Discrete Fourier Transform (RDFT) in detail and revealed that the algorithm used to realise the DFT computation is inherently parallel, and lends itself for practical implementation in a parallel VLSI system. The aim was to develop a generic spectral processing framework which can adequately support the RDFT, however, be flexible enough to accommodate the additional operators specific to spectral method computations.

To minimise hardware redundancy, a parallel framework was presented which utilises shared resources such as the memory system and bus arbiters, as well as arithmetic resources such as subtraction and division. The hardware described is capable of handling data sourced both internally to the system (stored in memory), as well as externally (streamed input e.g. ADC), and the system is controlled by finite state machine controller which provides the control signals to the various resources as required. This chapter provided details on the state machine and described the functionality of the various states.

The core aim of this work is to develop a parallel framework to support spectral based processing, and this chapter provides a gate-level description of the hardware required to support the computations discussed in Chapter 3. The bulk of the processing is performed by multiple processing elements which operate synchronously, and each processing element computes the Fourier transform recursively, and includes error correction hardware capable of determining and correcting the output automatically on-the-fly. The volume of data produced from this system is shown of be in excess of 29 Gb/s (64 point complex DFT), and it is suggested that multi-gigabit transceivers are employed for data transfer.

University of Cape Town

Finally, this chapter provides an overview of the two hardware test platforms available, and development software used throughout this study. The two hardware platforms were the Xilinx ML506 development system, and a ROACH development system, both providing a different FPGA for system development testing. The prototype system was built and gateware produced for both Xilinx Virtex 5 (VSX50T and VSX95T) FPGA devices, and both successfully executed the prototype system. The chapter which follows concentrates on the analysis of results obtained for both devices, and considers system accuracy, power utilisation, and benchmarking of the proposed system.

University of Cape Town

# Chapter 5

# Analysis of Results

University of Cape Town

## 5.1 Introduction

This chapter discusses and analyses the results obtained from the hardware testing in Chapter 4. Consideration is first given to the resource utilization within the FPGA platforms, as well as the operating clock frequency attainable and processing latencies incurred. Power analysis, system accuracy as well as additional front-end data streaming support is also discussed. Finally the system is benchmarked and compared against three alternative FFT arrangements: Multi-core CPU, GPUs, and an alternative FPGA implementation. The chapter concludes with a discussion on FPGA-to-ASIC design conversion.

## 5.2 Contributions

- Presents a performance comparison of the recursive DFT with respect to an FFT FPGA implementation, as well as an FFT on multicore CPU and GPU hardware
- Presents the resource utilisation for two different FPGA devices.
- Discusses system accuracy, clock frequency and processing latency, and provides a power analysis.

- Provides the specifications for additional external hardware for an analog-to-digital front end.

- Discusses FPGA to ASIC conversion

## 5.3 Hardware Implementation Results

### 5.3.1 Resource Utilisation

The focus of Chapter 4 has been on the hardware implementation of the system described in Chapter 3. As is expected of most hardware systems, a trade off exists between system performance and the hardware required to realise the system. The two hardware platforms available for use in this study have been detailed in Tables 4.3 and 4.4 in Chapter 4, and due to resource constraints, the actual design size mapped to FPGA resources had to be reduced to 4 DFT points. The resources used for the two platforms (Table 5.1) are similar for *BRAM* and *Slice Registers* and *Slice LUTs*, however differs significantly for *DSP48Es* usage.

The larger of the two FPGAs available (XC5VSX95T) offers up to 640 *DSP48Es* slices, where the XC5VSX50T FPGA can only offer 240. Mapping limitations became apparent for bonded IOBs, as the number available on either package was insufficient for this design. In practice, the outputs of each processing element would be handled by high-speed transceivers (not a requirement for this work), and would therefore not require a full set of I/O ports. Doubling the DFT points from 4 to 8 would be possible for XC5VSX95T device if *IOBs* were not an issue. To map a full design a next generation FPGA such as a Virtex 6 or Virtex 7 from Xilinx would be required for a full design placement.

University of Cape Town

### 5.3.2 Clock Frequency and Processing Latency

The post place-and-route timing report generated for the XC5VSX95T FPGA indicates a maximum operating frequency of 82.2MHz (or 12.162ns per cycle), and a maximum path delay between any node of 3.064ns. The number of cycles required per computation was originally assessed in Simulink (hardware simulation) to be 13 cycles per update. To verify this in hardware, it was necessary to capture the state transition of the flag used to signal the state machine that the processing elements have completed the computation for the given input sample (discussed in 4.6.5.1).

The ROACH hardware platform offers the facility of accessing data registers in the FPGA through an SSH connection to the Linux kernel running on the Power PC core. To enable this feature, it is necessary to include some hardware blocks from the CASPER toolchain which allow the data registers to be accessible by the Power PC core.

The cycle count per update was computed using a clocked and latched counter to capture the current count state. Accessing this register post computation would reveal a cycle count of 13 cycles (identical to Simulink hardware simulation), however if the register were accessed during the processing operation, a smaller value would be reported. To overcome this issue, the system latches the output cycle count after one full computed update, and is later disabled for following updates.

University of Cape Town

Table 5.1: Resource Utilization

| *Resource* | $Utilization(\%)$ $XC5VSX50T(4ptDFT)$ | $Utilization(\%)$ $XC5VSX95T(4ptDFT)$ |
|---|---|---|
| Block RAM/FIFO | 6% | 4% |
| Slice Registers | 35% | 20% |
| Slice LUTs | 25% | 14% |
| DSP48Es | 83% | 37% |
| Bonded IOBs | 85% | 64% |

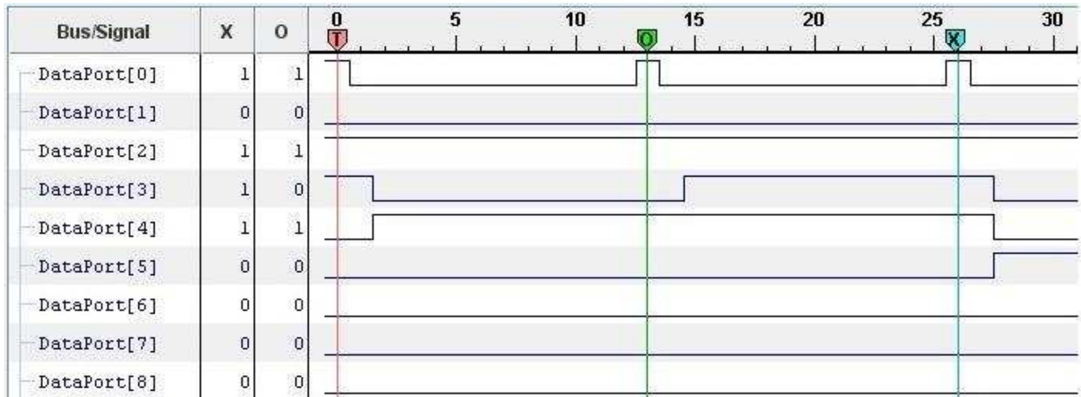



Figure 5.1: Processing Latency - ChipScope Analyser

University of Cape Town

A double verification of results was also performed through the implementation of the ChipScope analysis system. A ChipScope probe was attached to the status line from the processing element and during operation captured and buffered. The results were then read out at a slower rate by the JTAG interface and interpreted using the ChipScope analysis software. The results captured from the ChipScope analyser are shown in Figure 5.1.

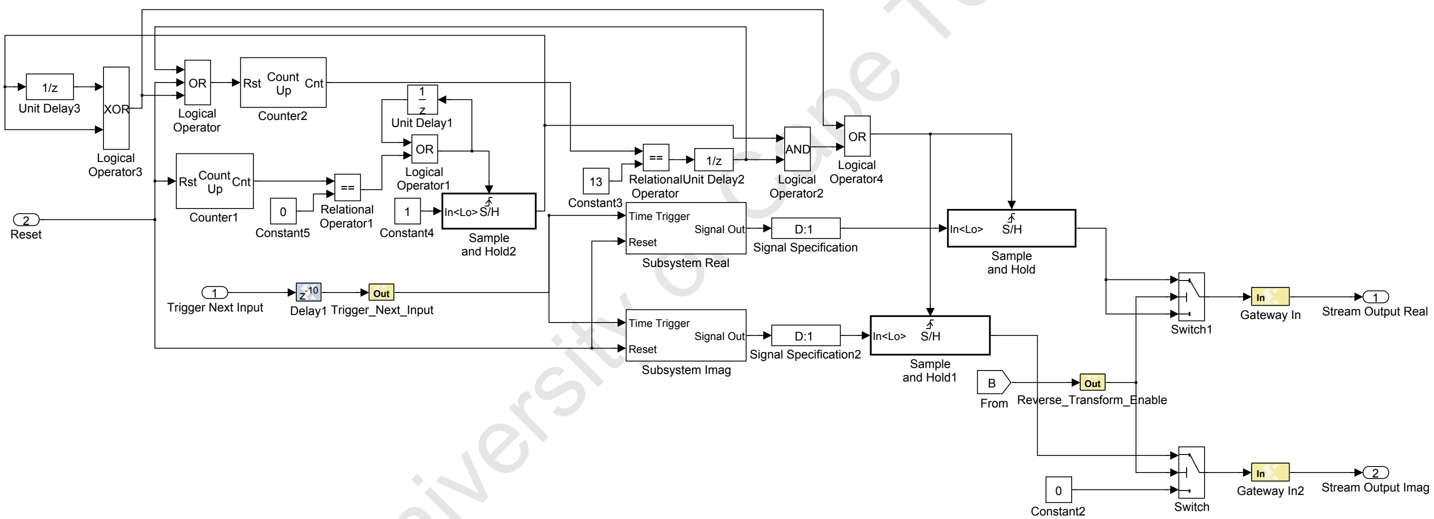
The top trace (*DataPort [0]*) represents the activity of the status line which feeds back into the state machine of the main system. The other traces are a partial view of the BRAM address lines used to access pre-loaded data for computation. Returning to the status line represented by the top trace, the pulsating state change is periodic lasting 13 cycles per pulse. The three markers *'T'*,*'O'* and *'X'* indicate two occurrences (with a time period of 13 cycles between between *'T'* and *'O'*, and *'O'* and *'X'*).

### 5.3.3 System Accuracy

The system specifications listed in Table 4.1 specify a word width of 36-bits with an arithmetic precision of 24-bits. To ensure consistency in computations as well as to limit word growth (multiplication and addition stages), the quantization for all hardware blocks is set to unbiased rounding. For accuracy of system model simulations when compared to hardware results, the software simulations described in Chapter 3 perform the same rounding technique. The accuracy of the output obtained from the final hardware implementation is a function of the word length (discussed in Section 3.7.2.1), and can be adjusted for the final hardware implementation.

To assess the hardware implementation system accuracy with respect to the simulated model, the same chirp signal dataset was used for input to the simulation system as well as the hardware system. The simulation model for the recursive DFT was written in Matlab and utilised a vector array of generated data for testing. The hardware simulation (Simulink) and hardware FPGA testing (Simulink Co-Hardware) required an additional supporting system front-end to stream the dataset sequentially and synchronously with the toggling of the status line from the processing element 0. This front-end is shown in Figure 5.2.

University of Cape Town




University of Cape Town

Figure 5.2: Supporting Data Streaming Front-End

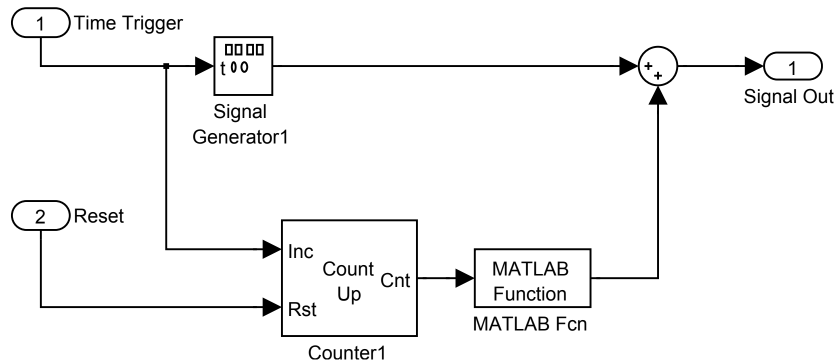



Figure 5.3: Supporting Data Streaming Front-End

The streaming front-end shown in Figure 5.2 consists of two sections: The reset and trigger control and the data access section (Figure 5.3). The reset control accepts a main system reset line for the resetting of the entire system (and subsystems), and also contains a logical subsystem to reset the counter labeled *Counter 2*.

The processing latency has already been characterized and is known to require 13 cycles per sample to produce a valid output. This a priori knowledge is used to create a timing system consisting of a counter and comparison logic. When this counter reaches a count state of 13, and the initial coefficient transfers have completed (a second counter running for a few hundred cycles before rolling over to zero to enable the logical *AND* accounts for this time), the *sample-and-hold* modules are permitted to trigger to hold the current input on the output port.

The input data supplied to the *sample-and-hold* module is fed from another subsystem triggered by the status complete line coming from the processing elements. This subsystem (*Subsystem Real* and *Subsystem Imag* in Figure 5.2) runs an embedded Matlab module (Figure 5.3) which executes a script that iterates through the input dataset (saved to disk) and returns the next successive sample.

University of Cape Town

The subsystem shown in Figure 5.3 includes a counter which increments by one for every trigger event that enters the system from the processing elements. The output of the counter is used an an indexing point for the Matlab function to iterate through the data vector array stored on disk.

To allow the option to induce white noise into the system, a signal generator is included, and the sum of both data sources is presented at the output. In the system one level above (Figure 5.2), two modules are included to permit a complex dataset to be used in testing. The outputs from the sample-and-hold modules are connected to a set of multiplexing switches, which in turn supply the input stage of the system running on the FPGA (Co-Hardware Simulation), or Simulink model (if a simulation).

The results obtained from the various models underwent a sample-for-sample comparison to assess accuracy of results for both the forward and reverse Fourier transform implementations. The sample-for-sample comparison of the Simulink hardware model and the FPGA implementation revealed that no differences existed between the two sets of results.

Furthermore, these results were compared against the Matlab model (using the same input dataset) which consisted of both a recursive DFT implementation, as well as verification with respect to an FFT restricted to the same arithmetic word lengths in processing. The results of the Matlab model, FFT and FPGA implementation were all identical.

University of Cape Town

Table 5.2: Power Analysis (XC5VSX95T)

| $Resource$ | $Power(W) : 4pt$ | $Projected\ Power(W) : 64pt$ |
|---|---|---|
| Clocks | 0.23801W | 3.80816W |
| Logic | 0.00098W | 0.01568W |
| Signals | 0.00433W | 0.06928W |
| I/O | 0.15990W | 2.5584W |
| BRAMs | 0.01880W | - |
| DSP48s | 0.00482W | 0.07712W |
| Total Quiescent Power | 1.282W | - |
| Total Dynamic Power | 0.42683W | 6.82928W |
| Total Power | 1.70883W | - |

### 5.3.4 Power Analysis

An important aspect to consider when assessing system performance is the power utilization characteristic of the system. The analysis provides a detailed breakdown (Table 5.2) for each set of resources within the FPGA (both dynamic and quiescent power), and is provided for a fully synthesized and routed 4 point RDFT design on a Xilinx Virtex 5 XC5VSX95T FPGA. The quiescent power usage for all resources is the largest portion (81.2%) of the overall power budget for a 4 point design, with dynamic power covering the remaining 18.7%.

A projection for a 64 point is not a linear scaling for all resources, as not every resource would be required for a scaled design. Resources such as BRAM's are not increased for an increase in parallelism, and therefore cannot be included in the projection. Scaling the 4 point design results linearly (applicable resources) indicates that the single largest consumer of dynamic power is the clock resources, followed by signal routing resources. The clock resources are required to support a large fan-out, and therefore utilize many buffer stages - all requiring power. Signal routing (especially among many processing elements) is the second largest consumer, followed by the DSP slices.

University of Cape Town

## 5.4 System Benchmarking

To characterize system performance, it is necessary to compare the system with respect to other Fourier transform implementations. Direct comparison is not possible given the framework and design feature differences, however a performance metric in a gross sense can be achieved by considering FFT implementations for various platforms.

The platforms under consideration are: FFT FPGA implementations, FFT GPU implementations, and FFT CPU implementations. As a core computation for a spectral based system, the FFT is not directly suitable for use as a general purpose Fourier transform computation algorithm, as it does not offer the necessary support for sample-by-sample updating, nor re-usable hardware for the continued use of other Fourier operators.

To compute sample-by-sample updating, entire vector sets are required to be loaded after every sample update, even though only a single sample input has changed. The recursive DFT by comparison, requires only the difference between the incoming and outgoing samples. The characteristics of each considered algorithm in this study was covered in detail in Chapter 2, and most will not be considered in a benchmarking comparison due to the lack of widespread implementations. The FFT offers a wide set of implementation results, and while it does not offer all the desired feature for the framework discussed in this study, it does offer a useful point of comparison.

University of Cape Town

### 5.4.1 FPGA

The first benchmark comparison is that of a dedicated FPGA implementation. The comparisons assess both the Xilinx and Altera FFT IP, and draw comparisons with respect to processing latency, loading/offloading time, maximum operating frequency, maximum simultaneous outputs, and word length. A comparison is also given for both block based processing as well as sample-by-sample updating.

#### 5.4.1.1 Block Based Processing

The FFT operates on a block based processing principle where a vector array of samples are acquired and sequentially streamed into the FFT hardware. The loading and offloading stages require $N$ cycles, where $N$ represents both the input vector length and the DFT length to be computed. The recursive DFT requires a similar initial loading. The output will be updated after every sample input to the system, except the output will only be valid and match the result from the FFT after the first $N$ iterations.

Two sets of comparisons are made for FPGA FFT implementations. The first draws a comparison for an equal length recursive DFT and FFT as both FFT length are 64 ($N = 64$). The recursive DFT (RDFT) used a data length of 36-bits, and the FFT 18-bits. The 64 point FFT was generated withing the Xilinx core generator for the same Virtex 5 FPGA used in this study. Table 5.3 details the results for this implementation.

University of Cape Town

Table 5.3: FFT Results: 64 Point DFT

| $DFTType$ | $Device$ | $Data\ Width(bits)$ | $f_{Max}(MHz)$ | $Time(\mu S)$ |
|---|---|---|---|---|
| FFT | Virtex 5 | 18 | 458 | 0.6 |
| RDFT | Virtex 5 | 36 | 82.2 | 10.12 |

There are a few interesting entries which stand out in the results for block processing. Firstly, the transform computational time for the recursive DFT is more than an order of magnitude longer for the same transform length. This

results primarily as the updated transform is computed for each sample (where as the FFT computes for the full input vector). The recursive DFT requires 13 cycles per computation, however could be reduced to 7 cycles if no error correction is required with 18-bit arithmetic [91](It should be noted that initial work of this study produced a prototype system that did not implement error correction, and used 18-bit data lengths. The same fundamental algorithm and framework was used).

The operating rate of the recursive DFT is 5.5x slower (computing using 36-bit data lengths). The data length influences the number of cycles required by the DSP slices when computing, and any computational length greater than 18-bits increases the number of required cycles [92]. Similar performances are obtained for larger implementations (256,1024,4096) on alternative FPGA devices [93, 94], however the maximum frequency varies between 231MHz and 442MHz.

Overall, a dedicated FFT implementation tailored for the underlying FPGA resources performs significantly better by comparison to the the recursive DFT. The comparison is not so straightforward, as it is required to take into account additional design architecture features such as additional operator support, custom Fourier coefficients, as well as sample-by-sample updating. If FPGA resources were not an issue, the recursive DFT could implement any $N - wide$ DFT incurring no more than $N$x13 cycles to fully compute the transform. The results shown in [93, 94] indicate that the transform time increases approximately linearly with the transform size. It is clear however, that the strength of the recursive DFT does not necessarily lie in the direct computation of the DFT for any length $N$, and is therefore not a practical approach for such dedicated applications.

#### 5.4.1.2 Sample-by-Sample Updating

A different picture emerges when assessing performance for sample-by-sample updating (see Table 5.4). For each sample input, the recursive DFT requires only 13 cycles, where as the FFT implementation requires 275 cycles. The FFT uses 211 cycles for initial loading and transformation, and a further 64 cycles

University of Cape Town

to export the resulting output. In this operational case, the recursive DFT is 3.8x faster per output update over the FFT implementation which is operating on a clock 5.5x faster. In effect, operationally, the recursive DFT is 20.9x faster than the equivalent length FFT (without taking into account that the recursive DFT computes using a longer bit length and implements error correction). It is clear that in this situation, the recursive DFT is the algorithm and FPGA implementation of choice.

Table 5.4: FFT Results: 64 Point DFT

| $DFTType$ | $Device$ | $Data\ Width(bits)$ | $f_{Max}(MHz)$ | $Time(\mu S)$ |
|---|---|---|---|---|
| FFT | Virtex 5 | 18 | 458 | 0.6 |
| RDFT | Virtex 5 | 36 | 82.2 | 0.158 |

### 5.4.2 Multicore CPU

Further benchmark comparisons can be made against commodity hardware such as high-end CPUs and GPUs. In [6], the authors use a high-end Windows PC equipped with an Intel QX9650 3.0GHz quad-core processor and 4GB of DDR3 1600 RAM for experimentation. The processor hardware consists of two dual-core dies in one package with each pair of cores sharing a 6 MB L2 cache. The FFT is implemented using Intels Math Kernel Library (MKL) version 10.2 on the CPU, and the MKL tests utilized four hardware threads and used out-of-place, single precision transforms. The results for a 1D power-of-two FFT (averaged over multiple runs) is illustrated in Figure 5.4

University of Cape Town

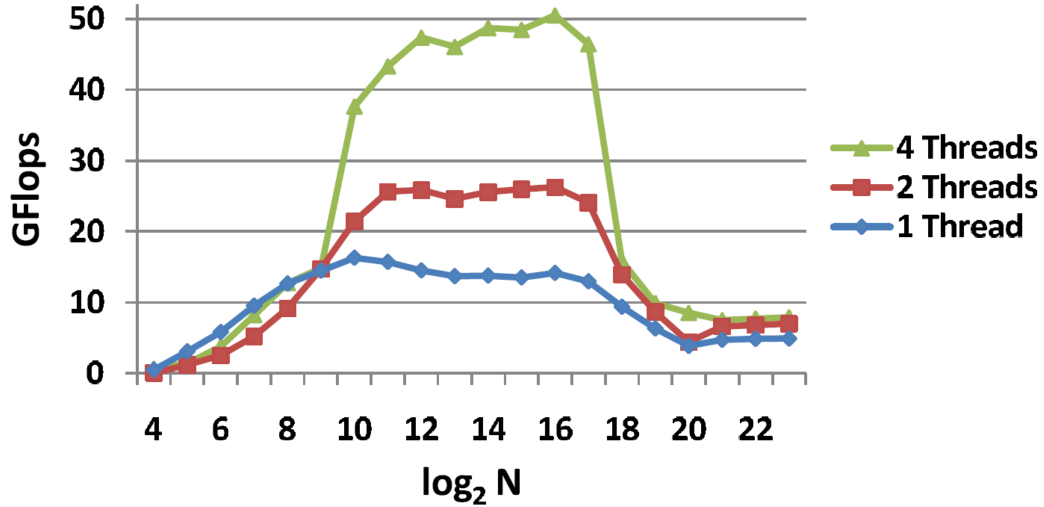



Figure 5.4: Intel MKL: Single 1D FFT per Thread [6]

The illustration in Figure 5.4 represents the results for 1,2 and 4 threads operating on the quad-core CPU, and the results are given in Giga FLoating-point OPerations per Second (GFLOPS). By comparison, the recursive DFT implementation on FPGA hardware was limited to fixed-point arithmetic, and does not perform floating-point operations in the current implementation.

Traditionally, fixed point arithmetic is stated in terms of Millions of Instructions Per Second (MIPS), and not FLOPS. It is not uncommon to have multiple instructions per singular operation, so in an effort to perform a more accurate comparison, an operational count can be performed on the recursive DFT implementation. In this manner, a Operations Per Second (OPS) comparison can be performed between the two systems. Computing the OPS for the recursive DFT gives a total of 10 real operations for the forward transform (using Equation 3.15 where for example, $f_{in} - f_{out}$ is considered one real arithmetic operation, and $F[u] + \frac{f_{in}-f_{out}}{N}$ is considered two real arithmetic operations due to the complex valued $F[u]$ term - assuming $\frac{f_{in}-f_{out}}{N}$ is already computed).

University of Cape

The error correction engine is another source of computation, and using Equation 3.41, the total real arithmetic operations for the error correction is 18. Summing the total operations, and computing the update rate gives:

$$\begin{aligned} Total\ OPS &= Total\ OPS_{RDFT} + Total\ OPS_{EC} \\ &= 10 + 18 \\ &= 28 \end{aligned} \tag{5.1}$$

Working at a clock rate of 82.2MHz with 13 cycles required per sample update provides the number of updates possible per second. Using Equation 5.1, the total operations per second per processing element can be computed.

$$\begin{aligned} OPS_{PE} &= \frac{Clock\ Rate}{Cycles\ Per\ Update}(Total\ OPS) \\ &= \frac{82.2 \times 10^6}{13}(28) \\ &= 1.767 \times 10^8\ OPS\ per\ processing\ element \end{aligned} \tag{5.2}$$

Using a total of 64 processing elements:

$$\begin{aligned} OPS_{64} &= 1.767 \times 10^8\ \times\ 64 \\ &= 11.3\ GOPS \end{aligned} \tag{5.3}$$

Using the results in Figure 5.4, an equivalent length FFT ($2^6 = 64$) yields less than 10 GFLOPS. Applying Equation 5.3 hypothetically for all DFT lengths illustrated in Figure 5.4 (assuming the hardware real-estate permitted such design scaling) provides the results shown in Figure 5.5 (shown up to $2^{12}$) and tabled in Table 5.5.

University of Cape Town

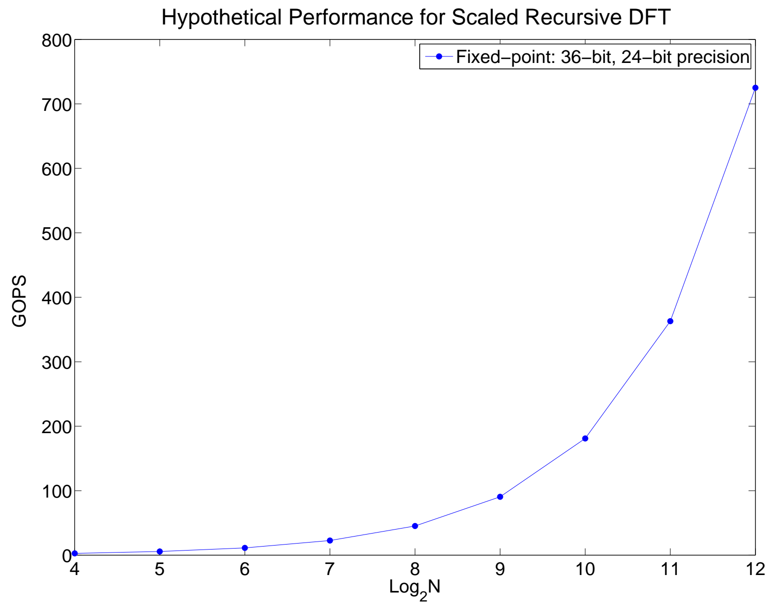



Figure 5.5: Hypothetical Performance for Scaled RDFT

University of Cape Town

Table 5.5: Hypothetical Performance for Scaled RDFT

| $DFT Length$ | $GOPS$ |
|---|---|
| $2^4$ | 2.83 |
| $2^5$ | 5.67 |
| $2^6$ | 11.33 |
| $2^7$ | 22.66 |
| $2^8$ | 45.32 |
| $2^9$ | 90.65 |
| $2^{10}$ | 181.3 |
| $2^{11}$ | 362.59 |
| $2^{12}$ | 725.18 |

Comparing the results obtained for OPS versus FLOPS, the sheer parallelism available for the recursive DFT plays a significant role in concurrent operations that can be performed. This concurrency allows the performance scaling to climb exponentially, however it should be noted that to achieve such performance a significantly larger design would need to be synthesized and placed on an FPGA. Currently no FPGA exists capable of handling such design resource requirements (but future process technologies may yield such devices).

Similarly, an alternative method to compute performance is to view operational time per DFT update versus the DFT length (a method employed for performance comparison in FFTW algorithms [95]). It was computed in equation 5.1 that the total operational count for the recursive DFT with error correction is 28. An update requires 13 cycles, and when operating at 82.2MHz (period 12.165ns), a full update per DFT point requires:

$$\begin{aligned} Update\ Time_{PE} &= Clock\ Period \times Cycles\ Per\ Update \\ &= 12.165(ns) \times 13 \\ &= 158.15(ns)\ per\ processing\ element \end{aligned} \tag{5.4}$$

It has been shown that the recursive DFT supports wide-scale parallelism and provided $N$ processing elements can be supported in hardware and can operate simultaneously, the operational time would not scale with $N$. Using the approach in [95], the GOPS for an RDFT (length $N = 64$) can be computed as follows:

$$\begin{aligned} GOPS_{Time} &= \frac{OPS\ per\ processing\ element \times N}{Time\ for\ one\ RDFT\ in\ ns} \\ &= \frac{28 \times N}{158.15} \\ &= 11.33 GOPS \end{aligned} \tag{5.5}$$

University of Cape Town

The results shown from the computation in Equation 5.5 and Equation 5.3 match when adopting slighty different approaches, and while no new information has been added, it maintains a standardised method of performance computation for a given DFT length.

Turning attention back to the results from the MKL implementation, it can be seen that the results plateau for all threads for FFT lengths from approximately $N = 2^{10}$ to $N = 2^{17}$ (with a peak performance at 52 GFLOPS). Since the pairs of cores share the 6MB L2 cache, performance begins to degrade at about $N = 2^{18}$ due to increased number of cache conflicts between cores.

Assessing the error performance (Figure 5.6), overall, the MKL has lower error than the GPU algorithms (to be discussed next). Compared to the recursive DFT, the floating-point precision yields more accurate results. The recursive DFT accuracy was discussed in Section 3.7 and simulation results show squared error (not mean) to indicate an instantaneous error magnitude. The traces in Figure 3.25 to 3.28, illustrate a mean square error projection. By contrast, the results shown in Figure 5.6 are Root Mean Square Error (RMSE) results.

Taking the square root of the results shown in Figures 3.25 to 3.28 allocate all computed errors in the order of $10^{-3}$ (the order of $10^{-5}$ shown is due to the absence of the square root). The two orders of magnitude difference is to be expected for fixed-point versus single precision computations. The hardware resource requirements is however lower if implementing fixed-point instead of floating point arithmetic, and could be reduced if lower precision could be tolerated. The overall error for both CPU and GPU continue to grow as the FFT size grows - a likewise characteristic borne by the recursive DFT.

University of Cape Town

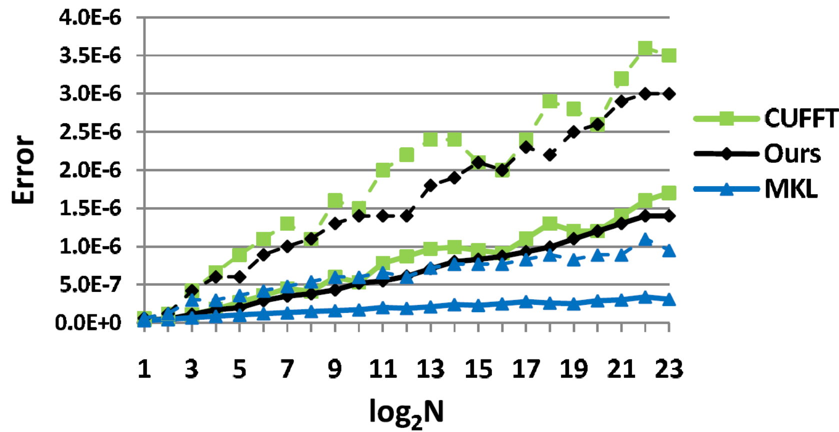



Figure 5.6: FFT RMSE for CPU and GPU Implementations. The maximum error is shown with the dashed lines [6].

### 5.4.3 GPU

Another point of comparison for FFT performance assessment is the use of the Graphics Processing Unit (GPU). The same work discussed for the CPU implementation of the FFT [6] also focused on the GPU implementations, where the focus was on NVIDIA GPU 8800GTX, 8800GTS, and GTX280 hardware.

The authors compared custom FFT algorithms to NVIDIAs CUDA FFT library (CUFFT) version 1.1 for the GPU, and charted performance for both single and batch implementations of FFT's. The execution time is obtained by taking the minimum time over multiple runs for each implementation. It should be noted that the time for library configuration and transfers of data between the GPU is not included in the timings.

University of Cape Town

The performance obtained from multiple GPU implementations is illustrated in Figures 5.7 and 5.8. The CPU implementation (1 thread) is also included, and is labeled 'MKL'. The peak performance for a single FFT run (Figure 5.7) resulted from the GTX280 GPU, and peaked at approximately 90 GFLOPs. A significant improvement in performance was achieved for batch FFT's (Figure 5.8), with a peak performance of 250 to 300 GFLOPs (the dashed line was achieved using an older set of drivers).

The recursive DFT implementation in this work does not perform batch processing runs in its current gateware form. Comparing to the single FFT performance, the projected results for the recursive FPGA implementation could substantially outperform the GPU, if enough resources were available to support the implementation size. Resource requirements would grow if single floating point precision were used in the recursive DFT, making the implementation a more difficult feat to achieve.

Cape Town

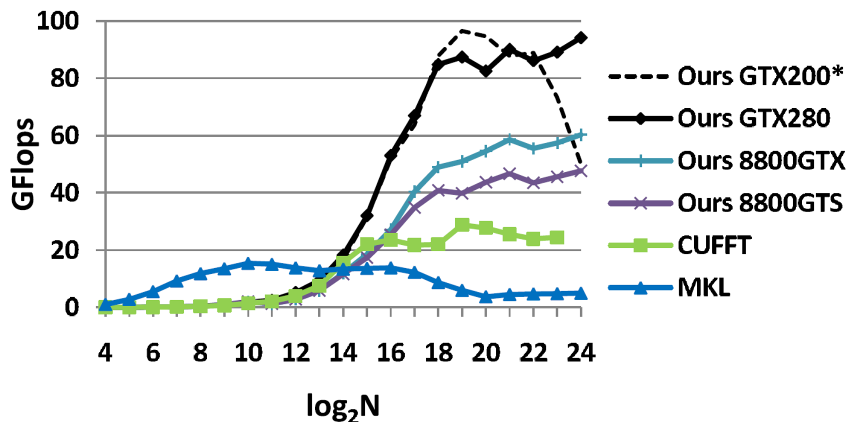



Figure 5.7: Single FFT GPU Results [6]. The dashed line indicates results achieved using an older set of drivers.

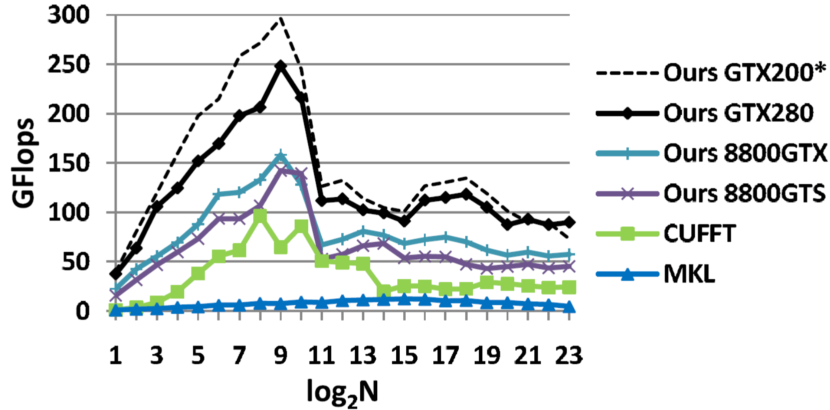



Figure 5.8: Batch FFT GPU Results [6]. The dashed line indicates results achieved using an older set of drivers.

To compute the accuracy of the computations, a forward transform followed by an inverse transform on uniform random data was performed. The results are compared to the original input and the root mean squared error (RMSE) and maximum error are divided by 2. The algorithms are designed for single-precision complex arithmetic as the currently available GPUs only support single-precision arithmetic. The results are jointly illustrated in Figure 5.6, and are of the order $10^{-6}$. As mentioned in Section 5.4.2, the order of error for the recursive DFT (forward or reverse transform) is of the order $10^{-3}$. The difference in accuracy can be attributed to the single floating point precision which is supported by GPU hardware.

University of Cape Town

## 5.5 FPGA to ASIC Projection

The results discussed in this chapter are indicative of the performance possible from all the studied systems, however the underlying platforms on which they operate differ substantially. CPU and GPU hardware are different architecturally, however share the common attribute that both architectures are specific, fixed, and manufactured using suitable transistor densities. The transistors placed on the die are therefore specific to the purpose in the design.

An FPGA by comparison consists of many configurable logic cells and hardware blocks such as multipliers and adders, and has configurable interconnects to allow for a myriad of logical configurations, permitting a very wide range of possible sythesizable circuits. As a result of the coarser but more flexible architecture, performance in terms of power, circuit density, and operational speed are affected.

Studies have shown that the delay of an FPGA lookup table (LUT) can be 12-14 times the delay of an ASIC (Application Specific Integrated Circuit) gate, and that the gate density of an ASIC is approximately 45 times greater than that of an FPGA [96]. It was also found that the final operating frequency of a design can vary depending on the initial value of the random seed given to the placement tool. The comparison study showed that for an FPGA design consisting of both logic and DSP elements, the FPGA design on average (geometric mean) would be 28 times larger than an equivalent ASIC implementation (90nm process). Similarly, the speed advantage of an ASIC would be between 3.4 and 4.5 times faster (depending on the FPGA speed grade). Lastly, the power benefit attributed to an ASIC design would be on average 12 times lower.

In effect, these results indicate that a design sythesized, placed and routed for an FPGA can operate 3.4 to 4.5 faster using 28 times smaller area on a 12 times smaller power budget when implemented as an ASIC. Using an average of a 4 times speed difference, the equivalent ASIC implementation of the current recursive DFT (64 point) on an FPGA can be projected (Table 5.6).

University of Cape Town

Table 5.6: Projected FPGA to ASIC Conversion

| *Attribute* | *FPGA* | *Equivalent ASIC* |
|---|---|---|
| Frequency | 82.2MHz | 328.8MHz |
| Area | 64 Points | 1792 Points |
| Dynamic Power | 6.82928W | 0.569W |

The scaling of the theoretical operating frequency as well as the number of DFT points would have a significant impact in the number of operations possible per second. This will also contribute significantly to the data handling and exporting, and extremely high-speed interfaces would be required to handle this volume of data. It is not the purpose of this study to focus on data handling techniques and appropriate interfaces, however it is worthwhile considering as it can severely reduce the realistic throughput possible. The performance comparisons are supplied to indicate expected performance, and it can be concluded that despite platform and implementation differences, the transformation of a recursive DFT FPGA to ASIC design would yield further benefits.

University of Cape Town

## 5.6 Additional Hardware Requirements

In Chapter 3, a test case application example of processing an input signal (chirp) originating from a theoretical Analog-to-Digital Converter (ADC) front-end was considered. It it worthwhile reflecting back onto this example to define the hardware requirements for such as system based on the operating limitations determined from the actual implementation of the recursive DFT (for real-time operation).

In Section 5.3.2 it was highlighted that the maximum operating clock frequency obtainable with this current design on the two available FPGAs was 82.2MHz, and the computational latency per sample was 13 cycles. After the initial transfer of coefficients, the system begins processing, requiring a new sample every 13 cycles. This provides sufficient time to mask the transfer of input data from an ADC front-end to the input stage of the FPGA system. A system operating at 82.2MHz with a 13 cycle latency would allow a maximum input sample rate (complex) of 6.32Msps at a maximum word length of 36 bits. The design discussed in this study implemented pipelining in key stages such as the multipliers, however could still be optimized to reduce the overall latency, and increase the maximum clock rate. Modifications such as additional pipelining would assist in increasing the maximum input sample rate, and results would scale as the maximum operating clock rate is improved.

University of Cape Town

## 5.7 Chapter Summary

This chapter discusses and analyses the results obtained from the hardware testing in Chapter 4. Consideration was given to the resource utilisation within the FPGA platforms, as well as the operating clock frequency attainable and processing latencies incurred. Power analysis, system accuracy as well as additional front-end data streaming support were also discussed. Finally the system is benchmarked and compared against three alternative FFT arrangements: Multi-core CPU, GPUs, and an alternative FPGA implementation. The chapter concludes with a discussion on FPGA-to-ASIC design conversion.

This chapter initially considered the utilisation of resources when built for the two FPGA devices. It was shown that both devices were insufficient in DSP48E resources to allow a full design map, and therefore the design had to be reduced to enable the design to fit. Once built, the design was capable of operating up to 82.2MHz with a maximum path delay between nodes of 3ns. The number of cycles required per update was determined through Simulink to be 13, and later verified on hardware using ChipScope.

To enable the input data streaming feature, it was necessary to develop a front end capable of providing a input sequence in a sample-by-sample fashion from a re-usable data source. To achieve this, a data streaming front-end was developed in Simulink and included in the design. This sub-system allowed any arbitrary data sequence to be included and processed, and emulated a real-world ADC implementation.

An important performance metric to any system is the power utilisation. This chapter discusses the power usage of the various resources in the FPGA (VSX95T), and showed that a large portion of the power budget is allocated to quiescent power. Projecting the data for a 64 point system, it was shown that the total dynamic power would be approximately 6.8W.

This chapter continued with the performance analysis of the system, and performed a benchmark comparison of the recursive Fourier transform implemented on an FPGA with the FFT implementations on a multi-core CPU, GPU, and FPGA. The results showed that a dedicated FFT performed on an FPGA significantly outperformed the recursive implementation when assessing from a block processing perspective. The sample-by-sample updating reversed this result, and showed the recursive implementation with error correction is approximately 20x faster for the equivalent length FFT.

University of Cape Town

To assess the multi-core CPU and GPU results, an alternative comparison metric for the recursive DFT was required. The multi-core CPU and GPU results compute in floating-point precision, where as the recursive DFT on the FPGA operates using fixed-point arithmetic, and a comparison in operational cycles would not be sufficient. For a more accurate comparison, it was necessary to consider the recursive DFT in terms of Operations Per Second (OPS), rather than cycles. It was shown that given a clock frequency of 82.2MHz, the number of OPS per processing element was $1.767 \times 10^8$. This result was computed for various length DFT's, and compared to the multi-core CPU and GPU implementations for the equivalent DFT length.

Comparing to 4 threads on the CPU, the recursive DFT FPGA implementation was able to surpass the performance of the CPU running more than an order of magnitude faster. The GPU comparison showed similar prospects, however the GPU using batch processing conditions was able to outperform the recursive DFT only up to a DFT length of approximately $2^{10}$. Computational error was also considered, and it was shown that floating-point precision of the multi-core CPU and GPU permitted accuracy of two orders of magnitude better than the recursive DFT implementation using fixed-point arithmetic ($10^{-5}$ versus $10^{-3}$).

Finally, the chapter considers an FPGA to ASIC projection. The design tested in this work was mapped to fit FPGA resources, and it has been shown that higher gate densities, improved clock speeds, and lower power utilization are possible when implementing the same design as an ASIC. The projection showed that it could be possible to implement a DFT design of 1792 points and exceed an operational frequency of 328MHz using less than a $10^{th}$ of the power.

University of Cape Town

# Chapter 6

# Conclusions

University of Cape Town

This study began with the introduction of the current state of computing from a generic perspective. Over the past 40 years of computing history, many significant advances have been made in areas such as processor architecture, on-chip and off-chip memory, as well as data transfer interfaces. The ability to transfer data through high-speed interfaces and access memory within a few tens of cycles has contributed to the overall improvement of computing platforms, but it is the throughput of the processing architectures, when presented with a set of instructions, which plays a significant role in the overall performance of the system.

It can be argued that to obtain the best performance of a computing platform, application code should be programmed and compiled with the underlying architecture and memory system in mind. Masking memory latencies, predicting cache misses, and leveraging specific vector instructions became the norm for squeezing performance from a given system. This has been the dogma and practice since the dawn of computing, and even in recent times with the advent of parallel architectures and multi-core processing systems, very little has changed.

In the days before wide-scale parallelism where sequential processors dominated, much reliance existed on the hardware vendors to continue to scale performance as transistor densities improved. The ability to obtain a speedup for legacy software by simply waiting 18 months for newer, faster hardware to emerge became the 'free lunch' of the computing industry, and paradoxically its Achilles

heel. The lack of preparation from the software community to adopt and define parallel software coding practices and standards has seen the development of compilers and supporting languages for parallelism lagging behind.

Today great advances have been made, however there is still the tenet of 'design first, figure out how to program later' that still lingers. After 40 years of history to guide current research, a burning question still haunts the engineers of today and tomorrow - Is this *still* the correct approach? A question of this magnitude cannot have a simple answer. Reading the current state-of-the-art, it is clear that much research is still going on in all areas of computing, and it seems that over time, the community at large will slowly piece together the computing puzzle, for which the full final image is yet to be seen or known. This study was aimed to help answer this important question in part - one piece at a time.

The focus of this study was however not based on determining an efficient generic parallel processing architecture. It instead adopted a different design approach, where a class of computing was first selected and analysed, before determining suitable hardware tailored to the class being considered. Holistically, computing consists of many different classes, each different enough in computational and data handling requirements to stand alone.

The class under investigation in this work was *Spectral Methods*, which by its very nature, has its own processing and data communication requirements. The purpose of this study was to investigate the processing and data handling requirements of the *Spectral Methods* class, and design a suitable framework to support this class. This approach was different from past traditions - the hardware framework is based on software requirements, and in a sense is designed for the processing requirements, rather than the other way around.

It can be argued that this type of approach makes the hardware implementation indistinguishable from software requirements, as choice algorithms are embedded in hardware. This may not be a general rule and is based on computational specifications. In this study a fine line exists between a software instruction

University of Cape Town

and a hardware implementation. The underlying computation in *Spectral Methods* is the Fourier transform, which in turn can be assessed to reveal the underpinning of complex arithmetic.

No study would be complete without first reflecting back on the study objectives, hypothesis and research questions. Recalling the hypothesis stated in Section 1.4 provides the necessary starting point.

*A Many Processing Element (MPE) framework, tailored to fundamental spectral method motif class operations, promotes scalable performance for spectral method motif class applications.*

Consideration of the hypothesis leads to the following research questions:

- What are the fundamental operations of the spectral methods motif class?
- Can a spectral methods motif be implemented using many simple processing elements?
- Is parallelism scalable? If so, to what degree?
- What effect would finite word-widths have on fixed-point arithmetic? Can computational errors be determined and corrected?
- Is performance improvement possible using spectral method based MPE's, as compared to a specific hardware accelerated approach?

It is hypothesized that the spectral methods class of computing contains a handful of fundamental computations which make up the processing characteristic of the class; and the conjecture is that implementing these fundamental computations into a parallel processing framework, comprising of many processing elements, will achieve computing performance which is scalable to the degree of parallelism implemented.

University of Cape Town

The first research question pertaining to the fundamental operations was answered in Chapter 2 when the focus was on the computations characteristic to the spectral method class. It was determined that the fundamental computation in the class is the Fourier transform, with a tight coupling of related Fourier space operations such as integration and differentiation, to name just a few. The use of additional Fourier space operators is application dependent, however they have been shown to occur frequently enough to be considered fundamental to the class, with many having a strong reliance on complex arithmetic.

The hypothesis went on to state that a many processing element framework should be used and naturally lead to asking if in fact a spectral method motif can be implemented using many simple processing elements, as well as to what degree. To answer these questions requires careful consideration of the available techniques capable of achieving the computations already listed, and makes it imperative to employ a scalable parallel technique to perform Fourier transformation and related Fourier operations.

The goal of a high-throughput system through parallelism with minimal data dependencies steered the study to identify a technique to compute the Fourier transform, while making it possible to use the underlying hardware to support additional Fourier space operations. Investigations revealed that parallel scalable algorithms to compute the Fourier transform do exist, and with suitable modifications could be made to perform both forward and reverse Fourier transforms with error correction abilities for finite-bit arithmetic.

The Fourier transform technique should, at the most basic level, encompass the same typical processing as required by the Fourier space operations to promote hardware re-usability and ideally encourage local processing within the processing element in order to minimise data transfers. The Fourier technique, which provided all these features, was the recursive Fourier transform which allowed the updating of the output on a sample-by-sample basis.

University of Cape Town

Techniques such as the Fast Fourier Transform (FFT) operate in a block-data fashion where a set of samples is first required before the computation can be performed. The ability to compute a Fourier update sample-by-sample opens up a new opportunity for increased time-frequency resolution, while still maintaining the ability to compute either a forward or reverse transform on blocks of data, if required. The sample-by-sample updating required the recursion of results which in turn introduced the negative side-effect of arithmetic error accumulation for finite-bit arithmetic. If left unchecked, this error would grow unbounded, eventually saturating the respective output.

The recursive Fourier transform exhibits no data dependencies during processing, making it capable of achieving $N - wide$ parallelism ($N$ representing the number of DFT points). The wide-scale parallelism possible through the use of the recursive Fourier transform promotes high throughput for Fourier based transformations, however it should be noted that not all additional Fourier space operations can be locally processed, and could require transfer to neighbouring processing elements for further use - a task achieved through processor interconnection hardware.

The processing considered in this study was limited to fixed-point arithmetic, which in turn raises the question of the effect of finite-bit word widths on the results. If errors become inherent due to arithmetic and coefficient quantization, can these computational errors be determined and corrected? The effect of finite-bit word length arithmetic was carefully analysed, and showed that induced errors can be bounded and corrected under certain conditions.

To bound the error growth, this study further develops an error computation and correction technique, making it possible to determine the error at any time instant for use in correcting the respective Fourier output. The results showed that it is possible to error correct and limit the error in any output for finite-bit arithmetic, and achieve bounded square error results in the order of $10^{-6}$ for 36-bit word lengths using 26-bit precision (root mean square error of the order $10^{-3}$). Furthermore, it was shown that a 14-bit difference in word lengths between

University of Cape Town

the overall output fixed-point length, and the imposed error correcting enabling threshold ($\upsilon$), would ensure the output error remains bounded and predictable.

It is useful to express the error correction in a generic model for analysis, and this study provides an analytical model of the error for the computation of both a forward and reverse Fourier transform. The results revealed that the error correction technique used to determine the state of the error at any time iteration is accurate in a gross sense, and that it is possible to track and correct Fourier outputs on-the-fly.

This study also contributes a hardware implementation of the error correction technique with automatic on-the-fly correction (implemented in conjunction with hardware to compute both the forward and reverse Fourier transform). The full system was simulated (both a software simulation and hardware simulation) and later verified using a Field Programmable Gate Array (FPGA), where the results obtained matched the hardware and initial software simulations performed to verify the design.

A final question posed to help validate the hypothesis revolves around the performance of the framework as a whole, and how it would stand up against alternative implementations of multi-core CPUs, GPUs and FPGAs. Benchmarking the system revealed that performance gains are possible when compared to both a specific hardware accelerated approach, such as a hardware implemented FFT, and a generic hardware approach, such as using commodity multi-core processors with optimized algorithms. Focussing on the FPGA results, it was shown that a dedicated FFT performed on an FPGA significantly outperformed the recursive implementation when assessing from a block processing perspective, whereas the sample-by-sample updating approach showed the recursive implementation with error correction to be approximately 20x faster for the equivalent length FFT.

The multi-core CPU and GPU implementations showed multi-gigaFLOP performance, however were typically bounded by the number of possible threads, or cache size limitations. The multi-core CPU and GPU comparison required

University of Cape Town

an alternative comparison metric for the recursive DFT (Operations Per Second - OPS), and if each processing element is considered individually, the OPS per element is $1.767 \times 10^8$. Scaling this for multiple DFT lengths showed that if 4 threads are running on the multi-core CPU, the recursive DFT FPGA implementation was able to surpass the performance of the CPU running more than an order of magnitude faster.

The GPU by comparison (under batch processing conditions) was able to outperform the recursive DFT only up to a DFT length of approximately $N = 2^{10}$, however the projected recursive DFT results are theorized for many DFT sizes, as in many cases implementation is impractical because FPGA hardware with sufficient resources do not yet exist. Computational error was also evaluated, and it was shown that floating-point precision of the multi-core CPU and GPU permitted accuracy of two order's of magnitude better than the recursive DFT implementation using fixed-point arithmetic ($10^{-5}$ versus $10^{-3}$).

The recursive DFT by comparison promoted scalable performance, with the possibility of exceeding the GPU and CPU performance, albeit at fixed-point precision, and not the more resource intensive single precision floating point, as supported by the GPU hardware. To explore the possibility of more practical implementations of recursive DFTs of larger sizes, the study considered the practical gains and losses attributed to FPGA to ASIC design conversions. Previous research indicated that performance benefits for power, area, and speed are all possible when converting an FPGA design to a functionally equivalent ASIC design, and suggests that further performance benefits can be obtained when considering this route. This in turn shows even greater promise for performance, as gate densities of approximately 28×, and speedups of approximately 4× would lead to even greater performance for the same design.

Reflecting back on the hypothesis and research questions collectively, the consolidated positive answers of the research questions in turn verify the validity of the hypothesis for this study. The primary objective of this study (stated in Section 1.5) was to develop a parallel hardware framework specific to the spec-

University of Cape Town

tral methods motif, and one which will be generic to applications which reside within the spectral methods class. This study has been successful in the objective stated, and has showed that the design paradigm of ‘assess first, design later’ can in fact be beneficial for at least one class of computing. The framework and associated processing elements exhibited in this study are tailored to the specific computational requirements of the spectral methods class, however are generic to applications which reside in the class.

This study opened up with a brief overview of computing history, and highlighted many past exploitations which have ultimately led to a brick wall. There has been no obvious alternative to computing except through parallelism, which in turn has brought the industry to an exciting inflection point. Moore’s law still prevails offering improved transistor densities, and is constantly being given a new lease on life through newer technologies such as Fin Field Effect Transistors (FinFETs) and Ultra Thin Body Silicon On Insulator (UTBSOI) transistors which promise at least two to three more generations of transistor density scaling.

The adoption of parallelism in the form of many processing cores brings with it its own challenges and choices. The future of processing has been touted as heterogenous, meaning that a single ‘one size fits all’ architecture simply won’t work. There needs to be diversity and all processing classes need to be considered to produce a truly hetergenous system. An analogy of an incomplete jigsaw puzzle picture representing the holy grail of computing was used earlier to convey the point that over time and persistent effort, the picture will slowly become clearer as more pieces are added. This study aimed to contribute at least one piece to the puzzle grid of computing. It is clear that the race to the end is far from over, and much work still needs to be done, and it will be through the creative efforts, tenacity, and sheer willingness to explore all avenues that the puzzle will become complete - one study at a time.

University of Cape Town

## 6.1 FutureWork

The core of this study has been focused on the development of a processing framework for spectral method class of computations. The body of this work assessed the class, elected a suitable Fourier transform algorithm, and developed a hardware implementation for both the recursive DFT and an error correction technique.

The Fourier transform is the dominant computational requirement for the class, and the nature of the recursive DFT and its computational requirements allow for a framework which can support additional Fourier operators. To support additional Fourier operators, the internal architecture of each processing element should provide support, where each processing element would require a combination of accumulators, multiplexers, data registers and control lines from the system controller. Recalling some of the listed Fourier operators:

- Reciprocal Scaling Property
- Time Shift Property
- Frequency Shift Property
- Integration Property
- Time Domain Differentiation Property
- Convolution Property
- Area Property

It is the purpose of this section to highlight a theoretical modification of the existing framework discussed to support the additional Fourier operators. The architectural variations shown are aimed at the complex arithmetic section of the processing elements, however would also require appropriate control lines from the system controller for data transfer and input selection. It should be noted the additions for each Fourier property have not been vigorously tested, and are included to indicate the versatility and applicability of the framework for wider-scale spectral computations.

University of Cape Town

### 6.1.1 Reciprocal Scaling Property

Consider:

$$f(\alpha t) \Longleftrightarrow \frac{1}{|\alpha|} F\left(\frac{\omega}{\alpha}\right) \tag{6.1}$$

The hardware required to implement this function is shown in Figure 6.1, and remains the same in principle as implemented for the forward or reverse Fourier transform (many registers and control logic have been abstracted away for explanation simplicity). The heart of all computations is the complex multiplier, and the peripheral circuitry provide purely a supportive role. To perform the operation, the scaling term $\alpha$ is required to be known. Scaling $t$ in the time domain by $\alpha$ has the dual effect of adjusting $\omega$ by $\frac{1}{\alpha}$, and multiplying the $F\left(\frac{\omega}{\alpha}\right)$ Fourier term by $\frac{1}{|\alpha|}$ in the Fourier domain.

To obtain $F\left(\frac{\omega}{\alpha}\right)$, the Fourier term can be shifted by the system controller by requesting the processing elements swap Fourier terms through the processor interconnect, and then multiply by $\frac{1}{|\alpha|}$. The $\frac{1}{|\alpha|}$ can be imported to each processing element through the input bus system. The transfer of Fourier results prior to reciprocal scaling can be performed through the interconnect network, and through addressing each of the respective processing elements that are due to receive data. The transfer is executed to ensure the results stay in order. The additional hardware (shown in red) represents the supporting periphery. The second register included allows for the new operation to be stored without overwriting the original Fourier computation (stored in first register). The 'Cast' block performs truncation of the output word from the complex multiplier into 36-bit word lengths.

University of Cape Town

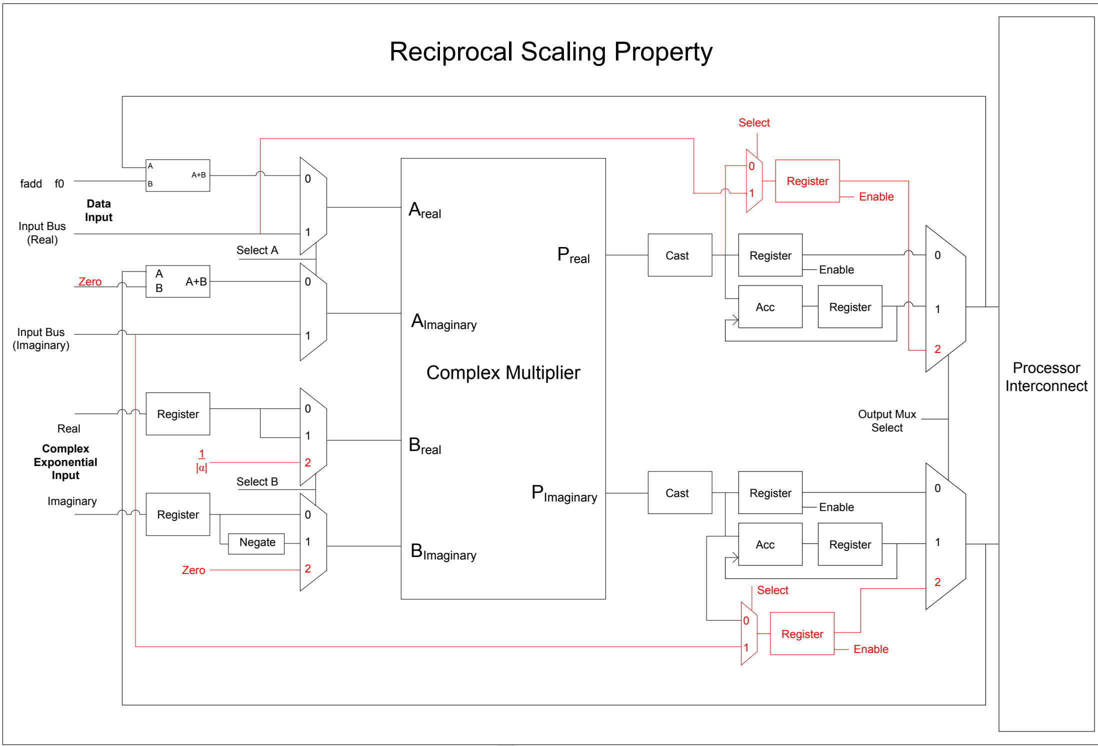



Figure 6.1: Reciprocal Scaling Property

University of

### 6.1.2 Time Shift Property

Consider:

$$f(t - t_0) \Longleftrightarrow \exp\left[-j\omega t_0\right] F(\omega) \tag{6.2}$$

The supporting hardware that implements the 'Time Shift Property' is illustrated in Figure 6.2. The only notable addition to the framework is the addition of the two input registers for the complex multiplier. The scaling factor support for 'Reciprocal Scaling' has been purposely omitted for clarity, but in reality it would be present on the input to the multiplexer. The two additional registers (shown in red) are used to store the complex exponential required for complex multiplication (this is a local only computation - no transfers are required among the processing elements). The first set of registers were used to store the complex exponential to compute the Fourier transform, and may be required for reverse transform computation.

Cape Town

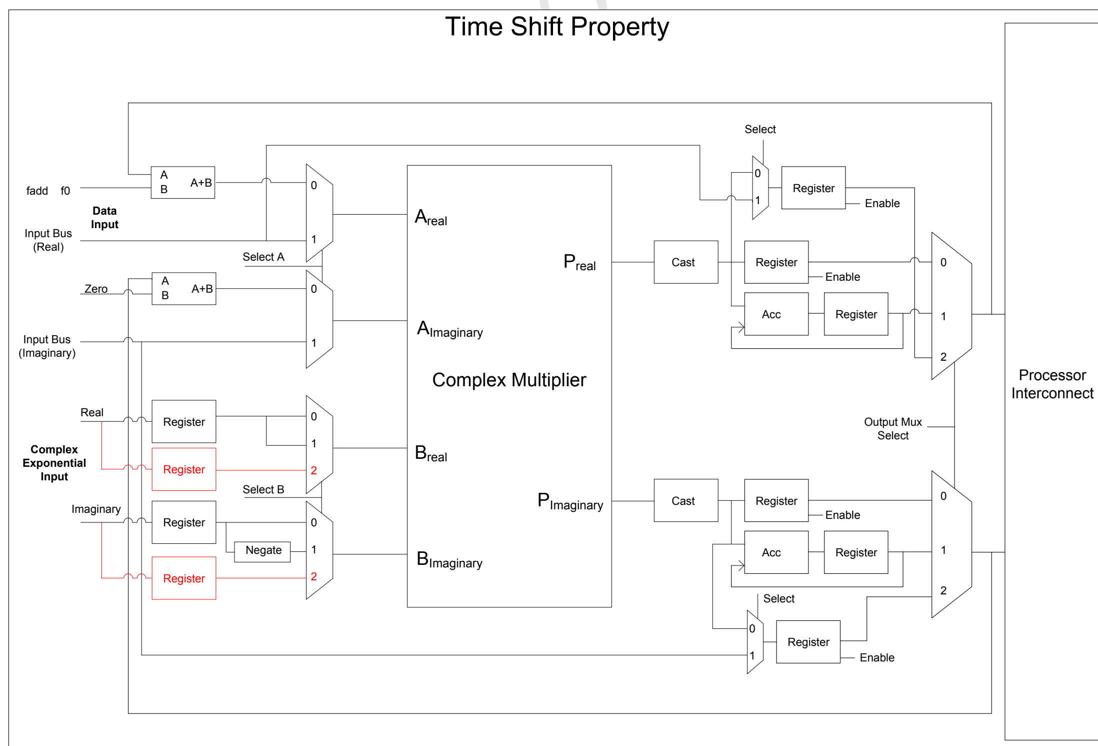



Figure 6.2: Time Shift Property

### 6.1.3 Frequency Shift Property

Consider:

$$f(t)\exp\left[-j\omega t_0\right] \Longleftrightarrow F(\omega-\omega_0) \tag{6.3}$$

In this operation it is only a requirement to shift Fourier domain outputs among the processing elements through the processor interconnect. A shifting term $\omega_0$ is used to represent the relative shift. The system controller would require to know the value for the shifting term, and in turn address and request a transfer of outputs to be placed on the bus. The addressed processing elements would capture the respective output on the bus when addressed, and can use the register-multiplexer combination (marked in red in Figure 6.3) to store the new output for the respective DFT point.

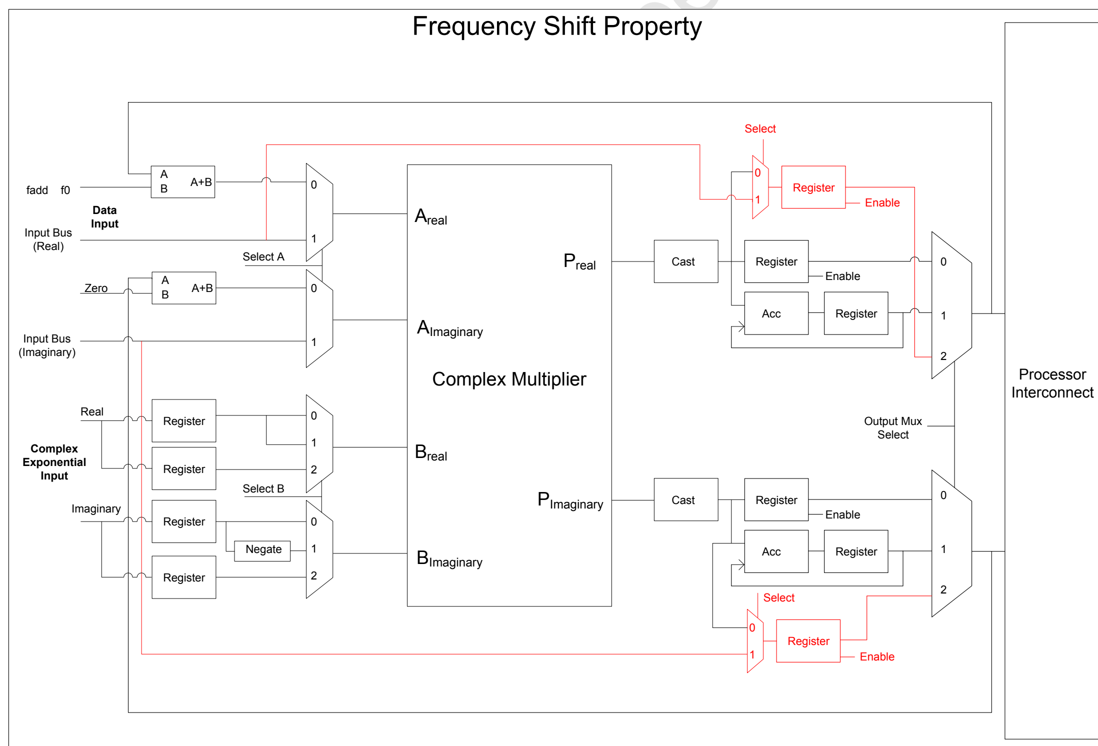



Figure 6.3: Frequency Shift Property

### 6.1.4 Integration Property

Consider:

$$\int_{-\inf}^{t} f(\tau)d\tau \Longleftrightarrow \frac{1}{j\omega}F(\omega) + \pi F(0)\delta(\omega) \tag{6.4}$$

To perform the integration property, two steps are required. The first step performs the partial computation of $\frac{1}{j\omega}F(\omega)$ which is relevant to all processing elements. This is a local only computation, and is simply the complex product of $\frac{1}{j\omega}$ and $F(\omega)$. These results are stored in the second register (marked red in Figure 6.4)).

The second part of the computation is local only to the $F(0)$ term due to the $\delta(\omega)$ term. The $F(0)$ term is required to perform the complex product of $\pi$ and $F(0)$. The $\pi$ term is loaded as an input to the complex multiplier for $F(0)$ (all other processors can be idle or can multiply by 1). During the first stage of processing, the partial result for $F(0)$ is stored in the accumulator, and after the second round of complex multiplication, the two partial results can be added in the accumulator stage. The final result is stored in the supplied register following the accumulator.

University of Cape Town

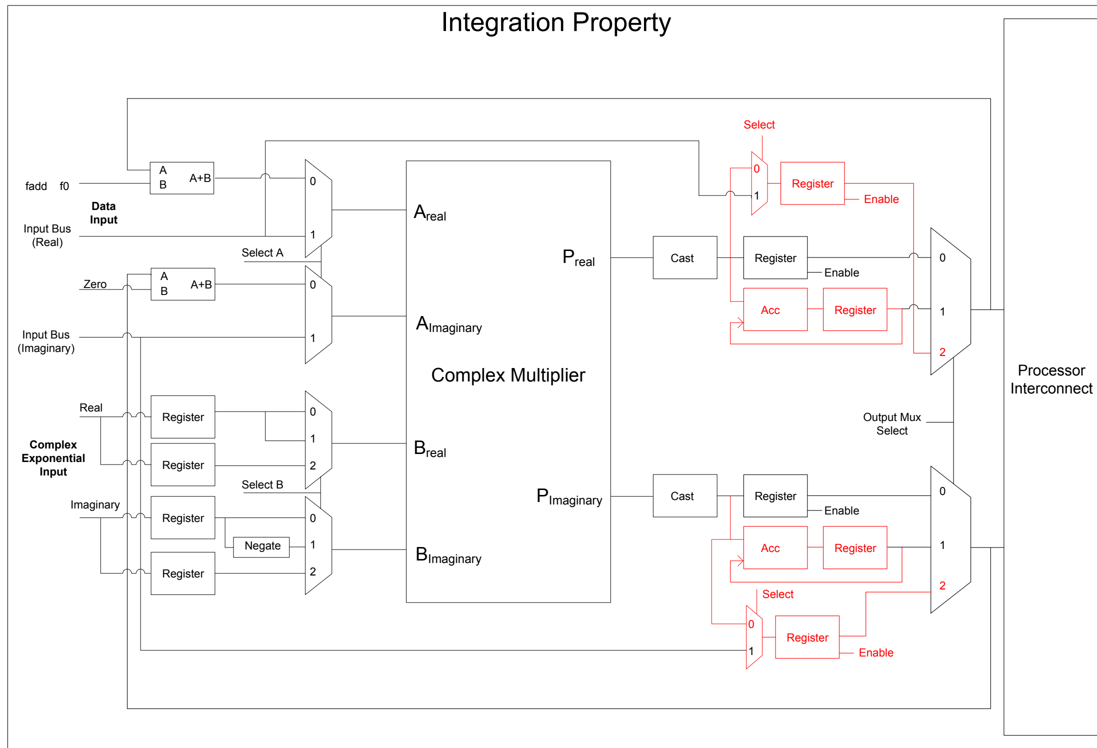



Figure 6.4: Integration Property

University of

### 6.1.5 Time Domain Differentiation

Consider:

$$D^n f(t)) \Longleftrightarrow (j\omega)^n F(\omega) \tag{6.5}$$

The hardware discussed for the previous operations remains the same for this property. The computation is simply a complex product for the current Fourier term $F(\omega)$, and a complex term $(j\omega)^n$, which is imported over the system bus from the system controller. The output is stored in a register and is available for further computations.

### 6.1.6 Convolution Property

Consider:

$$f(t) * h(t) = F(\omega)H(\omega) \tag{6.6}$$

To perform the convolution operation, the same existing hardware can be used. The operation is fundamentally a complex product (in Fourier space), and requires the $H(\omega)$ term to be loaded into the processing elements through the system bus. The resulting output is stored locally for further processing.

### 6.1.7 Area Property

Consider:

$$\int_{-\inf}^{\inf} f(t)dt = F(0) \tag{6.7}$$

The area property is the simplest additional operator that can be supported. The result of integrating the time domain representation $f(t)$ is the same as the Fourier $F(0)$ for the same sequence. No additional computing is required.

University of Cape Town

# References

University of Cape Town

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006. xiii, 2, 3, 4, 6, 7, 9, 10, 12, 17, 21, 22

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. D. Kubiatowicz, E. A. Lee, N. Morgan, G. Necula, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick, "The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View," Tech. Rep. UCB/EECS-2008-23, EECS Department, University of California, Berkeley, March 2008. xiii, 7, 8, 9, 10, 12, 17

[3] E. L. Kaltofen, "The Seven Dwarfs of Symbolic Computation." Dept. of Mathematics, North Carolina State University, Raleigh, North Carolina 27695-8205, USA, August 2011. xiii, 7, 9, 17

[4] "Xilinx ML506 FPGA Development Kit." xv, xvi, 172

[5] "CASPER Reconfigurable Open Architecture Computing Hardware (ROACH) Development System." xv, xvi, 170, 173

[6] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete Fourier transforms on graphics processors," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*

*International Conference for*, pp. 1 –12, 15-21 2008. xv, 190, 191, 196, 197, 198

[7] Xilinx, "Xilinx Virtex-5 Family Overview Product Specification (DS100 v5.0)." Online, February 2009. xvi, 171

[8] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, pp. 114–117, April 1965. 1

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. 1

[10] K. Olukotun and L. Hammond, "The Future of Microprocessors," *Queue*, vol. 3, no. 7, pp. 26–29, 2005. 1, 2, 4, 21

[11] M. Garland and D. B. Kirk, "Understanding throughput-oriented architectures," *Commun. ACM*, vol. 53, pp. 58–66, November 2010. 2, 6

[12] H. Sutter and J. Larus, "Software and the Concurrency Revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005. 2

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370 – 1380, 2008. General-Purpose Processing using Graphics Processing Units. 2, 6, 21

[14] F. Franchetti, M. Puschel, Y. Voronenko, S. Chellappa, and J. Moura, "Discrete fourier transform on multicore," *Signal Processing Magazine, IEEE*, vol. 26, pp. 90–102, November 2009. 2, 21

[15] Y.-K. Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard, "Signal processing on platforms with multiple cores: Part 1 - Overview and methodologies," *Signal Processing Magazine, IEEE*, vol. 26, pp. 24 –25, november 2009. 2, 3

University of Cape Town

[16] S. K. Moore, "Multicore CPUs: Processor Proliferation," *IEEE Spectrum*, pp. 36–39, January 2011. 2, 3

[17] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic, "The Bulk Multicore architecture for improved programmability," *Commun. ACM*, vol. 52, pp. 58–65, December 2009. 2

[18] Y. Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard, "Signal Processing on Platforms with Multiple Cores: Part 2-Applications and Design," *Signal Processing Magazine, IEEE*, vol. 27, pp. 20 –21, march 2010. 3

[19] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," *SIGARCH Comput. Archit. News*, vol. 38, pp. 302–313, June 2010. 3

[20] J. L. Trff, "What the parallel-processing community has (failed) to offer the multi/many-core generation," *Journal of Parallel and Distributed Computing*, vol. In Press, Corrected Proof, pp. –, 2009. 3

[21] X. Wen and U. Vishkin, "FPGA-Based Prototype of a PRAM-On-Chip Processor," in *ACM Computing Frontiers*, (Ischia, Italy), May 2008. To appear. 3

[22] S. Singh, "Computing without processors," *Commun. ACM*, vol. 54, pp. 46–54, August 2011. 3

[23] S. Borkar, "Thousand core chips: a technology perspective," in *DAC '07: Proceedings of the 44th annual conference on Design automation*, (New York, NY, USA), pp. 746–749, ACM, 2007. 3, 6, 22

[24] R. Biswas, L. Oliker, and J. Vetter, "Revolutionary technologies for acceleration of emerging petascale applications," *Parallel Computing*, vol. In Press, Corrected Proof, pp. –, 2009. 3, 6, 22

[25] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto, "Maximizing power efficiency with asymmetric multicore systems," *Commun. ACM*, vol. 52, pp. 48–57, December 2009. 3

University of Cape Town

[26] E. Larsson and O. Gustafsson, "The impact of dynamic voltage and frequency scaling on multicore dsp algorithm design [exploratory dsp]," *Signal Processing Magazine, IEEE*, vol. 28, pp. 127 –144, may 2011. 3

[27] A. Merigot and A. Petrosino, "Parallel processing for image and video processing: Issues and challenges," *Parallel Computing*, vol. 34, no. 12, pp. 694 – 699, 2008. Parallel Processing for Image and Video Processing. 4

[28] H. Corporaal and M. Arnold, "Using Transport Triggered Architectures for Embedded Processor Design," *Integr. Comput.-Aided Eng.*, vol. 5, no. 1, pp. 19–38, 1998. 4, 5

[29] J. Heikkinen, J. Takala, and H. Corporaal, "Dictionary-based program compression on customizable processor architectures," *Microprocessors and Microsystems*, vol. In Press, Uncorrected Proof, pp. –, 2008. 5

[30] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *SIGARCH Comput. Archit. News*, vol. 38, pp. 37–47, June 2010. 5

[31] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang, "Anton, a special-purpose machine for molecular dynamics simulation," *Commun. ACM*, vol. 51, no. 7, pp. 91–97, 2008. 5

[32] J. Kuskin, C. Young, J. Grossman, B. Batson, M. Deneroff, R. Dror, and D. Shaw, "Incorporating flexibility in Anton, a specialized machine for molecular dynamics simulation," in *Proceedings of the 14th International Symposium on HighPerformance Computer Architecture (HPCA-14), Salt Lake City, UT, 2008*, 2008. 5

University of Cape Town

[33] B. Gorjiara, M. Reshadi, and D. Gajski, "Merged Dictionary Code Compression for FPGA Implementation of Custom Microcoded PEs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 2, pp. 1–21, 2008. 5

[34] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," *Application Specific Processors, 2008. SASP 2008. Symposium on*, pp. 101–107, June 2008. 6

[35] A. Plaza, J. Plaza, A. Paz, and S. Sanchez, "Parallel Hyperspectral Image and Signal Processing," *IEEE Signal Processing Magazine*, vol. 28, pp. 119–126, May 2011. 6

[36] P. Bonzini, G. Ansaloni, and L. Pozzi, "Compiling custom instructions onto expression-grained reconfigurable architectures," in *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, (New York, NY, USA), pp. 51–60, ACM, 2008. 6

[37] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33, pp. 70–77, April 2000. 6

[38] J. Helmschmidt, E. Schuler, P. Rao, S. Rossi, S. di Matteo, and R. Bonitz, "Reconfigurable Signal Processing in Wireless Terminals," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, (Washington, DC, USA), p. 20244, IEEE Computer Society, 2003. 6

[39] M. H. Cho, C.-C. Cheng, M. Kinsy, G. E. Suh, and S. Devadas, "Diastolic arrays: throughput-driven reconfigurable computing," in *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, (Piscataway, NJ, USA), pp. 457–464, IEEE Press, 2008. 6

[40] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz, "RAMP Blue: A Message-Passing Manycore System in FPGAs," in *International Conference on Field Programmable Logic and Applications*, August 2007. 6

University of Cape Town

[41] S. Kyo, T. Koga, L. Hanno, S. Nomoto, and S. Okazaki, “A low-cost mixed-mode parallel processor architecture for embedded systems,” in *ICS ’07: Proceedings of the 21st annual international conference on Supercomputing*, (New York, NY, USA), pp. 253–262, ACM, 2007. 6

[42] K. Murakami, A. Fukuda, T. Sueyoshi, and S. Tomita, “An overview of the Kyushu University reconfigurable parallel processor,” *SIGARCH Comput. Archit. News*, vol. 16, no. 4, pp. 130–137, 1988. 6

[43] J. Boyd, *Chebyshev and Fourier Spectral Methods*. Dover, New York, 2nd ed., 2000. 12, 25

[44] L. N. Trefethen, “Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations,” 1996. 12, 13

[45] B. Fornberg, *A practical guide to pseudospectral methods*. Cambridge University Press, 1998. 12

[46] R. N. Bracewell, *The Fourier Transform and its Applications*. McGraw-Hill, New York :, 2d ed. ed., 1978. 12, 23

[47] M. T. Heath, *Scientific Computing An Introductory Survey*. McGraw Hill, 2 ed., 2002. 12

[48] A. Abed, F. Weinachter, H. Razik, and A. Rezzoug, “Real-time implementation of the sliding DFT applied to on-line’s broken bars diagnostic,” in *Electric Machines and Drives Conference, 2001. IEMDC 2001. IEEE International*, pp. 345 –348, 2001. 12

[49] L. N. Trefethen, *Spectral Methods in Matlab*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000. 13

[50] J. Hesthaven, S. Gottlieb, and D. Gottlieb, *Spectral methods for time-dependent problems*. Cambridge monographs on applied and computational mathematics, Cambridge University Press, 2007. 13

[51] E. Jacobsen and R. Lyons, “The sliding DFT,” *IEEE Signal Processing Magazine*, vol. 20, pp. 74–80, Mar. 2003. 19, 44, 46, 50

University of Cape Town

[52] E. Jacobsen and R. Lyons, “An update to the sliding DFT,” *IEEE Signal Processing Magazine*, vol. 21, pp. 110 – 111, January 2004. 19, 44, 46

[53] B. G. Sherlock and D. M. Monro, “Moving Discrete Fourier Transform,” *IEE Proceedings*, vol. 139, pp. 279–282, Aug. 1992. 19, 44, 46, 50, 52

[54] A. Brown, “Running Fourier Transforms,” *Electronics World*, p. 959, Nov. 1998. 19, 44, 46, 50, 64

[55] H. Kamei, T. Harada, and H. Kawarada, “A fast algorithm of running Fourier Transform,” *Electronics and Communications in Japan (Part I: Communications)*, vol. 71, no. 8, pp. 1–10, 1988. 19, 44, 46

[56] J.-H. Kim and T.-G. Chang, “Analytic derivation of the finite wordlength effect of the twiddle factors in recursive implementation of the sliding-DFT,” *IEEE Transactions on Signal Processing*, vol. 48, pp. 1485 –1488, May 2000. 19, 44, 65, 66

[57] K. Liu and C.-T. Chiu, “Unified parallel lattice structures for time-recursive discrete cosine/sine/Hartley transforms,” *IEEE Transactions on Signal Processing*, vol. 41, pp. 1357 –1377, mar 1993. 23

[58] N. Murthy and M. Swamy, “On the computation of running discrete cosine and sine transform,” *IEEE Transactions on Signal Processing*, vol. 40, pp. 1430 –1437, jun. 1992. 23

[59] A. V. Oppenheim and R. W.Schafer, *Digital Signal Processing*. Prentice Hall, Jan. 2002. 24, 27, 29, 34, 46, 50, 58, 61, 99, 103, 111, 112, 114

[60] S. Kuo and B. Lee, *Real-time digital signal processing: implementations, applications, and experiments with the TMS320C55X*. Wiley, 2001. 29

[61] G. Jullien and N. Wigley, “Implementing the Arithmetic Fourier Transform,” in *Signals, Systems and Computers, 1989. Twenty-Third Asilomar Conference on*, vol. 1, pp. 493 – 496, 1989. 31

[62] G. Sadasiv, “The arithmetic Fourier transform,” *ASSP Magazine, IEEE*, vol. 5, pp. 13–17, Jan 1988. 31

University of Cape Town

[63] D. Tufts and H. Chen, “Iterative Realization of the Arithmetic Fourier Transform,” *Signal Processing, IEEE Transactions on*, vol. 41, p. 152, jan 1993. 31

[64] N. Wigley and G. Jullien, “On implementing the arithmetic Fourier transform,” *Signal Processing, IEEE Transactions on*, vol. 40, pp. 2233 –2242, sep 1992. 31

[65] R. J. de Sobral Cintra and D. M. de Oliveira, “A Short Survey on Arithmetic Transforms and the Arithmetic Hartley Transform,” *Revista da Sociedade Brasileira de Telecomunica oes*, vol. 19, August 2004. 31

[66] J. W. Cooley and J. W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series,” *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. 33

[67] C. van Loan, *Computational frameworks for the fast fourier transform.* Frontiers in applied mathematics, SIAM, 1992. 33

[68] S. He and M. Torkelson, “A New Approach to Pipeline FFT Processor,” in *Proceedings of the 10th International Parallel Processing Symposium*, (Washington, DC, USA), pp. 766–770, IEEE Computer Society, 1996. 36

[69] B. M. Baas, “An Energy-Efficient Single-Chip FFT Processor,” in *In Symposium on VLSI Circuits*, pp. 164–165, 1996. 36

[70] R. Preuss, “Very fast computation of the radix-2 discrete Fourier transform,” *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 30, pp. 595–607, Aug 1982. 36

[71] J. Greene and R. Cooper, “A parallel 64K complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine processor,” in *Laboratory, University of Tennessee*, 2005. 36

[72] A. C. Chow, G. C. Fossum, and D. A. Brokenshire, *A Programming Example: Large FFT on the Cell Broadband Engine.* IBM, May 2005. 36

University of Cape Town

[73] M. Hegland, "Block Algorithms for FFTs on Vector and Parallel Computers," *Parallel Computing: Trends and Applications. Amsterdam, The Netherlands: Elsevier*, pp. 129–136, 1994. This bibtex comes from the cite in Discrete Fourier Transform on Multicore paper (bibtex 5230808). 36

[74] F. Franchetti, Y. Voronenko, and M. Püschel, "FFT Program Generation for Shared Memory: SMP and Multicore," in *Supercomputing (SC)*, 2006. 36

[75] L. H. Jamieson, P. T. Mueller, Howard, and J. Siegel, "FFT algorithms for SIMD parallel processing systems," *Journal of Parallel and Distributed Computing*, vol. 3, pp. 48–71, 1986. 36

[76] D. H. Bailey, "FFTs in external of hierarchical memory," in *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), pp. 234–242, ACM, 1989. 36

[77] L. Chen, Z. Hu, J. Lin, and G. R. Gao, "Optimizing the Fast Fourier Transform on a Multi-core Architecture," pp. 1–8, June 2007. 36

[78] P. D'Alberto, P. A. Milder, A. Sandryhaila, F. Franchetti, J. C. Hoe, J. M. F. Moura, M. Püschel, and J. Johnson, "Generating FPGA Accelerated DFT Libraries," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 173–184, 2007. 36

[79] J. M. Palmer, B. Y. University, J. M. Palmer, D. Michael, J. Wirthlin, D. Clark, N. Taylor, B. Y. University, D. M. Chabries, I. A. F. College, and J. M. Palmer, "The Hybrid Architecture Parallel Fast Fourier Transform (HAPFFT)," 2005. 36

[80] B. G. Sherlock, "Windowed discrete Fourier transform for shifting data," *Signal Processing*, vol. 74, no. 2, pp. 169 – 177, 1999. 46

[81] J. Firth, *Discrete transforms.* Chapman & Hall, 1992. 48

[82] D. Kar and V. Bapeswara Rao, "A CORDIC-based unified systolic architecture for sliding window applications of discrete transforms," *IEEE Transactions on Signal Processing*, vol. 44, pp. 441 –444, feb. 1996. 48

University of Cape Town

[83] R. Gray and D. Neuhoff, "Quantization," *IEEE Transactions on Information Theory*, vol. 44, pp. 2325 –2383, Oct. 1998. 55, 56

[84] V. Madisetti, *The Digital Signal Processing Handbook.* Boca Raton, FL, USA: CRC Press, Inc., 2nd ed., 2009. 64

[85] D. W. Allan, "Time and frequency (time-domain) characterization, estimation, and prediction of precision clocks and oscillators," in *IEEE Transactions on Ultrasonics, Ferroelectrics & Frequency Control*, vol. UFFC-34, pp. 647–654, Institute of Electrical and Electronic Engineers, Nov. 1987. 90, 91, 93

[86] D. A. Howe, D. W. Allan, and J. Barnes, "Properties of signal sources and measurement methods," in *Proceedings of the 35th Annual Symposium on Frequency Control, 1981*, pp. 1–47, Unknown, 1981. 91

[87] J. O. Smith, *Mathematics of the Discrete Fourier Transform (DFT).* http://www.w3k.org/books/: W3K Publishing, 2007. 112, 113

[88] J. Terry and J. Heiskala, *OFDM Wireless LANs: A Theoretical and Practical Guide.* Sams, December 2001. 120

[89] Xilinx, "7 Series FPGAs Overview (DS180 v1.8) - Advanced Product Specification." Online, September 2011. 169

[90] "CASPER Simulink Libraries (2011)." 175

[91] A. van der Byl, M. Inggs, and R. Wilkinson, "A Many Processing Element Framework for the Discrete Fourier Transform," in *Proceedings of the 2010 International Conference on Field-Programmable Technology*, pp. 425–428, IEEE, December 2010. 189

[92] Xilinx, "LogiCORE IP Complex Multiplier v3.1." Online, December 2009. 189

[93] Xilinx, "LogiCORE IP Fast Fourier Transform v8.0 (DS808)." Online, March 2011. 189

University of Cape Town

[94] Altera, “FFT MegaCore Function User Guide Version 11.0.” Online, May 2011. 189

[95] “FFT Benchmark Methodology.” Online (http://www.fftw.org/speed/method.html), Date last accessed 6 July 2012. 194

[96] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” in *FPGA ’06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, (New York, NY, USA), pp. 21–30, ACM, 2006. 199

University of Cape Town