NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

A COMPRESSION ALGORITHM FOR FIELD PROGRAMMABLE GATE ARRAYS IN THE SPACE ENVIRONMENT

by

Caleb J. Humberd

December 2011

Thesis Advisor:	Frank E. Kragh
Thesis Co-Advisor:	Herschel Loomis

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704–0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704–0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2011	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE A Compression Algorithm for Field Programmable Gate Arrays in the Space Environment			**5. FUNDING NUMBERS**
6. AUTHOR(S) Caleb J. Humberd			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943–5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ______N.A.__________.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

The focus of this thesis is a lossy Fourier-transform-based compression algorithm for implementation on field programmable gate arrays in the space environment. The algorithm computes the fast Fourier transform (FFT) of a real input signal, determines the energy in user-defined time and frequency ranges of interest, and transmits only those frequency domain portions of the signal that exceed the predefined thresholds. Error detection against single event upsets for the FFT is implemented by comparing the sum of the squares of the input to the scaled sum of the squares of the FFT output, which should be equal according to Parseval's Theorem. Error correction is implemented by duplicating the FFT calculation and error detection and choosing the output of the FFT that is not in error.

14. SUBJECT TERMS Fast Fourier Transform, Software Defined Radio, Field Programmable Gate Array, Compression, Lossy Data Compression, Fault Tolerant, Single Event Upset			**15. NUMBER OF PAGES** 111
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01–280–5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

A COMPRESSION ALGORITHM FOR FIELD PROGRAMMABLE GATE ARRAYS IN THE SPACE ENVIRONMENT

Caleb J. Humberd
Lieutenant, United States Navy
B.S., United States Naval Academy, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2011

Author: Caleb J. Humberd

Approved by: Frank E. Kragh
Thesis Advisor

Herschel H. Loomis, Jr.
Thesis Co-Advisor

Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The focus of this thesis is a lossy Fourier-transform-based compression algorithm for implementation on field programmable gate arrays in the space environment. The algorithm computes the fast Fourier transform (FFT) of a real input signal, determines the energy in user-defined time and frequency ranges of interest, and transmits only those frequency domain portions of the signal that exceed the predefined thresholds. Error detection against single event upsets for the FFT is implemented by comparing the sum of the squares of the input to the scaled sum of the squares of the FFT output, which should be equal according to Parseval's Theorem. Error correction is implemented by duplicating the FFT calculation and error detection and choosing the output of the FFT that is not in error.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
COTS	Commercial Off-the-Shelf
CW	Continuous Wave
DFT	Discrete Fourier Transform
DIF	Decimation in Frequency
DIT	Decimation in Time
DSP	Digital Signal Processing
DTFT	Discrete Time Fourier Transform
EDIF	Electronic Data Interchange Format
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GCLK	Global Clock Buffer
GUI	Graphical User Interface
HDL	Hardware Description Language
IC	Integrated Circuit
ICON	Interface Control
IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
ILA	Integrated Logic Analyzer
IO	Input / Output
IOB	Input / Output Block
I-Q	In-phase and Quadrature
ISE	Integrated Software Environment
JTAG	Joint Test Action Group (IEEE 1149.1)
LUT	Look-Up Table
MATLAB	Matrix Laboratory

MUX	Multiplexer
ORS	Operationally Responsive Space
RAM	Random Access Memory
RF	Radio Frequency
ROI	Range(es) of Interest
RPR	Reduced Precision Redundancy
SDR	Software Defined Radio
SECDED	Single Error Correction, Double Error Detection
SEU	Single Event Upset
SOI	Signal(s) of Interest
TCL	Tool Control Language
TMR	Triple Modular Redundancy
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VIO	Virtual Input / Output
XUP	Xilinx University Program

EXECUTIVE SUMMARY

Because of their high cost and long lead-time of development, manufacture, and launch, satellites are designed for a long service life in order to recoup investment. This means that the technology aboard operational satellites lags behind the state-of-the-art technology on the ground. In fact, a satellite may be rendered obsolete because the technology onboard has been surpassed by the technology of the ground infrastructure. In addition, the long lead-time results in delays in delivery of vital services to end-users. The Office of Operationally Responsive Space (ORS) is charged with developing strategies to mitigate these twin problems of obsolescence and untimeliness.

One of the methods for increasing the flexibility of communications satellites is through the use of software defined radio (SDR). Because the function of the radio is programmable, the satellite's software can be updated to keep up with the pace of technology on the ground. Field programmable gate arrays (FPGAs) are becoming the design choice for including the processing required to implement an SDR because they can approach the performance of application specific integrated circuits (ASICs) while exceeding the flexibility of general purpose processors (GPPs).

Computing in space has special considerations due to radiation effects. High energy radiation causes several degradations to computational hardware. Single-event upsets (SEUs) are caused when a high-energy particle deposits enough charge in a memory element to change its state. This has special significance to FPGAs used for onboard processing since the configuration of the calculation being conducted is stored in memory. FPGA applications must be designed to detect and correct errors caused by SEUs.

The traditional method to implement error correction is through triple modular redundancy (TMR), where the digital circuit is triplicated and a voter circuit decides the correct calculation based on a majority vote. In order to reduce the logic resources required for error correction, reduced precision redundancy (RPR) can be used to protect arithmetic calculations. In RPR, one full-precision result is calculated along with lower-

precision upper and lower bounds. The full precision calculation is deemed correct if it is between the bounds, otherwise the result returned is a lower-precision average of the two bounds.

Design for FPGA applications differs from writing programs for execution by GPPs. The focus of this thesis is the design flow of FPGA design, from high-level design entry and simulation through low-level functional simulation to hardware implementation. Xilinx System Generator, a high-level graphical design and simulation tool for use within The MathWorks' Simulink® environment, was used to design and test the algorithm. Once the design was behaving as desired, very high speed integrated circuit hardware description language (VHDL) code was generated by System Generator. This code was imported into the Xilinx integrated software environment (ISE) design suite for syntax checking and compilation. Mentor Graphics' ModelSim software was used with ISE for simulating the function of the algorithm defined by the HDL code.

The design of a lossy Fourier-transform-based compression algorithm for implementation on a FPGA-based SDR in the space environment is the focus of this thesis. The compression algorithm was designed by Wright and further refined by Livingston, both students at the Naval Postgraduate School. The algorithm computes the fast Fourier transform (FFT) of a wideband signal, divides it into user-defined time-frequency ranges of interest (ROI), and calculates the energy in those ROIs. If the energy exceeds the user-defined threshold, then the signal within that ROI is forwarded for downlink.

The FFT calculation that comprises the first stage of the compression algorithm is implemented using Xilinx FFT intellectual property (IP) blocks. The original algorithm designed by Wright uses the FFT v4.1 block and was targeted for the Virtex™-II Pro. In order to implement the algorithm on a Virtex™ FPGA, Livingston modified the algorithm to use the FFT v1.0 block. The performance of these FFT algorithms is examined and compared to the performance of the FFT v3.2 block, which is supported for implementation on Virtex™-II FPGAs.

The FFT used in the compression algorithm is a Xilinx, Inc. IP black box. TMR or RPR cannot be implemented directly within the FFT. Livingston implemented error detection by making use of Parseval's Theorem. Parseval's theorem states that the energy prior to the Fourier transform is equal to the energy after the transform. The sum of the squares of the input points to the transform is compared to the sum of the magnitudes squared of the output points scaled by the length of the FFT. If these values are not equal to within a pre-defined tolerance, the calculation is determined to be in error. This method of error detection was analyzed, and the algorithm was modified to be used with the FFT v3.2 block.

The previously designed error detection algorithm was used as the basis for an error correction algorithm. The error detection circuit was duplicated, and a voting circuit compared the error flags. The output of the FFT not flagged as in error was passed to the ROI analysis and data formatting portions of the compression algorithm. Adjustments were made to the error correction algorithm to reduce the logic resources required for implementation in order to allow the algorithm to be implemented in a Virtex™-II FPGA. Redundant sum-of-squares circuitry was removed from the input to the FFT blocks. The voting circuit was redesigned to implement delays required for aligning data with control signals with random access memory (RAM) rather than logic resources.

The error correcting circuit was tested by inserting a circuit into the output of one of the FFTs which after a set period of time would switch from the correctly calculated FFT points to an erroneous constant. It was shown that before the error correcting circuitry, erroneous values were present at the output of the FFT. After the error correcting circuitry, it was shown that the values present in the output were corrected. The process of iterative design of the error correcting and voting circuits to fit within the logic resources of the target FPGA validated the use of System Generator as a rapid prototyping, high-level design tool.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

There are so many to whom I owe a debt of gratitude for the help and support you lent me during my research and writing of this thesis:

To Professor Frank Kragh, for introducing me to the world of software radio. Thanks especially for your encouragement during the final push to get this thesis written.

To Professor Hersch Loomis, for your help with understanding reconfigurable processors and fault tolerant computing.

To Donna Miller and Ron Aikins, for your help in the Communications Research Lab, and with my numerous computer questions on both sides of the campus.

To my fellow Space Systems Engineering students, for helping me through a course of study including mechanics, structures, thermodynamics, and a whole host of subjects foreign to me.

To my parents, Mary and Charlie, for inspiring in me from a young age (and still to this day) a sense of curiosity and love of learning.

To my loving ~~fiancé~~ wife, Nichole, for putting up with the nights and weekends I spent on this thesis – and still marrying me!

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Satellites are highly complex machines that have long acquisition lead times. Due to the high costs of design, construction, and launch, satellites tend to have long design lives and are often pressed to continue service well beyond their intended life. Satellites in service may be ten, fifteen, or even twenty years old; however, technology continues to advance. The technology aboard otherwise operational satellites may become obsolete because the ground infrastructure has progressed beyond the envisioned capabilities when the satellite was launched [1].

This is a concern for the Department of Defense because the long lead-time in designing, building, and launching spacecraft means that satellites are not available to respond to new demands from warfighters. Operationally Responsive Space (ORS) is a strategy, presented in [1], to provide advanced space-based technology in a more timely manner. ORS provides solutions in three categories: tier 1 involves the re-use of on-orbit assets, tier 2 comprises the rapid launch of existing spacecraft using commercial off-the-shelf (COTS) parts, and tier 3 is the rapid development of new spacecraft designs.

Flexibility of the payload is one design strategy to implement tier 1 solutions. Reconfigurable payloads also allow on-orbit troubleshooting and re-engineering in the event of spacecraft damage or design flaws, mitigating risk of spacecraft mission non-completion. On-board processing, specifically the signal processing and routing used in software defined radio (SDR) applications, provides the flexibility to keep pace with the advance of communication technology in ground-based systems without launching a new satellite [2], [3].

Wright described an FPGA-based SDR for space applications in [4]. His design consisted of a Fourier transform-based bandwidth reduction algorithm with user-defined time-frequency parameters for extraction of signals-of-interest (SOI) from a wide-bandwidth front-end. This design was further refined by Livingston in [5], making it more resource-efficient, splitting the design for a multiple field programmable gate array (FPGA) implementation as well as adding error detection. In this research error correction is added, building on the previously designed error detection as well as continuing development of the multiple FPGA design.

A. OBJECTIVES

The objectives of this research are twofold. First is to continue progress on the existing design. The previous design effort concluded with two algorithms: one that was designed for a three-FPGA implementation and one that incorporated error detection. In this research, error correction, based on the previously designed error detection algorithm, was implemented.

The second objective is to explore the design process for rapid code iteration of an FPGA design using available design tools to implement a signal processing application on an FPGA and utilizing fault-tolerant design principles. The single-chip design was implemented from graphical design and simulation through hardware description language (HDL) design and simulation, and ultimately to hardware implementation and verification.

B. DESIGN PROCESS

FPGA design generally takes place using high-level design tools such as block diagrams and schematics. From this high-level concept, the design is developed in a HDL such as VHDL (very high speed integrated circuit [VHSIC] HDL) or Verilog. Unlike a high-level language for general-purpose processors, where commands are executed sequentially, an HDL specifies the interconnections between logic cells within the FPGA where logic level signals are processed in parallel. This parallelism is what lends FPGAs to real-time signal processing applications; however, it also renders HDL designs more difficult to program and debug and even more difficult to decipher (if not well commented) than sequentially-executed computer software. There are design tools to mitigate these drawbacks to HDL designs. Code modularity is one such tool. The Xilinx Integrated Software Environment (ISE) uses the terminology of “cores” to describe these pre-built modules that can be integrated into a design [6].

High-level design tools have also begun to be used for industrial design. By implementing design ideas at a more abstract level, the iterative process of improving and fielding the design can be accelerated while at the same time reducing the instance of syntax and code-level implementation errors. Because this method of algorithm design is more abstract than design in an HDL, it is possible to build a design that cannot be implemented in hardware. For this reason, it is a design tool that cannot be used alone [6].

The design was first modeled using the graphical method described in the previous paragraph in Simulink® using the System Generator plugin from Xilinx. Once the design was working in the Simulink® environment, the design was compiled in VHDL. This VHDL instantiation was then added to a Xilinx ISE project, where it was integrated with other VHDL modules and cores and targeted to a specific FPGA, in this case a Virtex™-II Pro. The VHDL project was then simulated using ModelSim. Penultimately, the ISE project was compiled to bitcode, which defines the resulting FPGA configuration. This bitcode is finally loaded onto the target FPGA. The target FPGA for this research is the Virtex™-II Pro XC2V30 on the Xilinx University Program (XUP) development board, with onboard universal serial bus (USB) Joint Testing Action Group (JTAG), codified as IEEE 1149.1. JTAG is the industry-standard debugging interface for printed circuit boards and internal sub-blocks of integrated circuits (ICs) [6].

C. THESIS ORGANIZATION

A presentation of Fourier analysis, including the development of the fast Fourier transform is given in Chapter II, as well as properties of Fourier transforms which are useful to the development of a compression algorithm. The concept behind a data compression algorithm is presented. Radiation effects, especially single-event upsets (SEUs), are presented, the special implications to FPGA designs are discussed, and strategies used implement fault tolerance, error detection, and error correction are presented.

Chapter III, Development Environment, is an overview of the software tools used for this design. The general design flow for the development of an FPGA application is presented along with why each tool was chosen for each design process step. The strengths of each tool are explained, and the limitations are discussed. Also addressed are software setup and use for this design process.

Chapter IV, Fast Fourier Transform (FFT) Computing, contains an investigation of the Xilinx FFT intellectual property (IP) blocks. Previous analysis of FFT v4.1 and FFT v1.0 are reviewed, and FFT v3.2 is investigated as a replacement for FFT v4.1 for implementation on a Virtex™-II FPGA.

Chapter V, Error Detection, is a review of the design work conducted in [5] to add an error detection circuit to the FFT IP blocks. A modification to the design in which FFT v3.2 is substituted for FFT v4.1 is presented.

A design that corrects errors produced by FFT IP blocks is presented in Chapter VI. Several iterations of the design are presented, with a focus on logic resource minimization.

Conclusions from this design work, as well as recommendations for future work to continue to refine both this design as well as the design process, are presented in Chapter VII.

II. BACKGROUND

The compression algorithm discussed in this thesis is based on the Fourier transform of the received signal. It allows the user to define specific time and frequency ranges of interest in order to discard information deemed not important. Fault tolerance is implemented using the principles of triple modular redundancy (TMR) as well as a property of Fourier transforms defined by Parseval's theorem.

A. THE FOURIER TRANSFORM

1. Fourier Analysis

The basis of the compression algorithm is the Fourier Transform. After [5] the relationship between the time domain and frequency domain of a continuous signal is

$$X(f) = FT\{x(t)\} = \int_{-\infty}^{+\infty} x(t)e^{-j2\pi ft}dt\,, \tag{II.1}$$

where $X(f)$ is the Fourier transform of $x(t)$.

The time domain signal is sampled by multiplying by an infinite series of impulse functions separated by the sample interval T_s. In the frequency domain this sampled signal is a convolution of the original signal with an infinite series of impulse functions separated by the sample frequency $f_s = 1/T_s$. The sampled time domain signal is $x[n] = x(nT_s)$, where n is an integer. The discrete time Fourier transform (DTFT) is defined in [8] as

$$X(f) = DTFT\{x[n]\} = \sum_{-\infty}^{+\infty} x[n]e^{-j2\pi fnT_s}\,. \tag{II.2}$$

The time domain signal $x(t)$ exists for all time, from $-\infty$ to $+\infty$. A time-limited signal is defined as the previous signal multiplied by a rectangle function $\mathrm{rect}\left(\frac{t - T_o/2}{T_o}\right)$. The rect function is defined as

$$\mathrm{rect}(u) = \begin{cases} 1 & if\ |u| < 1/2 \\ 0 & otherwise \end{cases}\,. \tag{II.3}$$

This multiplication in the time domain corresponds to convolving the frequency domain signal by the $\operatorname{sinc}(fT_0)$ function. The sinc function is defined as

$$\operatorname{sinc}(u) = \begin{cases} 1 & if\ \ u = 0 \\ \dfrac{\sin(\pi u)}{\pi u} & otherwise \end{cases} . \tag{II.4}$$

The number of samples included in the time window $N = T_o / T_s$ is an integer. Limiting the transform to this time-limited window gives the Discrete Fourier Transform (DFT) [8],

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-jk(2\pi/N)n}, k = 0,1,..,N-1, \tag{II.5}$$

where $k = fN / f_s$.

A summary of Fourier Theory is shown in Figure 1.

Properties of Fourier transforms that are of consideration for a compression algorithm are the properties of conjugate symmetry and the reversibility of the transform. Conjugate symmetry implies

$$X[k] = X^*[N-k] \tag{II.6}$$

for real $x[n]$ input. The other property of use is that the inverse DFT (IDFT)

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{jk(2\pi/N)n}, k = 1,2,...,N-1 \tag{II.7}$$

can be calculated [8].

One property that can cause an undesirable effect is the fact that multiplication in the frequency domain transforms to circular convolution in the time domain,

$$X[k]Y[k] \overset{\mathcal{F}}{\rightleftharpoons} x[n] * y[n]. \tag{II.8}$$

This property must be considered during bin analysis since the frequencies of interest are selected by multiplying by a rectangle function. During decompression this multiplication is transformed into a periodic convolution with a sinc function, which causes distortion of the data if not compensated for [5], [8].

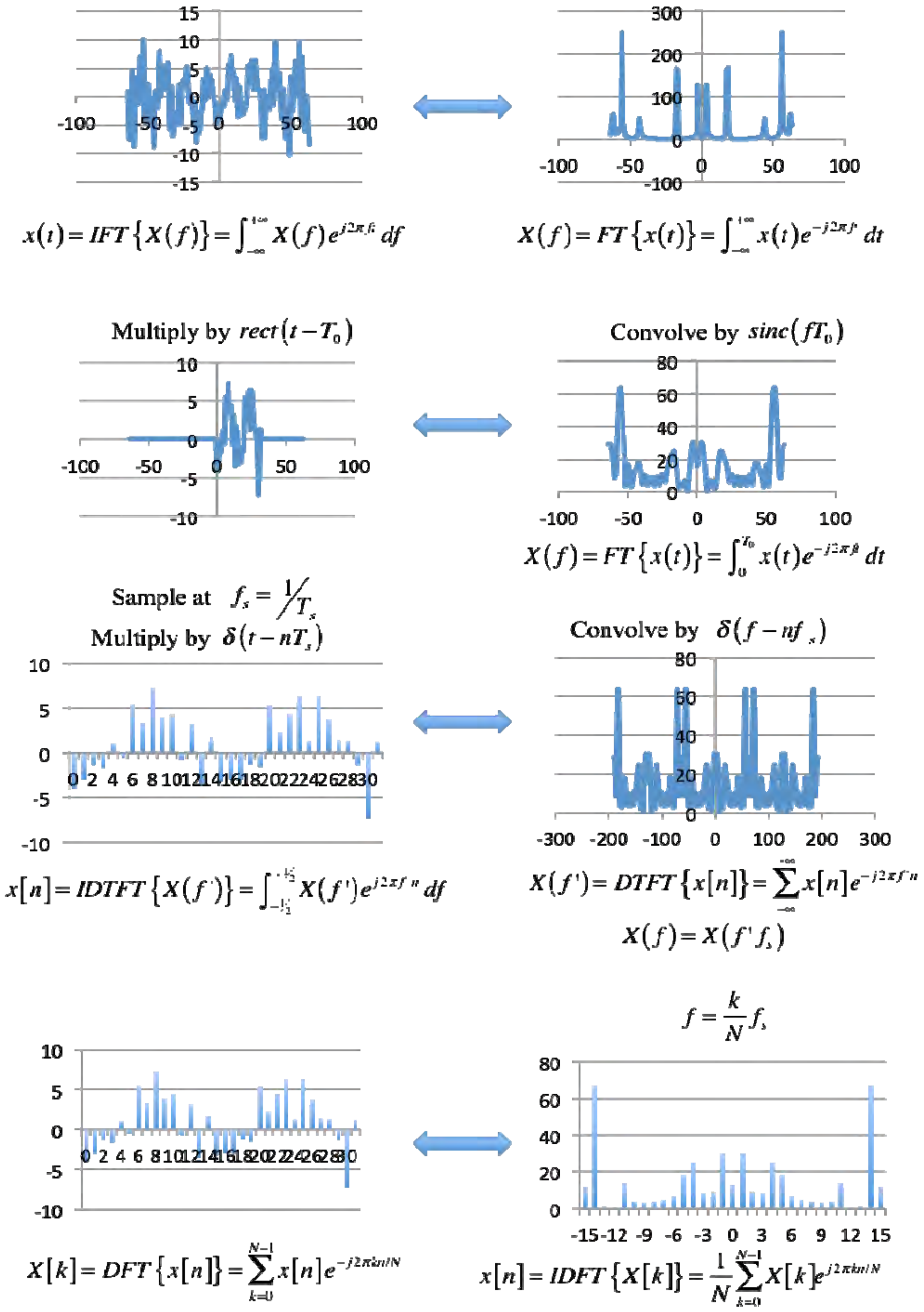


Figure 1. Fourier Theory (After [8]).

One other property of Fourier transforms that can be used to implement error detection is the inner product property, also known as Parseval's theorem [5]:

$$\sum_{n=0}^{N-1} x^*[n]y[n] = \frac{1}{N}\sum_{k=0}^{N-1} X^*[k]Y[k]. \tag{II.9}$$

From this equation we see that the energy of the signal prior to the transform is equal to the energy of the signal after the transform scaled by the factor $1/N$. By using this property of Fourier transforms, we can relate the input and output of a transform in a less computationally intensive manner. In [5], Equation II.9 is manipulated into the form

$$\frac{1}{N}\sum_{n=0}^{N-1} \mathrm{Re}^2\{x[n]\} = 2\sum_{k=0}^{N/2-1}\left(\frac{\mathrm{Re}\{X[k]\}}{N}\right)^2 + \left(\frac{\mathrm{Im}\{X[k]\}}{N}\right)^2 \tag{II.10}$$

which can be implemented computationally.

2. The Fast Fourier Transform

The DFT is a discrete input, discrete output function and is suitable for computation. As discussed in [7], direct computation of the DFT requires $O(N^2)$ computations. By taking advantage of symmetries within the DFT, we can reduce the number of required computations to $O(N\log_2 N)$ with a class of algorithms known as fast Fourier transforms (FFT).

To simplify notation when discussing the DFT and FFT, the phase factor, or "twiddle factor," is written

$$w_N = e^{-j(2\pi/N)}. \tag{II.11}$$

The DFT has symmetry about $N/2$ such that

$$w_N^{k+N/2} = -w_N^k. \tag{II.12}$$

The DFT also has periodicity of N such that

$$w_N^k = w_N^{k+N}. \tag{II.13}$$

For $N = 2$ and substituting the phase factor given by Equation (II.11) into the DFT given by Equation (II.5) we get

$$X[k] = \sum_{n=0}^{1} x[n] w_2^{nk} = x[0] w_2^{0k} + x[1] w_2^{1k} \qquad \text{(II.14)}$$

where

$$w_2 = e^{-j(2\pi/2)} = e^{-j\pi} = -1, \qquad \text{(II.15)}$$

$$w_2^{0k} = (-1)^0 = 1, \qquad \text{(II.16)}$$

and

$$w_2^{1k} = (-1)^k . \qquad \text{(II.17)}$$

For $k = 0...1$, the resulting DFT points are

$$\begin{aligned} X[0] &= x[0] + x[1] \\ X[1] &= x[0] - x[1] \end{aligned} . \qquad \text{(II.18)}$$

The signal flow graph, or "butterfly" operator, shown in Figure 2 represent the resulting operation.

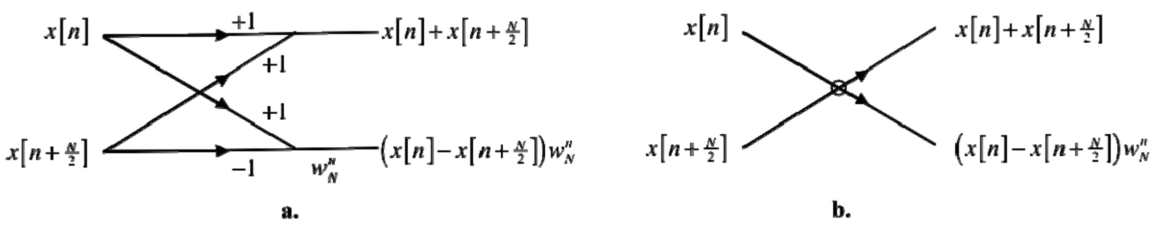


Figure 2. Radix-2 FFT signal flow graph (a.) and shorthand notation (b.) (After [8]).

Similarly, as discussed in [7], the radix-4 FFT can be calculated by dividing the input into four summations. The radix-4 FFT achieves a 25 percent reduction in the number of complex multiplies required over the radix-2 FFT. The signal flow graph of a radix-4 FFT is shown in Figure 3.

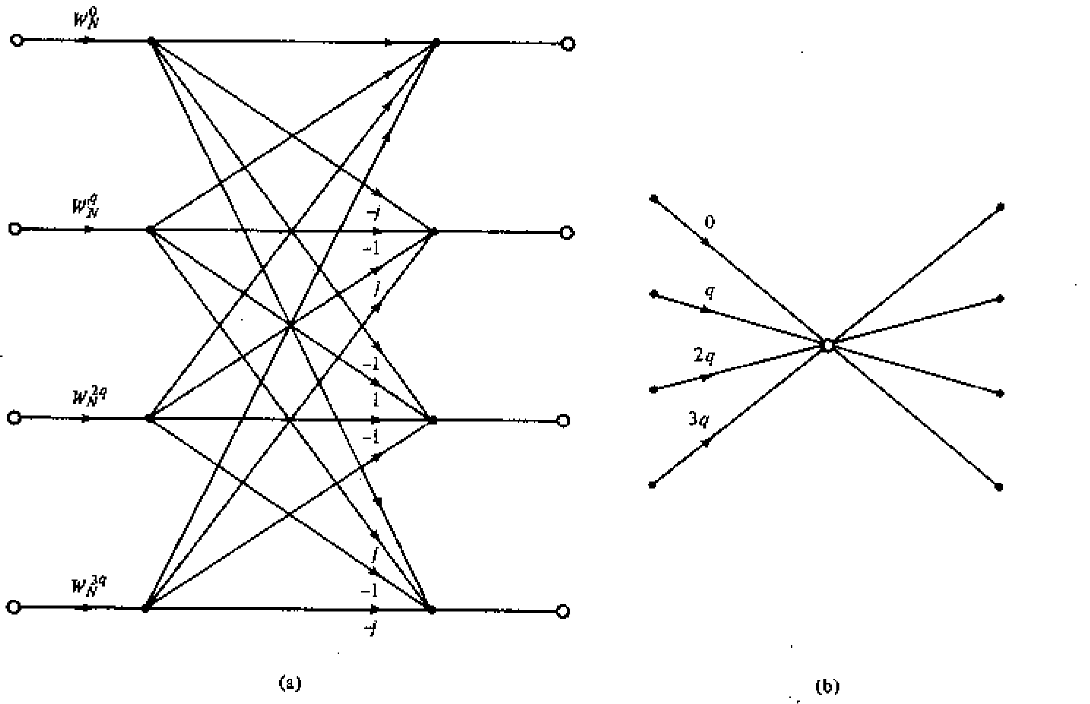


Figure 3. Radix-4 FFT signal flow graph (a.) and shorthand notation (b.) (From [7], [9]).

The FFT algorithm can be used for any signal with a length that is a power of two. To do so, the butterflies are cascaded. An eight-point, radix-2, decimation in time (DIT) FFT is shown in the signal flow graph in Figure 4.

When implementing the FFT in software or in an FPGA, the algorithm can be initiated with either in-place addressing or constant geometry. For constant geometry addressing, shown in Figure 4, the output of the butterfly is written into the same memory locations that the input was read from. This results in either the input or output points' memory locations being out of order in memory. In normal order addressing, shown in Figure 5, the input and output values are in order in memory. However, since the butterfly operators cannot write to the same memory locations from which they read their input values, this geometry requires additional memory resources to implement [5], [8].

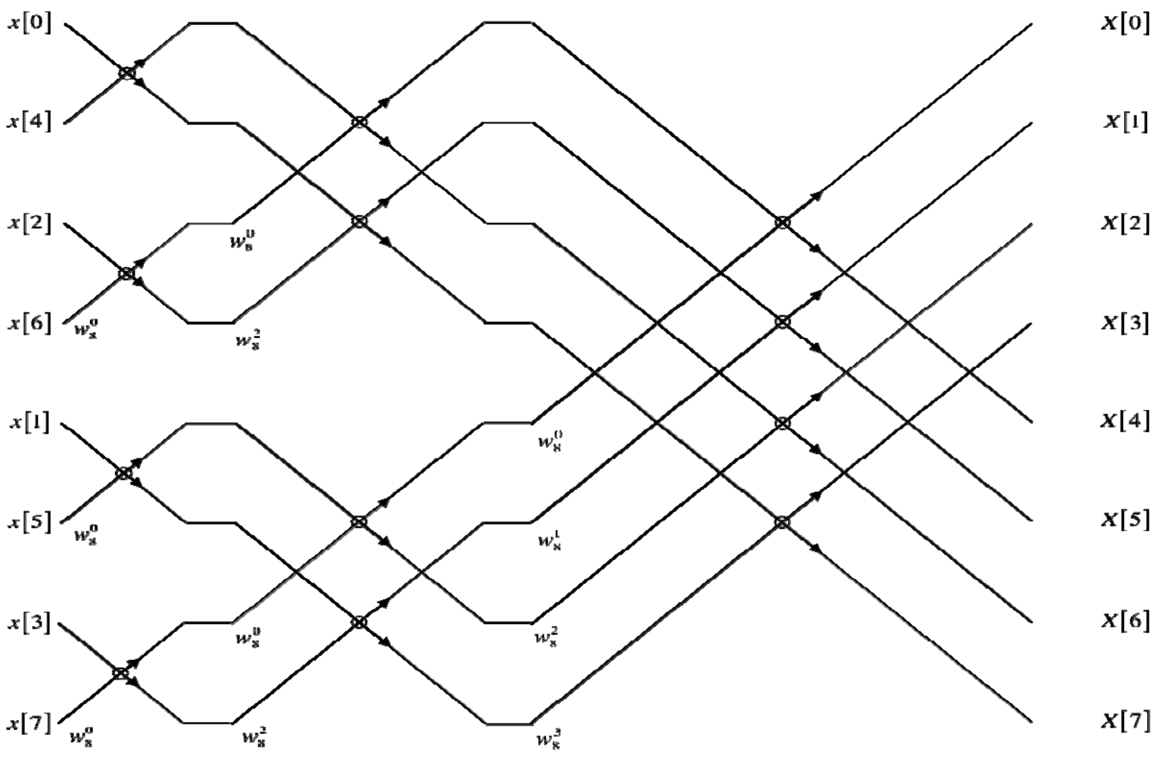


Figure 4. Eight point constant geometry decimation in time (DIT) FFT (After [8]).

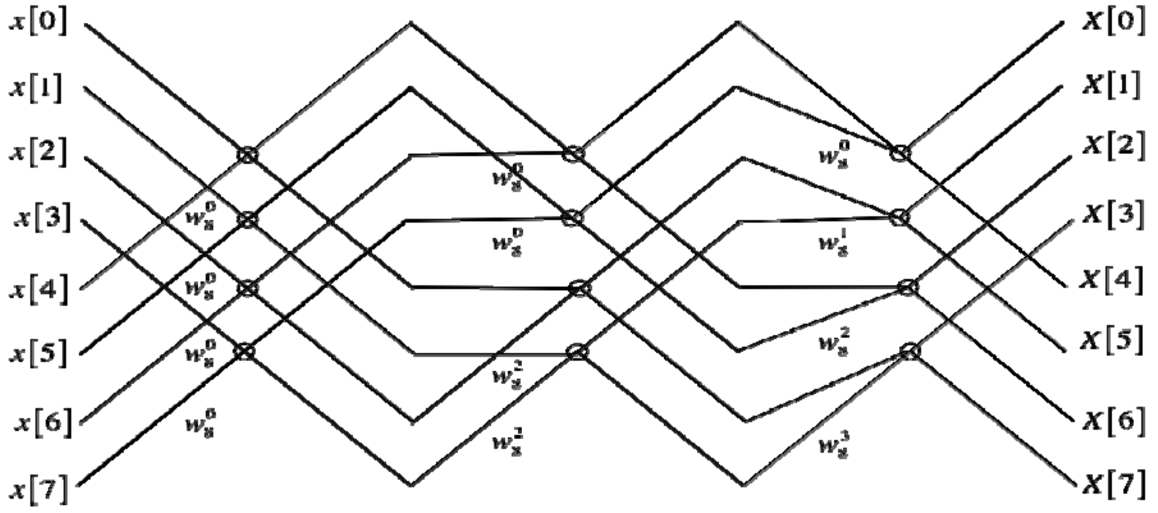


Figure 5. Eight point normal order DIT FFT (After [8])

B. COMPRESSION

The bandwidth of a SOI may be much less than the passband of the radio frequency (RF) front-end of a wideband digitizing SDR. In addition, the signal-of-interest may not be a continuous wave (CW) signal but pulsed or otherwise interrupted periodically such that there are periods of silence in the SOI. Therefore, the SOI can be represented by fewer bits than if the entire passband is digitized. In order to achieve this reduction in the downlink data rate, the received and digitized signal is divided into user-defined time and frequency ranges of interest, referred to as “bins.” The energy present in each bin is compared against an operator-specified threshold, and the points representing the signal of interest are only downlinked when bin energy exceeds the threshold. This method for reducing downlink data rate was presented in [4], and a conceptual representation of this is illustrated in Figure 6. Shown in the first plot is a time-varying signal. The colored contours represent the energy contained in the signal as a function of time and frequency. In the second plot, the red boxes denote the user defined time-frequency ranges of interest (ROI) overlaid on the signal. The third plot shows the signal that would be included in the downlink in the red boxes, while the blue box indicates a bin of interest in which the energy did not meet the required threshold and no data is downlinked.

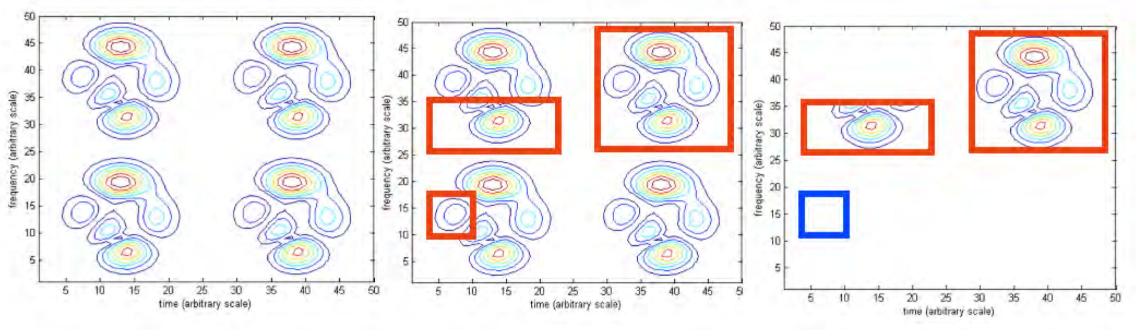


Figure 6. Compression (After [4]).

The algorithm developed by Wright and presented in [4] to conduct this compression is illustrated in Figure 7. The FFT of the time domain signal is calculated, and the frequency domain points are passed to the bin energy calculation and bin energy threshold algorithms. The data management block then reads the FFT points in the bins meeting the energy threshold and formats the data points for downlink. There are four bins available with user defined time and frequency ROI.

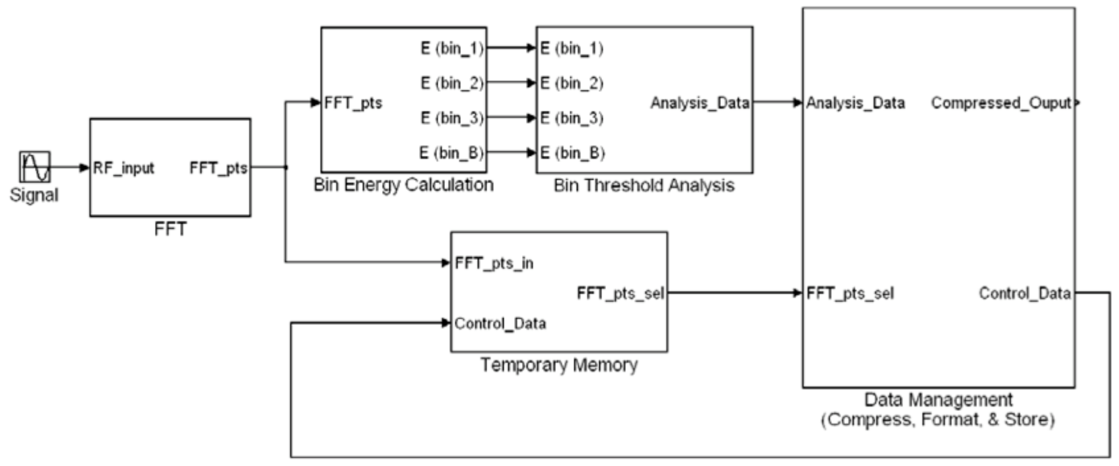


Figure 7. Compression Algorithm Block Diagram (From [4]).

C. COMPUTING IN THE SPACE ENVIRONMENT

As discussed in [10], choices for implementing onboard processing include general-purpose processors (GPPs), application-specific integrated circuits (ASICs), and FPGAs. For performing the same calculations on streams of data, such as real-time digital signal processing (DSP), an ASIC provides the most performance to power consumption of these alternatives. Their downside is that they require extensive development before they are produced, and they cannot be modified beyond their intended purpose. GPPs are the most flexible option; however, their strength lies in sequential operations in which the calculations performed on the data differ from one operation to the next. Their throughput for performing the same calculation on streaming data is lower than ASICs for the same clock speed. FPGAs offer the capability to process streaming data as efficiently as an ASIC with the ability to modify the application for which they are being used [10].

1. Fault Detection

FPGAs come at the cost of higher power consumption as well as being susceptible to their programming being altered by single-event upsets (SEU) due to high-energy particles present in the space environment. FPGA application design for space applications must take these issues into consideration. A SEU occurs whenever a high-energy particle impacts the semiconductor material and deposits enough charge to change the state of a single bit. The effect of this unintended state change depends on whether the affected bit is in data memory is a data bit in the midst of the calculation circuitry or is part of the FPGA configuration memory [10].

Parseval's theorem states that the energy into the DFT must equal the energy out. Because of this property, it is possible to detect whether or not the calculation of the DFT contains an error. In [5], Livingston implemented error detection for a FFT through application of Equation (II.10). In this application the accumulated (summed) squared input points to the DFT are compared to the accumulated squares of the output points, scaled by $1/N$. If the input to the DFT is limited to a real, as opposed to in-phase and quadrature (I-Q) signal, then the output only has to accumulate squares of the first half of the output points and scale by $2/N$. This saves memory requirements as well as latency in the error detection.

2. Fault Correction

Electronics in the space environment must be designed to tolerate high-energy particle radiation as discussed in [11]. Some of this tolerance must be implemented at the physical level: shielding is used to reduce the incidence of radiation on the electronics, and specialized semiconductor design techniques are used to mitigate the long- and short-term effects of radiation on the electronics. Beyond these methods, the algorithm implementation in software and hardware must also be made radiation tolerant.

Errors in data memory are usually checked and corrected using a parity scheme, such as a single error correction, double error detection (SECDED) Hamming code. When data is read into memory, parity check bits are calculated and stored in a separate memory location. When the data is read out, parity is calculated again and compared against the original parity bits. If the parity check bits are not equal, an error occurred, and if a single error, the check bits can be decoded to point to the location of the error [12], [13].

Configuration errors must be detected and corrected by a circuit which itself may be in error. This has been done traditionally through triple modular redundancy (TMR). In TMR, the calculation is performed three times, the results are compared, and any erroneous result is discarded. The price of TMR is more than three times the required logic resources, since the calculation circuitry must be triplicated, and a voting circuit is required to check the results. The voting circuit shown in Figure 8 compares the result of each bit from each of the three calculations. This circuit can correct any one error from the inputs and can detect when the voting circuit itself is in error [12].

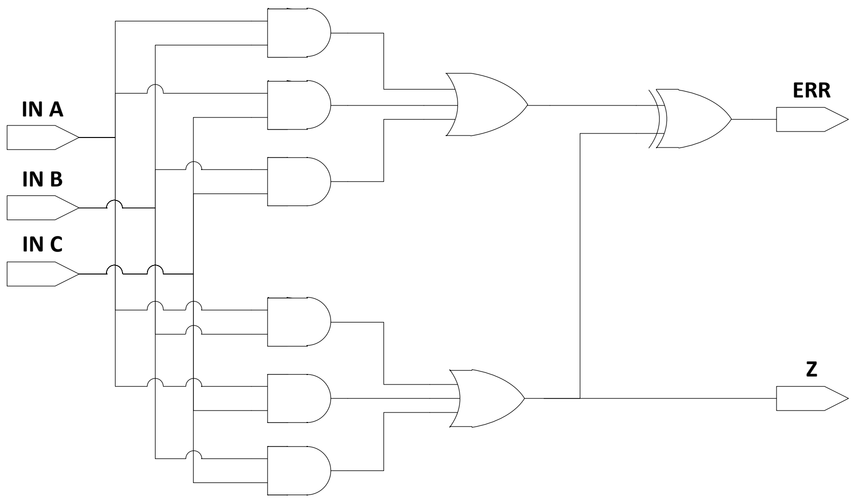


Figure 8. Bitwise Majority Voter (After [14]).

A method of redundancy requiring fewer resources, Reduced Precision Redundancy (RPR), was discovered by Snodgrass and presented in [12], and expanded upon by Sullivan in [14]. In RPR the full precision result is calculated once, while a lower-precision upper and lower bound are calculated. If the full precision calculation is outside of the calculated bounds, it is assumed to be in error and the reduced precision bound is returned. This technique results in lower logic resource use and power consumption at the cost of a more complex voting circuit as well as a less precise result when the full precision calculation is found to be in error. Reduced precision redundancy is also limited to arithmetic processes. Logic control operations must be protected using TMR.

D. TARGET FIELD PROGRAMMABLE GATE ARRAYS

This algorithm is targeted for the Virtex™ family of FPGAs manufactured by Xilinx. A summary of the devices considered is shown in Table 1. These devices are fully described in [15], [16], [17], and [18].

Table 1. Target FPGAs.

Family	**Device**	**Configurable Logic Blocks**	**Multipliers**	**Block RAM**
Virtex™	xcv1000	6144	0	32 (512 Byte)
Virtex™-II	xc2v3000	3584	96	96 (18 kB)
Virtex™-II Pro	xc2vp30	3424	136	136 (18 kB)
Virtex™-4	xc4vlx25	2688	48 (Xtreme DSP blocks)	72 (18 kB)

The devices considered include the older Virtex™ and Virtex™-II devices, which are in use on legacy space platforms. The newer Virtex™-4 device is considered because it is also available in a radiation hardened version. The Virtex™-II Pro device is included for comparison since that device was the one available for this research.

The oldest device, the xcv1000, has the most configurable logic blocks (CLBs). This device does not include the embedded multipliers, which degrades its ability to perform DSP calculations such as the FFT. This device can conduct algorithms that require multiplication; however, the multipliers must be constructed from the available

CLBs. Also, this device has the least amount of available block random access memory (RAM), which significantly limits the amount of data that can be processed at one time [15].

The newer devices all contain fewer CLBs but include embedded 18x18 bit multipliers, reducing the demand for CLBs. These devices also include significantly more RAM than the Virtex™. This increased on-chip RAM allows faster processing since algorithms do not need to make as many off-chip data accesses. The Virtex™-4 has the fewest number of embedded multipliers; however, they are arranged in XtremeDSP blocks in which the 18x18 bit multiplier is followed by a 48-bit accumulator. The Virtex™-II Pro includes a PowerPC® core, which was not used in this research [16], [17], [18].

E. SUMMARY

Several high level concepts that support the development of a Fourier transform-based, fault tolerant compression algorithm were introduced in this chapter. A summary of Fourier analysis was discussed, and the development of the fast Fourier transform was presented. Challenges associated with spaceborne computing were discussed, and the Triple Modular Redundancy and Reduced Precision Redundancy methods for correcting errors caused by single event upsets were presented. A number of design tools that enable the designer to develop the design at a high level, and then remove the layers of abstraction down to implementing the compression algorithm in hardware are introduced in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DEVELOPMENT ENVIRONMENT

Design for FPGAs is quite different and distinct from writing a sequential computer program. The tools used for development of FPGA applications are designed to allow the designer to control the sequential and parallel behavior of the logic. A design to be implemented on an FPGA is mapped out by the designer as a block diagram, flow chart, or schematic. From this high-level abstraction of the desired behavior of the design, the designer develops components of the circuit in an HDL. Design in an HDL allows full control over the hardware-level behavior of the algorithm while still allowing abstractions that aid the designer, such as human-readable variable names. This control at the hardware level comes at the price of a more difficult to understand design. Unless the HDL files are well documented, anyone other than the original designer may never be able to understand the purpose of the design [6].

To increase the ease and speed with which a design can be developed and altered, high-level design tools have become popular. With these tools, HDL components and their interconnections are represented graphically. As discussed in the introduction to [6], abstracting away the complexities of an HDL allows the designer to focus on the function of the design. The analogy is that a programmer writing a DSP algorithm will code most of the algorithm in a high-level language and only code portions with strict performance requirements in assembly code. In the same fashion, once the behavioral design is complete, the high-level design tool outputs a lower-level instantiation of the design for functional development, simulation, and eventual implementation in hardware. At each step in the design process, results of the verification are used to refine the initial design. This design flow, from high level algorithm development, HDL-level implementation, and hardware implementation is illustrated in Figure 9, along with the design tool used in that step.

This chapter is an overview of the design tools used for this research. An overview of the software packages, their features, and their operation are presented. Also discussed are the relative advantages as well as the potential pitfalls of each software package. The design tools discussed are summarized in Table 2 at the end of the chapter.

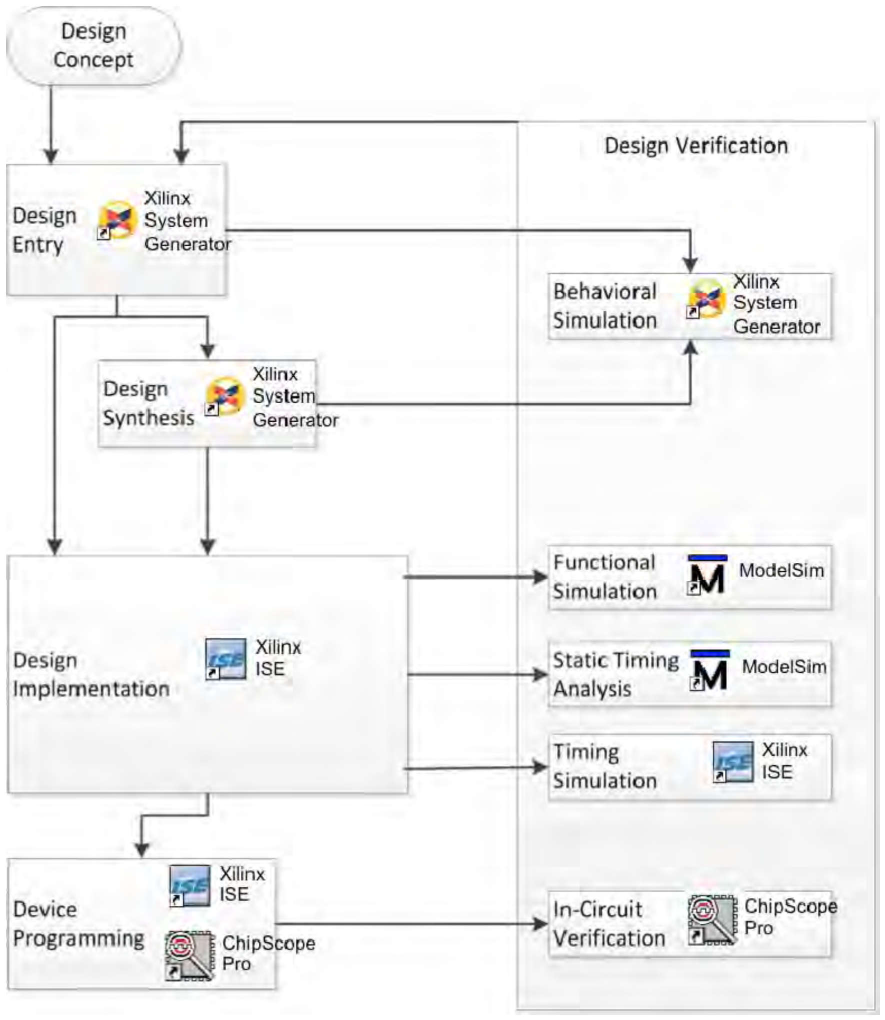


Figure 9. FPGA Design Process, and Associated Tools (After [6]).

A. MATLAB / SIMULINK

MATLAB (Matrix Laboratory) is a high-level numerical analysis oriented programming language developed by The Math Works®. Simulink® is a graphical modeling and simulation application which works within the MATLAB environment.

The primary use of MATLAB for this design was the generation of input signals and the analysis and display of the output. MATLAB is designed for numerical analysis and the manipulation of large arrays of numbers. MATLAB scripts, or M-files, were the primary means for setting up environment variables. The plotting tools were used for generating displays of input and output. In multiple FPGA instantiations of the algorithm, M-files were also used to script the execution of the separate models and control the routing of signals from one module to the next, simulating the function of the common backplane in hardware [5].

Models built in Simulink® are useful for investigating the behavior of applications destined for FPGA implementation; however, the timing of signals through the model is not accurate. Simulink® allows models to be run in discrete or continuous time; however, this design process made use of only discrete time modeling, where each time step represents one clock cycle of the FPGA.

B. XILINX SYSTEM GENERATOR FOR DSP

Xilinx System Generator is a plugin for Simulink, which adds the functionality to develop applications for Xilinx FPGAs using the high-level, graphical modeling and simulation environment of Simulink. The modeling environment uses Simulink® and MATLAB to generate input signals, pass the signals into the System Generator model, and collect output signals for post-processing and display [19].

An example System Generator design is shown in Figure 10. The plain white blocks are standard Simulink® blocks, while the blocks with the Xilinx “X” logo are System Generator blocks. The Gateway In and Gateway Out blocks represent bonded Input / Output Blocks (IOB) on the FPGA, and all blocks between the input and output represent functional segments of the user’s choice of HDL. The stand-alone System Generator block controls the instantiation and compilation of the block diagram into HDL [19].

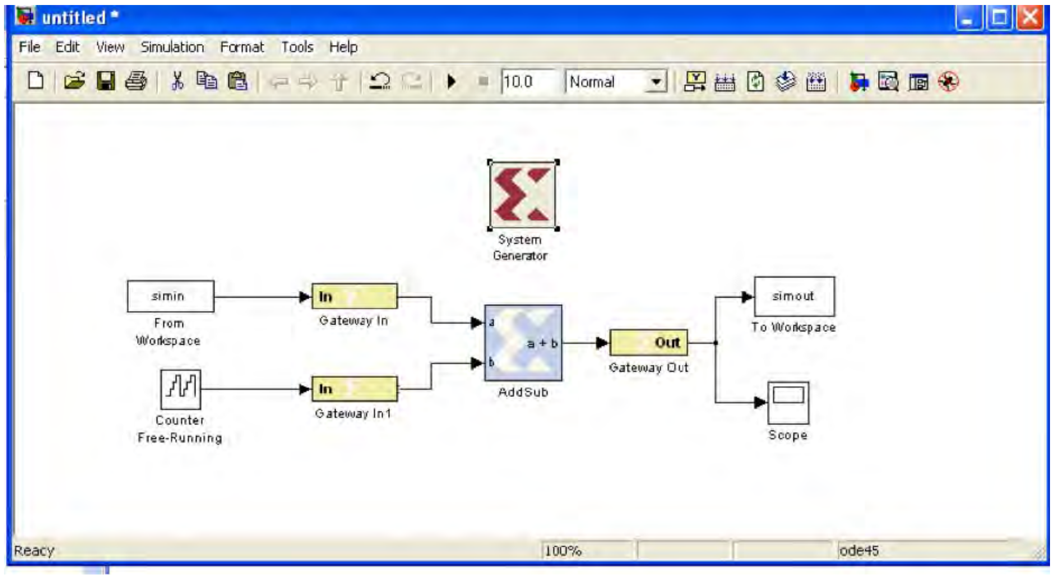


Figure 10. Example System Generator Design in Simulink

Simulation of applications for FPGA implementation using System Generator is more faithful to the actual behavior of the algorithm than a simulation using just Simulink. The reason is that the System Generator blocks represent segments of HDL code with the attendant constraints. System Generator outputs HDL which can then be modified and compiled with FPGA synthesis tools [19].

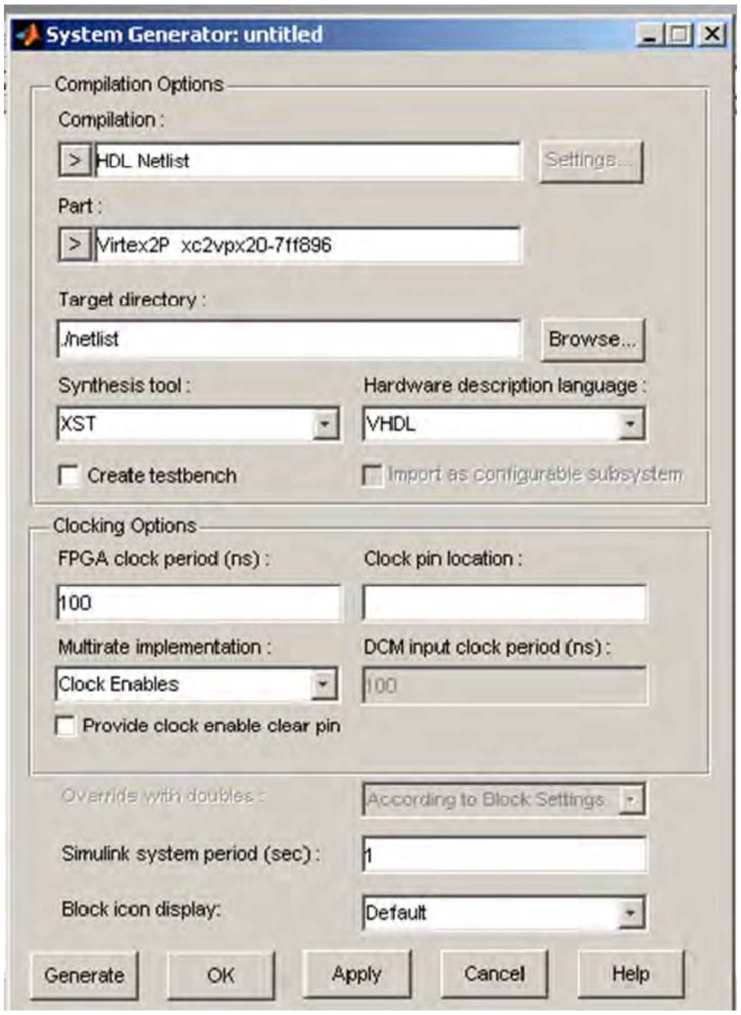
The version of System Generator used is version 10.1 because this is the last version that supports the original Virtex™ devices which are still in use in certain applications. As discussed in [20], this version of System Generator requires MATLAB R2007a / Simulink®6.6 or MATLAB R2007b / Simulink® 7.0. This version also requires Xilinx ISE version 10.1.

System Generator for DSP contains many blocks that are specific functions for DSP. The blocks under consideration for this design were the FFT v1.0, FFT v4.1, and FFT v3.2 blocks, discussed in Chapter IV. These pre-built blocks take advantage of specific on-chip DSP-specific logic resources (such as the XtremeDSP slices in the Virtex™-4). These blocks make use of Xilinx intellectual property (IP) core generation algorithms to optimize the performance of the design for the chosen FPGA [19].

If the desired function block does not exist, the MCode block may be used to implement the desired function. MCode blocks allow the insertion of a MATLAB M-file into a System Generator design to allow scripting and control of signal flow. This gives the designer a higher-level way to implement control over the logic, rather than designing a state machine from individual gates, allowing for increased flexibility of the control of the design. However, as discussed in [20], MATLAB algorithms, such as the MATLAB `fft()`function, cannot be implemented using this method. This design uses MCode blocks to implement state machines based on control signals generated by the logic circuitry to control the flow of signals within the design.

The System Generator block contains the synthesis and generation options available for the design. Shown in Figure 11, the user can specify the compilation target, the type of generation, and various design constraints. The Part menu allows selection of the FPGA on which the design is to be implemented with additional options for speed grade and pinout of the device. The Compilation menu allows the designer to select the output type. HDL Netlist outputs a VHDL or Verilog file along with a pre-populated Xilinx ISE project file, which was the output type used for this research. Other options available are the NGC Netlist wrapper file, which is the Xilinx proprietary format analogous to the industry standard electronic data interchange format (EDIF), compilation directly to bitstream for direct implementation on an FPGA, and hardware cosimulation [19], [20].

Hardware cosimulation uses a JTAG-configurable target device to load the generated bitcode of the current System Generator model to speed simulation and provide a check on whether the hardware implementation of the design matches the software simulation. Under the hardware cosimulation option in the Compilation menu is a list of all devices for which hardware cosimulation is supported. Instructions for adding the XUP development board are given in [21]. Generation produces a single System Generator block with inputs and outputs corresponding to the GATEWAY IN and GATEWAY OUT blocks of the source design. When the model is simulated, the new block causes System Generator to connect to the target device through the selected JTAG interface, upload the bitfile to the FPGA, insert the input waveforms, and read the output waveforms.


System Generator: untitled

Compilation Options

Compilation : HDL Netlist

Part : Virtex2P xc2vpx20-7ff896

Target directory : ./netlist

Synthesis tool : XST

Hardware description language : VHDL

Create testbench

Import as configurable subsystem

Clocking Options

FPGA clock period (ns) : 100

Clock pin location :

Multirate implementation : Clock Enables

DCM input clock period (ns) : 100

Provide clock enable clear pin

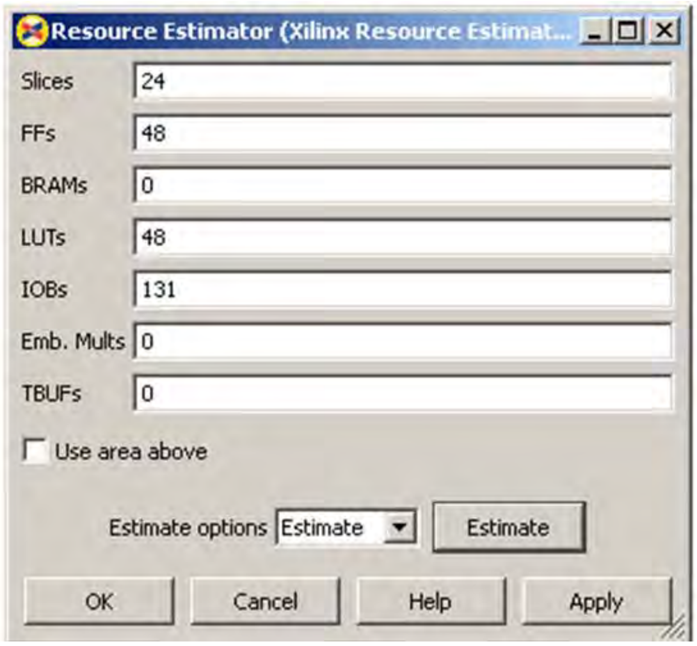
Override with doubles : According to Block Settings

Simulink system period (sec) : 1

Block icon display: Default

Figure 11. System Generator Options.

Another useful feature of System Generator is its ability to estimate the resources required to implement the design. The Resource Estimator block, shown in Figure 12, can be placed into the System Generator design to compile an estimate of the number of slices and other embedded logic resources required to implement the design. This function is useful for the designer to get an early estimate of the logic resources required for the design; however, the estimator is somewhat device agnostic. Conflicts in logic allocations can arise that make a design require more logic resources than the estimator indicates [20].



Slices	24
FFs	48
BRAMs	0
LUTs	48
IOBs	131
Emb. Mults	0
TBUFs	0

☐ Use area above

Estimate options: Estimate

Figure 12. System Generator Resource Estimator.

Another feature of System Generator used in this design was the HDL Testbench generation feature. As noted in [19] when this option is selected, in addition to the usual files, System Generator also produces a file `<design>_tb.vhd`, as well as data vectors and scripts which ModelSim uses for HDL simulation. The data vectors are produced using the data passed from Simulink® to System Generator through the GATEWAY IN blocks.

C. XILINX ISE

Xilinx ISE is the integrated development environment for developing applications for Xilinx FPGAs. Although in Xilinx ISE the designer has the option to start a FPGA design project from scratch, System Generator includes a pre-configured ISE project file among the generated files, named `<design>_cw.ise`. This is the starting point for modifying the design in ISE after generation with System Generator. [6]

The primary use of Xilinx ISE for this design was synthesizing the design to determine actual logic resource utilization of the various designs. Xilinx ISE was also used as the interface with ModelSim for behavioral simulation of the design as well as generating the bitstream, which contains the configuration information for the target FPGA, for hardware implementation [6].

D. MODELSIM

ModelSim, by Mentor Graphics, is a simulation environment for testing the functional behavior of an application in HDL. It is a separate program from Xilinx ISE that provides an alternate environment for testing and verifying the behavior of HDL designs. The simulation passes stimuli into the HDL file and displays the output. It is also possible to script the input / output process to speed the simulation process as well as increase testing flexibility [22].

In order to use ModelSim with Xilinx ISE, the Xilinx HDL simulation libraries must be compiled. This can be performed using a command line argument, as discussed in [5], or it can be done using menu options. On the left side of the screen, under "Sources," select the top-level file for the project. Under "Processes," expand the "Design Utilities" option and double-click "Compile HDL Simulation Libraries." These menu options are shown in Figure 13. If the libraries were compiled during installation, the user does not need to complete this step.

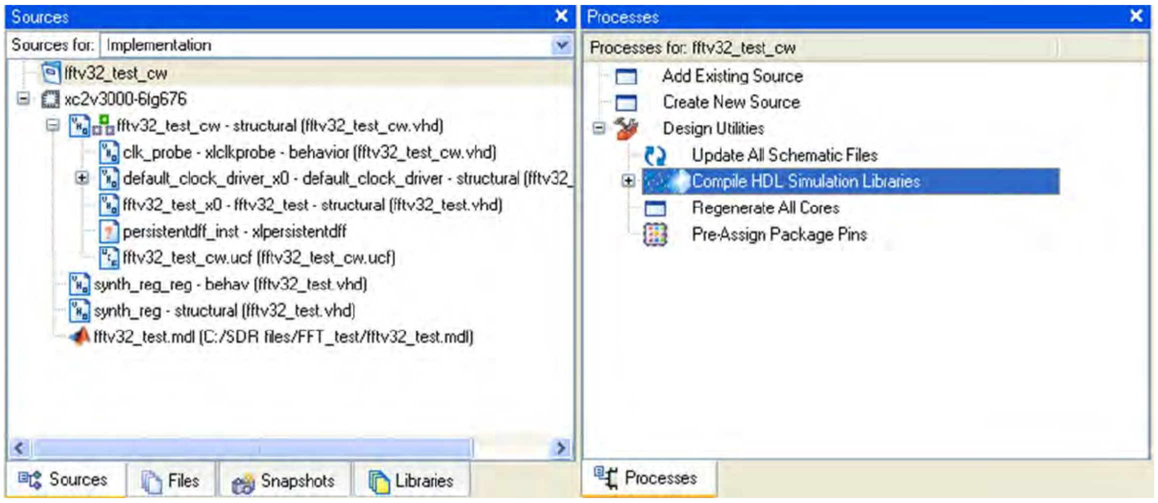

Sources
Sources for: Implementation
fftv32_test_cw
xc2v3000-6lg676
fftv32_test_cw - structural (fftv32_test_cw.vhd)
clk_probe - xlclkprobe - behavior (fftv32_test_cw.vhd)
default_clock_driver_x0 - default_clock_driver - structural (fftv32_
fftv32_test_x0 - fftv32_test - structural (fftv32_test.vhd)
persistentdff_inst - xlpersistentdff
fftv32_test_cw.ucf (fftv32_test_cw.ucf)
synth_reg_reg - behav (fftv32_test.vhd)
synth_reg - structural (fftv32_test.vhd)
fftv32_test.mdl (C:/SDR files/FFT_test/fftv32_test.mdl)
Sources
Files
Snapshots
Libraries
Processes
Processes for: fftv32_test_cw
Add Existing Source
Create New Source
Design Utilities
Update All Schematic Files
Compile HDL Simulation Libraries
Regenerate All Cores
Pre-Assign Package Pins
Processes

Figure 13. Compile HDL Simulation Libraries.

ModelSim uses Tool Control Language (TCL) based testbench files to control stimulus into the design under test. The System Generator option to automatically generate these testbench files was the primary method used to create the input waveforms for the model. Another option, described in [22], is to use the Waveform Editor to generate stimulus signals.

E. XILINX CHIPSCOPE PRO

Xilinx ChipScope Pro is a software package that works with Xilinx FPGAs and CPLDs to conduct hardware level test and debugging. As described in [23], ChipScope consists of three main components: the Xilinx CORE Generator™ tool, the core inserter, and the analyzer. The ICON (interface control) core controls the logic analyzer cores and provides a communications path to the JTAG boundary scan port. The integrated logic analyzer (ILA) core is used to monitor and analyze logic within the chip. The virtual input / output (VIO) core provides access to internal FPGA signals without requiring on-chip RAM. The relationship between the cores, the device under test, and the analyzer software is illustrated in Figure 14.

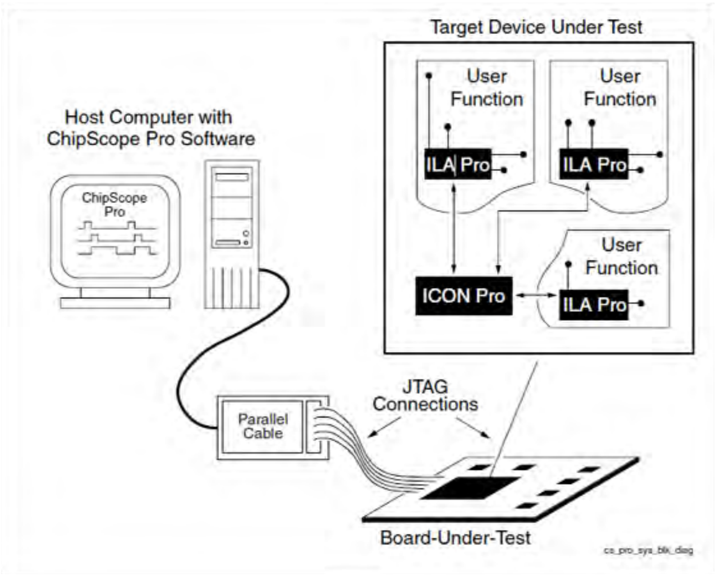


Figure 14. ChipScope Pro Block Diagram (From [23]).

ChipScope Cores are inserted into a design, providing access to in-chip signals. With System Generator, inserting cores is as simple as including the appropriate ChipScope block in the design. The limitation to this method is that only one ChipScope Core can be inserted in this manner, and it cannot be used at the same time as JTAG hardware co-simulation, which is discussed in the next section [19]. ChipScope Cores can also be generated with the CORE Generator and included in HDL source files from within Xilinx ISE. Finally, cores can be inserted into a finished design using the ChipScope Pro Core Inserter tool [23].

ChipScope Analyzer is two pieces of software used to interact with the inserted cores when the design is implemented in hardware. The server is a command line application that connects to the target device via JTAG. It is run automatically if the target device is connected to the local computer, which is the method used for this research. It is also possible to use the server application to connect to a target device over a network. The client application is the graphical user interface (GUI) used to debug the design on the target device. It is used to set data collection triggers and to display collected waveforms [23].

F. XILINX XUP VIRTEX™-II PRO DEVELOPMENT BOARD

The Xilinx XUP (Xilinx University Program) Virtex™-II Pro development board, shown in Figure 15, was used as the hardware target during this design process. It was used during two parts of the design process: first, during System Generator design with the hardware cosimulation option, and second after design implementation with Xilinx ISE with the hardware implementation of the bitcode [19], [21].

The XUP development board features a Xilinx Virtex™-II Pro xc2vp30–7ff896 FPGA. The development board uses the USB JTAG interface to provide access from the development software to configure the FPGA. When the device is connected and powered on, the device drivers are automatically uploaded from firmware on the board over USB to the host computer. Local administrator access is required to allow the drivers to be installed. The board can be powered either through a single 4.5–5.5V power supply through the center-positive barrel jack (J26) or through individual 1.5V, 2.5V, and 3.3V external power supplies. A single power supply was used for this application because the power requirements for this research were low. The preceding features of the XUP development board were the ones used for this research. The full range of features of the XUP development board is listed in [21].

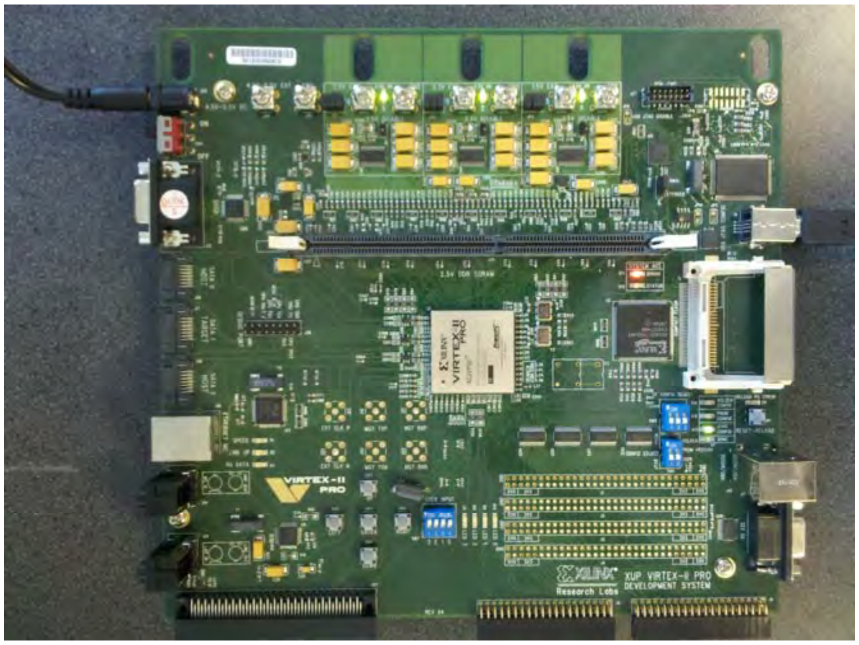


Figure 15. Xilinx University Program Development Board.

G. SUMMARY

The software tools discussed in this section were used to take the compression algorithm from concept through high-level design and all the way to hardware implementation. At each step in the design, the algorithm was tested, first for function, then for behavior, and finally for execution at the hardware level. In following chapters, the use of these design tools, to continue the development of the algorithm, is discussed.

Table 2. FPGA Development Tools.

Design Tool	**Version**	**Purpose**
MATLAB / Simulink	7.4.0 (R2007a)	• Generate input data • Analyze output data • Configure the model
Xilinx System Generator	10.1	• High-level design of the algorithm • Simulation of the algorithm at the functional level • Generate VHDL instantiation of the algorithm
Xilinx ISE	10.1	• Configure VHDL files • Insert UCF (constraint) files • Syntax check the VHDL files • Compile the VHDL file to FPGA bitcode
ModelSim	6.3g	• Simulate and troubleshoot the VHDL file at the behavioral level
ChipScope Pro	10.1	• Configure the target device with the compiled bitcode • Troubleshoot the design at the physical layer level
XUP Virtex™-II Pro development board	xc2vp30–7ff896	• Development board including FPGA and USB JTAG interface

THIS PAGE INTENTIONALLY LEFT BLANK

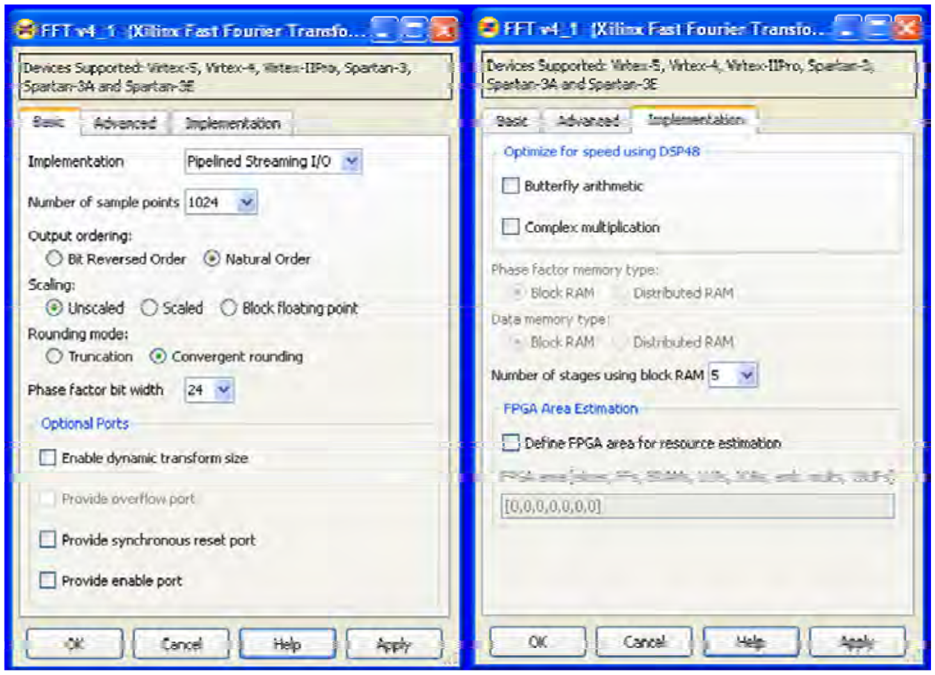
IV. FAST FOURIER TRANSFORM COMPUTING

The System Generator FFT IP Blocks were introduced in Chapter III. In this chapter, the analysis conducted on FFT v4.1 and FFT v1.0 in Wright's thesis [4] and Livingston's thesis [5] is reviewed. A new IP Block, the FFT v3.2 is introduced and analyzed to facilitate migration of the algorithm to implementation on Virtex™-II FPGAs. The configuration and timing differences between the FFT IP Blocks are examined, and the implications to the rest of the design are discussed.

A. FFT V4.1

The original SDR design presented in [4] made use of the FFT v4.1 Xilinx IP block. This version of the FFT allows for streaming input and output (IO) and calculates an FFT of length $N = 2^{(k)}$, $3 \le k \le 16$, where k is an integer. Other options, shown in Figure 16, include selecting natural order output, unscaled output, and phase factors with 24-bit precision. Natural order output was chosen for ease of use with the remainder of the algorithm at a cost of more required block random access memory (BRAM), as discussed in [24]. The highest precision phase factor and the most allowed BRAM were chosen.

The output signals from the FFTv4.1 block, when the input is a constant 1.0_{10} (base-10) and configured for $N = 2^{10} = 1024$, is shown in Figure 17. In the top plot, the real and imaginary outputs are shown, along with the output index, which increments from $0..1023$. As shown, the ready for data (*rfd*) signal is asserted after the first clock cycle at (a). The input index, *xn_index* increments from 0 to 1023 and reaches its final value at $t = 1025$. The FFT calculation begins at $t = 518$, indicated by the *busy* signal (b). The *e_done* signal, asserted 1 clock cycle before the FFT calculation is complete, is asserted at $t = 2162$, (c). The *valid* signal, indicating valid output data, is asserted at $t = 2163$, (d), along with the first point of the calculated FFT, (e). The next *e_done* signal (f) is asserted at $t = 3186$, and the first point of the next calculated FFT (g) is output at $t = 3187$. Both signals are asserted 1024 clock cycles after the preceding signal, indicating streaming IO. The total latency of the FFT v4.1 IP block, from the time the first input point is accepted at $t = 2$ to the time the first point of the calculated FFT is output at $t = 2163$, is $2163 - 2 = 2161$.


FFT v4_1 (Xilinx Fast Fourier Transfo...

Devices Supported: Virtex-5, Virtex-4, Virtex-IIPro, Spartan-3, Spartan-3A and Spartan-3E

Basic | Advanced | Implementation

Implementation: Pipelined Streaming I/O

Number of sample points: 1024

Output ordering: Bit Reversed Order [✓ Natural Order]

Scaling: [✓ Unscaled] Scaled Block floating point

Rounding mode: Truncation [✓ Convergent rounding]

Phase factor bit width: 24

Optional Ports

☐ Enable dynamic transform size

☐ Provide overflow port

☐ Provide synchronous reset port

☐ Provide enable port

OK Cancel Help Apply

FFT v4_1 (Xilinx Fast Fourier Transfo...

Devices Supported: Virtex-5, Virtex-4, Virtex-IIPro, Spartan-3, Spartan-3A and Spartan-3E

Basic | Advanced | Implementation

Optimize for speed using DSP48

☐ Butterfly arithmetic

☐ Complex multiplication

Phase factor memory type: [✓ Block RAM] Distributed RAM

Data memory type: [✓ Block RAM] Distributed RAM

Number of stages using block RAM: 5

FPGA Area Estimation

☐ Define FPGA area for resource estimation

[illegible]

[0,0,0,0,0,0,0]

OK Cancel Help Apply

Figure 16. FFT v4.1 Configuration Options (After [5]).

The calculated FFT shown in Figure 17 is

$$X[k] = \begin{cases} 1023 + j0 & k = 0 \\ 0 & k = 1..1023 \end{cases}, \qquad \text{(IV.1)}$$

which is consistent with Equation (II.5).

The FFT v4.1 test circuit shown in Figure 18 was implemented for a Virtex™-II Pro xc2vp30. The required logic resources are shown in Table 3. All circuits and supporting computer files presented in this thesis are listed in the Appendix.

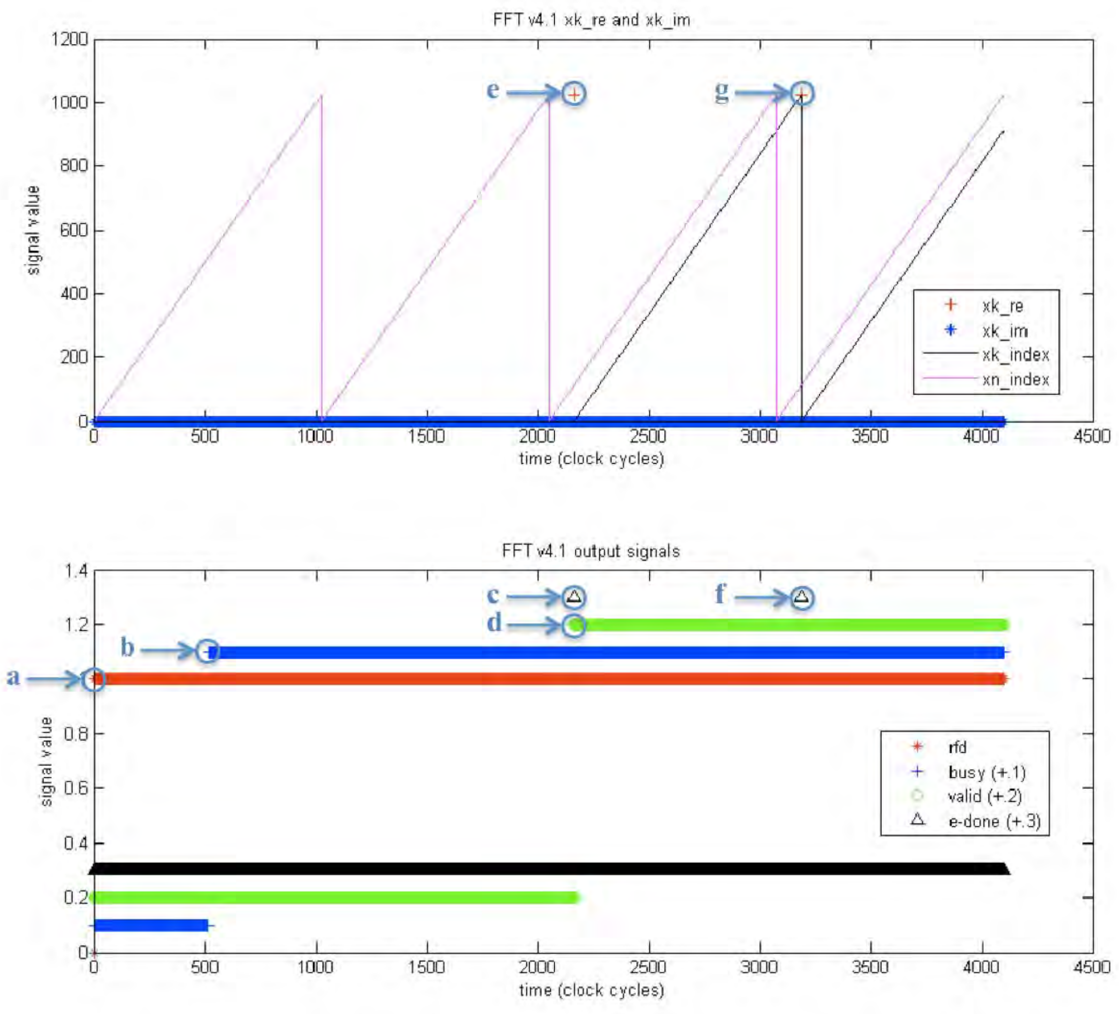


Figure 17. **FFTv4.1 Output Signals (After [5]).**

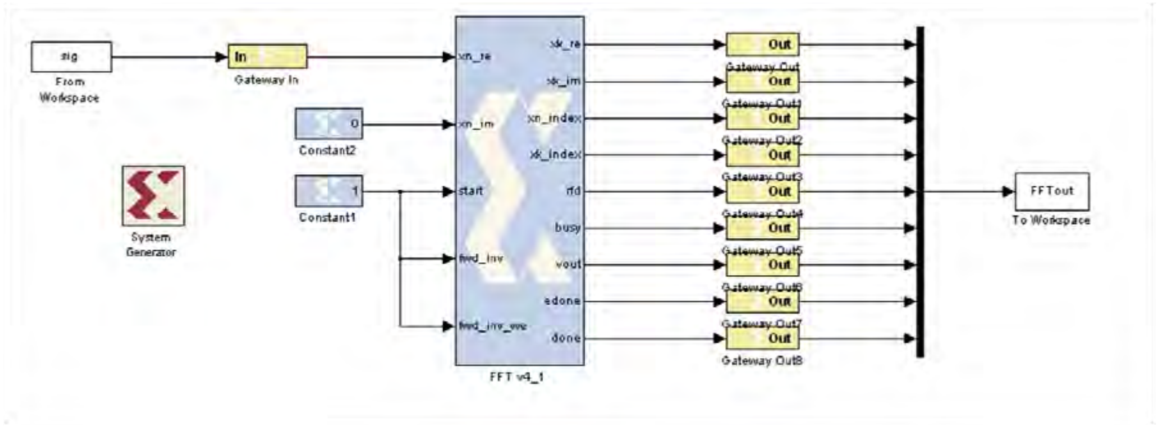


Figure 18. Circuit for Timing Analysis of FFT v4.1 (After [5]).

Table 3. FFT v4.1 Resource Utilization on a Virtex™-II Pro xc2vp30–7ff896 (After [5]).

Resource	**Used**	**Available**	**Percent**
Slices	3448	13696	25%
Flip-Flops	5945	27392	21%
4-input LUTs	4312	27392	15%
Bonded IOBs	81	556	14%
BRAMs	12	136	8%
MULT18x18s	32	136	23%
GCLKs	1	16	6%

B. FFT V1.0

The modified SDR presented in [5] made use of the Xilinx IP block FFT v1.0. This version of the FFT is the only FFT IP block supported by the Virtex™ family of devices. This version of FFT accepts streaming input on each clock cycle but only outputs calculated FFT points for *N* out of every 4*N* clock cycles. As noted in [25], the circuit precision is fixed at 16-bit 2’s complement numbers for input and output. The timing of the FFT v1.0 IP block was tested using the circuit shown in Figure 19.

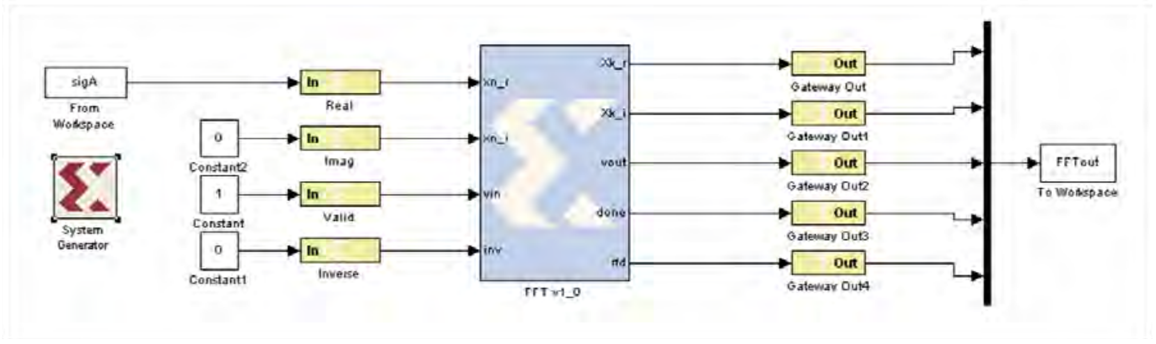


Figure 19. Circuit for Timing Analysis of FFT v1.0 (After [5]).

As shown in Figure 20, the circuit produces a valid output for 1024 cycles out of every 4096. The *vin* signal and real data begin at $t = 1$. The circuit asserts the *ready* signal at $t = 1$, and it remains asserted during the entire operation, indicating the FFT is accepting streaming input data. At $t = 8247$, the *done* signal is asserted for one clock cycle at (a) and the calculated FFT points are output at (b) from $t = 8247$ until $t = 9270$. The next set of data is ready for output at $t = 12343$, at (c). The total latency of the FFT v1.0 circuit is 8247.

The input to the test circuit is a constant 0.5_{10}. From Equation (II.5), the output should be

$$X\left[k\right] = \begin{cases} 512 + j0 & k = 0 \\ 0 & k = 1..1023 \end{cases} . \tag{IV.2}$$

This inconsistency is because the precision of FFT v1.0 is fixed at 16 bits, so the output is scaled by $1/N$ after each stage. If Equation (II.5) is scaled by $1/N$, the results are consistent.

The FFT v1.0 was implemented for a Virtex™ xcv1000 FPGA. The required logic resources are shown in Table 4.

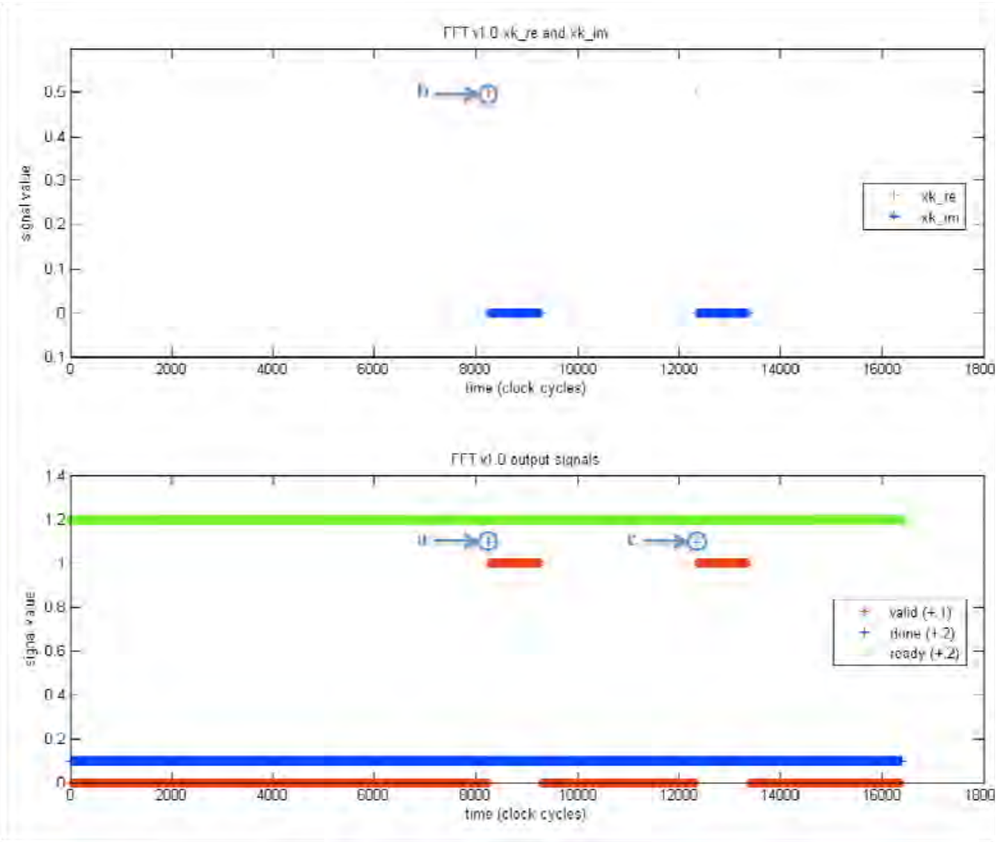


Figure 20. FFT v1.0 Output Signals (After [5]).

Table 4. FFT v1.0 Resource Utilization on a Virtex™ xcv1000–6fg680 (After [5]).

Resource	**Used**	**Available**	**Percent**
Slices	1285	12288	10%
Flip-Flops	2570	24576	10%
4-input LUTs	2247	24576	9%
Bonded IOBs	70	512	13%
BRAMs	16	32	50%
GCLKs	1	4	25%

C. FFT V3.2

As discussed in [24], the Xilinx FFT v4.1 IP block is not supported for the Virtex™-II family of FPGAs. In order to implement this algorithm on a Virtex™-II, the

FFT v3.2 IP block, described in [26] has to be used. As with the FFT v4.1, this version of the FFT allows for streaming IO and calculates an FFT of length $N = 2^{(k)}$, $3 \le k \le 16$, where *k* is an integer.

In order to investigate the timing constraints for this block, the circuit shown in Figure 21 was used. The input signal for investigating the timing was a constant DC input of 0.5 for all real inputs and 0 for all imaginary inputs. The start signal was held high for the entire analysis, and the inverse signal was held low.

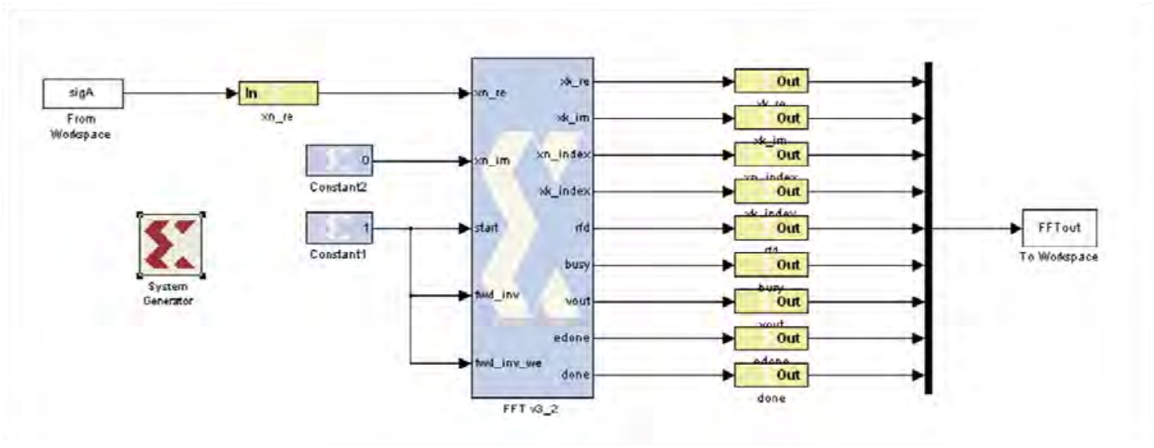


Figure 21. Circuit for Timing Analysis of FFT v3.2.

The output signals from the FFT v3.2 block, when the input is a constant 1.0_{10} and configured for $N = 2^{10} = 1024$, is shown in Figure 22. In the top plot, the real and imaginary outputs are shown, along with the output index, which increments from $0..1023$. As shown, the *rfd* signal is asserted after the first clock cycle at (a). The input index, *xn_index* increments from 0 to 1023 and reaches its final value at $t = 1025$. The FFT calculation begins at $t = 518$, indicated by the *busy* signal (b). The *e_done* signal, asserted 1 clock cycle before the FFT calculation is complete, is asserted at $t = 2146$, (c). The *valid* signal, indicating valid output data, is asserted at $t = 2148$, (d), along with the first point of the calculated FFT, (e). The next *e_done* signal (f) is asserted at $t = 3170$, and the first point of the next calculated FFT (g) is output at $t = 3172$. Both signals are asserted 1024 clock cycles after the preceding signal, indicating streaming IO. The total latency of the FFT v3.2 IP block, from the time the first input point is accepted at $t = 2$ to the time the first point of the calculated FFT is output at $t = 2148$, is $2148 - 2 = 2146$.

The calculated FFT shown in Figure 22 is

$$X\left[k\right] = \begin{cases} 1023 + j0 & k = 0 \\ 0 & k = 1..1023 \end{cases}, \qquad \text{(IV.3)}$$

which is consistent with Equation (II.5).

The FFT v3.2 test circuit was generated for a Virtex™-II xc2v3000. The resulting VHDL was compiled using ISE and simulated using ModelSim. The logic resource utilization is shown in Table 5.

The ModelSim waveform is shown in Figure 23. The `edone` signal is asserted on output index `0x3FE` of the previous output, the `done` signal is asserted on output index `0x3FF`, and the first point of the new computed FFT `0x0FFC00` is output with index `0x000`.

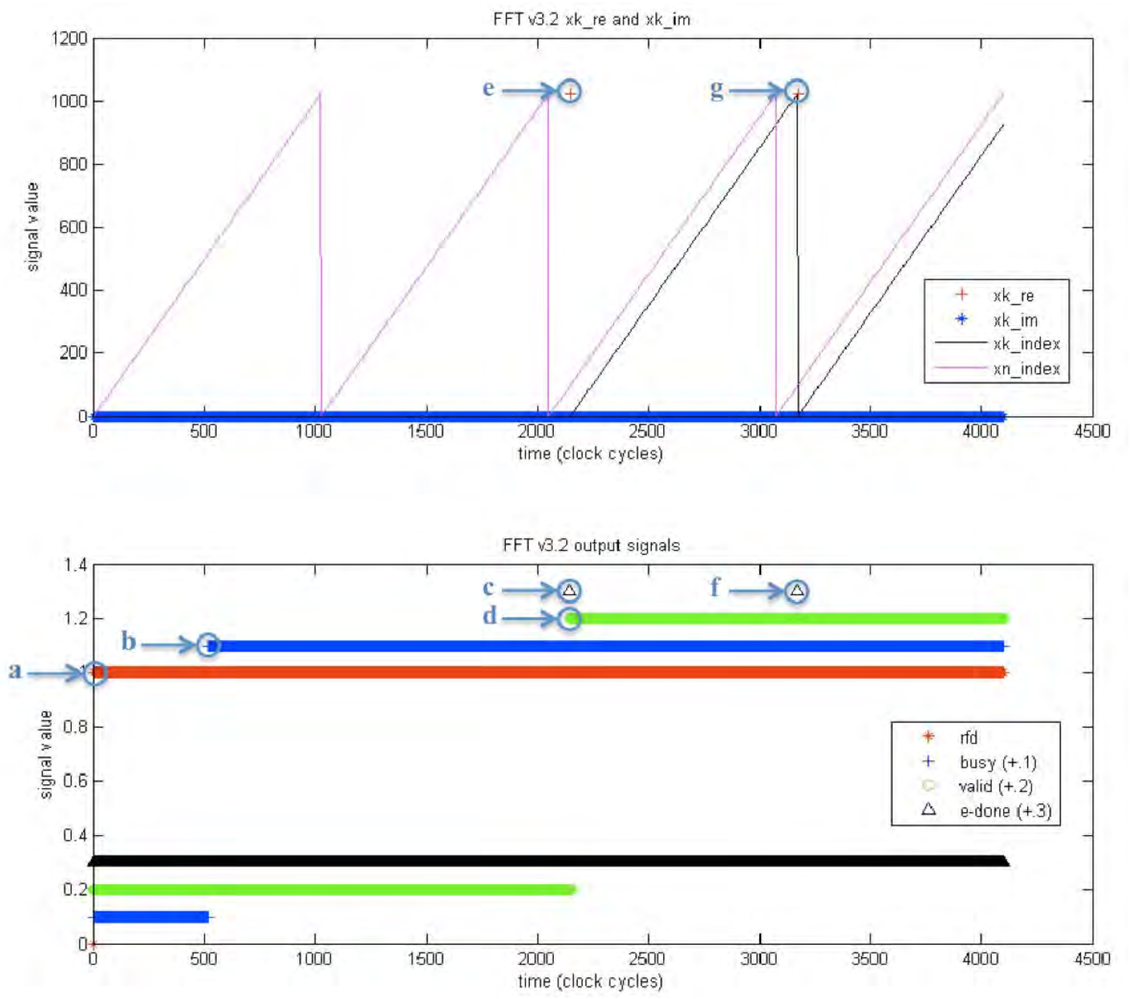


Figure 22. FFT v3.2 Output Signals.

Table 5. FFT v3.2 Resource Utilization on a Virtex™-II xc2v3000–6fg676.

Resource	Used	Available	Percent
Slices	3274	14336	22%
Flip-Flops	5520	28672	19%
4-input LUTs	4037	28672	14%
Bonded IOBs	81	484	16%
BRAMs	12	96	12%
MULT18x18s	32	96	33%
GCLKs	1	16	6%

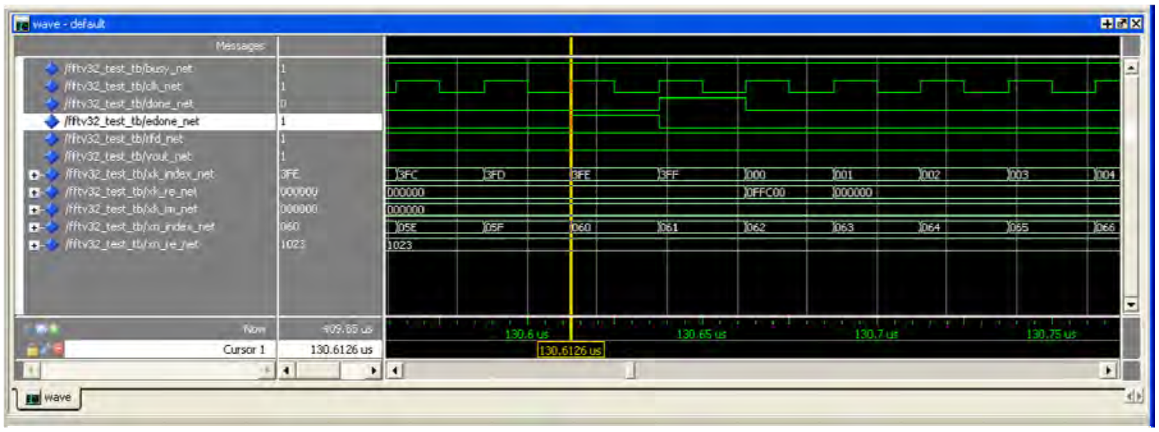

wave - default
Messages
/fftv32_test_tb/busy_net 1
/fftv32_test_tb/clk_net 1
/fftv32_test_tb/done_net 0
/fftv32_test_tb/edone_net 1
/fftv32_test_tb/rfd_net 1
/fftv32_test_tb/vout_net 1
/fftv32_test_tb/xk_index_net 3FE
/fftv32_test_tb/xk_re_net 000000
/fftv32_test_tb/xk_im_net 000000
/fftv32_test_tb/xn_index_net 060
/fftv32_test_tb/xn_re_net 1023
3FC 3FD 3FE 3FF 000 001 002 003 004
000000 0FFC00 000000
000000
05E 05F 060 061 062 063 064 065 066
1023
Now 409.65 us
Cursor 1 130.6126 us
130.6 us 130.65 us 130.7 us 130.75 us
130.6126 us
wave

Figure 23. FFT v3.2 ModelSim Simulation.

Because the timing of this IP block is similar to the timing of the FFT v4.1 IP block, the logic resource requirements for both were compared for a Virtex™-II Pro xc2v30, the results of which are shown in Table 6. In a side-by-side comparison, the FFT v4.1 used slightly more resources, due mainly to requiring more 4-input LUTs for signal routing. Because this algorithm was designed to be backward compatible with the Virtex™-II FPGA, implementation with FFT v3.2 was desirable since this FFT IP is compatible with both the Virtex-II and Virtex-II Pro [26]. If the design is to be forward compatible with the Virtex™-4 or -5 FPGAs, the FFT v4.1 is desirable due to its compatibility with the DSP48 logic primitives available on those devices [24].

D. SUMMARY

The performance of the FFT v4.1 IP block was verified. The FFT v4.1 was shown to accept streaming input and produce streaming output. The performance of the FFT v1.0 IP block was verified. It was shown that the FFT v1.0 IP block accepts streaming input but does not produce streaming output. The performance of the FFT v3.2 IP block was investigated. It was shown to behave similarly to the FFT v4.1 block, accepting streaming input and output. Timing differences between the FFT v4.1 and v3.2 and the implications to further design work were discussed. In the following chapter, a method of applying error detection to the FFT v4.1 block is analyzed, and modifications are made to use the FFT v3.2 block.

Table 6. Resource Comparison of FFT v3.2 and v4.1 on a Virtex™-II Pro.

Resource	Available	v3.2 Used	v3.2 Percent	v4.1 Used	v4.1 Percent
Slices	13696	3298	24%	3448	25%
Flip-Flops	27392	5520	20%	5945	21%
4-input LUTs	27392	4043	14%	4312	15%
Bonded IOBs	556	81	14%	81	14%
BRAMs	136	12	8%	12	8%
MULT18x18s	136	32	23%	32	23%
GCLKs	16	1	6%	1	6%

THIS PAGE INTENTIONALLY LEFT BLANK

V. ERROR DETECTION

As discussed in Chapter II, computing in space requires attention to detection and correction of errors caused by SEUs. In this chapter, the development of error detection for the compression algorithm presented in [5] is reviewed. Errors in the computation of the FFT are detected by comparing the energy in with the energy out using Parseval's theorem. Errors in Temporary Memory are detected by calculating and comparing data parity before and after storage.

A. FFT ERROR DETECTION WITH PARSEVAL'S THEOREM

In [5], Livingston presents a version of the compression algorithm that can detect whether the output is in error. As discussed in [11], the probability of an SEU is proportional to the area of the semiconductor. The two portions of the algorithm with the highest resource allocation and, hence, the biggest area are the FFT calculation and the temporary storage memory. The error detection applied to the FFT block was designed to check that the transform was computed successfully. The error detection applied to the temporary storage memory was to check that the computed time-frequency bins were not corrupted.

1. Analysis of the Original Error Detection Algorithm

As discussed in Chapter II, Parseval's theorem states that the energy into a Fourier transform must equal the energy out of the transform. The algorithm presented in [5] makes use of this property to determine if the FFT output is correct. The conceptual block diagram illustrating this setup is shown in Figure 24. The squares of the input points are summed. After the FFT is computed, the squares of the magnitude of the complex output are summed and scaled by $1/N$. If the FFT calculation is correct, the two sums will be equal to within a threshold determined by the precision of the sum-of-squares of the input points, and the error flag is set to 0. If the FFT calculation is not correct, the two sums will differ by more than the threshold, and the FFT output will be flagged as in error.

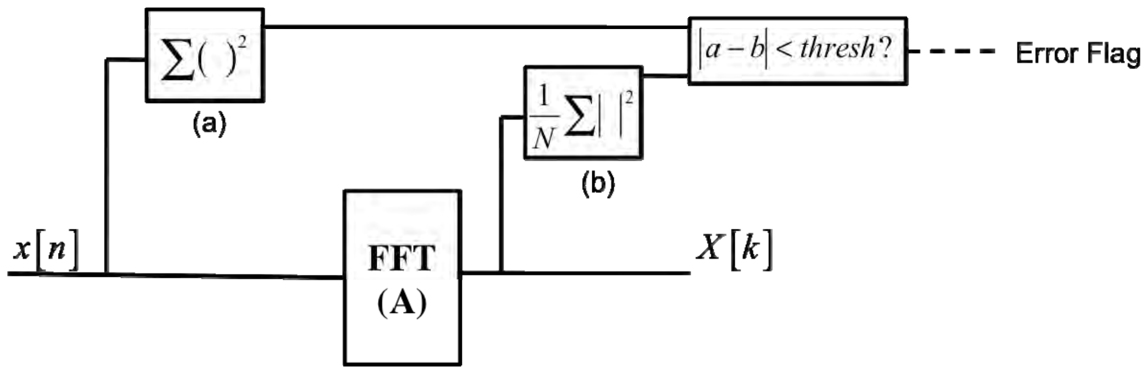


Figure 24. Error Detection with Parseval's Theorem (Conceptual) (After [5]).

The error checking in the algorithm presented in [5] has components in two blocks, the FFTv4.1 block and the Windowing Algorithm block. This is because the squaring function on the output is used twice in the Windowing Algorithm: to calculate the energy for comparison to the input for error detection and to calculate the energy for time-frequency bin energy calculation. For the purposes of analysis, the FFT and the error detection components were copied from the algorithm in `SDR1024Mod8C.mdl` and placed together in a single System Generator model, shown in Figure 25.

The left side of Equation (II.10) is computed using a Mult block, the Accum subsystem, and a Scale block. The input to the compression algorithm is assumed to be real, so the squaring is accomplished using a single Mult block. These values are input into the Accum subsystem, shown in Figure 26, and originally developed in [4]. This subsystem contains two Accumulator blocks, controlled by a MCode state machine.

The output of the Accum block is scaled by $1/N$ using the Scale block and delayed to match the output of the FFT. The output is already delayed by the Point Pwr subsystem, and only the first $1/N$ points are used in the error analysis. The required delay is therefore given by the expression given in [5] as

$$\text{FFT Latency} - \text{Point Pwr Latency} - N/2. \tag{V.1}$$

The right side of Equation (II.10) is calculated using Scale blocks, the Point Pwr subsystem, and the FFT Error Detection subsystem. The real and imaginary outputs of the FFT block are scaled by $1/N$. The square and sum of the result are completed by the Point Pwr subsystem, shown in Figure 27.

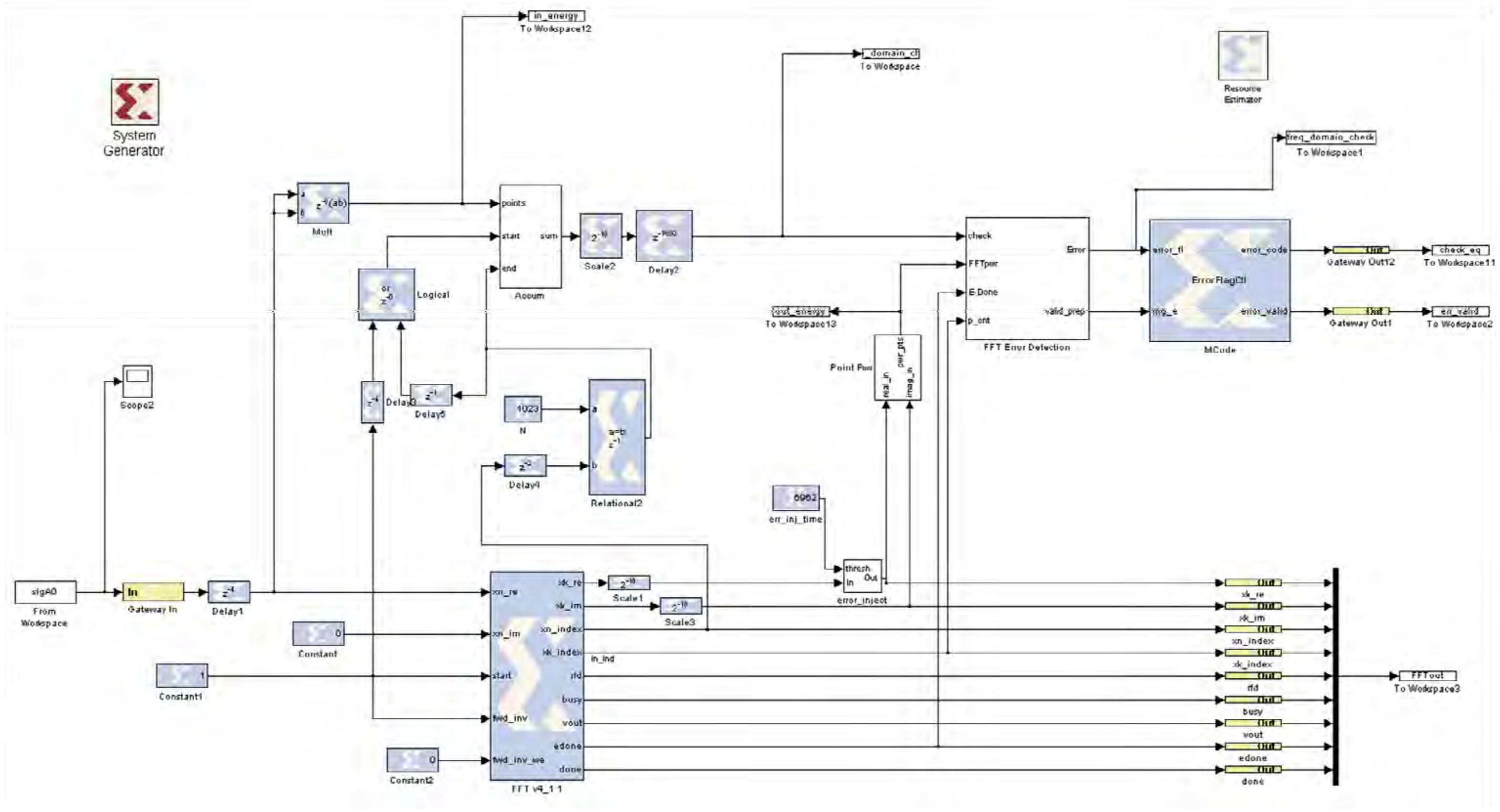


Figure 25. Error Detection applied to the FFTv4.1 IP block (After [5]).

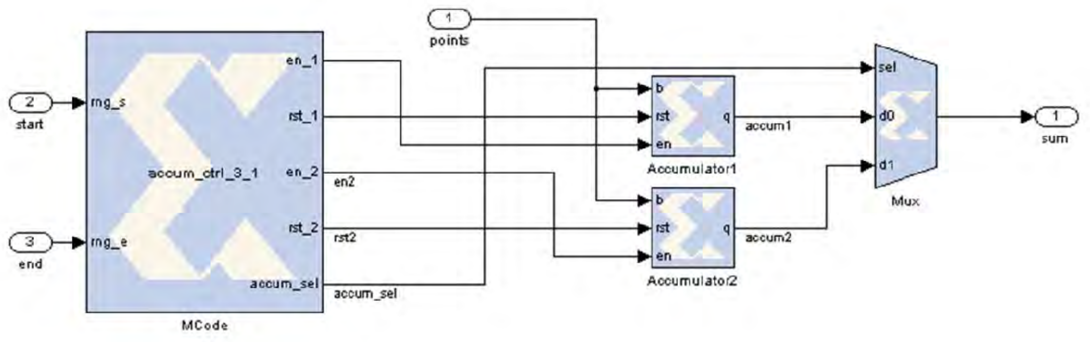


Figure 26. “Accum” Subsystem (From [4]).

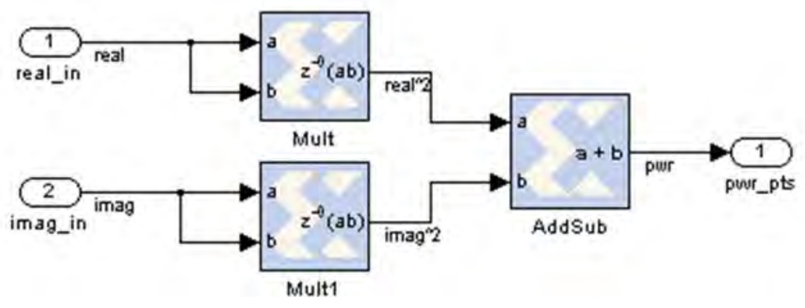


Figure 27. “Point Pwr” Subsystem (From [5]).

The FFT Error Detection subsystem, shown in Figure 29, contains another Accum to sum the squares. This subsystem is set to accept input for half the period of the FFT, effectively summing from $k = 0$ to $N/2-1$. Due to conjugate symmetry this is sufficient, since the sum of the magnitudes squared of the first half of the FFT are equal to those of the second half [5].

The two sides of Equation (II.10) are compared by subtracting the two values and comparing against a threshold. In this implementation, the threshold is set such that a correct output is declared when

$$-2^{-12} < a-b < 2^{-12}, \qquad \text{(V.2)}$$

which was determined experimentally by Livingston in [5]. This threshold is required since the precision of the sum-of-squares calculations for either side of Equation (II.10) are different.

In order to test the circuit, an error injection circuit was implemented in [5] using a counter and a MUX (multiplexer), shown in Figure 28. The input is the real values from the FFT block. These actual values are passed through the MUX until the freerunning counter exceeds the given threshold. Once that occurs, the output becomes the entered constant, in this case 2^{-6}.

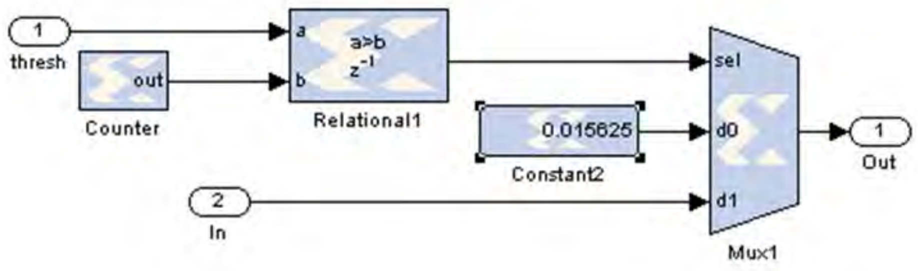


Figure 28. Error Injection circuit (After [5]).

The FFT v4.1 with error correcting circuit in Figure 25 was generated for a Virtex™-II Pro xc2vp30. The resources required are shown in Table 7.

2. Modification to FFT Error Detection

The error detection method developed in [5] was built around a compression algorithm based on the FFT v4.1. In order to support this algorithm on a Virtex™-II, the FFT v3.2 block must be used. As mentioned in Chapter IV, the latency of the two FFT versions is different, requiring modification to the circuit to use the block.

The FFT v3.2 block was inserted as shown in Figure 30. The delay after the accumulator was adjusted according to Equation (V.1), resulting in a new delay of 1631. This new compression algorithm was tested in the same manner as the original algorithm. The results were the same, with the output indices shifted by the latency difference between the FFT v3.2 and FFT v4.1.

The circuit in Figure 30 was generated for a Virtex™-II xc2v3000. The required resources are shown in Table 8.

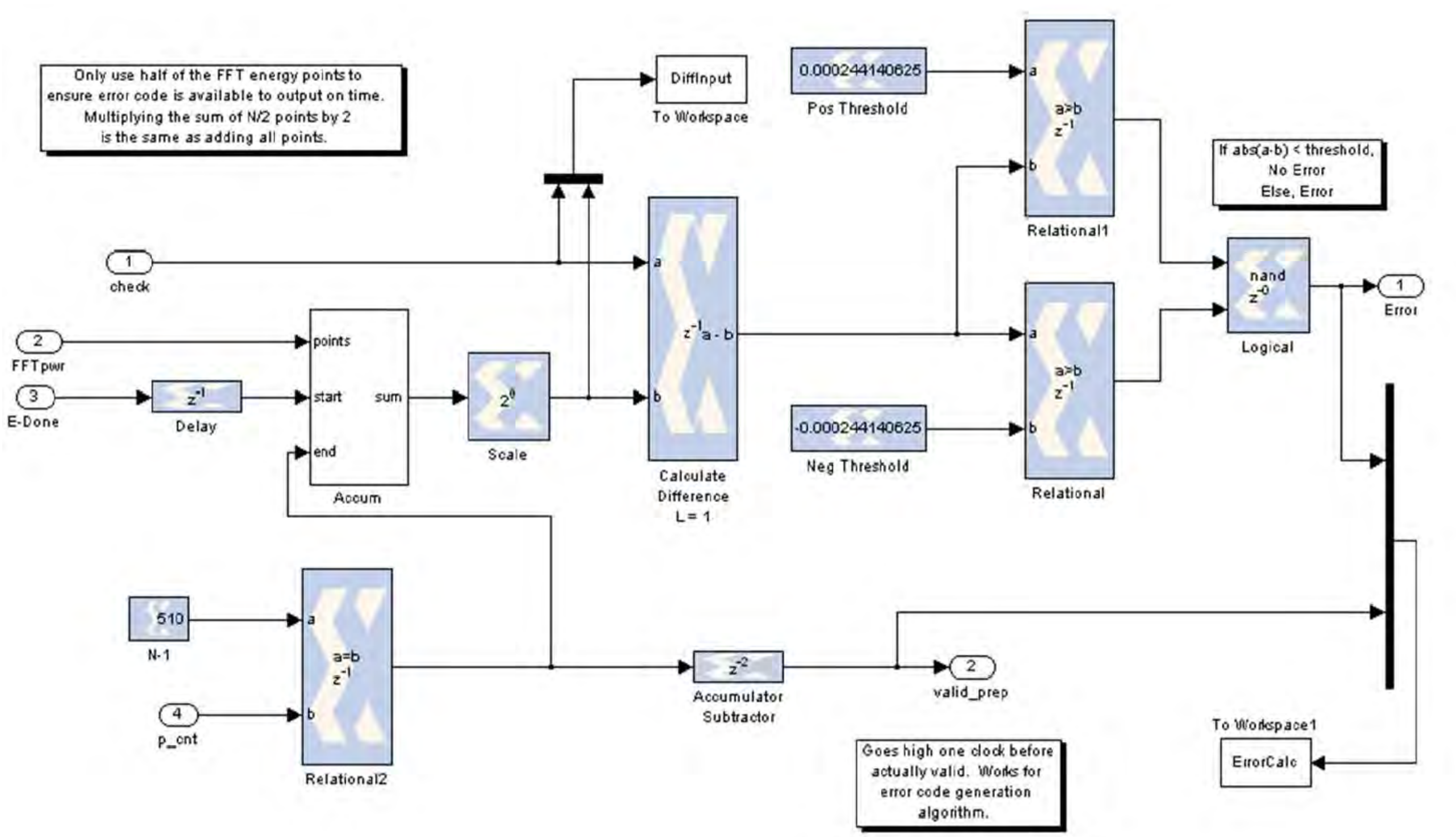

Only use half of the FFT energy points to ensure error code is available to output on time. Multiplying the sum of N/2 points by 2 is the same as adding all points.
DiffInput
To Workspace
0.000244140625
Pos Threshold
a>b
z^-1
Relational1
If abs(a-b) < threshold, No Error Else, Error
1 check
a
z^-1 a - b
b
Calculate Difference L = 1
2 FFTpwr
3 E-Done
z^-1
Delay
points
start
sum
end
Accum
2^0
Scale
-0.000244140625
Neg Threshold
Relational
nand z^-0
Logical
1 Error
510
N-1
a=b z^-1
Relational2
4 p_cnt
z^-2
Accumulator Subtractor
2 valid_prep
Goes high one clock before actually valid. Works for error code generation algorithm.
To Workspace1
ErrorCalc

Figure 29. “FFT Error Detection” Subsystem (From [5]).

Table 7. Resource Requirements for Compression Algorithm on Virtex™-II Pro xc2vp30.

Resource	Used	Available	Percent
Slices	9297	13696	67%
Flip-Flops	14924	27392	54%
4-input LUTs	14207	27392	51%
Bonded IOBs	96	556	17%
BRAMs	57	136	41%
MULT18x18s	61	136	44%
GCLKs	1	16	6%

Table 8. Resources Required for Modified Compression Algorithm on Virtex™-II xc2v3000.

Resource	Used	Available	Percent
Slices	7442	14336	51%
Flip-Flops	11356	28672	39%
4-input LUTs	11343	28672	39%
Bonded IOBs	270	484	55%
BRAMs	15	96	15%
MULT18x18s	36	96	37%
GCLKs	1	16	6%

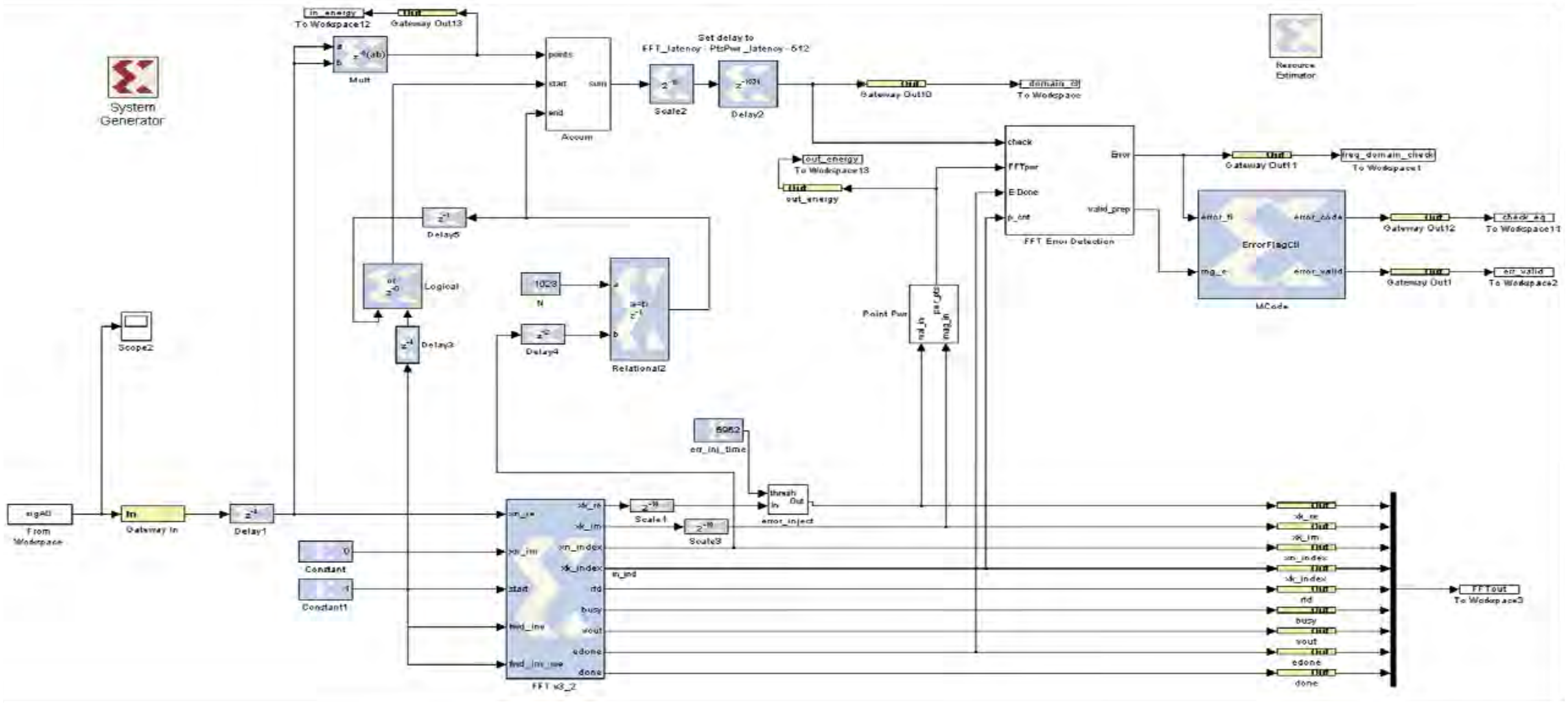


Figure 30. Error Detection Applied to the FFT v3.2 IP Block.

B. MEMORY ERROR DETECTION USING PARITY

The section of the compression algorithm with the second highest probability of SEU after the FFT is the Temporary Memory where calculated FFT points are stored after time and frequency bin calculations have been conducted. In order to detect whether the contents of the Temporary Memory have been corrupted, Livingston added a parity check feature to the algorithm presented in [5].

The temporary storage subsystem developed in [4] consists of two dual port RAM blocks for storing the real and imaginary FFT points while the bin energy is calculated. The values stored in memory are 35 bits, so the parity bit is calculated with the expression

$$P = \mathrm{XOR}\left\{\mathrm{Bit}\left[34\right], \mathrm{Bit}\left[33\right], \ldots, \mathrm{Bit}\left[1\right], \mathrm{Bit}\left[0\right]\right\}. \tag{V.3}$$

Equation (V.3) is implemented with a string of XOR gates and Bit Basher blocks, as shown in Figure 31. As discussed in [5], the string of Bit Basher blocks are used to separate each 35-bit value into 35 one-bit values, and the string of XOR gates is used to calculate a single parity bit.

The calculated parity bits are stored in a separate Dual Port RAM block using the same addressing signals from the data Dual Port RAM block. When the data is read out, the parity is calculated using another parity generator as shown in Figure 32, and this parity bit is compared to the previously calculated parity bit using an XOR gate. If the parity bits are not equal, an error flag is sent to the parity flag generation subsystem, which is described in [5].

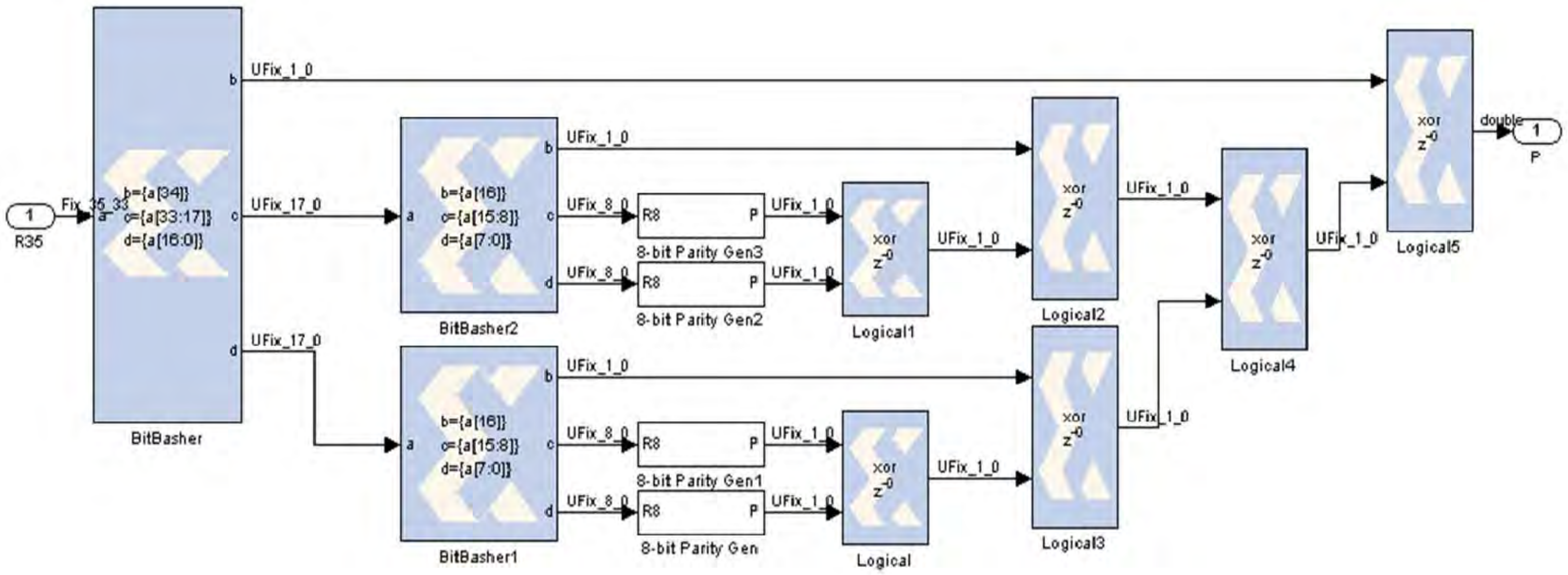


Figure 31. Parity Generator (From [5]).

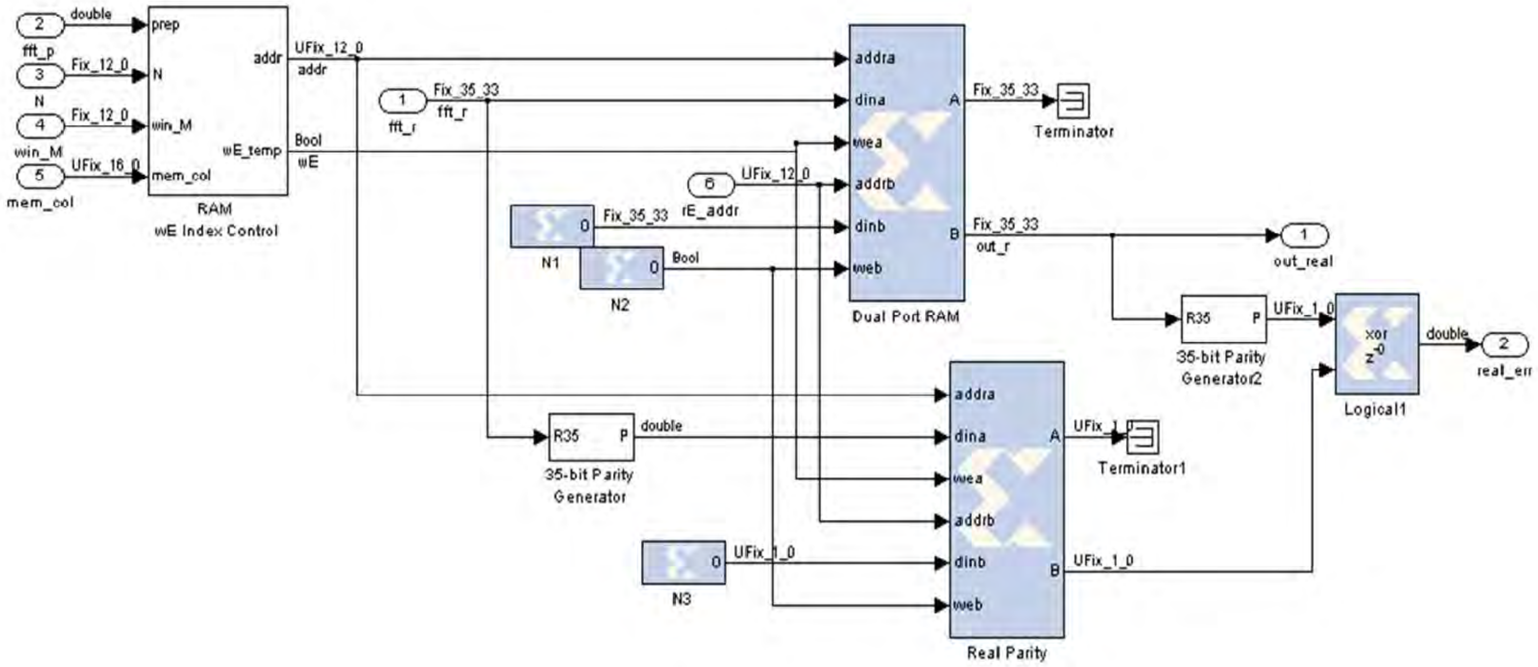


Figure 32. Temporary Memory Subsystem with Parity Check (From [5]).

C. SUMMARY

Methods for detecting errors in the calculation of an FFT when the designer does not have access to the internal circuitry of the FFT block were introduced in this chapter. The design and performance of an error detection circuit applied to the FFT v4.1 block was analyzed and the error detection was verified. The error detection circuit was modified for use with the FFT v3.2 block, and the error detection was verified. A method of checking memory for error using parity check bits was also analyzed, and the detection of errors verified. The error detection method discussed in this chapter is used in the next chapter to implement error correction for the FFT computation circuits.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. ERROR CORRECTION

The compression algorithm development described thus far has resulted in a design that can detect when the FFT calculation is in error. This design can be implemented in a Virtex™-II Pro FPGA. The goals of this design work were to make use of the error detection to implement error correction and implement the resulting design in a multiple-FPGA implementation using Virtex™-II FPGAs.

A. INITIAL ERROR CORRECTION DEVELOPMENT

The error detection developed in [5] set a flag if the output of the FFT block was in error. In order to use this feature to implement error correction, the FFT calculation block was duplicated, and a voter decides which FFT is correct, as shown in Figure 33.

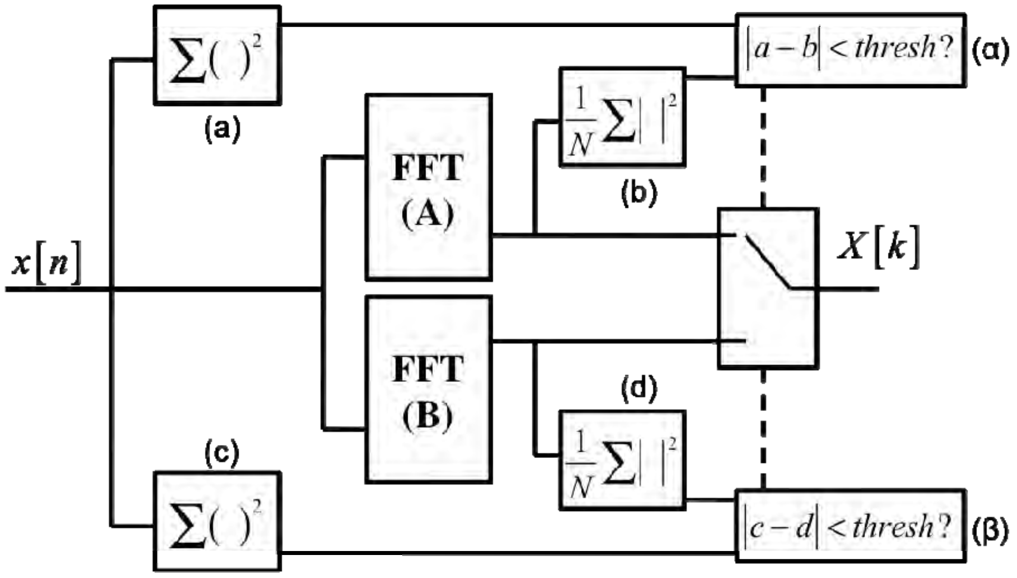


Figure 33. Error Correction (Conceptual).

In this implementation, the energy in each FFT is calculated separately. If the sums-of-squares are equal, it is assumed that there was no error in calculating the FFT. If they are not equal, it is assumed that the FFT calculated is in error and the other FFT output is chosen. Again, the assumption is that there is an SEU in only one subsystem in each calculation cycle. The relationship between the block containing the error and which FFT values are chosen for $X[k]$ are shown in Table 9.

Table 9. Truth Table for Error Correction Algorithm.

Error in Block:	**|a-b| <thresh**	**|c-d| <thresh**	**X[k]**	**X[k] correct**
Σ(a)	0	1	FFT B	1
Σ(b)	0	1	FFT B	1
FFT A	0	1	FFT B	1
Σ(c)	1	0	FFT A	1
Σ(d)	1	0	FFT A	1
FFT b	1	0	FFT A	1

The error correcting circuit is the System Generator model shown in Figure 34. The circuit consists of two subsystems, each containing an FFT v3.1 IP Block, as well as the error detection circuitry described in Chapter V. One of these subsystems contains the error injection circuit, also described in Chapter V. Each of these blocks computes the FFT of the input signal independently and checks whether the computation was in error. These values are passed to the new Voter subsystem.

The Voter subsystem, shown in Figure 35, consists of MUXs that act as switches to select output from either of the FFT computation subsystems. The VoterCtl block implements a state machine controlling which data the MUXs select for output. Inputs to the VoterCtl block are the error calculations from each of the FFT error detection circuits and the *x_error_valid* signal, which indicates when a new error calculation is ready. The voter also detects whether both FFTs are reporting an error, in which case the *voter_error* signal is asserted.

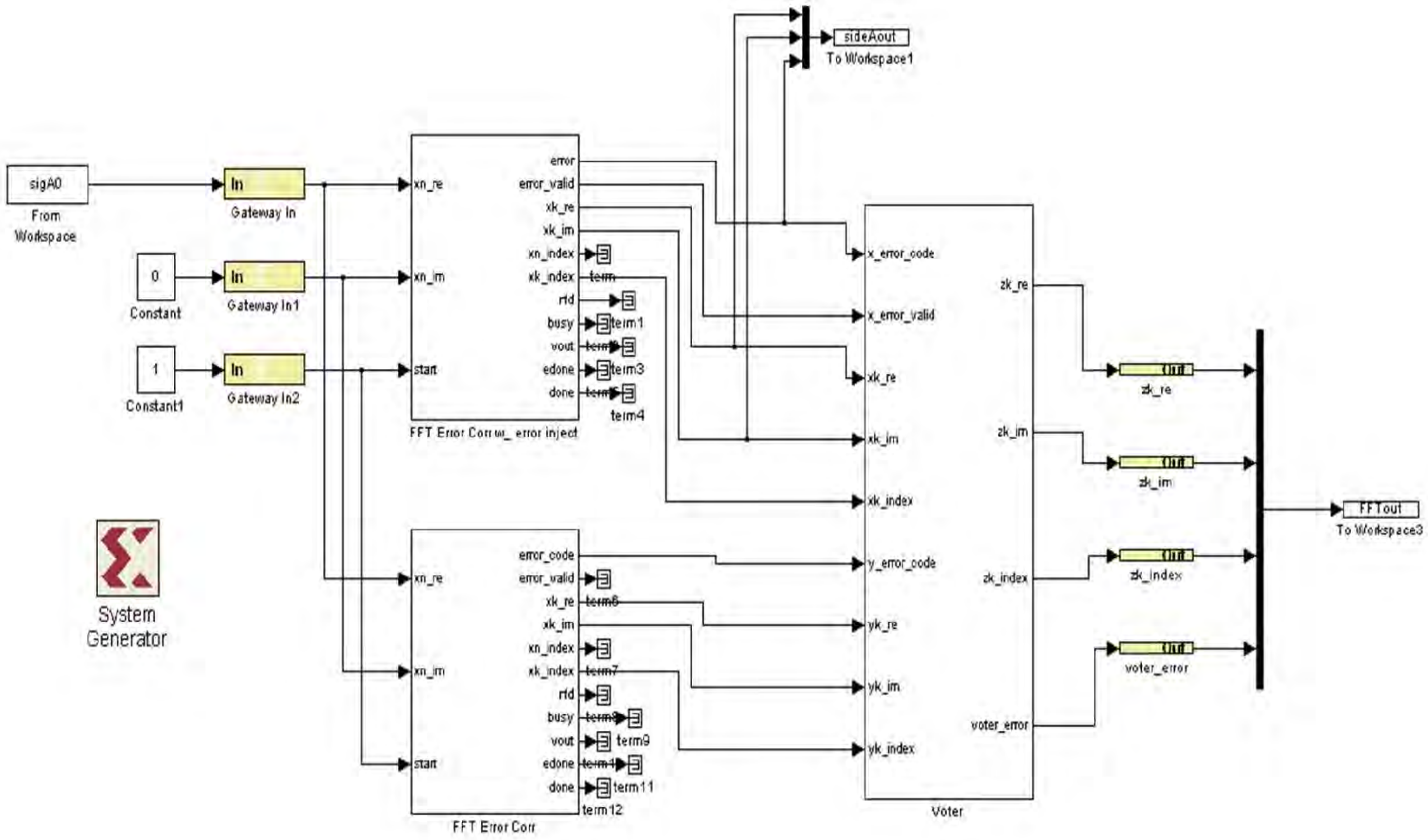


Figure 34. FFT Error Correcting Circuit.

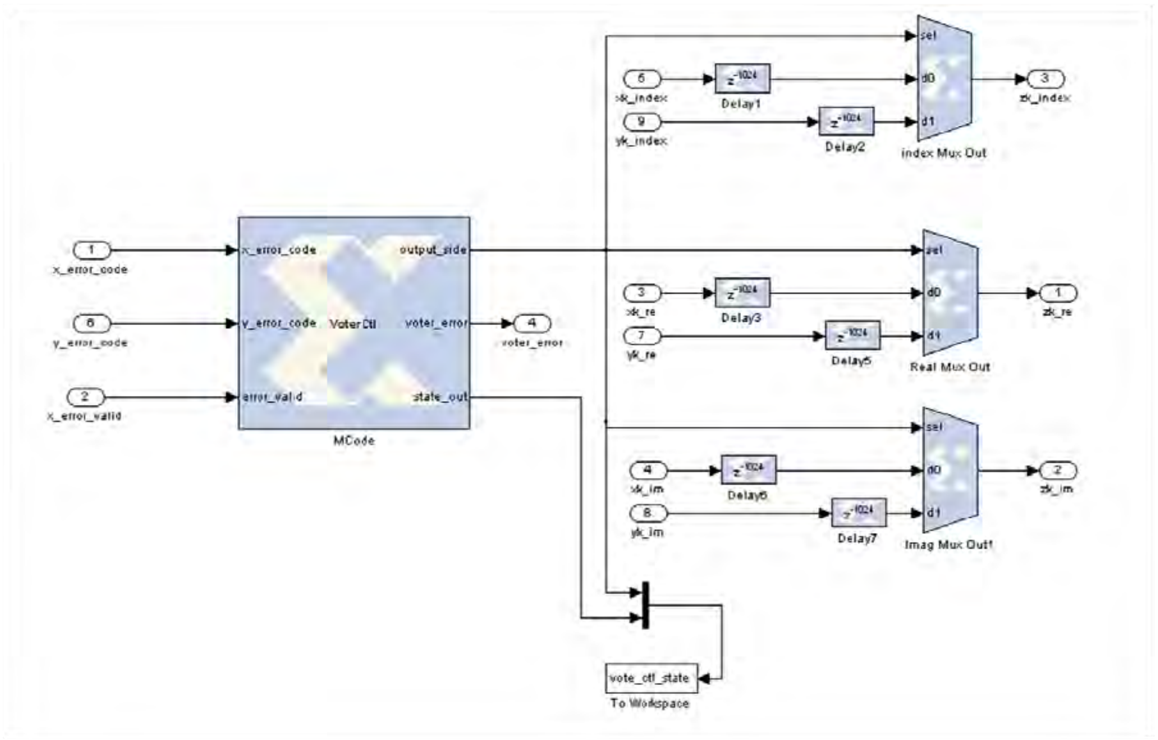


Figure 35. Voter Subsystem.

The error correction circuit was tested by injecting an error into the output of one of the FFTs. In the first plot of Figure 36, the error is shown where the real input transitions from the correct value of 0 to 2^{-6} at time $t = 13924$. The second plot shows that the circuit masks the error when the *output_side* signal transitions from 0 to 1. Note that the output signal is delayed one FFT period of $N = 1024$ due to latency in calculating the error.

This error correction algorithm was implemented for a Virtex™-II xc2v3000. The required resources are shown in Table 10. This shows that this algorithm requires 4109 more slices than are available on the target device, implying that it cannot be implemented on this device.

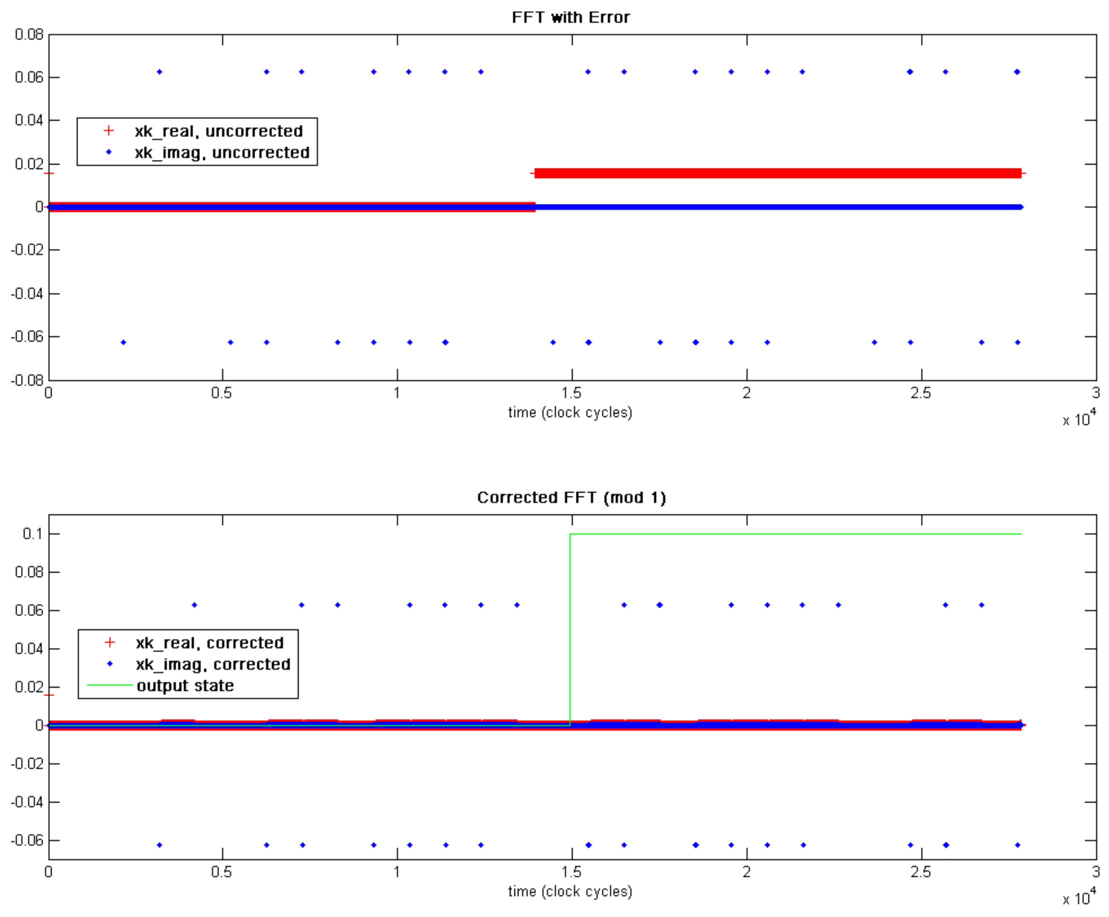


Figure 36. Error Correction of a Single Error

Table 10. Resource Utilization for Error Correcting FFT on Virtex™-II xc2v3000–6fg676.

Resource	**Used**	**Available**	**Percent**
Slices	18445	14336	128%
Flip-Flops	32250	28672	112%
4-input LUTs	32517	28672	113%
Bonded IOBs	131	484	27%
BRAMs	30	96	31%
MULT18x18s	72	96	75%
GCLKs	1	16	6%

B. MODIFICATION TO REDUCE RESOURCE REQUIREMENTS

1. Eliminate Redundant Multiply-and-Accumulate Circuitry

In order to implement error correction on a single Virtex™-II xc2v3000, an implementation using fewer resources was necessary. The first error correction algorithm had a redundant sum-and-accumulate circuit on the input. By eliminating this redundant circuitry, as shown in Figure 37, fewer resources would be required.

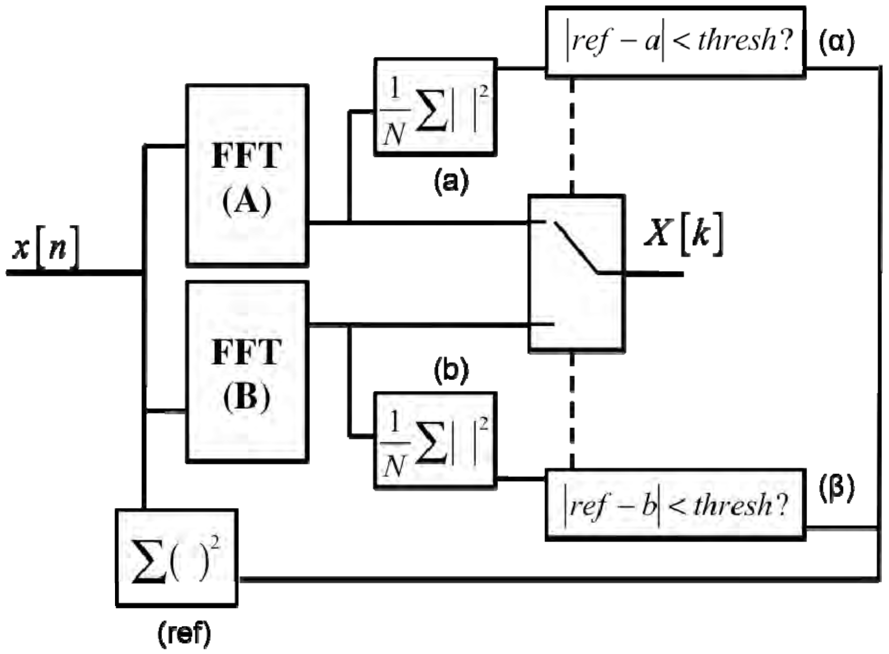


Figure 37. Modification to Error Correction (Conceptual).

The truth table in Table 11 shows that this configuration will also mask any single error. Even if the error occurs in the reference sum-and-accumulate circuit, the voter will recognize that when there are two apparent errors the error is not occurring within one of the FFTs.

The modified error correcting circuit was implemented as shown in Figure 38. In this modified error correction circuit, the multiply and accumulate circuitry was removed from the FFT subsystems, and one copy of it computes the energy of the input signal. This single calculation is routed into the error detection circuits in both FFT subsystems.

Table 11. Truth Table for Modified Correction Algorithm.

Error in Block:	**|a-b| <thresh**	**|c-d| <thresh**	**X[k]**	**X[k] correct**
Σ(ref)	0	0	FFT A	1
Σ(a)	0	1	FFT B	1
Σ(b)	1	0	FFT A	1
FFT A	0	1	FFT B	1
FFT B	1	0	FFT A	1

The circuit was tested in the same manner as the original error correction algorithm. Shown in the first plot of Figure 39, the error is injected into the real part of the FFT at time $t = 13924$. The second plot shows that this error is masked when the error is detected and the *output_side* signal transitions from 0 to 1 at time $t = 14950$, one FFT cycle after the error was injected. This switches the source data from the error FFT to the correct FFT.

This new design was implemented for a Virtex™-II xc2v3000. As shown in Table 12, this modified error correction algorithm uses fewer slices but still requires 2564 more than are available, implying that it cannot be implemented on this device.

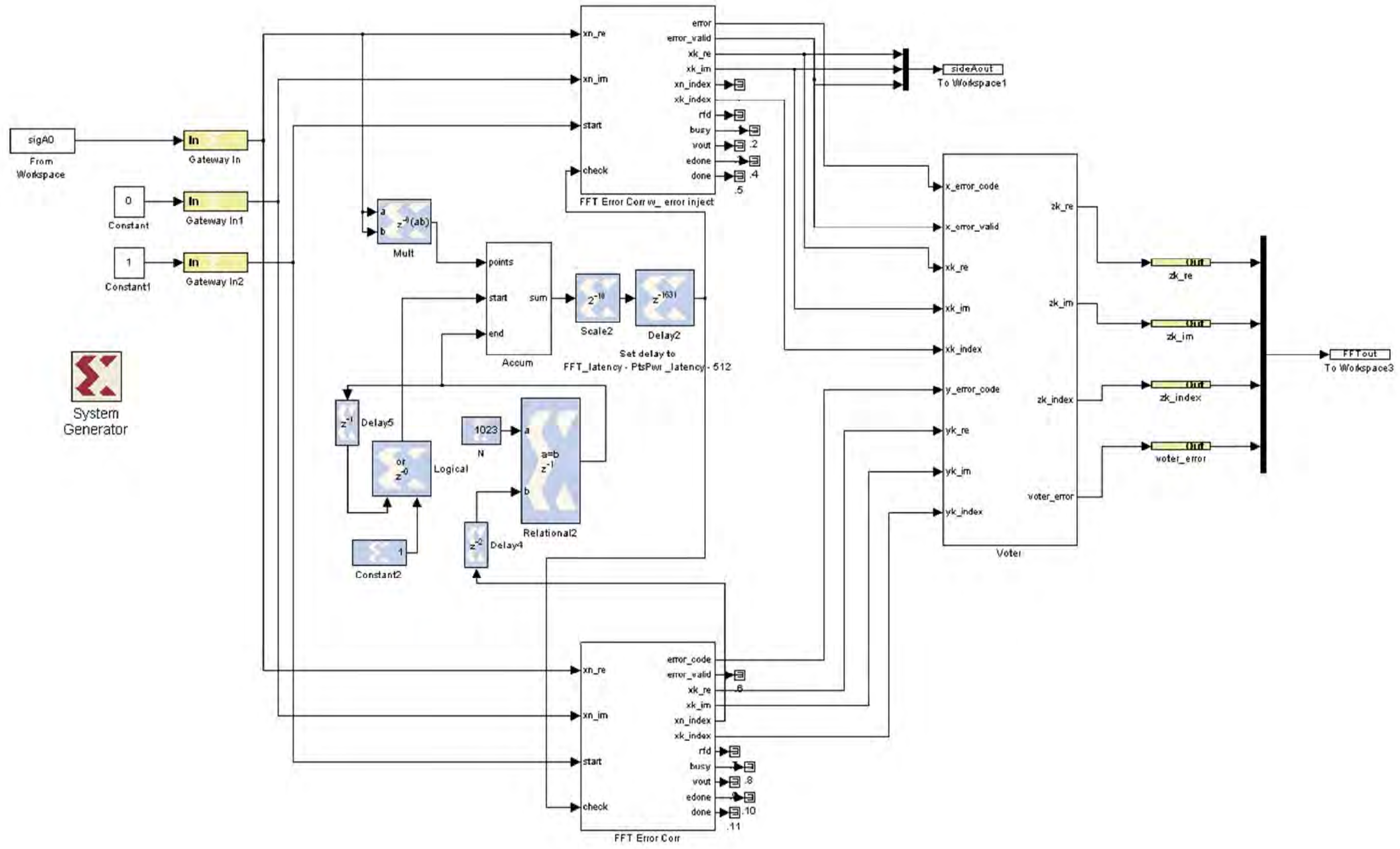


Figure 38. Modification to FFT Error Correction.

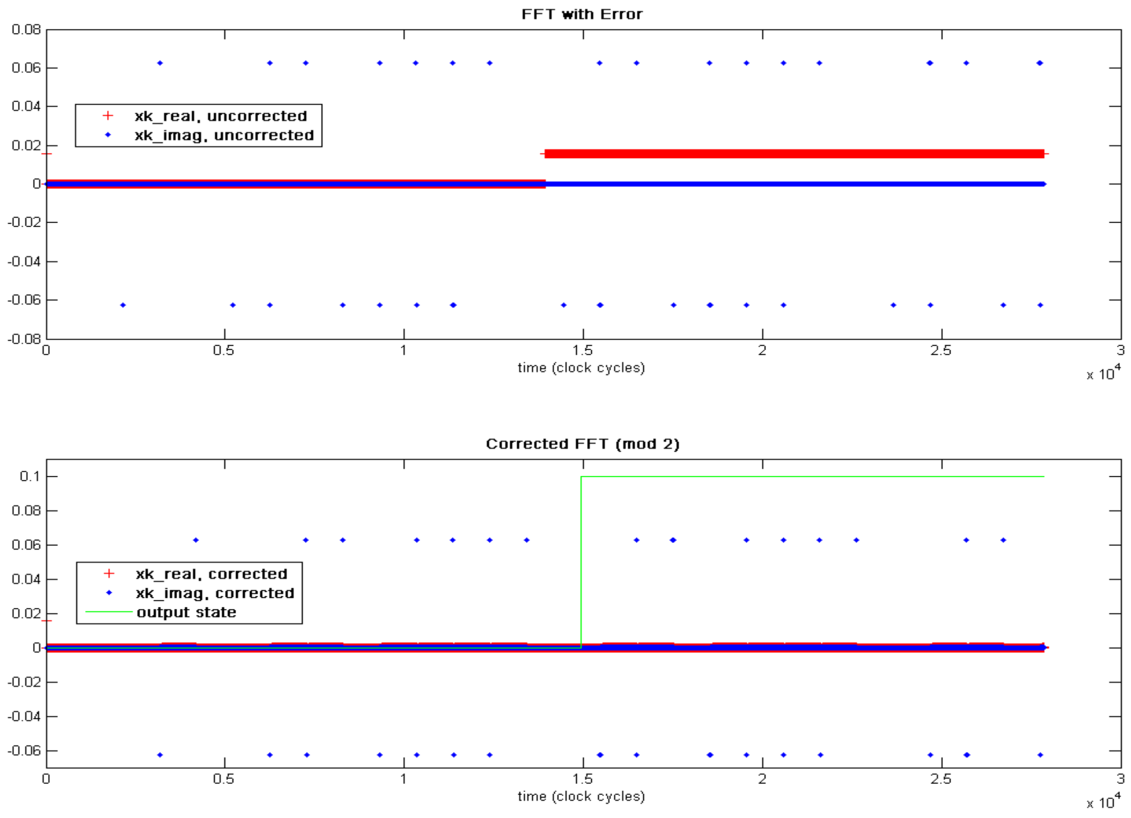


Figure 39. Error Correction of Single Error by the Modified Circuit

Table 12. Resources Required for Modified Error Correction Circuit on Virtex™-II xc2v3000.

Resource	Used	Available	Percent
Slices	16900	14336	117%
Flip-Flops	29335	28672	102%
4-input LUTs	29494	28672	102%
Bonded IOBs	132	484	27%
BRAMs	30	96	31%
MULT18x18s	68	96	70%
GCLKs	1	16	6%

2. Modify Voter to Reduce Usage of Slice Logic

As seen in Table 12, the number of flip-flops required to implement the error correcting algorithm exceeds the number available on the target FPGA. The device utilization summary generated by Xilinx ISE during synthesis stated that 17033 of the LUTs were used as shift registers, which is how delay blocks are implemented. Since the Voter subsystem contains six delays of length 1024, eliminating these delays would significantly reduce the requirement for LUTs. Since this algorithm only used 31% of the available BRAM, the decision was made to modify the voter to use memory rather than logic to implement the delay.

The updated Voter subsystem is shown in Figure 40. The delay blocks have been replaced by Single Port RAM blocks. The depth is set to $2^{10} = 1024$, so each value from the FFT can be read in. The address is provided by the *xk_index* signal. The Single Port RAM has been set to read before write, so that the value from the previous FFT period is read out before the new value is read in. The VoterCtl MCode block remains the same.

This Voter successfully masks all single-source errors. As shown in the first plot in Figure 41, the error is injected into the real output of the FFT at time $t = 13924$. The corrected output is shown in the second plot, where the Voter circuit switches the output from the FFT in error to the FFT computed correctly.

This implementation of error correction was generated for a Virtex™-II xc2v3000 FPGA. As shown by the resource requirements listed in Table 13, this new voter implementation is now within the resource limitations of the target FPGA.

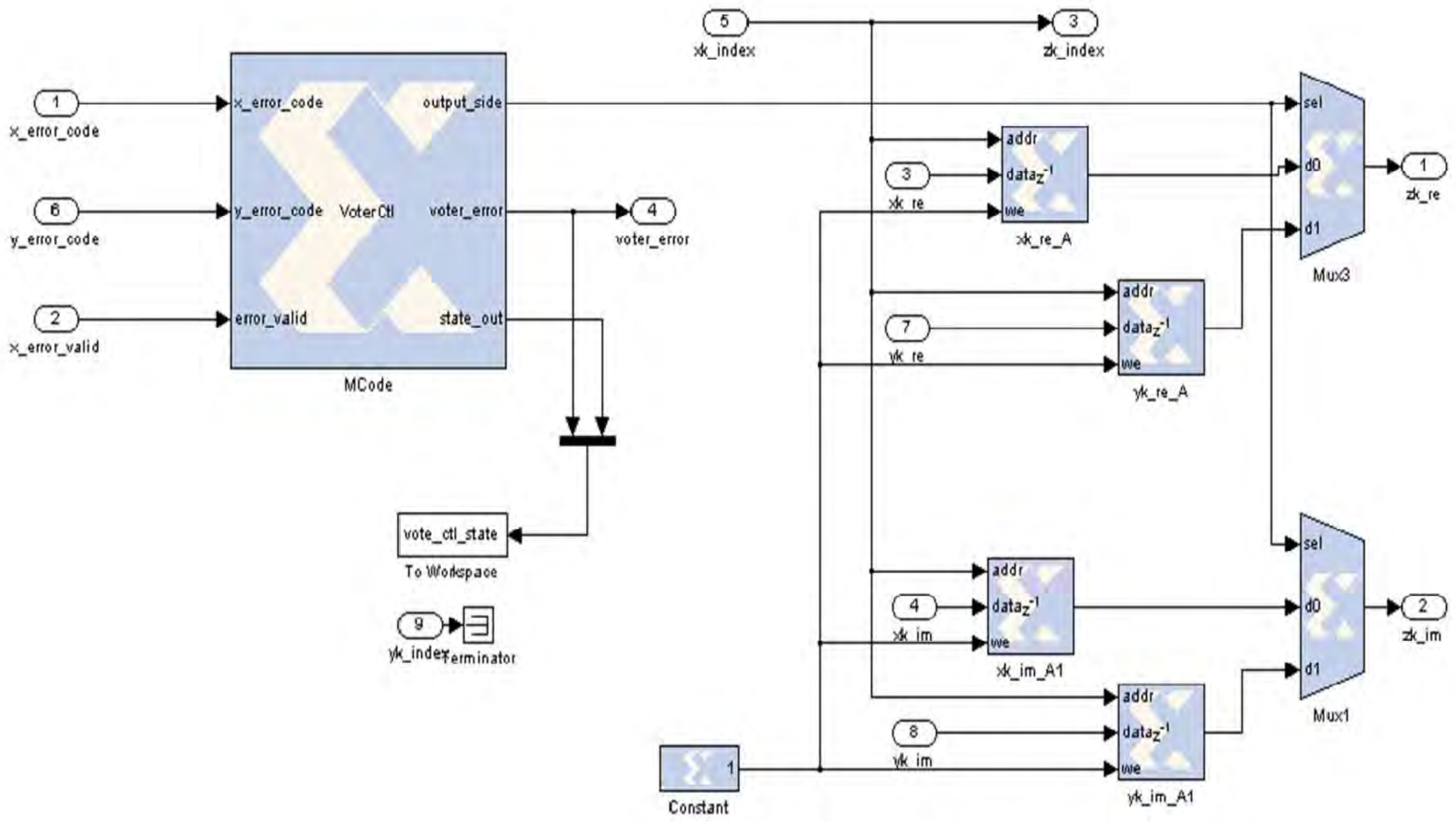


Figure 40. Memory-Based Voter Circuit.

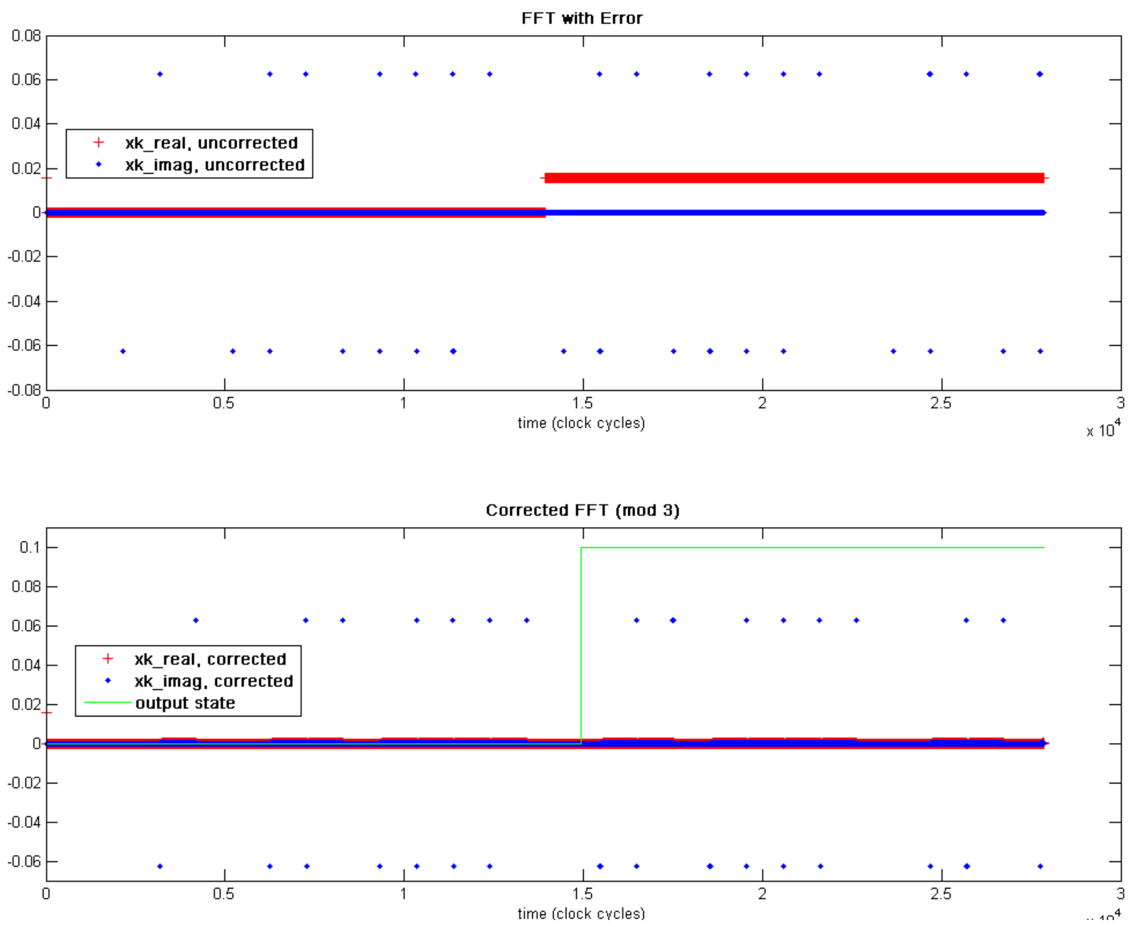


Figure 41. Error Corrected by Memory-Based Voter.

Table 13. Resources Required for Modified Voter Circuit on Virtex™-II xc2v3000.

Resource	Used	Available	Percent
Slices	11952	14336	83%
Flip-Flops	19575	28672	68%
4-input LUTs	19724	28672	68%
Bonded IOBs	131	484	27%
BRAMs	38	96	39%
MULT18x18s	68	96	70%
GCLKs	1	16	6%

C. PROTECTED FFT WITH THE COMPRESSION ALGORITHM

The protected algorithm was used as the input for the windowing and data formatting subsystems. This is a two-FPGA implementation. The FFT circuit used was the one developed in the previous section, modified as shown in Figure 42 to provide the required output signals.

The compression and data format subsystems from [5] were modified as shown in Figure 43. In this implementation, the FFT circuit has been removed and replaced by the input signals generated by the error correcting FFT. The error detection circuit has been removed from the Windowing Algorithm, and the ErrorFlagCtl MCode block has been moved outside of the Windowing Algorithm. The output of the ErrorFlagCtl block is required for the Format Output Subsystem. In the previous algorithm, this block counted the number of errors per time bin. In this algorithm, it performs the same task but is counting errors in the voter rather than errors in the FFT calculation.

The Windowing Algorithm retains the Point Pwr circuit for calculating FFT point energy. This is redundant to the Point Pwr circuit in the error correction circuit; however, the resources required for its implementation are available. The Temporary Storage subsystem retains the parity check system described in [5] and Chapter V.

The compression and data format model was generated for a Virtex™-II xc2v3000 FPGA. The resources required are shown in Table 14. Because this circuit uses so few resources, it would be possible to add some measure of redundancy to the circuit. The required number of BRAM is greater than a third of the available memory so directly implementing TMR would not be possible.

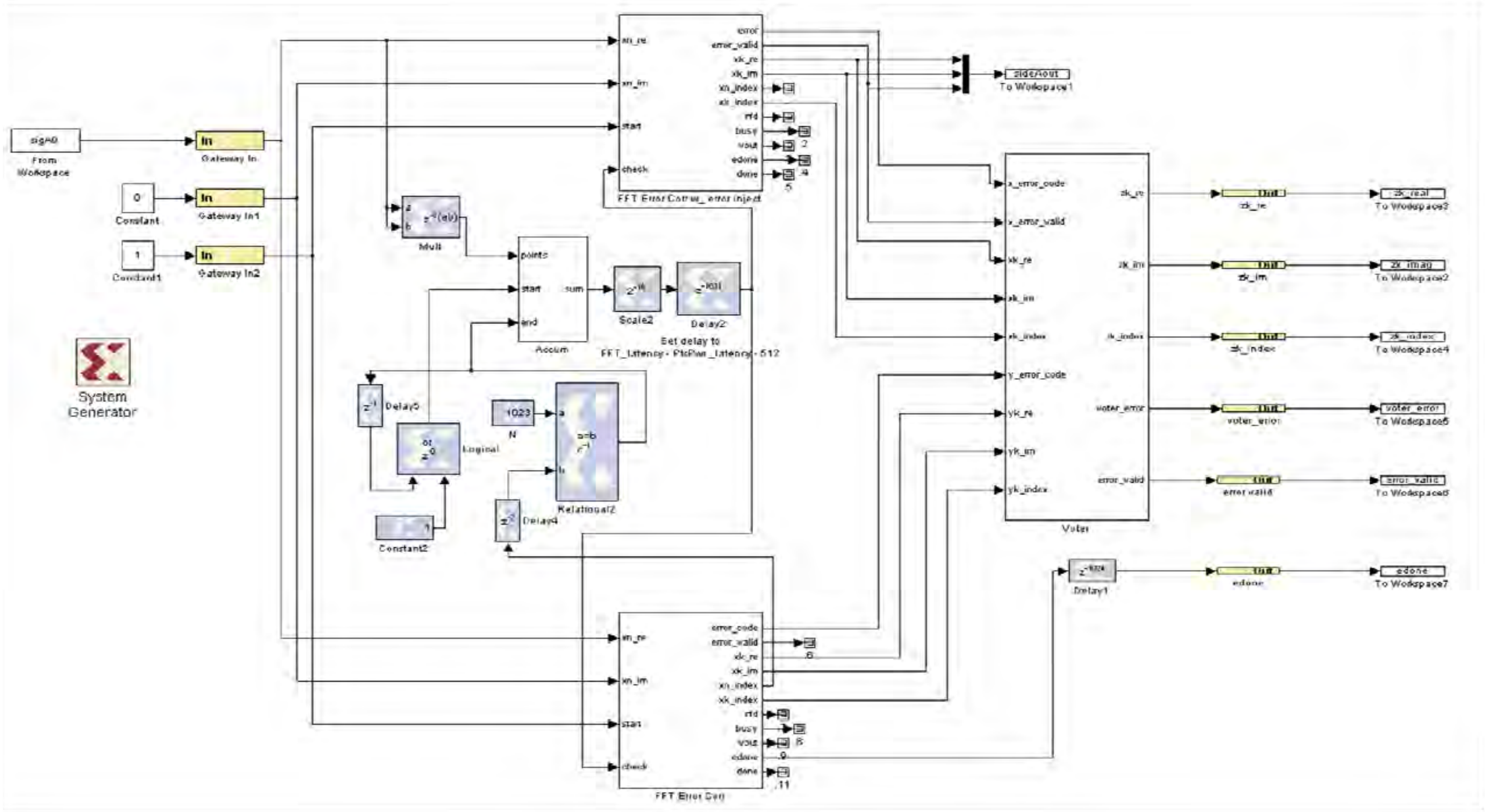


Figure 42. Modified Error Correcting FFT.

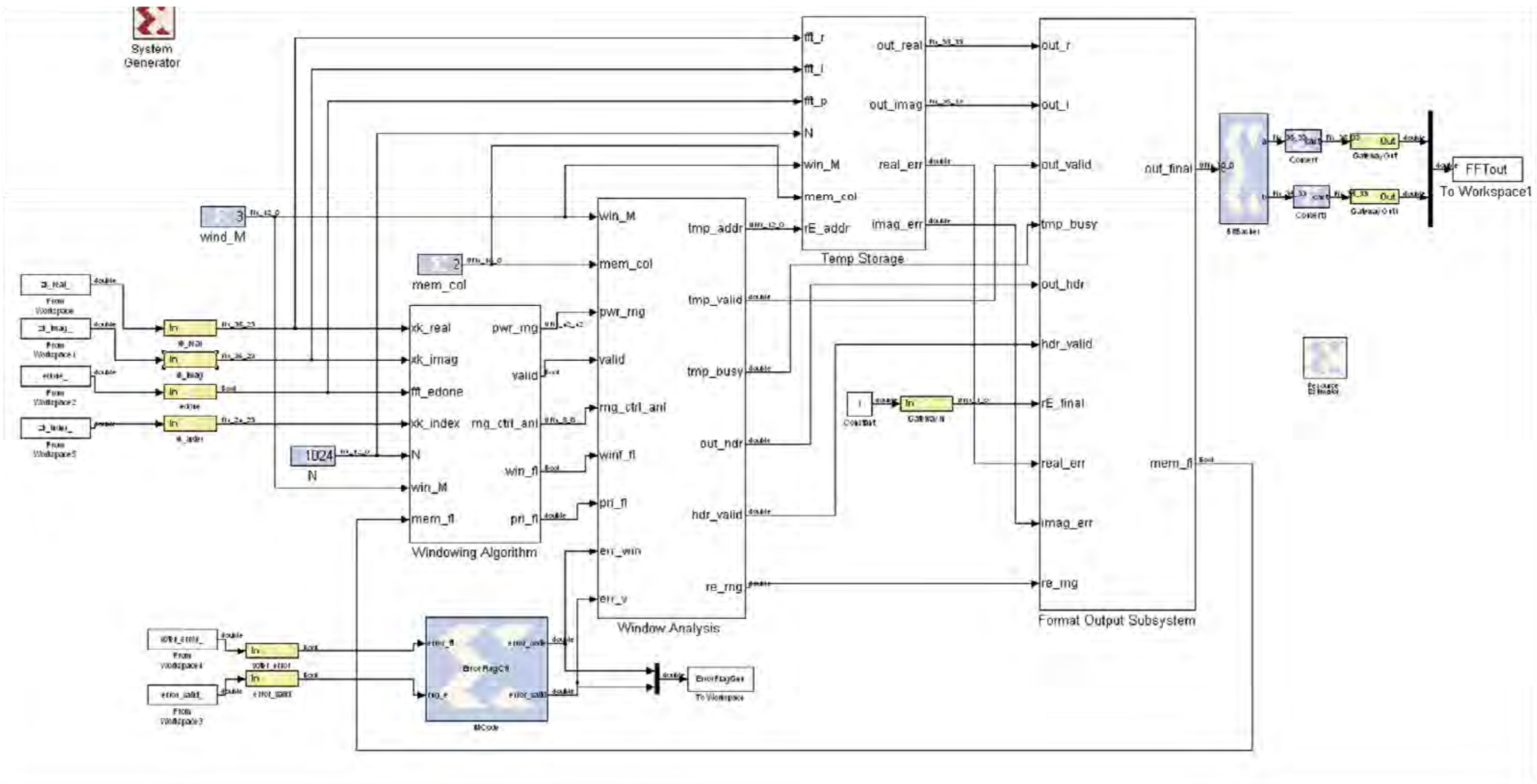


Figure 43. Modified Compression and Data Formatting Circuit (After [5]).

Table 14. Resources Required for Compression and Data Formatting Subsystems.

Resource	Used	Available	Percent
Slices	1262	14336	8%
Flip-Flops	625	28672	2%
4-input LUTs	2194	28672	7%
Bonded IOBs	145	484	29%
BRAMs	36	96	37%
MULT18x18s	9	96	9%
GCLKs	1	16	6%

D. SUMMARY

Methods for implementing error correction for the FFT v3.2 IP block using the error correction method developed in Chapter V were introduced in this chapter. The initial error correction design duplicated the error detection circuitry and used a voter to decide which FFT was computed correctly. This method resulted in errors being corrected and reduced the required resources; however, it required more logic resources than were available on the target FPGA. The next design iteration eliminated one of the redundant input power computation circuits. This design also corrected errors and reduced the required resources, but still required more logic resources than were available. The final design iteration modified the voter circuit to reduce its logic resource requirement by implementing delays using BRAM rather than slice logic. This design successfully corrected errors and could be implemented on the target FPGA. The error correcting FFT circuit was then integrated with the existing compression and data formatting subsystems, and the function was verified.

VII. CONCLUSION

In this chapter, the conclusions drawn from analysis of the original compression algorithm and the modification and design of an error-correcting FFT algorithm are presented. In addition, recommendations for continuation of the design are discussed.

A. SUMMARY

Several high level concepts that support the development of a Fourier transform-based, fault tolerant compression algorithm were presented. Relevant elements of Fourier analysis were discussed, and the development of the fast Fourier transform was presented. Challenges associated with spaceborne computing were discussed, and the Triple Modular Redundancy and Reduced Precision Redundancy methods for correcting errors caused by single event upsets were presented.

The software tools used for development of the compression algorithm were introduced, and the design process from concept, through high-level design, all the way to hardware implementation was presented. Capabilities and limitations of each software package were discussed, and some software setup notes were highlighted.

The performances of the FFT v4.1 and FFT v1.0 IP blocks used in the algorithms presented in [5] were verified. The performance of the FFT v3.2 IP block was investigated. It was shown to behave similarly to the FFT v4.1 block, accepting streaming input and output. Timing differences between the FFT v4.1 and v3.2 and the implications to further design work were discussed.

Methods for detecting errors in the calculation of an FFT when the designer does not have access to the internal circuitry of the FFT block were presented. The design and performance of an error detection circuit applied to the FFT v4.1 block were analyzed, and the error detection was verified. The error detection circuit was modified for use with the FFT v3.2 block, and the error detection was verified. A method of checking memory for error using parity check bits, presented in [5], was also analyzed, and its detection of errors verified.

An error correction scheme for the FFT v3.2 IP block was developed using the error correction method discussed in Chapter V. The initial error correction design duplicated the error detection circuitry and used a voter to decide which FFT was computed correctly. The algorithm proceeded through several design iterations to reduce the resource requirements to allow implementation on the target FPGA. The final design was shown to successfully correct single errors and was implementable on the Virtex™-II xc2v3000 FPGA. The final error correcting FFT circuit was then integrated with the existing compression and data formatting subsystems.

B. CONCLUSIONS

Use of high-level design tools makes rapid design iterations easier; however, they can mask some underlying problems. System Generator makes adjusting designs, adding or modifying components, and displaying output or intermediate signals convenient. Not all designs built and tested within the System Generator environment can be implemented in hardware. Some System Generator components behave differently or have different configuration options than their HDL instantiations. Use of this design tool greatly speeds the design entry – behavioral simulation iterative design loop; however, it does not replace thorough testing at the HDL and hardware levels.

Logic resources on an FFT have to be managed. In this design, delays of 35-bit wide busses for thousands of clock cycles were implemented using register-based delays. While this implementation functioned, it exceeded the logic resources available on the target FPGA. By implementing delays using memory rather than registers, it was possible to fit the algorithm on the intended platform.

It is possible to protect an IP FFT using Parseval’s theorem. By using the error detection algorithm developed in [5], error correction was implemented using a duplicate-and-check methodology. Errors injected into the output of one of the FFTs were successfully detected, and the output of the uncorrupted FFT was used to provide an error-free calculation.

C. RECOMMENDED FUTURE WORK

The design work conducted for this thesis focused on implementing error correction for the FFT in the algorithm presented in [4] and further developed in [5]. To further develop this algorithm into a practical design, the following are suggestions for additional work.

1. Test Error Correction FFT and Compression Algorithm

The two-FPGA algorithm presented at the end of Chapter VI was tested for functionality; however, a thorough investigation of proper operation under all input signal conditions is required.

2. Additional Error Correction Capabilities

The final circuit presented in Chapter VI made use of the error correcting FFT circuit. While the FFT circuit is the most likely point at which an SEU would cause an error, an error at any other point in the circuit would likely also cause an error in the output. Error correction could be implemented in the remaining subsystems. The Temporary Memory subsystem in this implementation employs a single parity bit check, which could detect a single error. This parity could be upgraded to a SECDED Hamming code generator and check in order to be able to correct single errors. The Windowing and Window Analysis subsystems, as well as the Format Output subsystem, could benefit from application of TMR. Use of the Xilinx TMRTool, an IP tool for implementing TMR into an FPGA design, is one possible avenue for future research.

3. Develop a Comprehensive Test Set

As discussed in [5], the signals used for testing the compression algorithm only serve to illustrate that the circuit functions. Development of a signal set that tests the compression algorithm’s limits are required to ensure the algorithm is a robust design. Also, development of a signal set that emulates real-world SOI against a noisy background will allow analysis of the performance of the algorithm under anticipated operating conditions.

4. Conduct Comprehensive Functional Testing

System Generator creates a ModelSim testbench for individual designs. In order to use ModelSim to simulate designs implemented on multiple FPGAs, an HDL wrapper file needs to be created which serves to route signals between the individual VHDL files created by System Generator. A testbench which feeds input signals into the wrapper file and collects output signals also has to be generated. Existing multiple-FPGA systems have an associated HDL wrapper and testbench, so the algorithm could be tested using these existing files.

5. Implement In Hardware

Ultimately, the algorithm must be implemented in hardware in order to be used in the real world. Further investigation into implementing the algorithm in the target FPGAs as well as using ChipScope to analyze in-circuit performance of the algorithm is required.

APPENDIX. COMPUTER FILES

Lists of files used in the design of the algorithms discussed in this thesis are contained in this Appendix. Instructions for simulating and synthesizing the designs are provided.

A. REQUIRED FILES

The following tables contain lists of files and directories contained on the software DVD. A copy of the DVD can be requested from the director of the Communications Research Laboratory. The directory structure of the DVD is listed in Table 15 and Table 16. The System Generator models used for the simulations discussed in Chapters IV, V, and VI are listed in Table 17. The required MCode files are listed in

Table 18. Supporting MATLAB script files are listed in Table 19.

Table 15. DVD directories.

Directory Name	**Purpose**
DurkeInit	Contains the files from [4].
DurkeInit\Mods	Contains the files from [5].
FFT_test	Contains the files from Chapter IV of this thesis used for timing analysis of FFT v4.1, FFT v1.0, and FFT v3.2.
FFT_error_correct	Contains the files from Chapters V and VI of this thesis used in analysis of the error detection algorithm from [5], as well as the development of the error correction algorithm.

Table 16. Subdirectories of FFT_error_correct.

Subdirectory Name	Purpose
err_chk	Contains the System Generator models and supporting MCode files for the FFT error detection algorithms only presented in Chapter V.
FFT_Err_Corr_mod1	Contains the System Generator model and supporting MCode files for the error correction algorithm presented in Chapter VI, Section A.
FFT_Err_Corr_mod2	Contains the System Generator model and supporting MCode files for the error correction algorithm presented in Chapter VI, Section B.
FFT_Err_Corr_mod3	Contains the System Generator model and supporting MCode files for the error correction algorithm presented in Chapter VI, Section C.
Mod8C	Contains the System Generator model and supporting MCode files for the error detection and compression algorithm presented in [5].
Mod9	An early attempt to insert FFT v3.2 into Mod8C.
Mod10	Contains the System Generator model and supporting MCode files for a modification to the compression algorithm presented in [5], inserting FFT v3.2 in place of FFT v4.1.
Mod11	Contains the System Generator models and supporting MCode files for the two-FPGA error correcting algorithm presented in Chapter VI, Section D.

Table 17. System Generator Models.

Model Name	**Chapter in Thesis**	**Description**
fftv41_test.mdl	IV.A	Circuit for analyzing FFT v4.1.
fftv10_test.mdl	IV.B	Circuit for analyzing FFT v1.0.
fftv32_test.mdl	IV.C	Circuit for analyzing FFT v3.2.
FFT_error_chk_only4_1.mdl	V.A.1	Error detection circuit based on FFT v4.1.
FFT_error_chk_only3_2.mdl	V.A.2	Error detection circuit based on FFT v3.2.
SDR1024Mod8C.mdl	V.B	Error detecting compression algorithm from [5], based on FFT v4.1.
SDR1024Mod10.mdl	not discussed	Error detecting compression algorithm based on *SDR1024Mod8C*, modified to use FFT v3.2.
FFT_error_corr3_2mod1.mdl	VI.A	Error correcting circuit based on FFT v3.2.
FFT_error_corr3_2mod2.mdl	VI.B.1	Modification to *FFT_error_corr3_2mod1* to eliminate redundant square-and-accumulate circuit.
FFT_error_corr3_2mod3.mdl	VI.B.2	Modification to *FFT_error_corr3_2mod2* to replace register-based delays in voting circuit with memory-based delays.
FFT_error_corr3_2mod3A.mdl	VI.C	Modification to *FFT_error_corr3_2mod3* to include output signals required for integration with compression circuit.
SDR1024Mod11B.mdl	VI.C	Compression circuit, based on *SDR1024Mod10* which takes FFT input from *FFT_error_corr3_2mod3A.*

Table 18. MCode files required for Compression and Error Detection Algorithms (After [5]).

Algorithm	**File Name (*.m)**	**Description**
accum_ctrl	*accum_ctrl_3_1*	Controls Accum subsystem.
ErrorFlagCtl	*ErrorFlagCtl*	Generates error codes.
out_hdr	*out_hdrMod2*	Generates downlink header, including FFT and parity error codes.
OutputCtl	*OutputCtlMod0*	Controls downlink buffer in Format Output subsystem.
ParityFlagCtl	*ParityFlagCtl*	Generates parity error code from parity check circuit in Temp Storage subsystem.
pwr_time	*pwr_time_MOD2*	Manages Time Windowing subsystem.
re_freq_wind	*re_freq_wind_Mod1*	Manages signals and addressing for ROIs stored in memory in Window Analysis subsystem.
re_tmp	*re_tmp_Mod1*	Manages signals and addressing in Temp Storage subsystem.
we_temp_fft	*we_temp_fft_Mod1*	Manages signals and addressing in Temp Storage subsystem.
we_time_win	*we_time_win_Mod1*	Manages signals and addressing in Temp Storage subsystem.
wind_anal	*wind_anal_Mod1*	Manages signals in Bin Analysis subsystem within Window Analysis subsystem.
VoterCtl	*VoterCtl*	Controls output of voter based on error flags generated by FFT Error Corr subsystems.

Table 19. **Supporting *.m files**

File Name	**Purpose**
fftv41_test_ctrl.m	Generates signals for and displays output of *fftv41_test.mdl*
fftv10_test_ctrl.m	Generates signals for and displays output of *fftv10_test.mdl*
fftv32_test_ctrl.m	Generates signals for and displays output of *fftv32_test.mdl*
ErrorCheck_testing_control.m	Generates signals for and displays output of *FFT_error_chk_only3_2.mdl* and *FFT_error_chk_only3_2.mdl*
fft_err_corr_mod1_test_ctrl.m	Generates signals for and displays output of *FFT_error_corr3_2mod1.mdl*
fft_err_corr_mod2_test_ctrl.m	Generates signals for and displays output of *FFT_error_corr3_2mod2.mdl*
fft_err_corr_mod3_test_ctrl.m	Generates signals for and displays output of *FFT_error_corr3_2mod3.mdl*
Mod11_control_testing.m	Generates signals for and displays output of *FFT_error_corr3_2mod3A.mdl* and *SDR1024Mod11B.mdl*
input_sig_gen.m	Generates time and frequency-varying signal. Required function for all test control *.m files. From [5]
ROI_ctrl.m	Generates ROI for Windowing Algorithm and Window Analysis subsystems in the compression algorithms. Required function for *Mod11_control_testing.m* From [5].

B. INSTRUCTIONS

The following instructions detail the method of executing the supporting *.m script files and running the simulations and are based on the instructions given in [5]. The *test_ctrl* files use Cell Mode execution, detailed in the "Rapid Code Iteration Overview," section of [27].

1. Examine System Generator Model

Open the System Generator model. Examine all MCode blocks and ensure the required MATLAB files are in the same directory as the System Generator Model. [5]

2. Execute Control Script and Run Simulation

Incrementally execute the *test_ctrl.m* file. Cell Mode allows intermediate steps between execution of the code. To run in Cell Mode, place the cursor in the desired cell and either choose the menu option "Cell ➔ Evaluate Current Cell," or type "Control+Enter." Once all environment variables have been set, run the simulation either by executing the cell containing the code `sim(model_name,clks);` or running the model within the Simulink® window. Finally, execute the remaining cells to display the output signals. The two-FPGA algorithm requires a data-reformatting step between running *FFT_error_corr3_2mod3A.mdl* and *SDR1024Mod11B.mdl*. The required code is contained in the *Mod11_control_testing.m* file [5].

3. Generate

System Generator will generate an HDL file and Xilinx ISE project file automatically, as detailed in Chapter III. The options are in the dialog box opened by double-clicking the System Generator token. Choose HDL Netlist under the "Compilation" menu and the desired FPGA under the "Part" menu. It is recommended to set the target directory to a new subdirectory under the directory containing the System Generator and MATLAB files, since System Generator creates numerous supporting files during generation. Select the "Create testbench" option to automatically generate a ModelSim testbench [5].

4. Compile

Open the generated ISE project file in Xilinx ISE. Compilation can usually be accomplished by double-clicking the “Synthesize-XST” process in the “Processes” pane on the left side of the screen. Multiple warnings may be generated during the compile process, however they do not impede the simulation process. After compilation, a device utilization summary, showing a summary of logic resources, will be displayed in the main screen. If compilation fails, error messages detailing the failure will be displayed.

5. Conduct Behavioral Simulation with ModelSim

After compilation, choose “Sources for Behavioral Simulation,” in the drop-down menu in the “Sources” pane. In the “Processes” pane, expand the “ModelSim Simulator” item and double-click the “Simulate Behavioral Model” process. This will launch ModelSim and begin the simulation using the signals contained in the generated testbench. If the simulation fails due to the error “`Library unisim not found`,” then HDL simulation libraries must be compiled as described in Chapter III.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] L. Doggrell, "Operationally responsive space – A vision for the future of military space," *Air and Space Power Journal*, vol. XX, no. 2, pp. 42-49, Summer 2006.

[2] A. A. Ghouwayel, Y. Louet, "FPGA implementation of a re-configurable FFT for multi-standard systems in software radio context," IEEE Transactions on Consumer Electronics, vol. 55, no. 2, May 2009.

[3] F. Iacomacci et al., "A software defined radio architecture for a regenerative on-board processor," NASA/ESA Conference on Adaptive Hardware and Systems, June 2008.

[4] D. Wright, "Field programmable gate array (FPGA) based software defined radio design," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2009.

[5] J. V. Livingston, "A field programmable gate array based software defined radio design for the space environment," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2009.

[6] *Programmable Logic Design: Quick Start Guide*, Xilinx, Inc., UG500 (v1.0), 08 May 2008

[7] A. Gavros, "use of the reduced precision redundancy (RPR) method in a radix-4 FFT implementation," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2010.

[8] R. D. Strum and D. E. Kirk, *First Principles of Discrete Systems and Digital Signal Processing*, Reading, MA: Addison-Wesley Publishing Company, 1989.

[9] CMLab DSP Research Group, Taiwan, "Fast Fourier Transform," http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html. (accessed 15 December 2011).

[10] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: a survey," *Journal of VLSI Signal Processing*, vol. 28, pp. 7-27, Kluwer Academic Publishers, The Netherlands, 2001.

[11] R. C. Olsen, *Introduction to the Space Environment: PH2514 Course Notes*, Naval Postgraduate School, pp. 224–225, January 2005.

[12] J. Snodgrass, "Low-power fault tolerance for spacecraft FPGA-based numerical computing," Ph.D. dissertation, Naval Postgraduate School, Monterey, CA, 2006.

[13] R. W. Hamming, "Error detecting and error correcting codes," *Bell Systems Technical Journal*, vol. 29, pp. 147-160, April 1950.

[14] M. A. Sullivan, “Reduced precision redundancy applied to arithmetic operations in field programmable gate arrays for satellite control and sensor systems,” M.S. thesis, Naval Postgraduate School, Monterey, CA, 2008.

[15] *Virtex™ 2.5V Field Programmable Gate Arrays: Product Specification*, Xilinx, Inc., Xilinx Product Specification DS003–1 (v2.5), 02 April 2001.

[16] *Virtex™-II Platform FPGAs: Complete Data Sheet*, Xilinx, Inc., Xilinx Product Specification DS031 (v3.5), 05 November 2007.

[17] *Virtex™-II Pro and Virtex™-II Pro X Platform FPGAs: Complete Data Sheet*, Xilinx, Inc., Xilinx Product Specification DS083 (v5.0), 21 June 2011.

[18] *Virtex™-4 Family Overview*, Xilinx, Inc., Xilinx Product Specification DS112 (v3.1), 30 August 2010.

[19] *System Generator for DSP, Release 10.1.1: User Guide*, Xilinx, Inc., April 2008

[20] *System Generator for DSP, Release 10.1.1: Getting Started Guide*, Xilinx, Inc., April 2008.

[21] *Xilinx University Program Virtex-II Pro Development System: Hardware Reference Manual*, Xilinx, Inc., UG069 (v1.0), 08 March 2005.

[22] *ModelSim® SE User’s Manual: Software Version 6.3j*, Mentor Graphics Corp., October 2008.

[23] *ChipScope Pro 10.1 Software and Cores User Guide*, Xilinx Inc., UG029 (v10.1) 24 March 2008.

[24] *Fast Fourier Transform v4.1*, Xilinx Inc., Xilinx LogiCore Product Specification DS260, 02 April 2007.

[25] *High Performance 1024-Point Complex FFT/IFFT v1.0*. Xilinx, Inc., Xilinx LogiCore Product Specification, 11 May 2001.

[26] *Fast Fourier Transform v3.2*, Xilinx Inc., Xilinx LogiCore Product Specification DS260, 11 January 2006.

[27] *MATLAB® 7: Desktop Tools and Development Environment*, The MathWorks, Inc., Natic, MA, September 2009, pp. 185-208.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Frank E. Kragh
   Naval Postgraduate School
   Monterey, California

4. Herschel Loomis
   Naval Postgraduate School
   Monterey, California

5. Donna Miller
   Naval Postgraduate School
   Monterey, California

6. Ron Aikins
   Naval Postgraduate School
   Monterey, California

7. Bieu Lu
   SPAWAR Systems Center
   San Diego, California

8. Shawn Kocis
   National Reconnaissance Office
   Chantilly, VA

9. Caleb J. Humberd
   SWOSCOLCOM
   Newport, Rhode Island