



**NAVAL
POSTGRADUATE
SCHOOL**

MONTEREY, CALIFORNIA

THESIS

**USE OF THE REDUCED PRECISION REDUNDANCY
(RPR) METHOD IN A RADIX-4 FFT
IMPLEMENTATION**

by

Athanasios Gavros

September 2010

Thesis Co-Advisors:

Herschel Loomis

Alan Ross

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
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 September 2010	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE Use of the Reduced Precision Redundancy (RPR) Method in a Radix-4 FFT Implementation			5. FUNDING NUMBERS	
6. AUTHOR(S) Athanasios Gavros				
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) Reduced precision redundancy (RPR), as a new method for improving fault tolerance in FPGAs, appears promising in replacing triple modular redundancy (TMR) to prevent the single event effects due to radiation in arithmetic processes. As a test of this approach, the RPR technique was used to implement a Radix-4 fast Fourier transform (FFT). This design was implemented in a Xilinx Virtex 2 FPGA in order to find the possible gain in speed and power as compared to the TMR method. This thesis deals with a 64-point Radix-4 in-place FFT, based on an improved FFT algorithm. The whole FFT structure was implemented based on self-designed modules and by manipulating the embedded Virtex II FPGA's modules. The point was to create a fast and small FFT module that could be altered according to specific application requirements. The implementation of the FFT was successful, managing to handle data in real time at a speed of 134MHz. Based on this FFT design, the next challenge was the implementation of TMR and RPR modules. The first attempt was the TMR structure, implemented by creating three identical replicas of the FFT and installing a voter per FFT stage. This implementation was unsuccessful due to space limitations. The next step was the alteration of the existing FFT and the creation of a smaller 8 x 8 bit butterfly module for the RPR structure. After the successful completion of this step, implementation of a RPR module with an 8/32 degree was commenced. Ambiguities and inefficient radiation protection were identified in this implementation. Finally, adopting a new RPR approach and a higher degree of 14/32, a smooth and correct RPR module was created that could work in real time, and handle data at a speed of 163MHz. Both TMR and RPR with a degree of 14/32 methods were compared, confirming the RPR's advantage in power consumption and in occupied FPGA's resources.				
14. SUBJECT TERMS Reduced Precision Redundancy (RPR), Triple Modular Redundancy (TMR), Field Programmable Gate Array (FPGA), Fast Fourier Transform (FFT)			15. NUMBER OF PAGES 135	
			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	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**USE OF THE REDUCED PRECISION REDUNDANCY (RPR) METHOD IN A
RADIX4 FFT IMPLEMENTATION**

Athanasios Gavros
Lieutenant Junior Grade, Hellenic Navy
Bachelor of Naval Science, Hellenic Naval Academy, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
September 2010**

Author: Athanasios Gavros

Approved by: Herschel H. Loomis, Jr.
Thesis Co-Advisor

Alan A. Ross
Thesis Co-Advisor

R. Clark Robertson
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Reduced precision redundancy (RPR), as a new method for improving fault tolerance in FPGAs, appears promising in replacing triple modular redundancy (TMR) to prevent the single event effects due to radiation in arithmetic processes. As a test of this approach, the RPR technique was used to implement a Radix-4 fast Fourier transform (FFT). This design was implemented in a Xilinx Virtex 2 FPGA in order to find the possible gain in speed and power as compared to the TMR method.

This thesis deals with a 64-point Radix-4 in-place FFT, based on an improved FFT algorithm. The whole FFT structure was implemented based on self-designed modules and by manipulating the embedded Virtex II FPGA's modules. The point was to create a fast and small FFT module that could be altered according to specific application requirements. The implementation of the FFT was successful, managing to handle data in real time at a speed of 134MHz.

Based on this FFT design, the next challenge was the implementation of TMR and RPR modules. The first attempt was the TMR structure, implemented by creating three identical replicas of the FFT and installing a voter per FFT stage. This implementation was unsuccessful due to space limitations. The next step was the alteration of the existing FFT and the creation of a smaller 8 x 8 bit butterfly module for the RPR structure. After the successful completion of this step, implementation of a RPR module with an 8/32 degree was commenced. Ambiguities and inefficient radiation protection were identified in this implementation. Finally, adopting a new RPR approach and a higher degree of 14/32, a smooth and correct RPR module was created that could work in real time, and handle data at a speed of 163MHz. Both TMR and RPR with a degree of 14/32 methods were compared, confirming the RPR's advantage in power consumption and in occupied FPGA's resources.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
	A. OBJECTIVE	2
	B. FFT DESIGN OVERVIEW	2
	C. BACKGROUND	2
	1. Space Environment and FPGA.....	2
	2. Fault Tolerance Methods—Redundancy	3
	D. ORGANIZATION OF THIS THESIS.....	4
II.	PREVIOUS WORK.....	5
	A. PREVIOUS THESIS	5
	1. Gkikas Thesis	5
	2. Sullivan Thesis.....	6
	B. PREVIOUS PAPERS	8
	1. Application Reports and Notes	8
	2. Papers.....	8
III.	THEORETICAL APPROACH	11
	A. PROBLEM DISCUSSION.....	11
	1. Discrete Fourier Transform.....	11
	2. Fast Fourier Transform	12
	a. Radix-2 Versus Radix-4 Versus Split-Radix	13
	b. In-Place Versus Constant Geometry Structure.....	15
	B. CONCEPTUAL DESIGN MODEL	16
	1. 64-Point Radix-4 in-Place DIF	16
	2. An Improved Radix-4 DIF FFT Algorithm.....	19
IV.	DESIGN IMPLEMENTATION DETAILS OF FFT	21
	A. DESCRIPTION.....	21
	1. Stage 1-2-3.....	22
	a. Stage Controller	24
	b. RAM.....	24
	c. Compute Factor.....	24
	d. BF Multiplier	24
	e. Multiplexer	31
	2. Main Controller	31
	3. Reversing Last Stage.....	31
	B. IMPLEMENTATION EFFORTS AND RESULTS	31
	1. Implementing a TMR	31
	2. Implementing a RPR—First Attempt—RPR Degree 8/32.....	32
	a. RPR Upper and Lower Modules.....	33
	b. Overflow Approach	34
	c. Ambiguity Phenomenon	35
	d. RPR Survey Problems.....	36
	3. Implementing a RPR—Second Attempt—RPR Degree 14/32.....	38

a.	<i>RPR Bound Modules</i>	38
b.	<i>Verifying Results</i>	39
V.	RESULTS	43
VI.	CONCLUSIONS AND RECOMMENDATIONS	47
A.	SUMMARY	47
B.	RECOMMENDATIONS FOR FUTURE STUDY	48
1.	Overflow Manipulator	48
2.	Implementing a 4 Recursive Butterfly FFT Instead of the 12 Butterfly FFT	48
APPENDIX A.	FFT IMPLEMENTATION	49
A1.	FFT MODULE—XILINX SCHEMATIC DESIGN	50
A2.	FFT MODULE—XILINX BEHAVIORAL AND STRUCTURAL DESIGN	52
A2.1.	Main Controller of TMR	52
A2.II.	Reversing Bit of Last Stage Module	53
A2.III.	Controller of First Stage Timescale 1ns / 1ps	54
A2.IV.	RAM	57
A2.V.	Multiplexer and Voter of Each Stage	60
A2.VI.	Compute Factor	60
A2.VII.	BF's Multiplier	65
A2.VIII.	Multiplier 32x32 Module	67
A2.IX.	BF Multiplier—CSA module	69
A2.X.	BF Multiplier—CLAH module	70
APPENDIX B.	TMR IMPLEMENTATION	75
B1.	TMR MODULE—XILINX SCHEMATIC DESIGN	76
B2.	TMR MODULE—XILINX BEHAVIORAL AND STRUCTURAL DESIGN	78
B2.I.	Multiplexer and Voter of Each Stage	78
APPENDIX C.	RPR IMPLEMENTATION	81
C1.	RPR MODULE—XILINX SCHEMATIC DESIGN	82
C2.	RPR MODULE—XILINX STRUCTURAL AND BEHAVIORAL DESIGN	85
C2.I.	Radiation Module	85
C2.II.	Multiplexer and Voter of Each Stage	89
C2.III.	BF—Delayer	92
C2.IV.	BF—Multiplier	93
APPENDIX D.	MATLAB FILE	95
D1.	MAIN FILE	95
	LIST OF REFERENCES	115
	INITIAL DISTRIBUTION LIST	117

LIST OF FIGURES

Figure 1.	Radix-2 DIF Butterfly (From [15])	13
Figure 2.	Radix-4 DIF Butterfly (From [16])	14
Figure 3.	Split Radix DIF (From [16])	14
Figure 4.	Radix-2 DIF FFT with in-place input and output (From [15]).....	15
Figure 5.	Radix-2 DIF FFT with constant geometry structure (From [15])	16
Figure 6.	16-Point Radix-4 DIF FFT Butterfly (From [8]).....	17
Figure 7.	64-Point Radix-4 DIF FFT	18
Figure 8.	64-Point Radix-4 in Shape DIF Signed Fixed Point 32Bit FFT—The Upper Level	22
Figure 9.	64-Point Radix-4 in Place DIF Signed Fixed Point 32bit FFT—Stages 1&2&3..	23
Figure 10.	BF Multiplier—Use of Embedded Multipliers—Part1	26
Figure 11.	BF Multiplier—CSA module—Part2.....	28
Figure 12.	BF Multiplier—CLAH Module—Part3 (From [19])	29
Figure 13.	Multiplier—Overview—Part1 & Part2 & Part3	30
Figure 14.	Reduced Precision Redundancy Stage Portrayal.....	34
Figure 15.	RPR Shield and 3-Bit Shifting Obstacle	37
Figure 16.	FFT and Radiation Module	41
Figure 17.	TMR Vs RPR Synthesis Results	44
Figure 18.	TMR Vs RPR—XPower Analyzer Comparison	45
Figure 19.	FFT Design—Entirely Layout.....	50
Figure 20.	FFT’s First Stage Layout.....	51
Figure 21.	RAM Structure	52
Figure 22.	TMR’s Entirely Concept Layout.....	76
Figure 23.	TMR’s First Stage Layout.....	77
Figure 24.	TMR’s BF Module	78
Figure 25.	RPR- Entirely Module Layout.....	82
Figure 26.	RPR Module—Three Stages Layout.....	83
Figure 27.	RPR—First Stage Layout.....	84
Figure 28.	RPR—Butterfly	85

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	Characteristics of 64-point Radix-4 FFT	20
Table 2.	Expected Errors of Precision Calculation	40
Table 3.	Errors Expected Due to the Import of Radiation in the Precise Module	42
Table 4.	Synthesis Results—Comparison between RPR and TMR in a Virtex II XC2V8000-5FF1152 FPGA	43
Table 5.	TMR Versus RPR—XPower Analyzer Results.....	44

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Field Programmable Gate Arrays (FPGA) are integrated circuits containing programmable logic components and interconnectors that are able to be used for the creation of complex logic functions. They are characterized by the unique ability to be updated or changed depending on current requirements. One of the main applications for FPGAs is in the space industry where arrays must be updated or reconfigured without being physically accessed. Difficulties arise in the use of such devices in the harsh space environment where guard methods against single event effects (SEE) from radiation are required.

A simple solution to this problem is the use of the Triple Modular Redundancy (TMR) method. This is a process that protects the whole FPGA configuration against SEE, with the compromise of demanding a significant amount of resources. Snodgrass recognized the problem and introduced, in his PhD dissertation in 2006 [1], a new method of fault tolerance. This method, referred to as the Reduced Precision Redundancy (RPR) method, is an alternative way of implementing and protecting the required design. It can be used only for arithmetic processes and requires fewer resources than TMR. The use of RPR requires a compromise between capacity demand and output's precision, depending on the "degree"—the measure of reduction of precision—of RPR.

The objective of this thesis was the creation of a fast Fourier transform (FFT) structure that could be implemented in a FPGA of the Virtex II family, adopting both methods of redundancy, TMR and RPR, in order to investigate the performance of RPR. First, a simple 64-point Radix-4 in-place FFT was implemented that could handle fixed-point two's complement numbers. This structure was tested with accurate results. Next, a TMR structure was designed by replicating three identical FFT structures and by importing a voter into the end of each stage. The design was successful, but the implementation failed to fit within due to size constraints of the Virtex II FPGA, revealing the significant demand for resources of the TMR method. The next step was the design of a RPR structure with a degree of $8/32$ – 8 bits reduced precision and 32 bits

precise result. The design was successfully implemented, but the protection against radiation failed due to ambiguities and errors that were not considered at that time. Taking into consideration the problems from the previous unsuccessful design, a new RPR structure with a degree of 14/32 was designed and implemented. This implementation worked correctly and protected the FFT structure efficiently.

Based on the research conducted in this thesis, an alternative RPR method is suggested, where there is no actual need for generating upper and lower bounds. Instead, the truncation and duplication of the precise number, in combination with theoretical boundary calculations, is sufficient. This alteration assists in simplifying the logic and decreasing the size of the voter.

Finally, both TMR and RPR methods were implemented successfully in a slightly larger Virtex II demonstrating the advantage of RPR over TMR in resource requirements and power consumption.

ACKNOWLEDGMENTS

I would like to express my gratitude and sincere appreciation to Professors Herschel H. Loomis, Jr. and Alan A. Ross for their guidance, encouragement and support in the completion of this work.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Modern satellites are capable of handling controls, communications, observation systems, and on-board payload data processing tasks. The inaccessibility of a satellite after launch and the need for periodic updates of the satellite's data handling procedures create a strong argument for using field programmable gate array (FPGA) instead of application-specific integrated circuit (ASIC) technologies. The fact that a FPGA can be reconfigured any time it needs to be is a major advantage and the main reason that they are preferred in a spacecraft circuit design.

Spacecraft computer systems must be able to operate reliably, despite the harsh radiation environment. High energy protons from the Van Allen radiation belt, cosmic rays from outer space, and highly charged ions from solar flares are just a few of the threats that need to be taken into consideration [2] in spacecraft circuit design. In order to prevent radiation effects, hardening of the device is a significant priority. But, continuously decreasing transistor sizes and the simultaneous increase in operating frequencies are obstacles to our hardening efforts.

Radiation can cause unwanted effects, such as the flipping of a memory cell's state in semiconductor devices, better known as soft errors or single event upsets (SEU). FPGAs are susceptible to errors in both data and architecture configuration caused by SEU. In order to prevent this faulty behavior, triple modular redundancy (TMR) is commonly used to ensure reliable operation. However, TMR is very costly in terms of chip area and power consumption. Current research is focused on introducing new methods of fault tolerance for space-borne reprogrammable computers. At the Naval Postgraduate School (NPS) in 2006, a new method of fault tolerance was introduced [1], referred to as reduced precision redundancy (RPR). RPR applies redundancy only to the most significant numerical bits of a circuit and in this way significantly decreases the needed chip area and power consumption over that required for TMR.

In this thesis, a Radix-4 64-point FFT is implemented in a Virtex II XC2V6000 FPGA chip using a Xilinx interface in order to further examine the effectiveness of the RPR method. The reason for the choice of this chip was its existence in the space-flight prototype processor, CFTP-2 [3].

A. OBJECTIVE

Reduced precision redundancy (RPR), a new method of fault tolerance in digital arithmetic processors, appears promising as a technique for replacing triple modular redundancy (TMR) against single event effects due to radiation. In order to demonstrate this promise, the RPR technique is used to implement a Radix-4 fast Fourier transform (FFT). This design is implemented with different degrees of RPR in a Xilinx Virtex II FPGA in order to find possible improvements in chip area, speed, and power consumption compared with the TMR method.

B. FFT DESIGN OVERVIEW

The basic design goal is to implement a FFT design based on the RPR concept in a Virtex2 XC2v6000. This efforts focuses on the creation of one FFT Radix-4 $N = 64$ point that can handle one 32-bit input signal in every clock period (in real time) and at the maximum possible frequency, combined with two RPR modules of the same philosophy with a degree of $8/32 = 0.25$.

C. BACKGROUND

1. Space Environment and FPGA

The space environment is harsh and has serious impact on any spacecraft that orbits, even for a short period of time. Thermal imbalances, erosion or surface damage are just a few of the threats. In this thesis, efforts to prevent or decrease the impact of a different threat, space radiation, are examined.

The space radiation environment is encountered by the majority of space missions and is the result of galactic cosmic rays (GCRs) of particles emitted by solar events and of particles trapped in the Earth's radiation belts [2]. GCRs are highly energetic, heavy

protons and ions, reaching energies in excess of 10 GeV/nucleon. The GCR environment in interplanetary space changes with the phase of the solar cycle. Particles emitted by solar events are an important contribution to radiation and are correlated with the eleven-year solar cycle. This is due to the fact that large solar events are more frequent during solar maxima than during solar minima. Particles trapped in the Earth's radiation belts, sometimes called the Van Allen belts, are most significant between the altitudes of approximately 1,000 km to 32,000 km. These particles consist of electrons, protons and heavy ions that are trapped in the Earth's magnetic field. Spacecraft shielding is capable of protecting against only some of these particles [2].

This radiation and the inefficiency of our shielding can cause unwanted effects on space-borne circuits, better known as single event effects (SEE). SEE can take many forms and some of them can be more destructive than others. In this thesis, the non-destructive SEE, single event upset, known as a soft error is considered. A soft error is the transient corruption of a single bit of data. Unfortunately, FPGAs are susceptible to soft errors, in both data and the architecture configuration, both of which are stored in memory.

2. Fault Tolerance Methods—Redundancy

One way to maximize a system's fault tolerance is to use triple modular redundancy (TMR). The basic design of TMR consists of three identical copies of the operation that is required to be "secure." The three identical system results are processed by a voting system to produce a single output. If any one of the three systems fails, the other two systems can correct and mask the fault. The disadvantage of using TMR is the high chip area occupation and the increased power consumption [4]. Snodgrass [1] suggested the concept of a new fault tolerance method, the reduced precision redundancy (RPR) method that allows a sacrifice in level of precision in an arithmetic calculation, in return for decreased power and capacity demand. Instead of implementing three identical copies of a circuit and voting the result, one fully functional copy of the circuit and two reduced precision copies, that will define the upper and lower limit of the function's output, are created. The voter then compares the three values and checks to determine if

the function's result is within the limits of the reduced precision values or if an error has occurred. In either case a result can be generated. Depending on where the exact point of the error is, the result will be a precise or less-precise output calculation.

D. ORGANIZATION OF THIS THESIS

Chapter II, Previous Work, describes all thesis and various previous work on which this thesis is based.

Chapter III, Theoretical Approach, describes the theoretical background and the algorithm choices that rising for the design of a FFT.

Chapter IV, Design Implementation Details and Description of the Implementation Effort, provides a description of the modules and data management strategies used to develop the FFT design and explains the implementation efforts for TMR and RPR modules.

Chapter V, Results, contains an analysis of the occupied resources and power consumption for both TMR and RPR modules.

Chapter VI, Conclusion and Recommendations, contains a summary of the total effort and recommendations for future work.

II. PREVIOUS WORK

A. PREVIOUS THESIS

This thesis is based on the dissertation of Joshua D. Snodgrass [1] and on two theses, from Nikolaos Gkikas [5] and Margaret A. Sullivan [4], respectively. These works reveal valuable information and results that are combined in the design efforts examined in this thesis.

1. Gkikas Thesis

Gkikas describes the design of a Radix-4 FFT for wireless communications (wireless local area networks (LANs)) [5]. He compares different algorithm structures, the complexity, memory needs, data flow and controller complexity of each of them, searching for the most suitable algorithm for his intended application.

First, he analyzes the difference between the decimation Fourier transform (DFT) and FFT, Radix-2, Radix-4 and Split-Radix algorithms. Based on theoretical approaches, he reports that the FFT provides a faster solution than the DFT. Comparing Radix-2 to Radix-4, he concludes that Radix-4 requires fewer multiplications, so it is the preferred solution. Investigating the Split-Radix algorithm, he calculates that it requires even fewer multiplications than the Radix-4, but it has the disadvantage of greater complexity. Continuing his investigation, he states some useful information about the decimation-in-time-frequency algorithm (DITF), the fast Harley transform and the quick Fourier transform. His final conclusion for his application is the adoption of the Radix-4 FFT.

Secondly, he analyzes the structure of a Radix-4, 64-point, in-place decimation in-frequency (DIF) FFT, identifying its major modules. He explains the basic principles of each module and writes a brief note about the interconnections and the possible controlling algorithms that he uses in his C++ implementation. His design contains four major components. The first component is a computer factor component that generates the needed factors. The second component is comprised of two 64-sample ROMs that store the computed phase factors and the twiddle factors. The third component is a

multiply accumulator that consists of one butterfly (BF) machine, one adder and two registers, to compute the real or imagery part of the stage's output. Finally, the fourth component is a controller that synchronizes all of the components.

In the third part of his thesis, Gkikas examines the structure of the butterfly machine, and based on the theory of the improved Radix-4 algorithm, derived from the Cooley and Tukey algorithm [6], he suggests the use of the factor terms. This structure is helpful because a minor modification to the factor's equations, while leaving the rest of the design intact, provides an inverse FFT (iFFT) algorithm module. Moreover, the use of factors improves data reusability and decreases the number of required computations. He also studies the phase factors and the correct use of them concerning this structure.

Finally, he examines three different structures of FFT, based on the number of butterfly machines that each of them includes. He first considers a 48-butterfly FFT, where each of the three stages has 16 butterflies and that each butterfly is used only once in every 64-point block of data. After that he examines the use of a 16-butterfly FFT, where all three stages use the same 16 butterflies, so that each butterfly is used three times, once per stage. Then, he explores the use of a 4-butterfly FFT, where each butterfly is used 12 times, four times per stage, for three stages.

2. Sullivan Thesis

Based on Snodgrass' dissertation [1], Sullivan illustrates the application of RPR as a new method of fault tolerance in FPGAs against single event effects [4]. She examines the use of different degrees of RPR depending on the arithmetic operation and compares the impact of implementation of those degrees to area consumption. Specifically, she categorizes the problems that are suitable for RPR implementation into two major divisions. These categories depend on the required computation, and are addition/subtraction or multiplication/division. Later she investigates the possibilities of implementing RPR in a FFT algorithm.

For each category, she describes the mathematical relationships, and depending on the two operands, she defines each case. She determines the lower and upper bounds, and at the same time, points to special cases that demand unique handling in order to

avoid possible errors or overflows. In a similar manner, she designs a RPR voter for each task, acknowledging the differences between addition voter and multiplication voter, and provides useful tips about the necessary signals, functions behavior and checking comparators of the module.

In the final portion of her research on the arithmetic operations, she comes to the conclusion that the use of RPR instead of TMR in the addition/subtraction operation does not guarantee less area occupancy in every case. After checking the results of FPGA's area comparison, she confirms that "... in order for RPR to be a more desirable fault-tolerance approach than TMR for a simple operation like addition or subtraction, the degree of RPR must be significantly less than 0.5 - and that for both the adder and the voter in a RPR addition process to be smaller than the analogous TMR modules the degree of RPR must be less than 0.25 [4]."

Moreover, she concludes that for the multiplication operation "... a RPR multiplication module requires 1/3 to 1/2 the FPGA slices of a TMR multiplication module depending on the degree of RPR [4]." However, she notes that the size of the multiplication RPR voter is extremely large when compared to the TMR voter module, a drawback that should not be underestimated. Another crucial question that she tries to answer is the dilemma of whether to test intermediate results or just the final result for error. She points out, "Any benefit of testing intermediate results in RPR processes must be considered against the additional space it requires [4]."

Later, she explains the basics of a FFT algorithm and reveals thoughts about different ways of implementing a RPR voter in a FFT. According to those thoughts, "...we may include one or more voters on the final or intermediate results [4]." Therefore, there are many choices; for example, we can implement a voter after each multiplication, or after each complex product or maybe at the end of each stage. She predicts that using an 8/32 degree of RPR in a FFT butterfly operation instead of a TMR will save 66 percent of occupied FPGA slices. Finally, she concludes that using a RPR voter for a few major points within the system will keep the area cost of the FPGA low compared to the cost of TMR.

B. PREVIOUS PAPERS

Various papers, application reports and notes were also considered in this thesis. The following notes are not an abstract from these reports. Only the ideas and thoughts that were considered useful for the design are discussed.

1. Application Reports and Notes

In [7], Wu describes the implementation of a Radix-4 DIF FFT using the Texas instrument TMS320C80 digital signal processor (DSP). Although the report is dated, it reveals basic FFT implementation principles in a parallel processor and points out possible errors and/or hazards due to data overflow. In [8], Delphin of ST Industries describes the implementation of the Radix-4 FFT Algorithm using the ST120 DSP. There she explains the basic structure of a FFT Radix-4 and a new algorithm that is derived from the Cooley and Tukey algorithm [6]. She discusses the correct order and the characteristics of each stage BF's factors and twiddle numbers. In order to save memory resources, she chooses the "in-place" FFT structure, but she notices the need for each stage's output digit reversing procedure. Finally, she provides a C++ example program that reveals the structure of specific parts of her design.

2. Papers

There are three relevant papers that discuss the design and implementation improvements of a Radix-4 FFT algorithm in a FPGA. Each of them provides important advice concerning the structure of the design considered in this thesis. In the first paper [9], Sun, Liu and Ji give a detailed description of the FFT's memory structure, RAM storage capabilities and restrictions of RAM used in a FPGA. They consider a new way of handling data addresses to allow manipulation of four input and output data streams, while at the same time, bypassing the limitations of modern two-port RAM hardware. In the second paper [10], Bouguezal discusses the profits of using the improved Radix-4 FFT algorithm in comparison to the Cooley-Tukey FFT algorithm [6]. In the third paper [11], Chao, Qin, Yingke and Chengde work on the design of a high performance FFT processor, based on a FPGA. They introduce two important ideas for the optimization of the FPGA, ideas that could be implemented in future iterations of the design presented in

this thesis. The first idea is the lifting scheme, which is a transform that reduces the number of real multiplications in a BF from four to three. While of benefit, note that it also increases the number of real additions from two to three. The second idea is the use of an adaptive overflow calculation. This novel approach ensures that no overflow takes place over the entire calculation. This is performed without limiting the data path width or decreasing the efficiency of the BF.

With important information from previous theses, application notes and papers considered, the theoretical background required for the creation of the FFT design is discussed in Chapter III.

THIS PAGE INTENTIONALLY LEFT BLANK

III. THEORETICAL APPROACH

A. PROBLEM DISCUSSION

In order to successfully implement a FFT in a FPGA, many aspects must be considered and wise decisions must be made concerning different algorithms, geometries and decimations. Each approach has advantages and disadvantages but the most suitable one must be selected. The goal is the implementation of a 64-point FFT that can handle data in real time, while occupying as few resources as possible. Before discussion of design optimization, where the three basic categories of decimations, geometries and algorithms must be considered, it is useful to describe the discrete Fourier transform.

1. Discrete Fourier Transform

The discrete Fourier transform (DFT) plays an important role in many applications of digital signal processing. The reason for its importance is the presence of efficient algorithms for computing the DFT.

The DFT sequence $\{X(k)\}$ of N complex-valued numbers, given a sequence of data $\{x(n)\}$ of length N , is computed as:

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

where:

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

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

For each value of k , N complex multiplications ($4N$ real multiplications) and $N-1$ complex additions ($4N-2$ real additions) are required. This indicates that in order to compute all N values of the DFT, N^2 complex multiplications and $N^2 - N$ complex additions are required.

Unfortunately, direct computation of the DFT is inefficient due to the fact that “...it does not exploit the symmetry and periodicity properties of the phase factor W_N [12].” These properties are summarized as:

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

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

The solution to this problem is the decomposition of the N -point DFT into successively smaller DFTs. This new approach, named divide-and-conquer, made the computation of the DFT more efficient, leading to fast Fourier transforms (FFT) algorithms.

2. Fast Fourier Transform

The fast Fourier transform (FFT) is an efficient algorithm that uses a reduced number of arithmetic operations as compared to DFT, “... eliminating redundancies that result from adding certain data sequence values after they have been multiplied by the same factors of fixed complex constants during the evaluation of different DFT transform coefficients. The efficiency is achieved at the expense of reordering the data sequence, but the additional expense is generally small compared to the reduction in multiplications and additions.” [13]

In order to use an efficient FFT algorithm based on the divide-and-conquer approach, the number of data points, N , is highly composite, meaning N can be described as $N = r_1 * r_2 * r_3 * \dots * r_u$, where $\{r_j\}$ are prime.

In the case where $r_1 = r_2 = r_3 = \dots = r_u = r$, then $N = r^u$ and the number r is called the radix of the FFT algorithm. Each FFT radix- r algorithm can be categorized into two groups, depending on the decimation chosen. The two groups are the decimation in-time (DIT) algorithms, where time samples are computed in alternating groups, and the decimation in-frequency (DIF) algorithms, where frequency samples are computed, separately, in alternating groups.

a. Radix-2 Versus Radix-4 Versus Split-Radix

When comparing the three major FFT algorithms, Radix-2 DIF, Radix-4 DIF and Split-Radix DIF, the following conclusions can be made [5], [14]:

The Radix-2 DIF algorithm (Figure 1) significantly decreases the number of complex multiplies compared to the DFT from N^2 to $(N/2)\log_2 N$ and complex additions from $N(N-1)$ to $N\log_2 N$. It is a very popular algorithm due to the symmetry and periodicity properties of its twiddle factors.

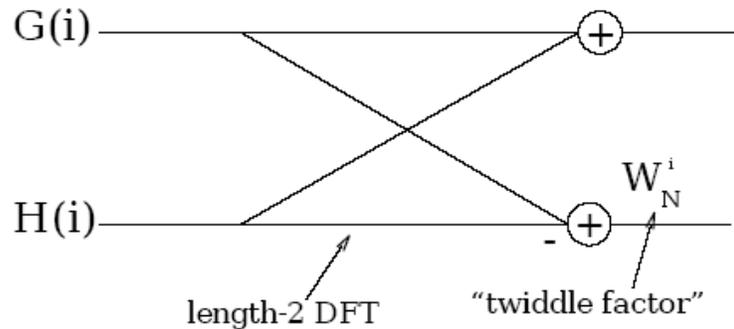


Figure 1. Radix-2 DIF Butterfly (From [15])

The Radix-4 DIF algorithm (Figure 2) yields an even greater decrease in the number of complex multiplies as compared to the Radix-2 DIF. The reduction is from $(N/2)\log_2 N$ to $(3N/8)\log_2 N$. Therefore, the Radix-4 DIF requires 75 percent as many multiplies as Radix-2 DIF and the same number of additions. It also demonstrates the same symmetry and periodicity properties of its twiddle factors.

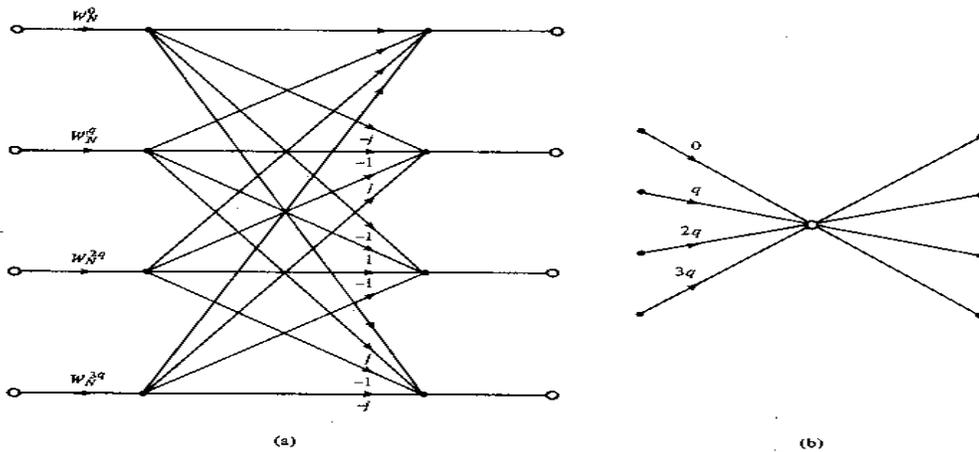


Figure 2. Radix-4 DIF Butterfly (From [16])

The Split-Radix DIF (Figure 3) is computationally superior to both the Radix-2 and Radix-4 algorithms when considering the number of required multiplications and additions. However, this algorithm has the disadvantage of structure irregularity, a disadvantage that prohibits its use in an implementation design effort [12]. So, after comparing these three popular algorithms, the Radix-4 DIF algorithm is selected for the design considered in this thesis.

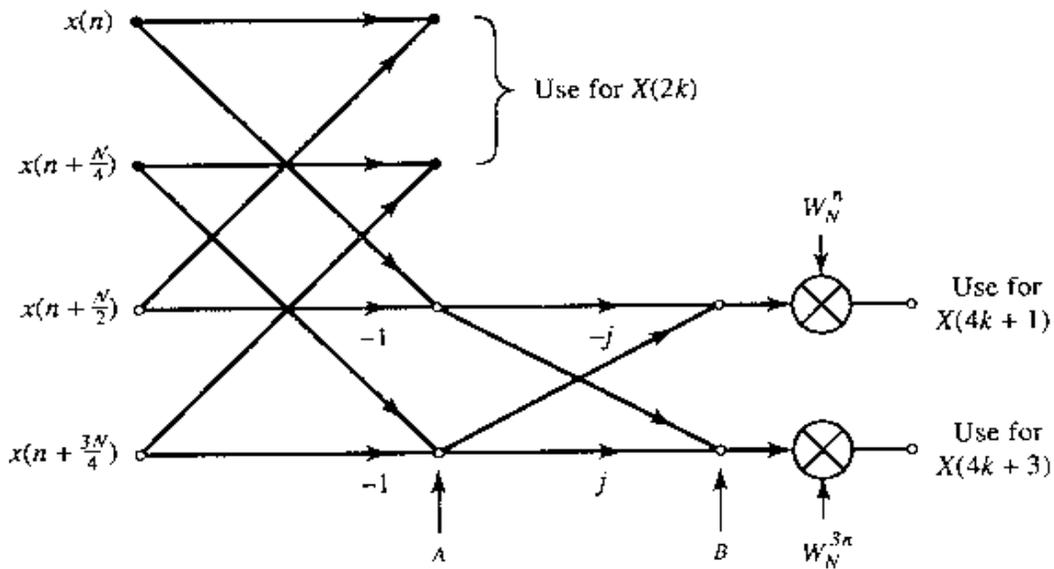


Figure 3. Split Radix DIF (From [16])

b. In-Place Versus Constant Geometry Structure

The term “in-place” (Figure 4) refers to the fact that each time a butterfly is computed, the correct set of data is read from memory and the products from the butterfly computation are written back into the same set of places in memory [17]. The advantage of the “in place” is the fact that the set of data that are needed each time is the same set of data that are produced by the butterfly. This means that data can be overwritten, in order to minimize memory requirements. This is very useful in implementing a FFT in an old FPGA, where the goal is using the least amount of required memory, vice achieving real time signal processing.

The term “constant geometry” (Figure 5) indicates that the connections between memory slots are the same in each stage. The advantage of this structure is the fact that it is less complicated than the “in-place” structure, reducing hardware size. Obviously, “constant geometry” seems more attractive for a highly parallel hardware implementation [18], but the potential of the “in-place” in an old FPGA results in its preference for the thesis design.

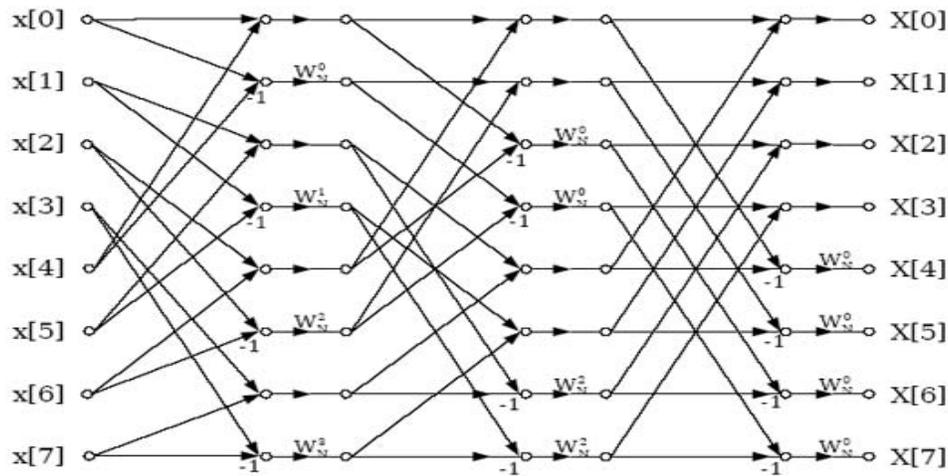


Figure 4. Radix-2 DIF FFT with in-place input and output (From [15])

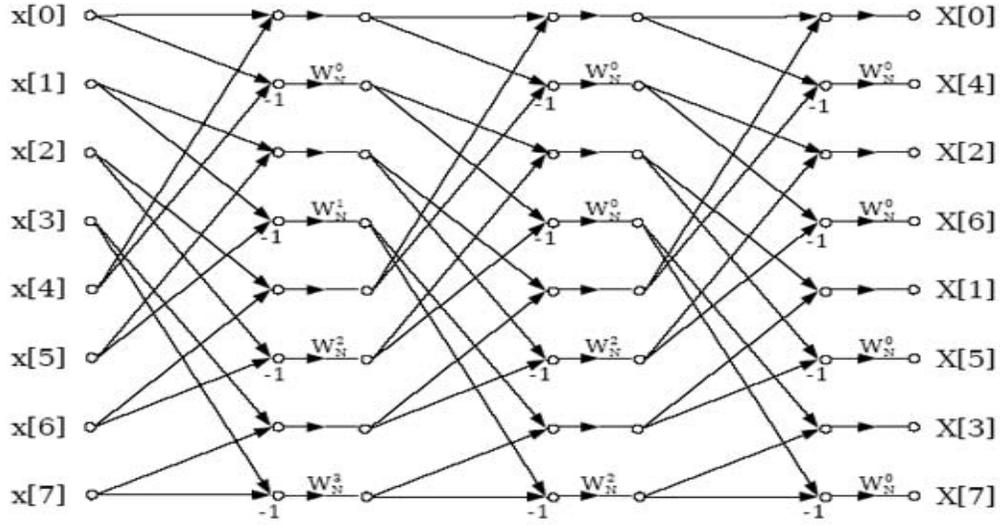


Figure 5. Radix-2 DIF FFT with constant geometry structure (From [15])

B. CONCEPTUAL DESIGN MODEL

In order to evaluate RPR concepts in a FFT, many aspects are considered in choosing an implementation. The final choice for this thesis is a 64-point Radix-4 in-place DIF FFT implementation in the Virtex-2 XC2V6000 FPGA. The following section provides a brief overview of the theoretical approach to this thesis design.

1. 64-Point Radix-4 in-Place DIF

The Radix-4 DIF FFT divides the 64-point DFT into four 16-point DFTs, then into 16 four-point DFTs [7-8]. The butterfly of a Radix-4 consists of four inputs and four outputs, as shown below:

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

where: $W_N^{\frac{N}{4}k} = (-j)^k$

$$W_N^{\frac{N}{2}k} = (-1)^k$$

$$W_N^{\frac{3N}{4}k} = (j)^k$$

Combining terms yields the following equation:

$$X(k) = \sum_{n=0}^{N/4-1} [x(n) + (-j)^k x(n + \frac{N}{4}) + (-1)^k x(n + \frac{N}{2}) + (j)^k x(n + 3\frac{N}{4})] W_N^{kn}$$

To get four-point DFT decomposition, we decompose again as:

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

This is called the butterfly and is repeated for all four-point bundles as shown in Figures 6 and 7.

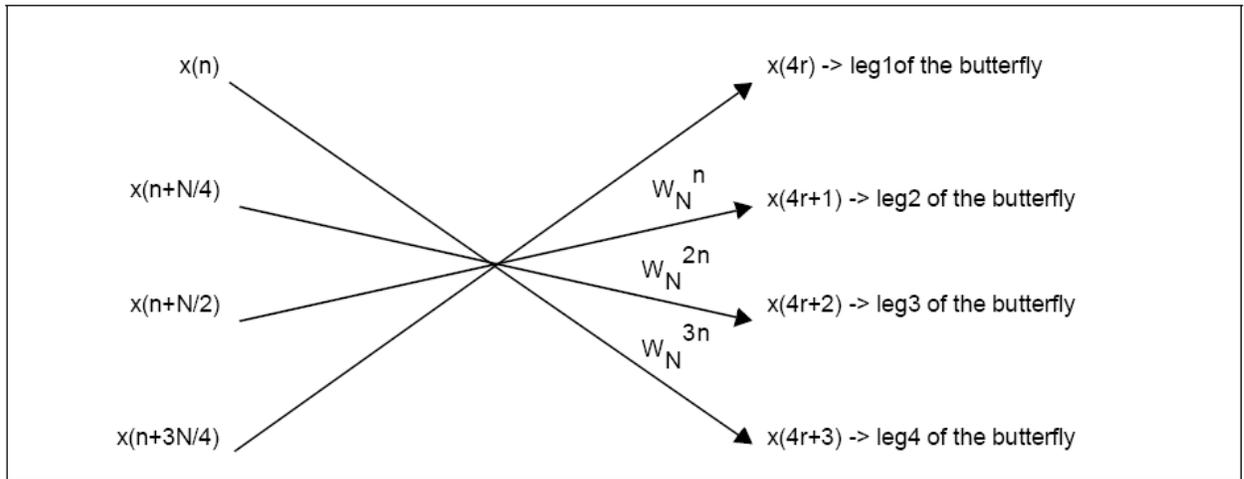


Figure 6. 16-Point Radix-4 DIF FFT Butterfly (From [8])

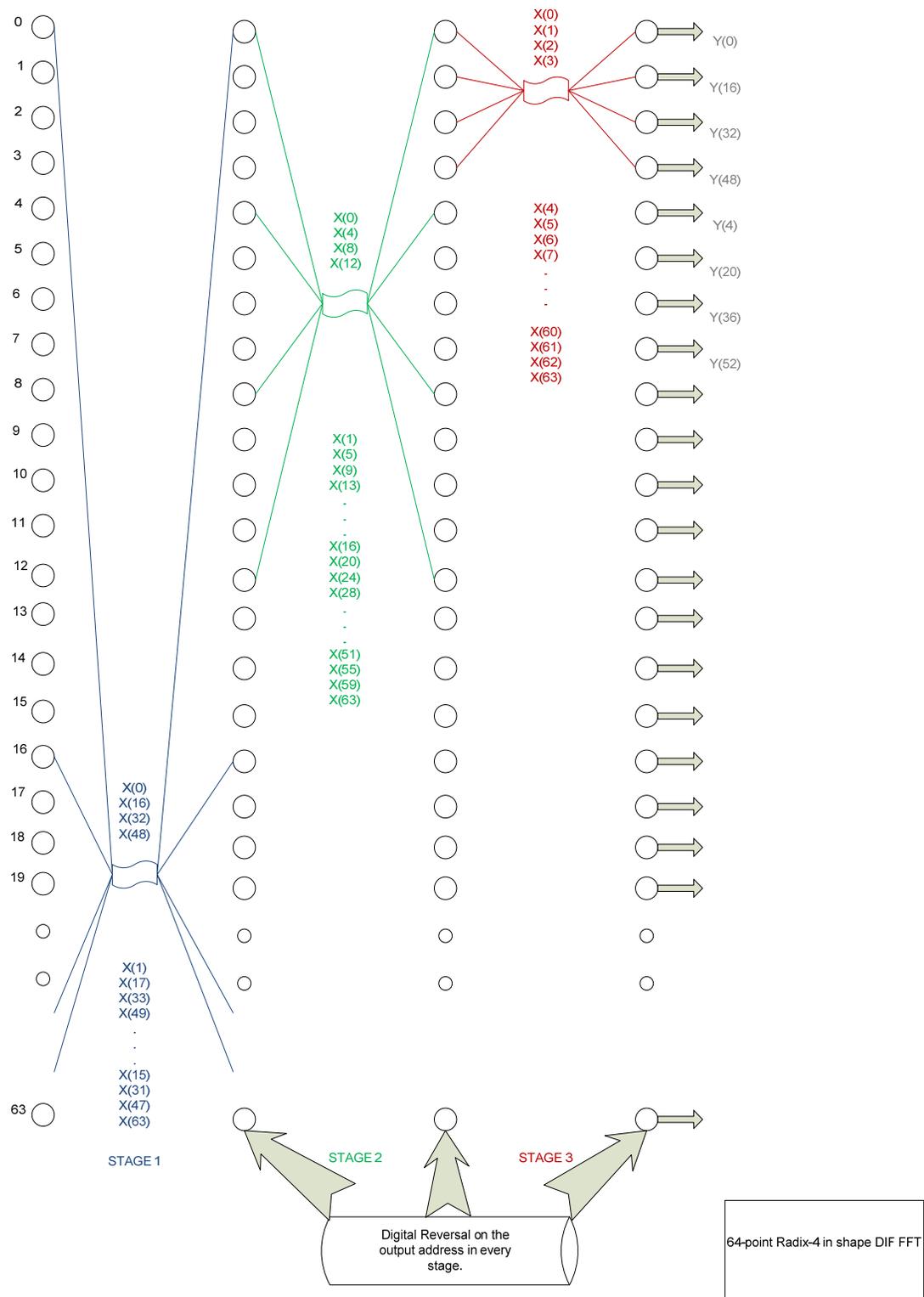


Figure 7. 64-Point Radix-4 DIF FFT

2. An Improved Radix-4 DIF FFT Algorithm

The improved Radix-4 DIF FFT algorithm is derived from the following equations.

The basic equations of the FFT's butterfly are:

$$\begin{aligned} A &= [a + b + c + d]W_N^0 \\ B &= [a - jb - c + jd]W_N^n \\ C &= [a - b + c - d]W_N^{2n} \\ D &= [a + jb - c - jd]W_N^{3n} \end{aligned}$$

where:

$$\begin{aligned} a &= a_r + ja_{im} \\ b &= b_r + jb_{im} \\ c &= c_r + jc_{im} \\ d &= d_r + jd_{im} \end{aligned}$$

and

$$W_N^k = W_{N-r}^k + W_{N-im}^k$$

Transforming the upper equations yields:

$$\begin{aligned} A &= [(a_r + ja_{im}) + (b_r + jb_{im}) + (c_r + jc_{im}) + (d_r + jd_{im})]W_N^0 \\ B &= [(a_r + ja_{im}) - j(b_r + jb_{im}) - (c_r + jc_{im}) + j(d_r + jd_{im})]W_N^n \\ C &= [(a_r + ja_{im}) - (b_r + jb_{im}) + (c_r + jc_{im}) - (d_r + jd_{im})]W_N^{2n} \\ D &= [(a_r + ja_{im}) + j(b_r + jb_{im}) - (c_r + jc_{im}) - j(d_r + jd_{im})]W_N^{3n} \end{aligned}$$

For the computation of the real and imaginary part of the first output of the butterfly, only the butterfly's inputs are added, without multiplying any phase factors.

$$\begin{aligned} A_r &= a_r + b_r + c_r + d_r \\ A_{im} &= a_{im} + b_{im} + c_{im} + d_{im} \end{aligned} \tag{III.1}$$

The real and imaginary parts of the B, C, D outputs produce six common factors:

$$\begin{aligned}
factor_1 &= a_r + b_{im} - c_r - d_{im} \\
factor_2 &= a_r - b_r + c_r - d_r \\
factor_3 &= a_r - b_{im} - c_r + d_{im} \\
factor_4 &= a_{im} - b_r - c_{im} + d_r \\
factor_5 &= a_{im} - b_{im} + c_{im} - d_{im} \\
factor_6 &= a_{im} + b_r - c_{im} - d_r
\end{aligned} \tag{III.2}$$

Expressing the B, C, D outputs of the butterfly in factor terms:

$$\begin{aligned}
B_r &= factor_1 * W_{N_r}^n + factor_4 * W_{N_{im}}^n \\
B_{im} &= factor_4 * W_{N_r}^n - factor_1 * W_{N_{im}}^n \\
C_r &= factor_2 * W_{N_r}^{2n} + factor_5 * W_{N_{im}}^{2n} \\
C_{im} &= factor_5 * W_{N_r}^{2n} - factor_2 * W_{N_{im}}^{2n} \\
D_r &= factor_3 * W_{N_r}^{3n} + factor_6 * W_{N_{im}}^{3n} \\
D_{im} &= factor_6 * W_{N_r}^{3n} - factor_3 * W_{N_{im}}^{3n}
\end{aligned} \tag{III.3}$$

where:

$$\begin{aligned}
W_{N_r}^{kn} &= \cos(2\pi kn / N) \\
W_{N_{im}}^{kn} &= -\sin(2\pi kn / N)
\end{aligned}$$

based on the reference of Table 1.

Table 1. Characteristics of 64-point Radix-4 FFT

Stage	1	2	3
Butterflies per group	1	4	16
Butterfly group	16	4	1
Twiddle Factor Exp.			
Leg1	0	0	0
Leg2	n	4n	16n
Leg3	2n	8n	32n
Leg4	3n	12n	48n
	n=0 to 15	n=0 to 3	n=0

Based on the theoretical background discussed above, it is clear that the best design implementation for this application is a 64-point Radix-4 in place FFT, which can handle 32-bit fixed point signals in real time.

IV. DESIGN IMPLEMENTATION DETAILS OF FFT

This chapter focuses on the design implementation of the FFT. The objective is to create a FFT that contains only modules, based on this research, designed in a Xilinx environment. This implementation is for a Virtex II FPGA XC2V-6000-6bf957.

A. DESCRIPTION

The FFT design contains three identical stage modules, one main controller and a last-stage reversing module, as shown in Figure 8. It receives as inputs, a 32-bit real number and a 32-bit imaginary number, both of which are fixed-point, normalized, two's complement numbers. It produces as outputs, a 32-bit real number and a 32-bit imaginary number. The detailed FFT design is included in Appendix A. According to analysis conducted in this thesis and based on an application report from Cheng [18], there is the possibility of a 3-bit overflow per stage, meaning a 9-bit overflow for the entire FFT. In order to avoid overflow issues, a simple solution was chosen. Input values were restricted to values less than 2^{-9} .

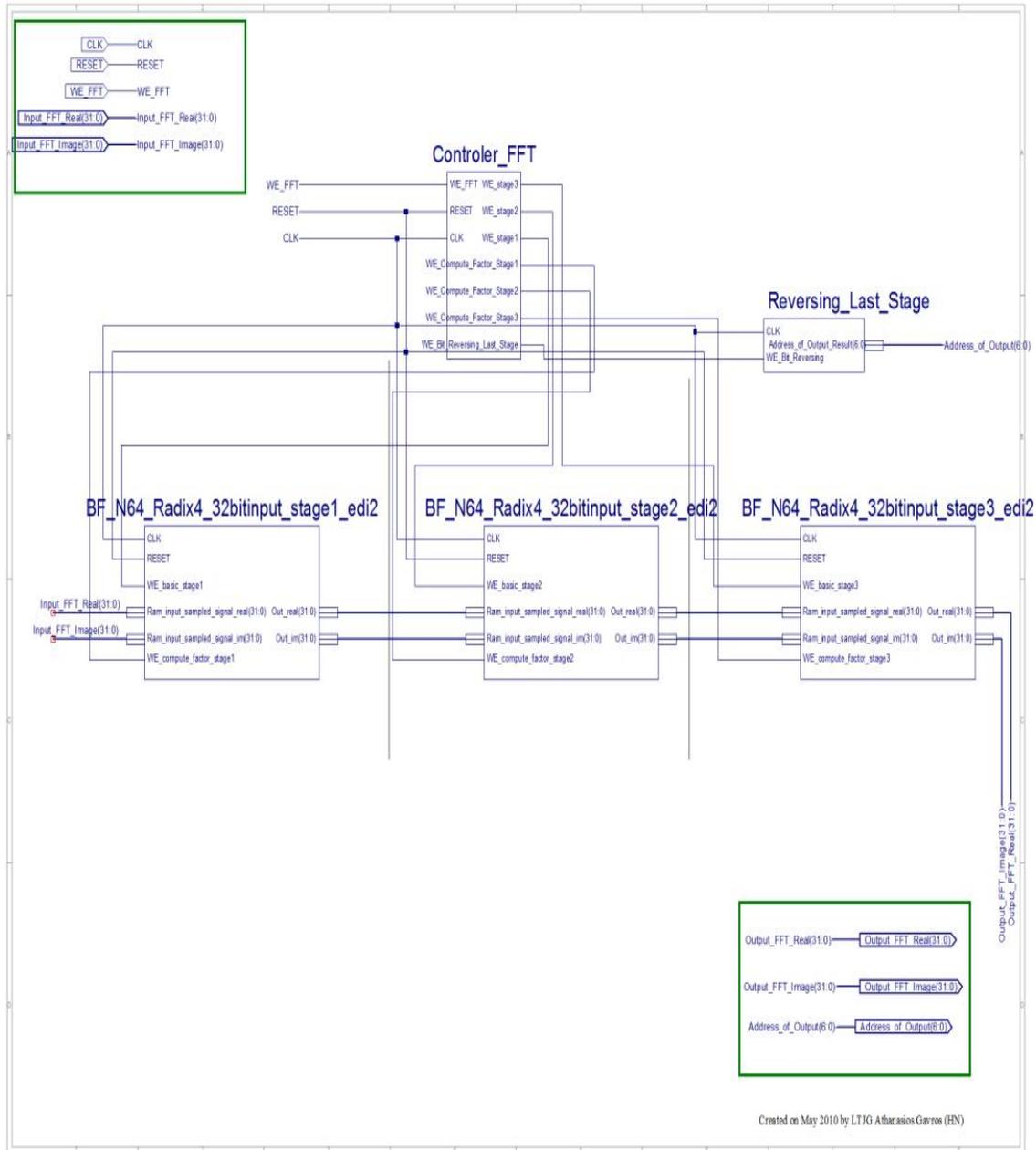


Figure 8. 64-Point Radix-4 in Shape DIF Signed Fixed Point 32Bit FFT—The Upper Level

1. Stage 1-2-3

All three stages are constructed identically, are pipelined, and their latency is 85 clock cycles per stage. These three stages are depicted in Figure 9. They include a stage controller, a RAM, a compute factor, a BF multiplier and a multiplexer module. The

stage controller generates addresses, the RAM receives those addresses and the input signals and sends the output signals to the compute factor. The compute factor computes the factors and sends them to the BF multiplier or to the multiplexer. The BF multiplier performs the necessary multiplications and sends the data to the multiplexer. The multiplexer chooses between incoming data from the compute factor or from the BF multiplier.

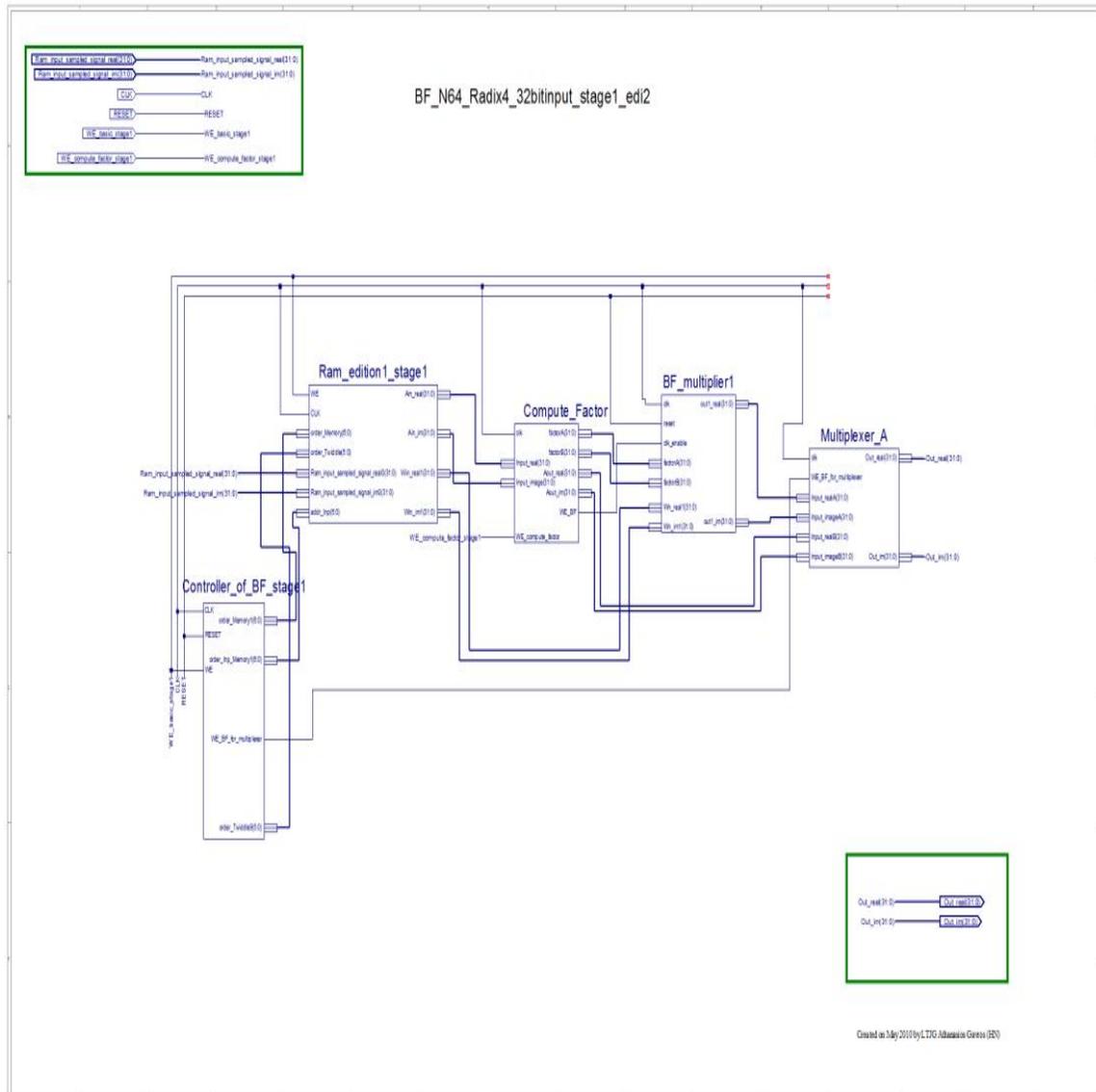


Figure 9. 64-Point Radix-4 in Place DIF Signed Fixed Point 32bit FFT—Stages 1&2&3

a. Stage Controller

The stage controller receives three and exports four signals. The stage controller of each stage waits for the write enable (WE) signal of the main controller. From this time on, it enters into a continuous process of creating input and output addresses for RAM, based on the algorithm depicted in Figure 7. It also generates the addresses for the ROM's twiddle factors and the switch command for the multiplexer.

b. RAM

The RAM module contains a ROM memory with the needed twiddle factors and two 128-point, 32-bit RAM from Xilinx CORE Generator tool. The RAM is the only part of the design that is made with the CORE Generator. The RAM receives 3 addresses, one for the incoming input signals, one for the required output signal and one for the required twiddle factors. Each of the real and imaginary input signals are stored in one half of the RAM and are exported only when the complete 64-point signals are collected. The RAM module exports four signals, the real and imaginary signal and the real and imaginary twiddle factor. Note that the ROM's stored twiddle factors are not rounded.

c. Compute Factor

The compute factor has four inputs and five outputs. It needs four pairs of signals in order to compute the proper factors and starts the procedure after commanded by the main controller. It computes three pairs of factors that are sent to the butterfly as demonstrated in Equation (III.2) and a pair of signals that are ready for the next stage as demonstrated in Equation (III.1). This pair of signals bypasses the BF multiplier and goes directly to the multiplexer.

d. BF Multiplier

The BF multiplier is the most complex module of the design. It receives 7 inputs, among them a pair of input factors from the compute factor and a pair of twiddle factors from RAM. It outputs the real and imaginary part of the computed signal.

The aim is to compute the parts of Equation III.3. This is demonstrated by the following computation:

$$\begin{aligned} B_r &= factor_1 * W_{N_r}^n + factor_4 * W_{N_{im}}^n \\ B_{im} &= factor_4 * W_{N_r}^n - factor_1 * W_{N_{im}}^n \end{aligned} \quad (IV.1)$$

Thus, four multiplies and two additions are required. For each multiply, the Virtex II embedded pipelined multiplier, MULT18x18S, is used. This choice was made to ensure maximum possible speed for our pipelined FFT. This was based on the fact that the embedded multipliers are probably faster than any behavioral multiplier that could be designed for this thesis work. The disadvantage of this choice is the limit bit number of each multiplier. The Virtex II embedded multiplier can handle 18-bit signed two's complement numbers and outputs a 36-bit result. However, the input signals in this design are 32-bit long. This requires the use of four embedded multipliers for each multiplication. To illustrate, computation of the first product, $factor_1 * W_{N_r}^n$ is shown in Figure 10:

32X32 PIPELINED MULTIPLIER

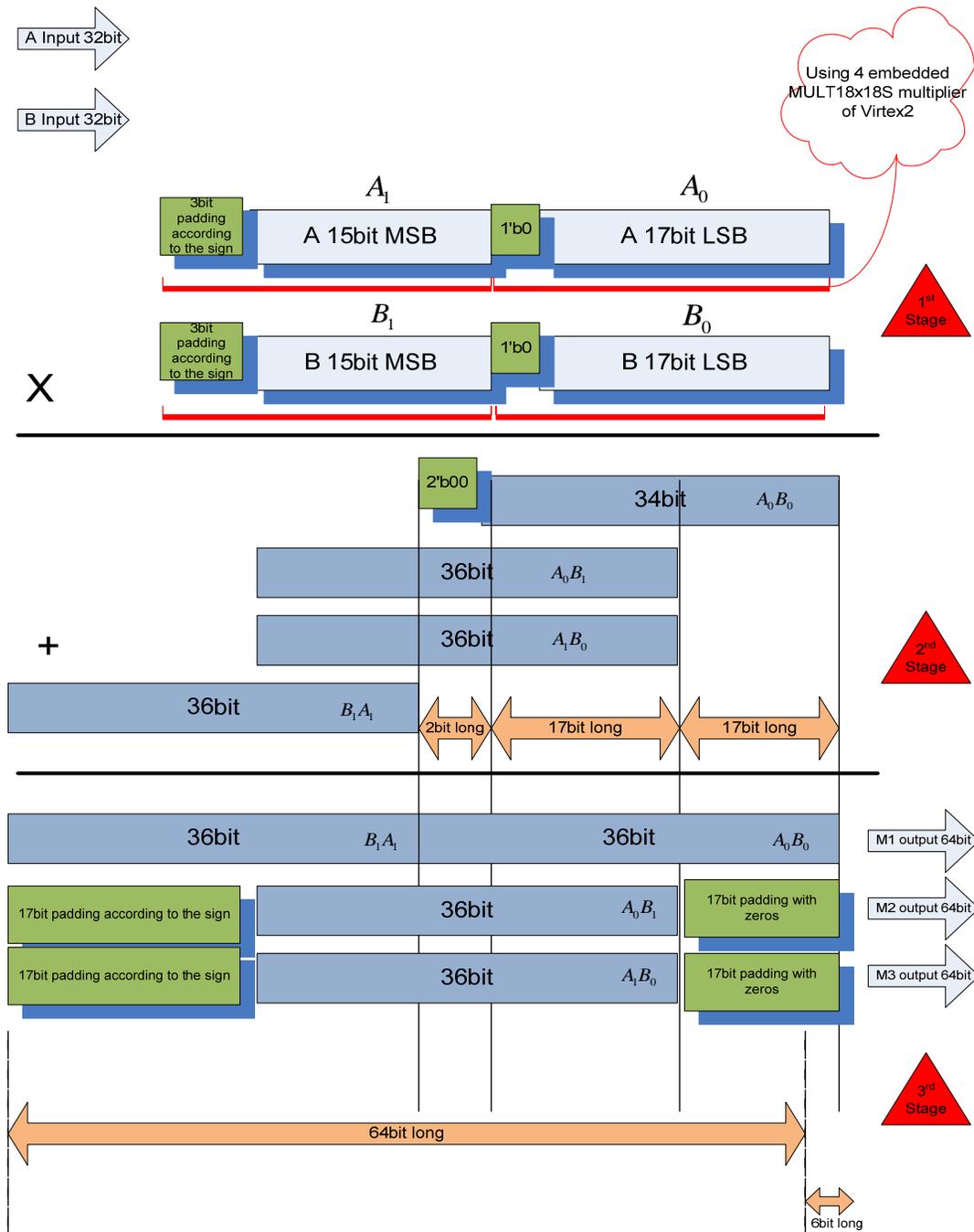


Figure 10. BF Multiplier—Use of Embedded Multipliers—Part1

Figure 10 clarifies the situation. It can be observed that after the import of the two 32-bit long signals, a factor and a twiddle factor, four different outputs are received from the embedded multipliers. Next, the signal is subject to the concatenation

or addition of bits in order to secure the correct multiplication of the signed fixed point numbers. Finally, in the third stage three 64-bit long signals are received.

At the fourth stage, those three signals are sent to the carry save adders module where the number of the desired signals is decreased from three to two. This is depicted in Figure 11. At the end of the fourth stage, two 64-bit long signals are available for import to the carry look ahead adder module. This module has a structure similar to that depicted in Figure 12. The carry look ahead adder adds the two 64-bit numbers and outputs the product that is being concatenated. The concatenation is based on the fact that although two 32-bit fixed point signals (with 30-bit fractional number) are imported, 36-bit fixed point numbers (with 30-bit fractional number) are being multiplied.

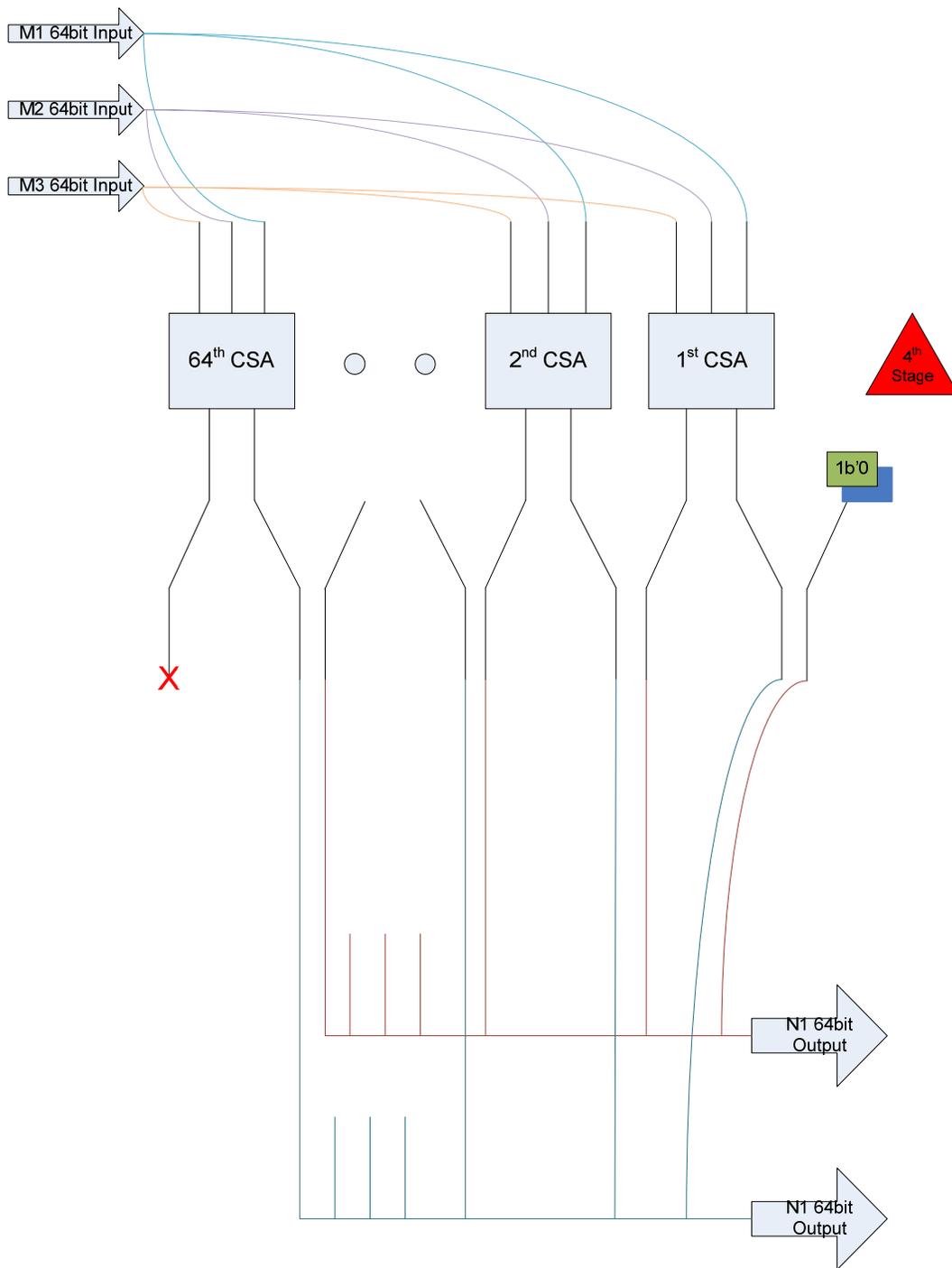


Figure 11. BF Multiplier—CSA module—Part2

64-Bit Carry Lookahead Adder

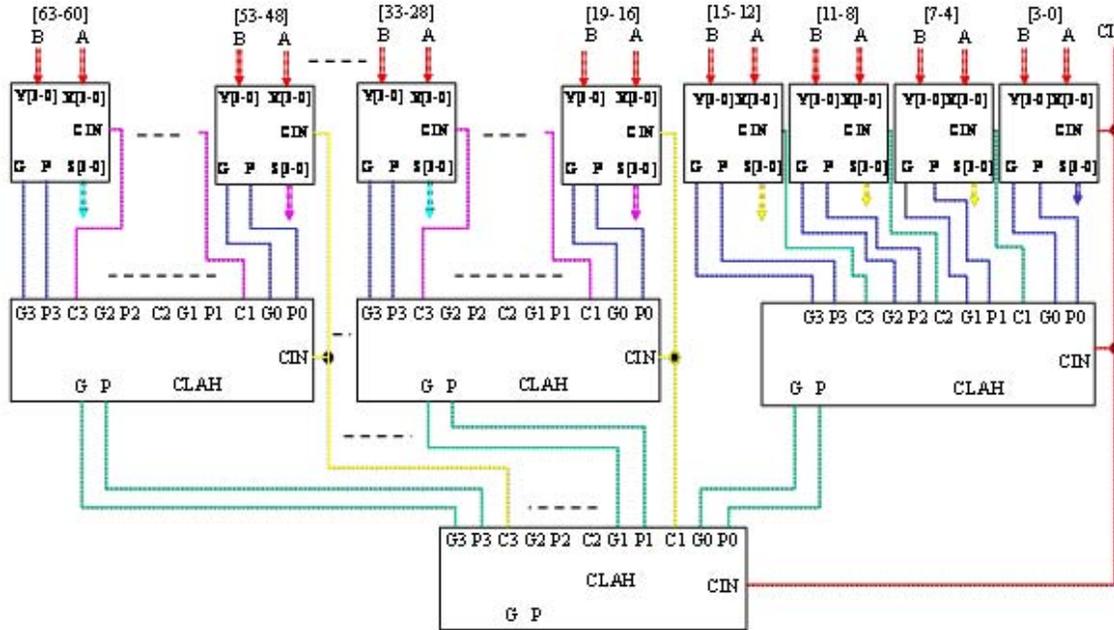


Figure 12. BF Multiplier—CLAH Module—Part3 (From [19])

So far, only one product of Equation IV.1 has been computed. However, four identical designs of the structure, which is shown in Figure 13, are used to compute the four products of Equation IV.1. Note that during the concatenation procedure, no rounding occurs.

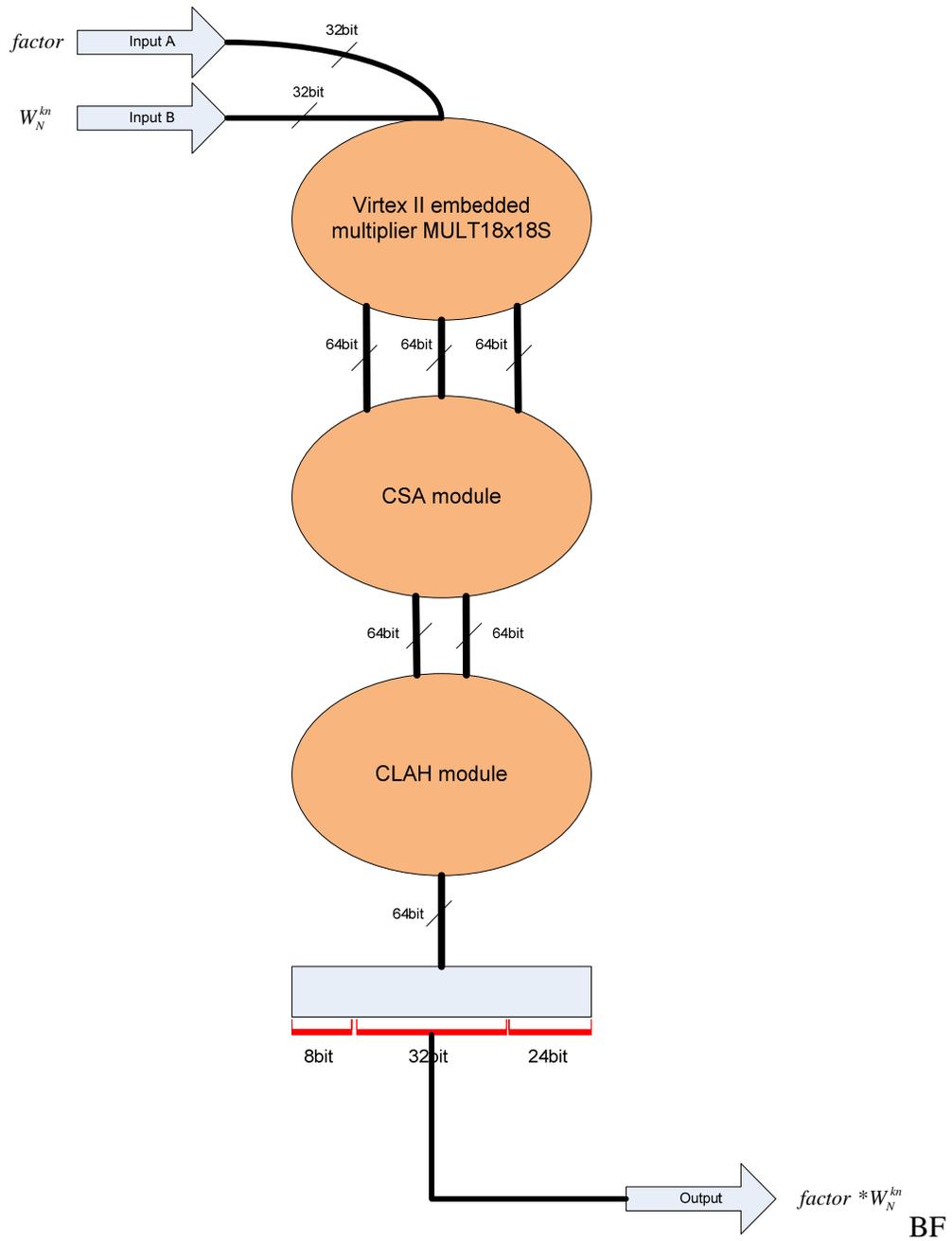


Figure 13. Multiplier—Overview—Part1 & Part2 & Part3

e. Multiplexer

The multiplexer receives a pair of input signals from the BF multiplier and a pair of signals directly from the compute factor module. The stage controller manipulates the multiplexer and determines which of the two pairs are going to be sent to the next stage.

2. Main Controller

The main controller is the module that manipulates the entire design by managing the controllers of each stage. It contains a reset command that restarts the entire system and two pairs of signals that awake on time the stage controllers and the compute factors of each stage.

3. Reversing Last Stage

The reversing last stage module generates the output address of the output signals. In order to avoid using another RAM for storing the signals in random order and outputting then in an arithmetic order, the preference is to export them in the in-place random order and indicate the correct order by using an address monitor.

B. IMPLEMENTATION EFFORTS AND RESULTS

In order to investigate the use of the RPR module it must first be compared to a TMR module that works under the same initial conditions. Therefore, a TMR 64-point Radix-4 in place DIF was designed and then implemented in the Virtex-2 XC2v6000 FPGA. Note that the voter in both cases (TMR and RPR) must be in the same position. In any other configuration a comparison would be ineffective.

1. Implementing a TMR

The design of a TMR is a simple process when compared to the design of a RPR. It is necessary to choose the frequency of the check points in the FFT and evaluate the values of the three identical modules through a small voter. Then the results are verified, with the expectation that at least two of the three values will be identical. In this manner, a FPGA is protected from unwanted space radiation.

As the frequency of the check points increases, the protection effectiveness increases. But, the increased use of voters raises the capacity demand of the design. A good compromise position for voters is to use one voter per FFT stage. The practical implication is that only three voters are required for a 64-point Radix-4 FFT.

At the conclusion of the implementation effort, a successful 64-point Radix-4 in-place DIF in a Virtex-2 XC2v6000 FPGA was designed and implemented. The entire design is included in Appendix B. The resources that were used at this time were much less than 12 percent of the slice resources of the FPGA, except from one, the embedded 18 x 18 pipelined multipliers. Exactly $3 \times 4 \times 4 = 48$ embedded multipliers were required from the total of 144 embedded multipliers available in the XC2v6000 FPGA. Based on calculation, if the goal is to create a TMR, three times the number of embedded multipliers is required in the primary design. Therefore, less than 36 percent of the FPGA's slices and $3 \times 48 = 144$ embedded multipliers should be required. Thus, the entire amount of available embedded multipliers was required for this design.

At this point all indications were positive and the design of the TMR was designed. After successfully synthesizing and simulating the design, an attempt was made to implement it in the FPGA. However, it was discovered that the FFT could not be implemented in the FPGA. The reason was based on the demand of the 6-block RAM of the module for six empty adjacent embedded multipliers. The embedded RAMs and multipliers were sharing the same routing resources in the Virtex II FPGA. Thus, the program required $144 + 6 = 150$ multipliers or 104 percent of the available resources and the implementation effort failed. Nevertheless, in order to compare the TMR and RPR source and power requirements, an implementation of both modules was attempted on the larger Virtex II XC2V8000 FPGA and is discussed in the next Chapter V.

2. Implementing a RPR—First Attempt—RPR Degree 8/32

Based on the previous concepts, a RPR module was created using, as its core, the primary design, with RPR voters embedded in the end of each stage. This design consisted of one precise module unit and two smaller average precision module units. The precise module unit was identical to the primary design module and handled 32-bit

real and imaginary inputs, while outputting 32-bit real and imaginary results. The average precise module units were distinct in the upper and lower bound units, in that each of them defined the upper and lower limits of the precise module.

a. RPR Upper and Lower Modules

The upper and lower average precision modules were almost identical. They truncated the 32-bit real and imaginary input into 8-bit numbers. In the upper module, the truncated number was increased by one bit on the least significant bit (LSB). The 8-bit number passed through the modified compute factor module and entered the modified BF multiplier. Inside the BF multiplier, due to the need for an 8 x 8 bit multiplier instead of the 32 x 32 bit precise module's multiplier, a single 18 x 18S embedded multiplier was used instead of four. There was no need for CSA or CLAH and the result of the multiplication occurred sooner than expected from the precise module unit. In order to keep the program synchronized, a delay was inserted after the 8 x 8 BF multiplier to equalize the latency, as depicted in Figure 14.

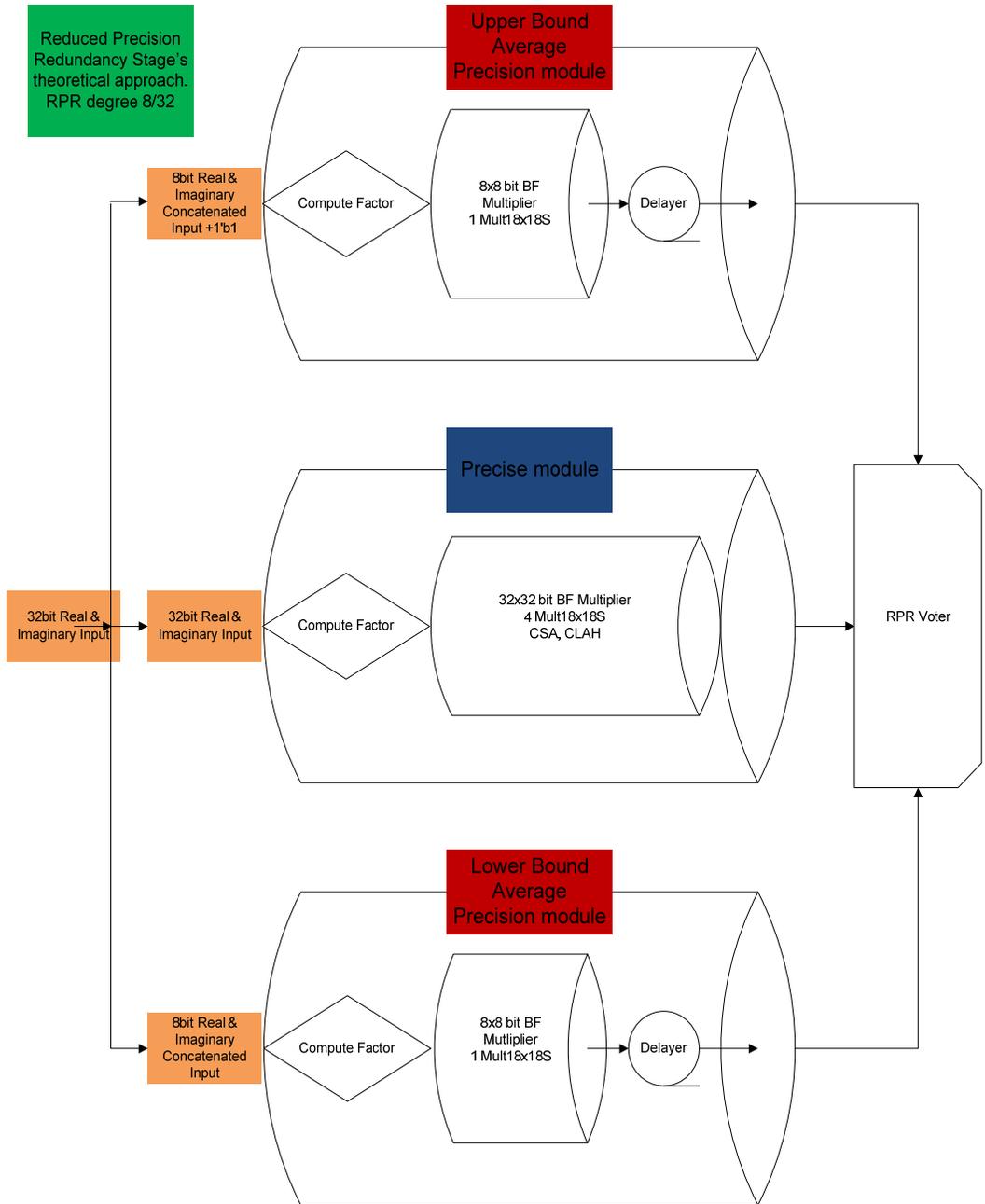


Figure 14. Reduced Precision Redundancy Stage Portrayal

b. Overflow Approach

In the primary design of the FFT, the overflow problem was recognized and the simplest solution was implemented. Therefore, the input was restricted to

normalized signals with values less than 2^{-9} . This secured the design from possible overflow events. In the first attempt at implementing RPR, a different approach was elected. Based on Cheng's work, "Autoscaling Radix-4 FFT for TMS320C6000" [20], a 3-bit shifting per stage approach was adopted in order to avoid overflow issues. The specific implementation was the adoption of 3-bit shifting in the input of each RAM's stage.

c. Ambiguity Phenomenon

In the design of the RPR modules, the upper bound is discriminated from the lower bound modules by adding one bit in the LSB of the truncated 8-bit number. This action directs us to the following equations:

$$\begin{aligned}
 A_{Lower} &\leq A_{Precise} < A_{Upper} \\
 B_{Lower} &\leq B_{Precise} < B_{Upper} \\
 C_{Lower} &\leq C_{Precise} < C_{Upper} \\
 D_{Lower} &\leq D_{Precise} < D_{Upper}
 \end{aligned}$$

Therefore, using the following equations:

$$\begin{aligned}
 factor_{Example} &= A + B - C - D \\
 Real &= factor_{Example1} * W_N^{real} - factor_{Example2} * W_N^{image} ,
 \end{aligned}$$

yields the following results for the upper, lower, and precise modules:

$$\begin{aligned}
 factor_{ExampleLower} &= A_{Lower} + B_{Lower} - C_{Lower} - D_{Lower} \\
 factor_{ExamplePrecise} &= A_{Precise} + B_{Precise} - C_{Precise} - D_{Precise} \\
 factor_{ExampleUpper} &= A_{Upper} + B_{Upper} - C_{Upper} - D_{Upper}
 \end{aligned}$$

But, these outcomes lead to the following ambiguity:

$$\begin{array}{ccc}
 & > & > \\
 factor_{ExampleLower} & ? & factor_{ExamplePrecise} & ? & factor_{ExampleUpper} \\
 & < & < & & <
 \end{array}$$

A solution to this problem is the creation of a unified upper and lower bound module where each equation loads upper or lower values depending on the sign.

$$\begin{aligned} factor_{Example_{Lower}} &= A_{Lower} + B_{Lower} - C_{Upper} - D_{Upper} \\ factor_{Example_{Precise}} &= A_{Precise} + B_{Precise} - C_{Precise} - D_{Precise} \\ factor_{Example_{Upper}} &= A_{Upper} + B_{Upper} - C_{Lower} - D_{Lower} \end{aligned}$$

And so:

$$factor_{Example_{Lower}} \leq factor_{Example_{Precise}} < factor_{Example_{Upper}}$$

d. RPR Survey Problems

Incorporating 3-bit shifting in each stage of the FFT introduces a total of 9-bit shifting at the final output. If 8-bit RPR shield (8/32 degree) is obtained, it is concluded that in a non-overflow case, that RPR is incapable of securing the precise module as illustrated in the following equations. Figure 15 makes clear the fact that in a non-overflow case, the RPR cannot sufficiently protect the third stage FFT.

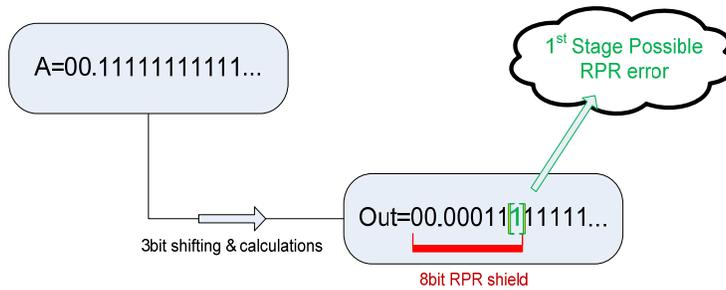
$$\begin{aligned} 3rd_Stage_Precise_Output(not_overflow) &< 2^{-9} \\ 3rd_Stage_RPR_Shield &\geq 2^{-6} \end{aligned}$$

Input < 1

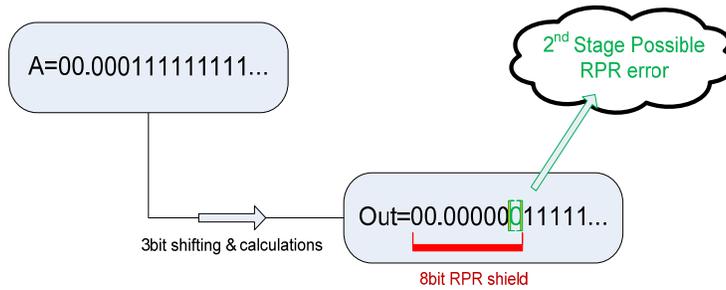
Let assume that $A = 00.11111111111111111111111111111111$ (near 0.99999)

Let assume that we are handling with a simplified case where $B=0, C=0, D=0, W_n_real=1, W_n_image=0$.

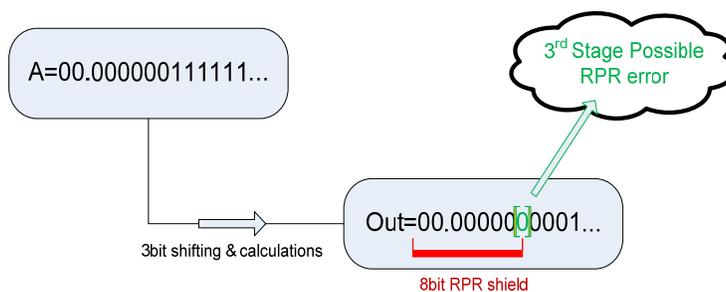
1 Stage



2 Stage



3 Stage



Final Output=00.00000000011111111111111111111111

Figure 15. RPR Shield and 3-Bit Shifting Obstacle

There are two possible solutions to this problem. Either a program must be created that recognizes and manipulates the possible overflow cases by shifting the input data by the appropriate amount of bits only when it is desired, or the degree of the RPR must be increased. In the first situation, the concept of an intelligent overflow manipulator is most desirable in order to preserve the 8/32 degree of the RPR. However, this manipulator will increase the complexity of the design due to the need for marking and shielding of the shifted bit amount throughout the entire FFT structure. In the second case, an increase of the RPR degree is not desirable, but seems rather attractive at this point because it minimizes alteration of the primary module.

3. Implementing a RPR—Second Attempt—RPR Degree 14/32

Taking into consideration the ambiguities and errors from the previous implementation, a RPR module was created, using the primary design as its core, with RPR voters embedded at the end of each stage. This design is included in Appendix C and consists of one precise module unit and two smaller low precision module units. The precise module unit is essentially identical to the previous design module (IV.B.2) and handles 32-bit real and imaginary inputs, while outputting 32-bit real and imaginary results. In this case, the only difference is that the bound module units are identical and not distinct into upper and lower bound units.

a. RPR Bound Modules

The RPR bound module truncates the 32-bit real and imaginary input into a 14-bit number. The 14-bit number passes through the modified compute factor module and enters the alternated BF multiplier. Inside the BF multiplier, due to the need for a 14 x 14 bit multiplier instead of the 32 x 32 bit precise module's multiplier, only one 18 x 18S embedded multiplier was used instead of four. There was no need for CSA or CLAH and the result of the multiplication occurred sooner than expected from the precise module unit. In order to keep the program synchronized, a delayer was used after the 14 x 14 BF multiplier. The major difference between the first and the second RPR implementation was the altered approach of the RPR function. Instead of using an upper and lower bound, two identical bounds were used that handle the truncated 14-bit value

of the 32-bit input. At the end of each stage, the voter inspects the duplicate truncated values and proceeds to a simple bit by bit comparison. If the values are identical, then the voter computes the expected upper and lower limits based on the truncated 14-bit output and on the theoretical expected errors as shown in Equation IV.3.

$$factor_{Precise} = A_{Precise} + B_{Precise} - C_{Precise} - D_{Precise}$$

$$factor_{Trunc} = A_{Trunc} + B_{Trunc} - C_{Trunc} - D_{Trunc} = \Psi$$

$$factor_{Trunc_Up} = (A_{Trunc} + 1'b1) + (B_{Trunc} + 1'b1) - C_{Trunc} - D_{Trunc} = \Psi + 2'b10$$

$$factor_{Trunc_Lo} = A_{Trunc} + B_{Trunc} - (C_{Trunc} + 1'b1) - (D_{Trunc} + 1'b1) = \Psi - 2'b10$$

In the worst possible case,

$$\Psi - 2b'10 \leq factor_{Precise} \leq \Psi + 2b'10 \quad (IV.2)$$

Using the same method,

$$Real_{Precise} = factor_{Precise} * W_{N_Precise} \overset{+}{real} \underset{-}{factor_{Precise} * W_{N_Precise} image}$$

$$Real_{Trunc} = factor_{Trunc} * W_{N_Trunc} \overset{+}{real} \underset{-}{factor_{Trunc} * W_{N_Trunc} image} = \Phi$$

Trying to compute the worst possible case, (IV.2)*1 + (IV.2)*1:

$$\Phi - 3'b100 \leq Real_{Precise} \leq \Phi + 3'b100 \quad (IV.3)$$

The voter compares the precise value to the expected upper and lower bounds and decides to choose either the precise value or the average (truncated) value.

b. Verifying Results

In order to verify the truth of the outputs of the implemented FFT structure, a MATLAB FFT simulation file was created. This file was used to compare the output forms of the three different sub-programs. The first subprogram was actually the built-in MATLAB FFT, the second was a clone of the Verilog design in the MATLAB environment, and the third was a MATLAB translator for the implemented

FFT results. Since MATLAB usually handles 64-bit numbers and due to the decision for 3-bit shifting of the input in every stage of the FFT, a possible error was identified in the precise calculation of the FFT between MATLAB's calculated outputs and Verilog's translated outputs. These expected errors are tabulated in Table 2.

Table 2. Expected Errors of Precision Calculation

FFTs Outputs	Expected Error
Output of the 1 st Stage	$\leq 2^{-25}$
Output of the 2 nd Stage	$\leq 2^{-22}$
Output of the 3 rd Stage	$\leq 2^{-19}$

The purpose of the entire design was the testing of the implemented FFT in a radiation environment. Prior to this, testing was conducted via simulation of possible radiation effects. For this reason, a radiation module (Figure 16) was imported into the FFT design that was capable of introducing error bits at the beginning of each stage, prior to the butterfly calculations.

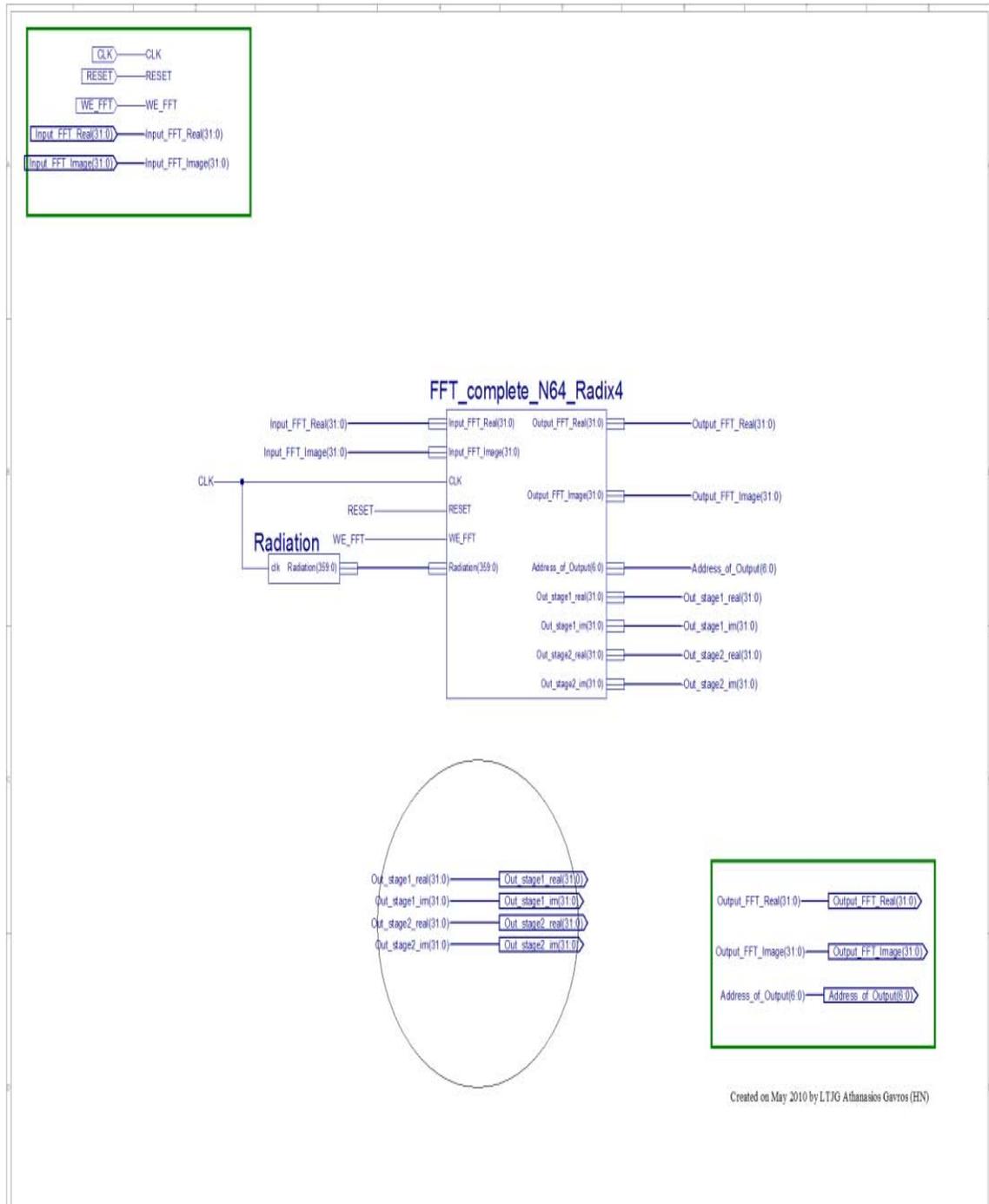


Figure 16. FFT and Radiation Module

The radiation module allows great flexibility in the introduction of error bits, giving the ability to decide in which stage, in which module (one of the truncated RPR or the precise value) and for how long this “form” of radiation lasts. Errors

introduced in the first stage have less impact on the precision of the average RPR in contrast to errors introduced in the last stage as shown in Table 3, again due to the three-bit shifting decision.

Table 3. Errors Expected Due to the Import of Radiation in the Precise Module

Radiation introduced in the precise module	Expected Error
Radiation introduced in the 1 st stage	$\leq 2^{-7}$
Radiation introduced in the 2 nd stage	$\leq 2^{-4}$
Radiation introduced in the 3 rd stage	$\leq 2^{-1}$

In this section, the implementation efforts of three different modules, TMR, RPR with 8/32 degree, and RPR with 14/32 degree were discussed. In the next chapter, the results analysis will be covered based on these designs.

V. RESULTS

The efforts for implementing a TMR version of the FFT in the Virtex II XC2V6000 were fruitless, forcing a revision to the plan. In order to compare TMR and RPR modules, the same FPGA chip had to be used, so it was essential to use a larger FPGA of the same family. The next larger, available Virtex II FPGA, the XC2V8000, was used.

First, the synthesis reports for the RPR and TMR modules were compared, as indicated in Table 4, where a significant difference in occupied resources is noted. RPR needs 74 percent of the slices that TMR uses, 76 percent of the slice flip flops, 68 percent of the four-input LUTs and 50 percent of embedded multipliers that the TMR uses. In Figure 17, the differences identified in the synthesis reports are presented.

Table 4. Synthesis Results—Comparison between RPR and TMR in a Virtex II XC2V8000-5FF1152 FPGA

Virtex II XC2V8000	TMR module	TMR's occupancy %	RPR module	RPR's occupancy %
# Slices	11028	23%	8236	17%
# Slice Flip Flops	18525	19%	14067	15%
# 4 input LUTs	17985	19%	12255	13%
# BRAMs	6	3%	6	3%
# MULT18x18S	144	85%	72	42%

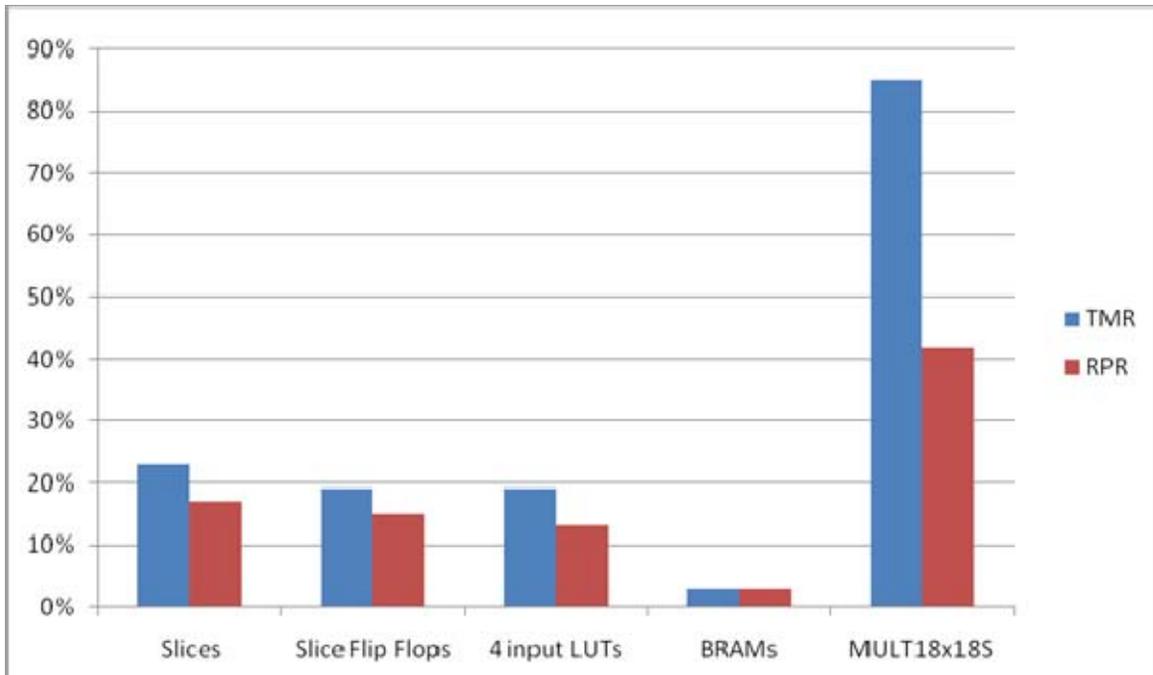


Figure 17. TMR Vs RPR Synthesis Results

Secondly, the Xilinx’s XPower Analyzer was used to investigate the power that each of the modules required for operation. When the outcomes from both cases were compared, an interesting conclusion resulted. Although both systems needed the same amount of quiescent power, RPR required 19 percent less dynamic power than TMR. These results are tabulated in Table 5 and depicted in Figure 18.

Table 5. TMR Versus RPR—XPower Analyzer Results

Virtex II XC2V8000	TMR module	RPR module
Total Quiescent Power	0.13785 W	0.13785 W
Total Dynamic Power	0.17408 W	0.14079 W
Total Power	0.31193 W	0.27864 W
Junction Temperature	28.3 degrees C	27.9 degrees C

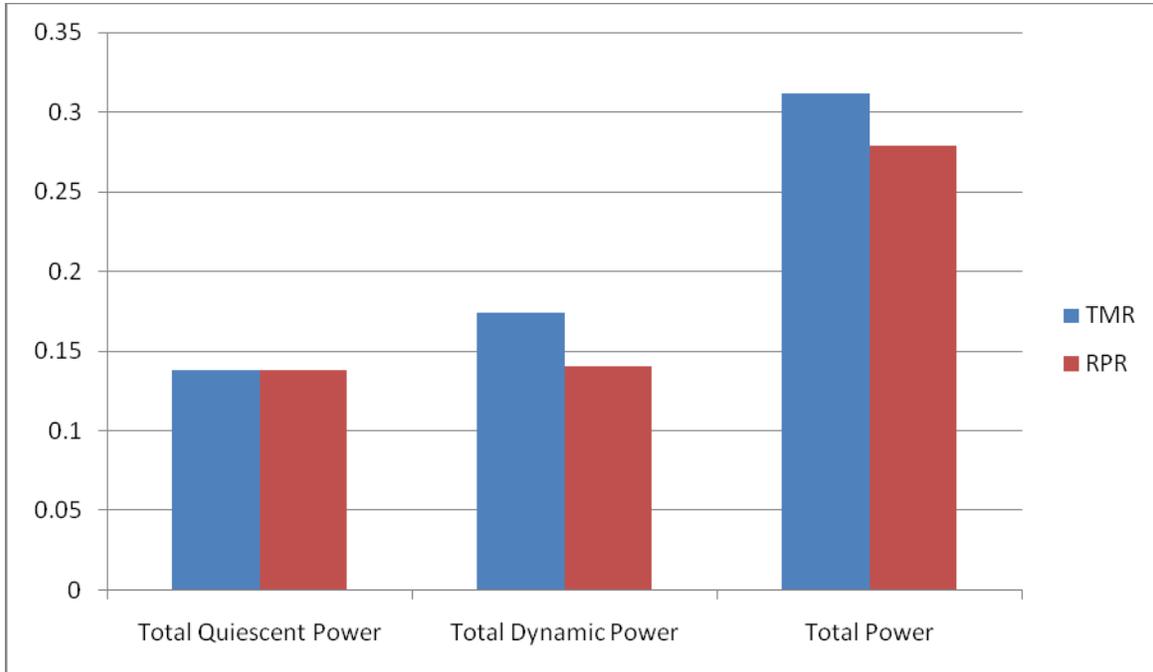


Figure 18. TMR Vs RPR—XPower Analyzer Comparison

After inspecting the synthesis and power reports from Xilinx for the two implemented modules, RPR with a degree of 14/32 and TMR, a more detailed conclusion can be made concerning the RPR method and is discussed in greater detail in Chapter VI.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY

The objective of this thesis was the creation of a FFT structure that would be implemented in a FPGA, adopting two different methods of redundancy, TMR and RPR. The purpose was to investigate the capabilities of RPR. First a simple 64-point Radix-4 in-place FFT was implemented that could handle fixed point, 2's complement numbers and was tested with accurate results. Next, a TMR structure was designed by replicating three identical FFT structures and by importing a voter at the end of each stage. The next stage was the design of a RPR structure with a degree of 8/32. The design was successfully implemented, but it failed to protect the system against radiation. Taking into consideration the problems from the previous unsuccessful design, the construction of a new RPR structure with a degree of 14/32 was conducted and resulted in a design that worked correctly and managed to protect the FFT structure efficiently.

One of the major concerns about the RPR method is the size of the voter. This thesis suggests a simple alteration from the method suggested from Snodgrass [1], where there is no need for generating upper and lower bounds. Instead, the truncated value of the precise number is formed and duplicated. This alteration helps simplifying things and decreases the size of the voter, since now the voter can easily, with a bit-to-bit comparison, verify the correctness of the truncated values and with a simple addition or subtraction can output the predicted theoretical boundaries of the precise value.

Although several interesting results were obtained, during this research, relative to the specific structure chosen (the 64-point Radix-4 in-place FFT), some conclusions with broad theoretical impact should also be mentioned. These findings can be summarized by the following statements: the RPR method is sufficient, it requires fewer resources, and is more power efficient than TMR when considering arithmetic operations. Additionally, with RPR, there are reduced resource requirements and power consumption over TMR, but there is a sacrifice in precision.

B. RECOMMENDATIONS FOR FUTURE STUDY

1. Overflow Manipulator

The current structure of this design did not permit a further decrease in RPR degree, but the addition of an intelligent overflow manipulator in each stage would allow abandonment of the high RPR degree and permit protection of each stage with the same small amount of bits that are desired. This would enable further investigation of the impact of RPR degree, and assist in understanding the trade-off of RPR degree, on precision, capacity size, and power consumption.

2. Implementing a 4 Recursive Butterfly FFT Instead of the 12 Butterfly FFT

The basic FFT module used in this thesis was required to handle data in real time. That forced use of a four-BF structure per stage for the Radix-4 FFT, creating a rather large FFT processor, requiring a significant number of embedded multipliers (33 percent) and a trivial amount of remaining resources (less than 13 percent in all cases) for a Virtex II XC2V6000 FPGA implementation. This revealed that this design had a significant weakness, the embedded multipliers. It is interesting to note that in considering the FFT design, a recursive four-BF multiple-stage expansion effort was evaluated. This would have enabled the ability to keep the majority of the FFT structure intact, altering only the controller. Of course, a replacement of the embedded multipliers would permit a deviation from the Virtex II family, but even so, the decrease in the number of required BFs, in addition to, the decrease in the demanded RAM resources would permit the development of a less resource demanding FFT, with RPR protection, at the price of sacrificing the operational sample rate.

APPENDIX A. FFT IMPLEMENTATION

This Appendix includes the 64-point, in-place, Radix-4, DIF FFT design as it is described in Chapter IV.A. The module receives two fixed-point 32-bit inputs (real and imaginary) and outputs two fixed-point 32-bit results. The design was implemented in a Virtex II XC2V6000 BF957 using Xilinx tools. The synthesis report confirmed that the design worked at a clock speed of 225MHz.

A1. FFT MODULE—XILINX SCHEMATIC DESIGN

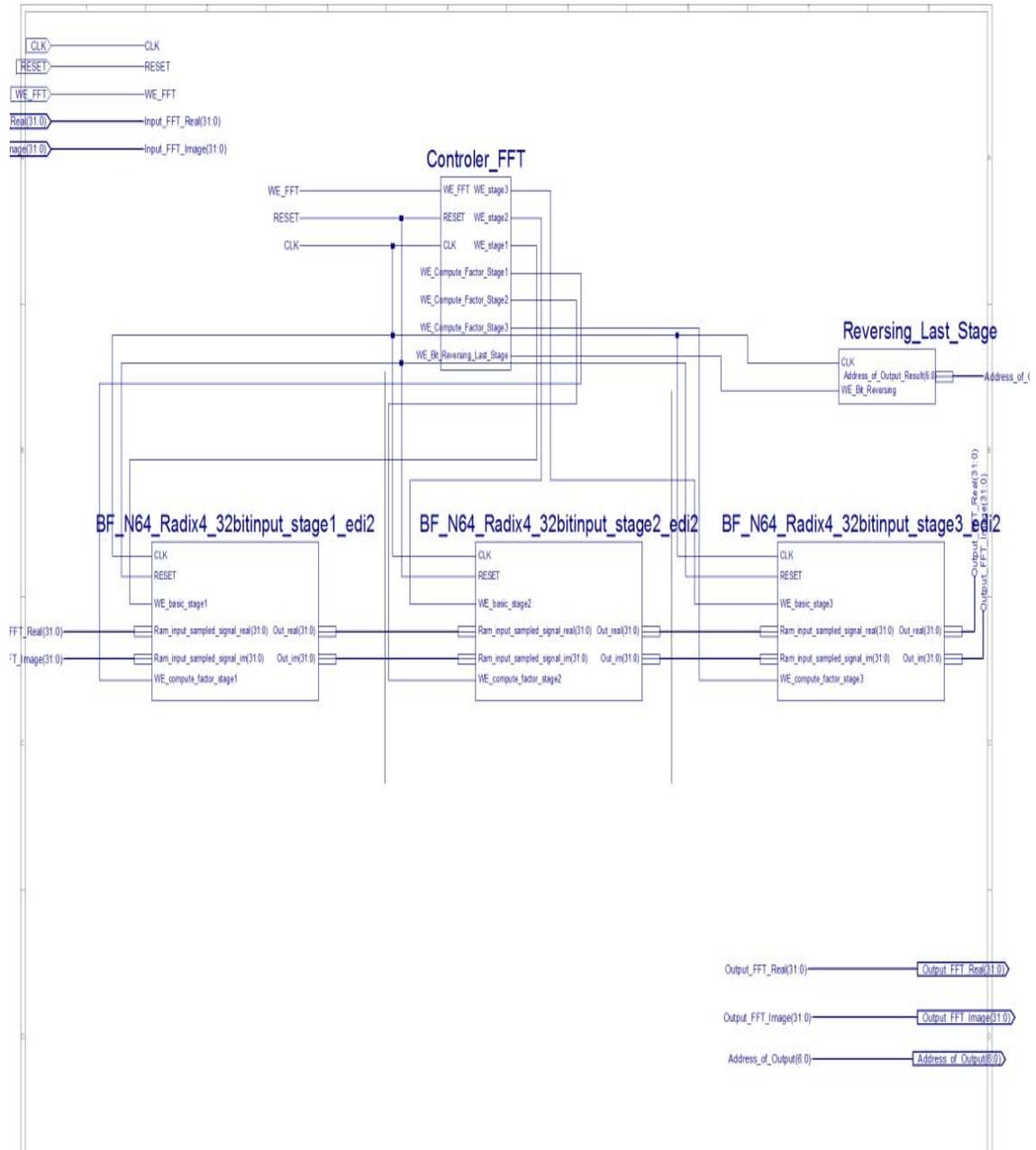
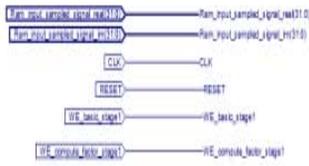


Figure 19. FFT Design—Entirely Layout



BF_N64_Radix4_32bitinput_stage1_ed1

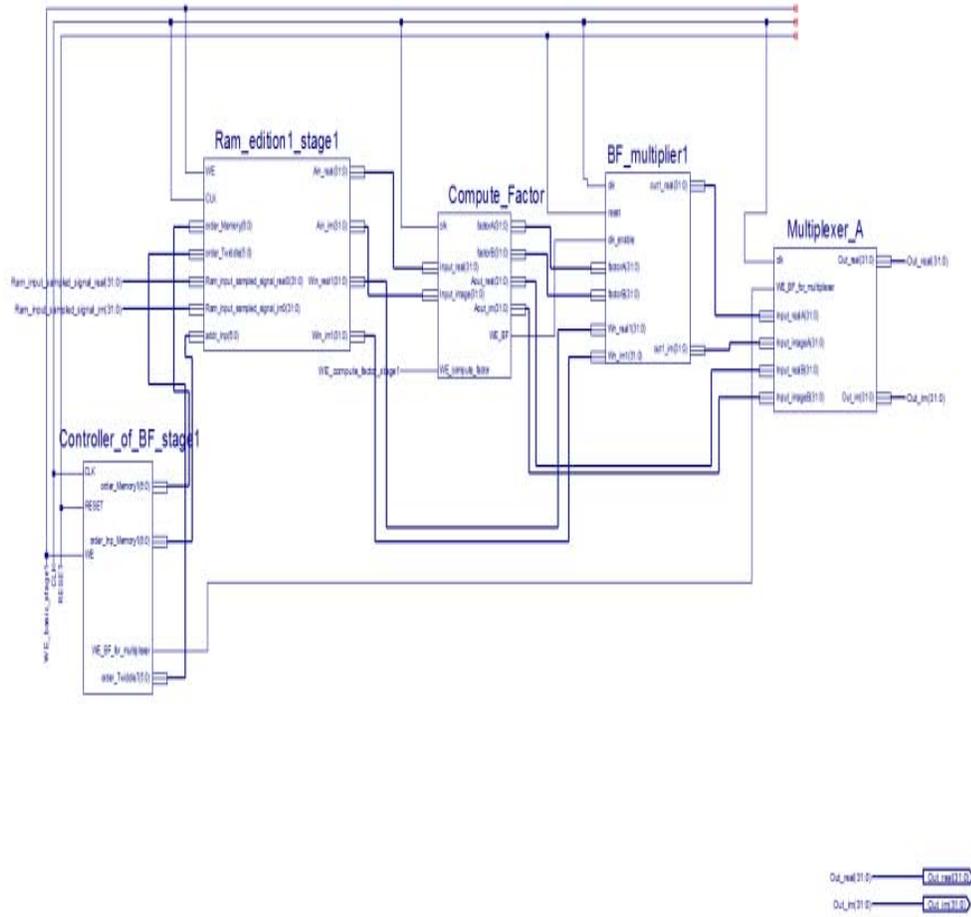


Figure 20. FFT's First Stage Layout

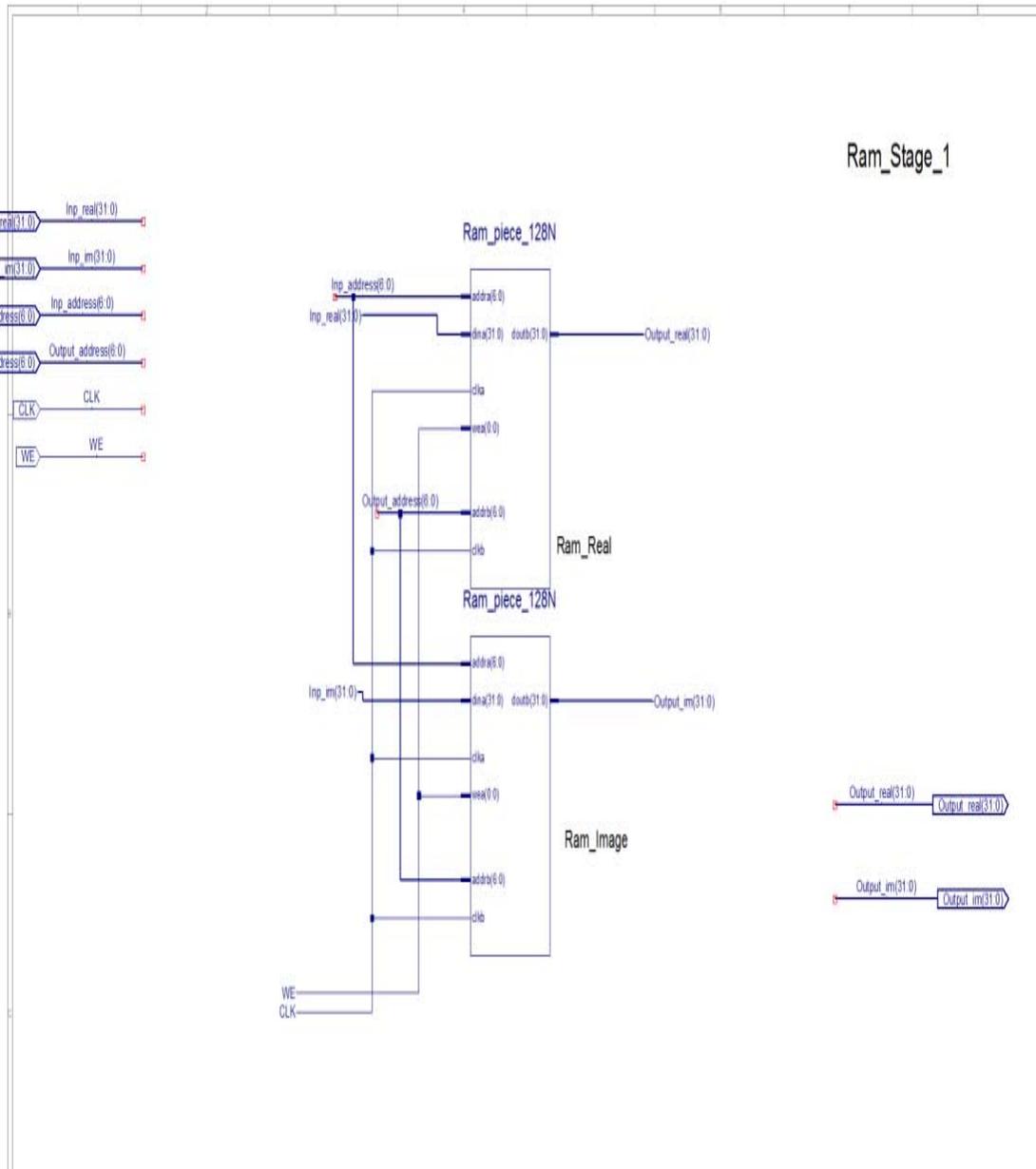


Figure 21. RAM Structure

A2. FFT MODULE—XILINX BEHAVIORAL AND STRUCTURAL DESIGN

A2.1. Main Controller of TMR

```
`timescale 1ns / 1ps
```

```
module
```

```
Controler_FFT(CLK,RESET,WE_FFT,WE_stage1,WE_stage2,WE_stage3,WE_Bit_Rev  
ersing_Last_Stage,WE_Compute_Factor_Stage1,WE_Compute_Factor_Stage2,WE_Co  
mpute_Factor_Stage3);
```

```

input CLK,RESET,WE_FFT;
output reg WE_stage1,WE_stage2,WE_stage3;
output reg WE_Compute_Factor_Stage1=0;
output reg WE_Compute_Factor_Stage2=0;
output reg WE_Compute_Factor_Stage3=0;
output reg WE_Bit_Reversing_Last_Stage=0;
reg [8:0] counter_FFT=0;

always @(posedge CLK)
begin

WE_stage1=WE_FFT;
if (RESET)
    begin
        counter_FFT<=0;
        WE_stage1=0;
        WE_stage2=0;
        WE_stage3=0;
    end
if (WE_stage1==1)
    begin
        if (counter_FFT==85) WE_stage2=1;
        if (counter_FFT==170) WE_stage3=1;
        if (counter_FFT==51) WE_Compute_Factor_Stage1=1;
        if (counter_FFT==112) WE_Compute_Factor_Stage2=1;
        if (counter_FFT==117) WE_Compute_Factor_Stage3=1;
        if (counter_FFT==256) WE_Bit_Reversing_Last_Stage=1;
        counter_FFT<=counter_FFT+1; //< !!!!
    end
//keep in mind that the controller of each stage needs 2 clocks from the time it is
triggered
// to the time that can handle the inputs.
end
endmodule

```

A2.II. Reversing Bit of Last Stage Module

```

`timescale 1ns / 1ps
module Reversing_Last_Stage(CLK,WE_Bit_Reversing,Address_of_Output_Result);
input CLK;
input WE_Bit_Reversing;
reg [7:0] order_Inp_Memory=0;
output reg [6:0] Address_of_Output_Result=0;
always @(posedge CLK)

```

```

begin
//I have to proceed in bit reversing in every stage. It is convenient to change the Input
//Address(reversing the bit) before the data are transferred into RAM of the next stage.
// I have to manipulate the 6bit . I have to make a bit
// reversal based on Radix4(four decimal system) => for example :300 -> 003 ,
// 012->210 etc.
if (WE_Bit_Reversing)
begin
    if (order_Inp_Memory==64)
    begin
        order_Inp_Memory=0;
    end
    Address_of_Output_Result
    ={order_Inp_Memory[6],order_Inp_Memory[1],order_Inp_Memory[0],order_Inp_Mem
    ory[3],order_Inp_Memory[2],order_Inp_Memory[5],order_Inp_Memory[4]};
    order_Inp_Memory=order_Inp_Memory+1'b1;
end
end
endmodule

```

A2.III. Controller of First Stage Timescale 1ns / 1ps

```

module Controller_of_BF_stage1(CLK,RESET,WE,WE_BF_for_multiplexer,
order_Memory1, order_Twiddle9, order_Inp_Memory1);

//based on - Fast Fourier Transform [64FFT] Univ. of the Ryukyus, Okinawa, Japan
//from the moment the WE_basic_stage1 is on (1), the Sampled_Input_signal must be
//imported on the second following CLK, meaning:
// WE_basic_stage1 is on at CLK0
// Ram_input_sampled_signal will be active at CLK2
input CLK,RESET,WE;
output reg [6:0] order_Memory1;//7bit each piece(due to 64 present)
output reg [5:0] order_Twiddle9;//6bit each piece(due to 64 present)
output reg [6:0] order_Inp_Memory1;//7bit each piece
output reg WE_BF_for_multiplexer;
reg [6:0] order_Memory;
reg [5:0]
order_Twiddle,order_Twiddle1,order_Twiddle2,order_Twiddle3,order_Twiddle4,order_
Twiddle5,order_Twiddle6,order_Twiddle7,order_Twiddle8;
reg [6:0] order_Inp_Memory;
parameter stage=1;
parameter N=64;
reg [5:0] counter=5'b00000;
reg [7:0] count_for_each_BF=8'b0;
reg [6:0] inpA=7'b0;

```

```

reg [5:0] counter2=6'b010010;//if I need 18 clks delay ???//mallon den 8a xreastw to
order_stage 2 Memory
reg [7:0] changer1=64
//in stage 1, Controller needs 1 clock in order to give the right output
always @(posedge CLK)
begin
    case (stage)
    1:
    begin
//order_Memory:generates the output address for stage1 Memory: 0,16,32,48 - 1,17,33,49
//keep in mind that order_Memory has to take into consideration, in which place
//is your data in RAM. First 64 slots or next 64 slots ((changer1: task)
//order_Twiddle: generates the output address for stage1 Twiddle: 0,0,0 - 0,1,2 - 0,2,4 - ...
//order_Inp_Memory:generates the input address for stage1 Memory: 0,1,2,3,4,...63 -
64,65..127
        if (WE)
        begin

                if (inpA==127) inpA<=0;
                inpA<=inpA+1;
                order_Memory<=count_for_each_BF+counter+changer1;
                order_Inp_Memory<=inpA;
                if (RESET)
                begin
                        counter<=5'b000000;
                        count_for_each_BF<=0;
                        counter2<=13;
                        inpA<=0;
                        end

                if (((inpA)>=63)&(inpA<127)) changer1<=0; // changer1 is responsible
for be the pointer in RAM for the two
                else changer1<=64;
                if (count_for_each_BF==0)
                begin
                        order_Twiddle<=32'b01000000000000000000000000000000;
                        WE_BF_for_multiplexer    <=1;
                        end
                if (count_for_each_BF==16)
                begin
                        order_Twiddle<=counter;
                        WE_BF_for_multiplexer    <=0;
                        end
                if (count_for_each_BF==32)
                begin

```

```

order_Twiddle<=counter<<1;//b2<=counter*2;
WE_BF_for_multiplexer    <=1;
end
if (count_for_each_BF==48)
begin
order_Twiddle<=(counter<<1)+(counter);//b3<=counter*3;
WE_BF_for_multiplexer    <=1;
end
count_for_each_BF<=count_for_each_BF+16;
if (count_for_each_BF==48)
begin
    count_for_each_BF<=0;
    counter<=counter+1;
    counter2<=counter2+1;
    if (counter==15)
        begin
            counter<=0;
        end
    // the following 2 if statements are for the order_Stage2_Memory
    if (counter2==15)
        begin
            counter2<=0;
        end
    end
end
end
endcase
end

//One stage registers for the commands
always @(posedge CLK)
begin
order_Inp_Memory1=order_Inp_Memory;
order_Memory1=order_Memory;
order_Twiddle1=order_Twiddle;
end
// The Wn_real and Wn_im are by-passing the "Compute Factor". So I have to "delay"
the Wn the same
// amount of clocks as if they were imported to "Compute Factor". In order to avoid
confusing
// parameter in Controler and in order to avoid using pipelined registers for the Wn in
Ram,

//Register for Order_Twiddle
always @(posedge CLK)

```

```

begin
    order_Twiddle2<=order_Twiddle1;
end
always @(posedge CLK)
begin
    order_Twiddle3<=order_Twiddle2;
end
    always @(posedge CLK)
begin
    order_Twiddle4<=order_Twiddle3;
end
    always @(posedge CLK)
begin
    order_Twiddle5<=order_Twiddle4;
end
    always @(posedge CLK)
begin
    order_Twiddle6<=order_Twiddle5;
end
    always @(posedge CLK)
begin
    order_Twiddle7<=order_Twiddle6;
end
    always @(posedge CLK)
begin
    order_Twiddle8<=order_Twiddle7;
end
    always @(posedge CLK)
begin
    order_Twiddle9<=order_Twiddle8;
end
endmodule

```

A2.IV. RAM

```

`timescale 1ns / 1ps
module Ram_edition1_stage1(
    order_Memory,
    order_Twiddle,
    Ain_real,Ain_im,
    Win_real1,Win_im1,
    Ram_input_sampled_signal_real0,
    Ram_input_sampled_signal_im0,
    addr_Inp,
    WE,
    CLK

```



```

        for (e=1;e<65;e=e+1)
        begin : L1
            assign twiddle_real[64-e][31:0]=twiddle_real_A[(e*32-1):((e-
1)*32)];
            assign twiddle_im[64-e][31:0]=twiddle_im_A[(e*32-1):(e-1)*32];
        end
    endgenerate

    assign Win_real1=twiddle_real[{1'b0,order_Twiddle[5:0]}];//here
you have to put the twiddle data
    assign Win_im1= twiddle_im[{1'b0,order_Twiddle[5:0]}];
endmodule

```

A2.V. Multiplexer and Voter of Each Stage

```

`timescale 1ns / 1ps
module Multiplexer_A(clk,Input_realA,Input_imageA,Input_realB,Input_imageB,
WE_BF_for_multiplexer,Out_real,Out_im);
input [31:0] Input_realA,Input_imageA,Input_realB,Input_imageB;
input clk,WE_BF_for_multiplexer;
output reg [31:0] Out_real;
output reg [31:0] Out_im;
always @(posedge clk)
begin
    if (WE_BF_for_multiplexer)
        begin
            Out_real    <=Input_realA;
            Out_im      <=Input_imageA;
        end
    else
        begin
            Out_real    <=Input_realB;
            Out_im      <=Input_imageB;
        end
end
endmodule

```

A2.VI. Compute Factor

```

`timescale 1ns/1ps

//11-12-09 32bit input, N=64, Radix4
module Compute_Factor(clk,WE_BF,Input_real,Input_image,factorA,factorB,Aout_real,
Aout_im,WE_compute_factor);
input clk,WE_compute_factor;
input [31:0] Input_real,Input_image;

```

```

output reg [31:0] Aout_real,Aout_im ;
output reg [31:0] factorA,factorB;
output reg WE_BF;//gives the order to BF_multiplier to start the procedure.
reg [31:0] factor1_a,factor2_a,factor3_a,factor4_a,factor5_a,factor6_a ;
reg [31:0] factor1_b,factor2_b,factor3_b,factor4_b,factor5_b,factor6_b ;

reg [1:0] counter=0;
reg [1:0] uncounter=0;
reg [31:0] Ain_real,Ain_im,Bin_real,Bin_im,Cin_real,Cin_im,Din_real,Din_im ;
reg stage1=0;
reg stage2=0;
reg [31:0] Areal_a,Aim_a;
reg [31:0] Areal_b,Aim_b;
reg [31:0] t1,t2,t3,t4,t5,t6,t7,t8;
reg [31:0]Aoutreal_reg_0,Aoutim_reg_0;
reg [31:0]Aoutreal_reg_1,Aoutim_reg_1;
reg [31:0]Aoutreal_reg_2,Aoutim_reg_2;
reg [31:0]Aoutreal_reg_3,Aoutim_reg_3;
reg [31:0]Aoutreal_reg_4,Aoutim_reg_4;
reg [31:0]Aoutreal_reg_5,Aoutim_reg_5;
reg [31:0]Aoutreal_reg_6,Aoutim_reg_6;
reg [31:0]Aoutreal_reg_7,Aoutim_reg_7;
reg [31:0]Aoutreal_reg_8,Aoutim_reg_8;
reg [31:0]Aoutreal_reg_9,Aoutim_reg_9;
reg [31:0]Aoutreal_reg_10,Aoutim_reg_10;
reg [31:0]Aoutreal_reg_11,Aoutim_reg_11;
reg [31:0]Aoutreal_reg_12,Aoutim_reg_12;
reg [31:0] Real [3:0];
reg [31:0] Im [3:0];
always @(posedge clk)
begin
//The 'counter' pairs the 4 input and gives the order to the 'stage1' to start
//the calculation of the needed factors
if (WE_compute_factor)
begin
if (counter==2'b00)
begin
Ain_real<=Input_real;
Ain_im<=Input_image;
stage1<=0;
end
else if (counter==2'b01)
begin
Bin_real<=Input_real;
Bin_im<=Input_image;

```

```

        end
    else if (counter==2'b10)
        begin
            Cin_real<=Input_real;
            Cin_im<=Input_image;
        end
    else if (counter==2'b11)
        begin
            Din_real<=Input_real;
            Din_im<=Input_image;

            stage1<=1;
        end
    counter=counter+1;
end
end
always @(posedge clk) // We compute all the factors here and also the Input and output
// that doesn't need multiplier. (Areal,Aim)
// In every 4 pairs due to Radix4, the one pair doesn't need multiplier
begin
    if (stage1)
    begin
        factor1_a      <= Ain_real+Bin_im;
        factor1_b      <= Cin_real+Din_im;
        factor2_a      <= Ain_real+Cin_real;
        factor2_b      <= Bin_real+Din_real;
        factor3_a      <= Ain_real+Din_im;
        factor3_b      <= Cin_real+Bin_im;
        factor4_a      <= Ain_im+Din_real;
        factor4_b      <= Cin_im+Bin_real;
        factor5_a      <= Ain_im+Cin_im;
        factor5_b      <= Bin_im+Din_im;
        factor6_a      <= Ain_im+Bin_real;
        factor6_b      <= Cin_im+Din_real;

        Areal_a        <=Ain_real+Bin_real;
        Areal_b        <=Cin_real+Din_real;
        Aim_a           <=Ain_im+Bin_im;
        Aim_b           <=Cin_im+Din_im;

        stage2<=1;
    end
end

always @(posedge clk)

```

```

begin
    t1<=Areal_a+Areal_b;
    t2<=Aim_a+Aim_b;
    t3<=factor1_a-factor1_b;
    t4<=factor4_a-factor4_b;
    t5<=factor2_a-factor2_b;
    t6<=factor5_a-factor5_b;
    t7<=factor3_a-factor3_b;
    t8<=factor6_a-factor6_b;
end

always @(posedge clk)
begin
    WE_BF<=0;
    if (stage2)
    begin
        if (uncounter==2'b01)
        begin
            Aoutreal_reg_0 <=t1;
            Aoutim_reg_0 <=t2;
            WE_BF<=1;
        end
        else if (uncounter==2'b10)
        begin
            factorA <=t3;
            factorB<=t4;
            WE_BF<=1;
        end
        else if (uncounter==2'b11)
        begin
            factorA <=t5;
            factorB<=t6;
            WE_BF<=1;
        end
        end
        else if (uncounter==2'b00)
        begin
            factorA <=t7;
            factorB<=t8;
            WE_BF<=1;
        end
        uncounter=uncounter+1;
    end

end
//my BF_multiplier1 needs 10 cycles in order to send the input to output

```

```

// that's why I need to keep my Aim, Areal for 11 cycles. #11||
//register1 for Areal,Aim
always @(posedge clk)
begin
    Aoutreal_reg_1          <= Aoutreal_reg_0;
    Aoutim_reg_1           <= Aoutim_reg_0;

end
always @(posedge clk)
begin
    Aoutreal_reg_2          <= Aoutreal_reg_1;
    Aoutim_reg_2           <= Aoutim_reg_1;

end

always @(posedge clk)
begin
    Aoutreal_reg_3          <= Aoutreal_reg_2;
    Aoutim_reg_3           <= Aoutim_reg_2;

end

always @(posedge clk)
begin
    Aoutreal_reg_4          <= Aoutreal_reg_3;
    Aoutim_reg_4           <= Aoutim_reg_3;

end

always @(posedge clk)
begin
    Aoutreal_reg_5          <= Aoutreal_reg_4;
    Aoutim_reg_5           <= Aoutim_reg_4;

end

always @(posedge clk)
begin
    Aoutreal_reg_6          <= Aoutreal_reg_5;
    Aoutim_reg_6           <= Aoutim_reg_5;

end

always @(posedge clk)
begin
    Aoutreal_reg_7          <= Aoutreal_reg_6;
    Aoutim_reg_7           <= Aoutim_reg_6;

end
end

```

```

always @(posedge clk)
begin
    Aoutreal_reg_8          <= Aoutreal_reg_7;
    Aoutim_reg_8           <= Aoutim_reg_7;

end
always @(posedge clk)
begin
    Aoutreal_reg_9          <= Aoutreal_reg_8;
    Aoutim_reg_9           <= Aoutim_reg_8;

end
always @(posedge clk)
begin
    Aoutreal_reg_10         <= Aoutreal_reg_9;
    Aoutim_reg_10          <= Aoutim_reg_9;

end
always @(posedge clk)
begin
    Aoutreal_reg_11         <= Aoutreal_reg_10;
    Aoutim_reg_11          <= Aoutim_reg_10;

end
always @(posedge clk)
begin
    Aout_real               <= Aoutreal_reg_11;
    Aout_im                 <= Aoutim_reg_11;

end
endmodule

```

A2.VII. BF's Multiplier

```

`timescale 1ns / 1ps
//02-01-10 32bit input, N=64, Radix4
module BF_multiplier1
(clk,reset,clk_enable,Wn_real1,Wn_im1,factorA,factorB,out1_real,out1_im);
input  [31:0] Wn_real1,Wn_im1 ;
input  [31:0] factorA,factorB ;
input  clk,clk_enable,reset;
// clk_enable = WE_BF from Compute Factor.
// only if it is enabled, the multiplier functions are going to worked.
// otherwise, every 4 signals, one is going to run through the multiple pipeline
// registered insted of the multipliers
output reg [31:0] out1_real;

```

```

output reg [31:0] out1_im;
wire [31:0] out1_real0;
wire [31:0] out1_im0;
reg [31:0] Wn_real0;
reg [31:0] Wn_im0;
reg [31:0] factorA1;
reg [31:0] factorB1;
wire [31:0] Sum1a ;
wire [31:0] Sum1b ;
wire [31:0] Sum2a ;
wire [31:0] Sum2b ;

//using the traditional way as described in N.Gkikas thesis
// each time computes only one pair of output.
// So every time uses the 4 multipliers of virtex2
// in every clk accepts new input for computing the 1 pair of FFT stage.
//11-12-09 32bit input, N=64, Radix4
//register1
always @(posedge clk)
begin
    if (clk_enable)
        begin
            Wn_real0          <=Wn_real1;
            Wn_im0           <=Wn_im1;
            factorA1         <=factorA;
            factorB1         <=factorB;
        end
end

end
// Based on the following procedure
//   for (n=0;n<3;n=n+1)
//       begin
//msb_keeper_real[63:0]= factor[1+n][31:0]*Wn_real[n][31:0] -
//factor[4+n][31:0]*Wn_im[n][31:0];
//msb_keeper_im[63:0]= factor[4+n][31:0]*Wn_real[n][31:0] +
//factor[1+n][31:0]*Wn_im[n][31:0];
//           out1_real=msb_keeper_real [63:32];
//           out1_im   =msb_keeper_im   [63:32];
//       end
//mult_32x32_edit1(clk,reset,clk_enable,Ainput,Binput,Sumfinal)

mult_32x32_edit2 i1(clk,reset,clk_enable,factorA1,Wn_real0,Sum1a);
mult_32x32_edit2 i2(clk,reset,clk_enable,factorB1,Wn_im0,Sum1b);

```

```

mult_32x32_edit2 i3(clk,reset,clk_enable,factorB1,Wn_real0,Sum2a);
mult_32x32_edit2 i4(clk,reset,clk_enable,factorA1,Wn_im0,Sum2b);
assign out1_real0=Sum1a+Sum1b;
assign out1_im0 =Sum2a-Sum2b;
//register 3
always @(posedge clk)
begin
    if (clk_enable)
    begin
        out1_real<=out1_real0;
        out1_im <=out1_im0;
    end
end
endmodule

```

A2.VIII. Multiplier 32x32 Module

```

`timescale 1ns/100ps
// This program is a 32x32 bit multiplier that uses the MULT18x18S multipliers of
// Virtex2 and send the 3 [...]bit results to the CSA. The multiple CSA give us 2 outputs
// that we send afterwards to the modified CLA in order to take the final result.
// we concatenate the final result in order to have an 32bit output the results were checked
// and are correct

// multiplier Edition 3
module mult_32x32_edit2(clk,reset,clk_enable,Ainput,Binput,Sumfinal);
input [31:0] Ainput,Binput;
input clk,clk_enable,reset;
wire [63:0] Sumfinal_not_ready;
output reg [31:0] Sumfinal;//reg
wire Areg_sign,Breg_sign;
reg [35:0] Ain,Bin;
wire [35:0] Ain0,Bin0;
reg [17:0] Ain1=18'b0;
reg [17:0] Ain2=18'b0;
reg [17:0] Bin1=18'b0;
reg [17:0] Bin2=18'b0;
reg [17:0] Ain1_part2,Ain2_part2,Bin1_part2,Bin2_part2;
wire [16:0] C1,C1a;
wire [10:0] C0;
wire [63:0] Addition1, Addition2, Addition3;
wire [63:0] Sum1,Sum2,Sum3;
reg [63:0] Addition1a, Addition2a, Addition3a;
wire [35:0] telos1,telos2,telos3,telos4;
reg [35:0] a_telos3,a_telos4;
assign C0=11'b000000000000;//10

```

```

assign
Ain0=Ainput[31]?{3'b111,Ainput[31:17],1'b0,Ainput[16:0]}:{4'b0000,Ainput[30:17],1'b
0,Ainput[16:0]};
assign
Bin0=Bininput[31]?{3'b111,Bininput[31:17],1'b0,Bininput[16:0]}:{4'b0000,Bininput[30:17],1'b
0,Bininput[16:0]};
always @(posedge clk)
begin
Ain1<={1'b0,Ain0[16:0]};
Bin1<={1'b0,Bin0[16:0]};
Ain2<=Ain0[35:18];
Bin2<=Bin0[35:18];
end
MULT18X18S A0B1 (.A(Ain1),
.B(Bin2),
.C(clk),
.CE(clk_enable),
.R(reset),
.P(telos3));//anti 71
MULT18X18S A1B0 (.A(Ain2),
.B(Bin1),
.C(clk),
.CE(clk_enable),
.R(reset),
.P(telos4));

always @(posedge clk)
begin
a_telos3<=telos3[35:0];
a_telos4<=telos4[35:0];
Ain1_part2<=Ain1;
Bin1_part2<=Bin1;
Ain2_part2<=Ain2;
Bin2_part2<=Bin2;
end
MULT18X18S A0B0 (.A(Ain1_part2),
.B(Bin1_part2),
.C(clk),
.CE(clk_enable),
.R(reset),
.P(telos1));
MULT18X18S A1B1 (.A(Ain2_part2),
.B(Bin2_part2),
.C(clk),
.CE(clk_enable),

```

```

.R(reset),
.P(telos2));//anti 71

assign C1=a_telos3[35]?17'b11111111111111111111:17'b00000000000000000000;
assign C1a=a_telos4[35]?17'b11111111111111111111:17'b00000000000000000000;
assign Addition1={telos2,telos1[33:6]};
assign Addition2={C1,a_telos3,C0};
assign Addition3={C1a,a_telos4,C0};

always @(posedge clk)
begin
    Addition1a<=Addition1;
    Addition2a<=Addition2;
    Addition3a<=Addition3;

end
csa_for_multiplier_beh_pip32x32_a
aat(Addition1a,Addition2a,Addition3a,Sum1,Sum2,clk);
//there is a register in csa , thats why I dont use one here.

multiplier_behav_pip_a aaw (Sum1,Sum2,Sum3,clk);
//there is a register in CLAH , thats why I dont use one here.

always @(posedge clk)
begin
Sumfinal<=Sum3[55:24];//final_not_ready[55:24];
end
endmodule

```

A2.IX. BF Multiplier—CSA module

```

`timescale 1ns / 1ps
//CSA . We create a 64 CSA that have 3 inputs and 2 outputs.
//We collect the inputs from the 4 multipliers of virtex2 MULT18x18S
//We send the two outputs to the Carry Lookahead adder of the
multiplier_behav_pip_32x23

module csa_for_multiplier_beh_pip32x32_a(Ain,Bin,Cin,Output1,Output2,CLK2);
input [63:0] Ain,Bin,Cin;
output reg [63:0] Output1,Output2;
input CLK2;
reg [63:0] Cout,Sum;
integer i;
always @(posedge CLK2)

```

```

        begin
//      Sum[1]=Ain[1]+Bin[1];
//      Cout[1]=1'b0;
        for (i=0;i<64;i=i+1)
            begin :L1
                Cout[i]<=(Ain[i]&Bin[i])|(Ain[i]&Cin[i])|(Bin[i]&Cin[i]);
                Sum[i]<= Ain[i]^Bin[i]^Cin[i];
            end
        end
    end
    always @(Sum)
        begin
            Output2[63:0]<=Sum[63:0];
            Output1[63:0]<={ Cout[62:0],1'b0};
        end
    endmodule

```

A2.X. BF Multiplier—CLAH module

```

`timescale 1ns / 1ps
module multiplier_behav_pip_a(Ain,Bin,Summary,CLK1);
input [63:0] Ain,Bin;
output [63:0] Summary;
input CLK1;
wire C0;
wire [15:0] Gener,Propagate ;
reg [3:0] Propagate3stage1,Gener3stage1 ;
wire [3:0] Propagate3stage,Gener3stage;
wire [11:0] doesntmatter ;
wire useless,useless1;
wire [7:0] doesntmatter1;
reg [63:0] Ain1,Ain2,Ain3,Ain4,Bin1,Bin2,Bin3,Bin4;
reg [15:0] Gener1,Gener2,Gener3,Propagate1,Propagate2,Propagate3 ;
reg [3:0] Cin1;
reg [15:0] Cjfinal;
wire [11:0] Cjstage5;
wire [2:0] Cin4stage;
assign C0=0;

//stage 1
    generate
        genvar e;
        for (e=0;e<16;e=e+1)
            begin : L1

                //assign A=

```

```

//assign B=Bin[3+e*4:0+e*4];

alu1 u1 (Ain[3+e*4:0+e*4],Bin[3+e*4:0+e*4],C0,Gener[e], Propagate[e]);
    end
endgenerate

//Register1
always @(posedge CLK1)
    begin
Gener1[15:0]<=Gener[15:0];
Propagate1[15:0]<=Propagate[15:0];
Ain1[63:0]<=Ain[63:0];
Bin1[63:0]<=Bin[63:0];
    end

//stage 2
generate
    genvar k;
    for (k=0;k<4;k=k+1)
        begin :L2
CLAH name1(Gener1[3+k*4:0+k*4],Propagate1[3+k*4:0+k*4],C0, Gener3stage[k],
Propagate3stage[k], doesntmatter[2+k*3:0+k*3]);

                end
        endgenerate

// Register2
always @(posedge CLK1)
    begin

Gener3stage1[3:0]<=Gener3stage[3:0];
Propagate3stage1[3:0]<=Propagate3stage[3:0];
Gener2[15:0]<=Gener1[15:0]; //we keep them in order to use them on stage 4
Propagate2[15:0]<=Propagate1[15:0]; //we keep them in order to use them on stage 4
//Cj_reg2[3:1]=doesntmatter[2:0]; //stores only the Cj from the first CLAH ???
Ain2[63:0]<=Ain1[63:0];
Bin2[63:0]<=Bin1[63:0];
    end

//stage 3
CLAH name2(Gener3stage1[3:0],Propagate3stage1[3:0],C0,useless,useless1,
Cin4stage[2:0]);

// Register3
always @(posedge CLK1)
    begin

```

```

Cin1[3:0]<={Cin4stage[2:0],C0};//input as the Cin to the CLAH of stage 4-Cin for
CLAH 1 is 0 (we dont have Cin in start)
Gener3[15:0]<=Gener2[15:0]; //we keep them in order to use them on stage 4
Propagate3[15:0]<=Propagate2[15:0];//we keep them in order to use them on stage 4
Ain3[63:0]<=Ain2[63:0];
Bin3[63:0]<=Bin2[63:0];
end

```

```

//stage 4
generate

```

```

    genvar i;
    for (i=0;i<4;i=i+1)
    begin :L3

```

```

        CLAH name3 (Gener3[3+i*4:0+i*4],Propagate3[3+i*4:0+i*4], Cin1[i],doesntmatter1[i],
        doesntmatter1[i+4],Cjstage5[2+i*3:0+i*3]);

```

```

        end
    endgenerate

```

```

// Register 4

```

```

always @(posedge CLK1)
    begin

```

```

        Cjfinal[15:0]<={Cjstage5[11:9],Cin1[3],Cjstage5[8:6],Cin1[2],Cjstage5[5:3],Cin1[1],Cjstage5[2:0],Cin1[0]};
        Ain4[63:0]<=Ain3[63:0];
        Bin4[63:0]<=Bin3[63:0];
    end

```

```

//stage 5

```

```

    generate

```

```

        genvar p;
        for (p=0;p<16;p=p+1)
        begin :L4

```

```

            alu2 u2 (Ain4[3+p*4:0+p*4],Bin4[3+p*4:0+p*4],Cjfinal[p],Summary[3+p*4:0+p*4]);

```

```

            end
        endgenerate

```

```

    endmodule

```

```

////////////////////////////////////

```

```

module alu1 (A,B,C0,Gener,Propagate);

```

```

//parameter n=4;
input C0;
input [3:0] A,B;//n-1
output reg Gener;
output reg Propagate;
wire [4:1] g; //n
wire [4:1] prop; //n
and and0 (g[1],A[0],B[0]);
and and1 (g[2],A[1],B[1]);
and and2 (g[3],A[2],B[2]);
and and3 (g[4],A[3],B[3]);
xor xor1(prop[1],A[0],B[0]);
xor xor2(prop[2],A[1],B[1]);
xor xor3(prop[3],A[2],B[2]);
xor xor4(prop[4],A[3],B[3]);
always@(A,B,C0)
    begin
        Propagate=prop[1]&prop[2]&prop[3]&prop[4];//i didnt put the C0
        Gener=
g[1]&prop[2]&prop[3]&prop[4]|g[2]&prop[3]&prop[4]|g[3]&prop[4]|g[4];
    end
endmodule

```

////////////////////////////////////

```

module alu2 (A,B,C0,Sum);
input C0;
input [3:0] A,B ;//n-1
output reg [3:0] Sum ;//n-1
reg Cout;
always@(A,B,C0)

    begin
        {Cout,Sum}=      A+B+C0;
    end
endmodule

```

////////////////////////////////////

```

module CLAH(G,P,C0,Generatea,Propagate,Cj);
input C0;
input [4:1] G,P;
output [3:1] Cj;
output Generatea,Propagate;
assign Propagate=P[1]&P[2]&P[3]&P[4];

```

```
assign Generatea=G[1]&P[2]&P[3]&P[4]|G[2]&P[3]&P[4]|G[3]&P[4]|G[4];
assign Cj[1]=(P[1]&C0)|G[1];
assign Cj[2]=G[2]|(P[2]&G[1])|(P[2]&P[1]&C0);
assign Cj[3]=G[3]|(P[3]&G[2])|(P[3]&P[2]&G[1])|(P[3]&P[2]&P[1]&C0);
endmodule
```

APPENDIX B. TMR IMPLEMENTATION

This appendix includes the TMR version of the 64-point, in-place, Radix-4, DIF FFT design as it is described in Chapter IV.B.1. All arithmetic calculations are protected by triple redundancy. The module receives two fixed-point 32-bit inputs (real and imaginary) and outputs two fixed-point 32-bit results. In the last step of each FFT's stage, the three computed values are inspected by a voter, based on the principle of triple redundancy. The design failed to be implemented in a Virtex II XC2V6000 BF957 and was finally implemented in a Virtex II XC2V8000 5FF1152 using Xilinx tools. The synthesis report from the XC2V6000 confirmed that the design needed 85 cycles per stage, a total of 255 cycles of latency at a clock speed of 134MHz.

In this appendix, only the schematic and the structural or behavioral modules that were changed or added after the creation of the basic FFT design (Appendix A) are illustrated.

B1. TMR MODULE—XILINX SCHEMATIC DESIGN

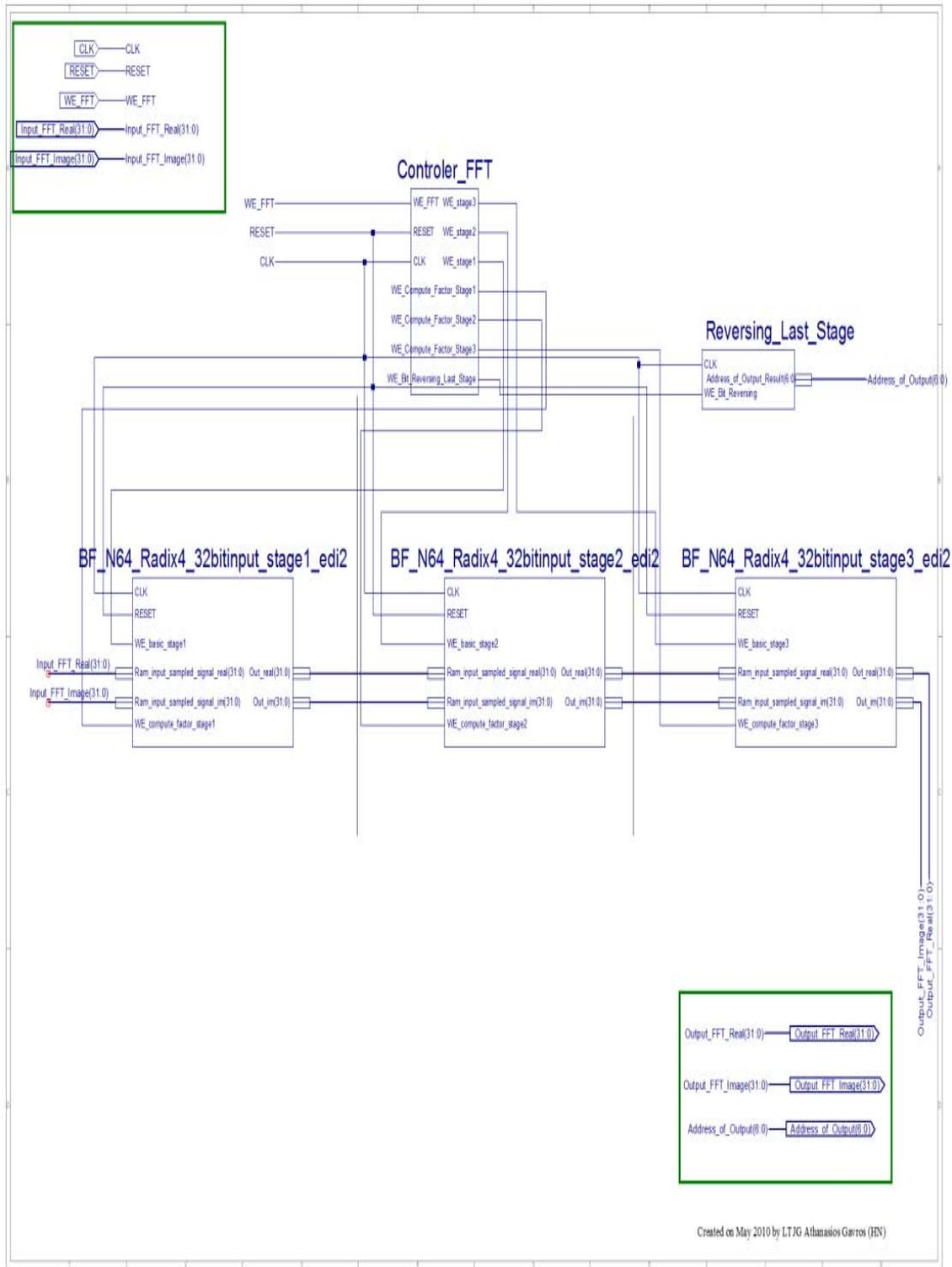


Figure 22. TMR's Entirely Concept Layout.

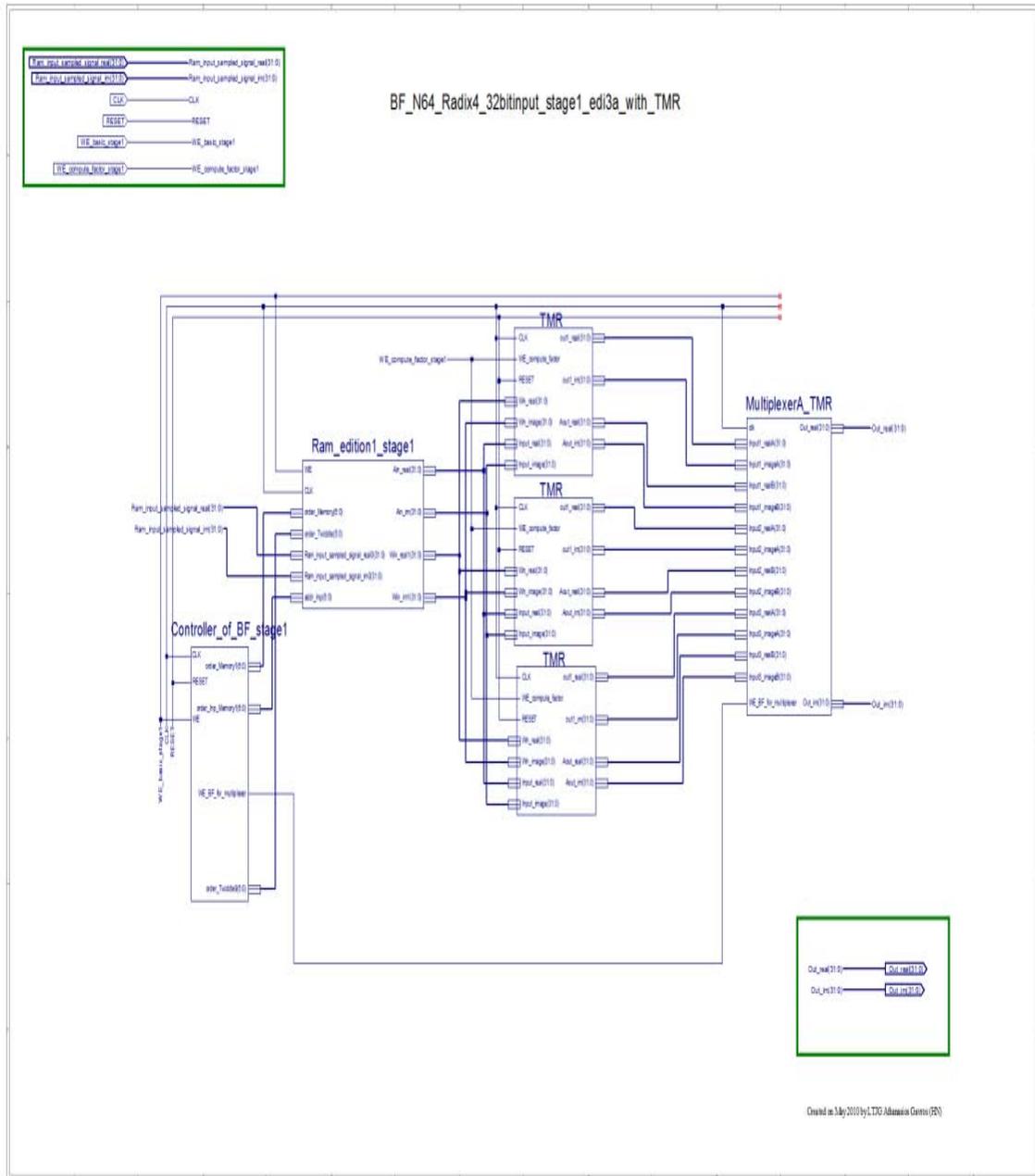


Figure 23. TMR's First Stage Layout.

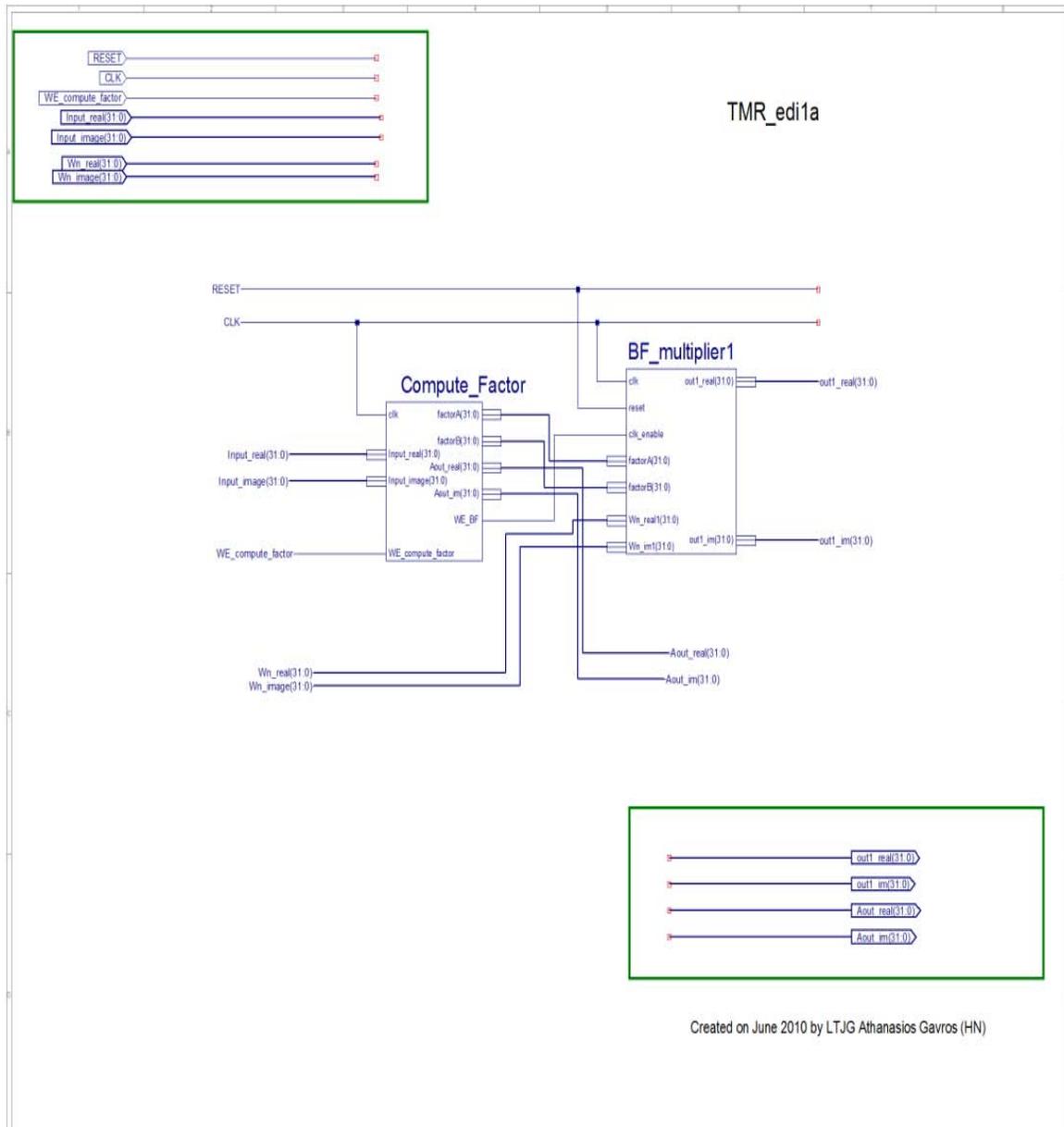


Figure 24. TMR's BF Module

B2. TMR MODULE—XILINX BEHAVIORAL AND STRUCTURAL DESIGN

B2.I. Multiplexer and Voter of Each Stage

``timescale 1ns / 1ps`

```
Module MultiplexerA_TMR(clk,Input1_realA,Input1_imageA,Input1_realB,
Input1_imageB, Input2_realA, Input2_imageA,Input2_realB,Input2_imageB,
Input3_realA, Input3_imageA,Input3_realB,Input3_imageB,
WE_BF_for_multiplexer,Out_real,Out_im);
```

```

input [31:0] Input1_realA,Input1_imageA,Input1_realB,Input1_imageB;
input [31:0] Input2_realA,Input2_imageA,Input2_realB,Input2_imageB;
input [31:0] Input3_realA,Input3_imageA,Input3_realB,Input3_imageB;

input clk,WE_BF_for_multiplexer;
output [31:0] Out_real;
output [31:0] Out_im;
reg [31:0] Before_Voter_real [2:0];
reg [31:0] Before_Voter_im [2:0];
wire [31:0] Out0_real,Out0_im;
always @(posedge clk)
begin
    if (WE_BF_for_multiplexer)
        begin
            Before_Voter_real[0]    <=Input1_realA;
            Before_Voter_im[0]      <=Input1_imageA;
            Before_Voter_real[1]    <=Input2_realA;
            Before_Voter_im[1]      <=Input2_imageA;
            Before_Voter_real[2]    <=Input3_realA;
            Before_Voter_im[2]      <=Input3_imageA;
        end
    else
        begin
            Before_Voter_real[0]    <=Input1_realB;
            Before_Voter_im[0]      <=Input1_imageB;
            Before_Voter_real[1]    <=Input2_realB;
            Before_Voter_im[1]      <=Input2_imageB;
            Before_Voter_real[2]    <=Input3_realB;
            Before_Voter_im[2]      <=Input3_imageB;
        end
end

assign
Out0_real=(Before_Voter_real[0]==Before_Voter_real[1])?Before_Voter_real[0]:Before
_Voter_real[1];
assign Out_real=(Out0_real==Before_Voter_real[2])?Out0_real:Before_Voter_real[0];
assign
Out0_im=(Before_Voter_im[0]==Before_Voter_im[1])?Before_Voter_im[0]:Before_Vo
ter_im[1];
assign Out_im=(Out0_im==Before_Voter_im[2])?Out0_im:Before_Voter_im[0];
endmodule

```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. RPR IMPLEMENTATION

This appendix includes the RPR version of the 64-point, in place, Radix-4, DIF FFT design as it is described in Chapter IV.B.3. All arithmetic calculations of the FFT are protected by the reduced precision redundancy method. The module receives two fixed-point 32-bit inputs (real and imaginary) and outputs two fixed-point 32-bit results. In the last step of each FFT's stage, the precise value and the truncated values are inspected by a voter, based on the principle of reduced precision redundancy. The design was first implemented in a Virtex II XC2V6000 BF957 and then in a Virtex II XC2V8000 5FF1152 using Xilinx tools. The synthesis report from the XC2V6000 confirmed that the design needed 85 cycles per stage, a total of 255 cycles of latency at a clock speed of 163Mhz.

In this chapter, only the schematic and structural or behavioral modules that were changed or added after the creation of the TMR design (Appendix B) are illustrated.

C1. RPR MODULE—XILINX SCHEMATIC DESIGN

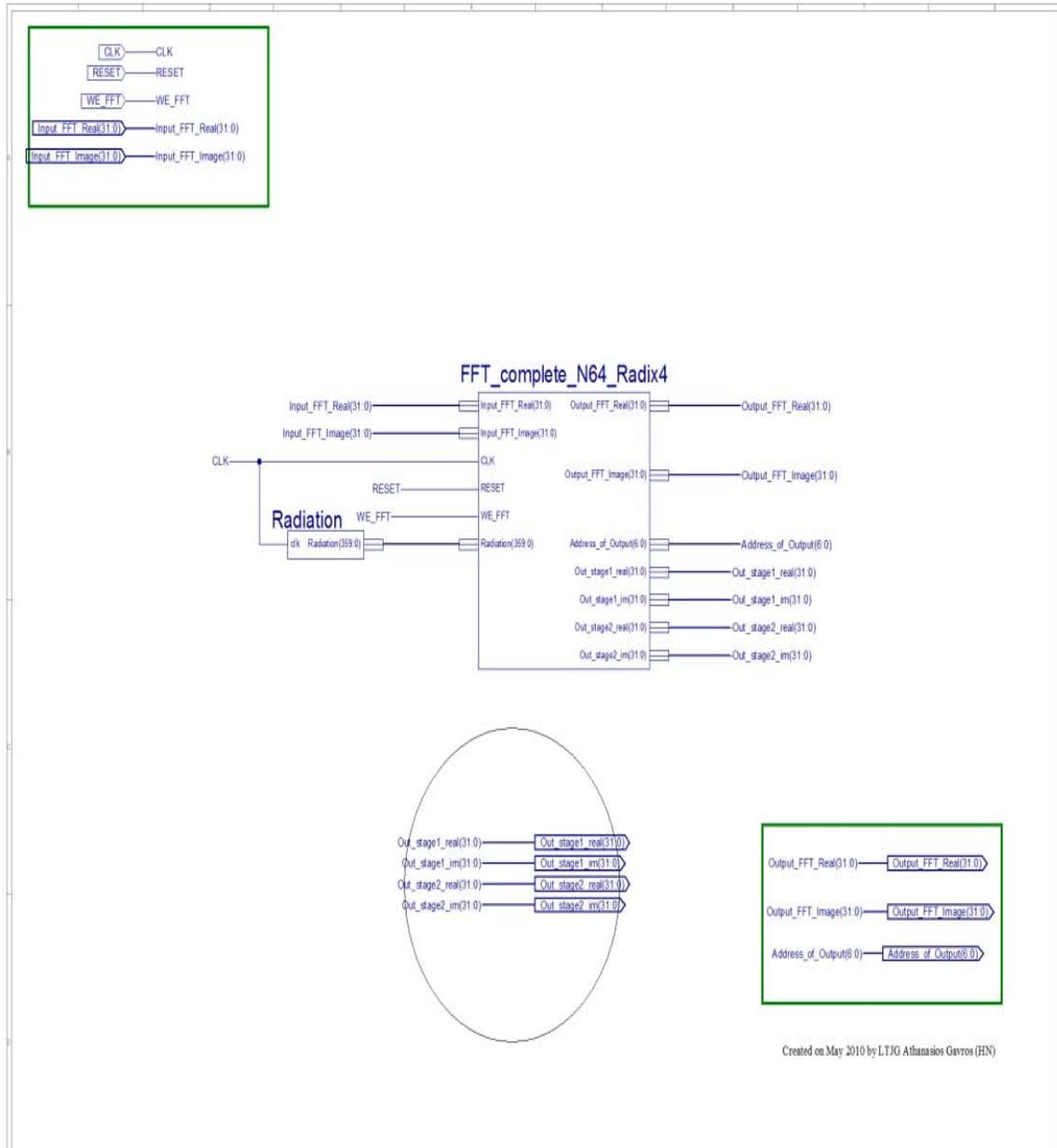


Figure 25. RPR- Entirely Module Layout

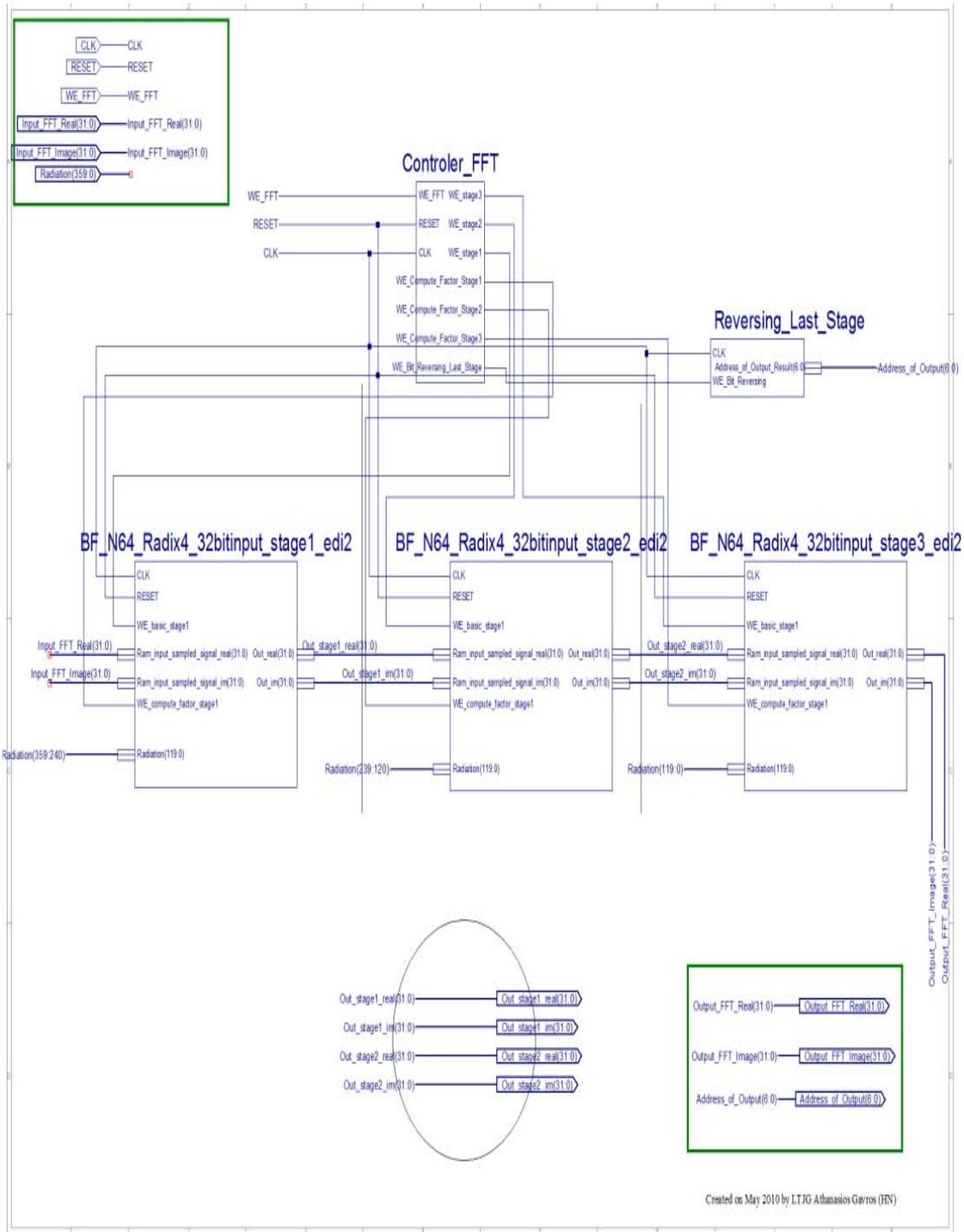


Figure 26. RPR Module—Three Stages Layout

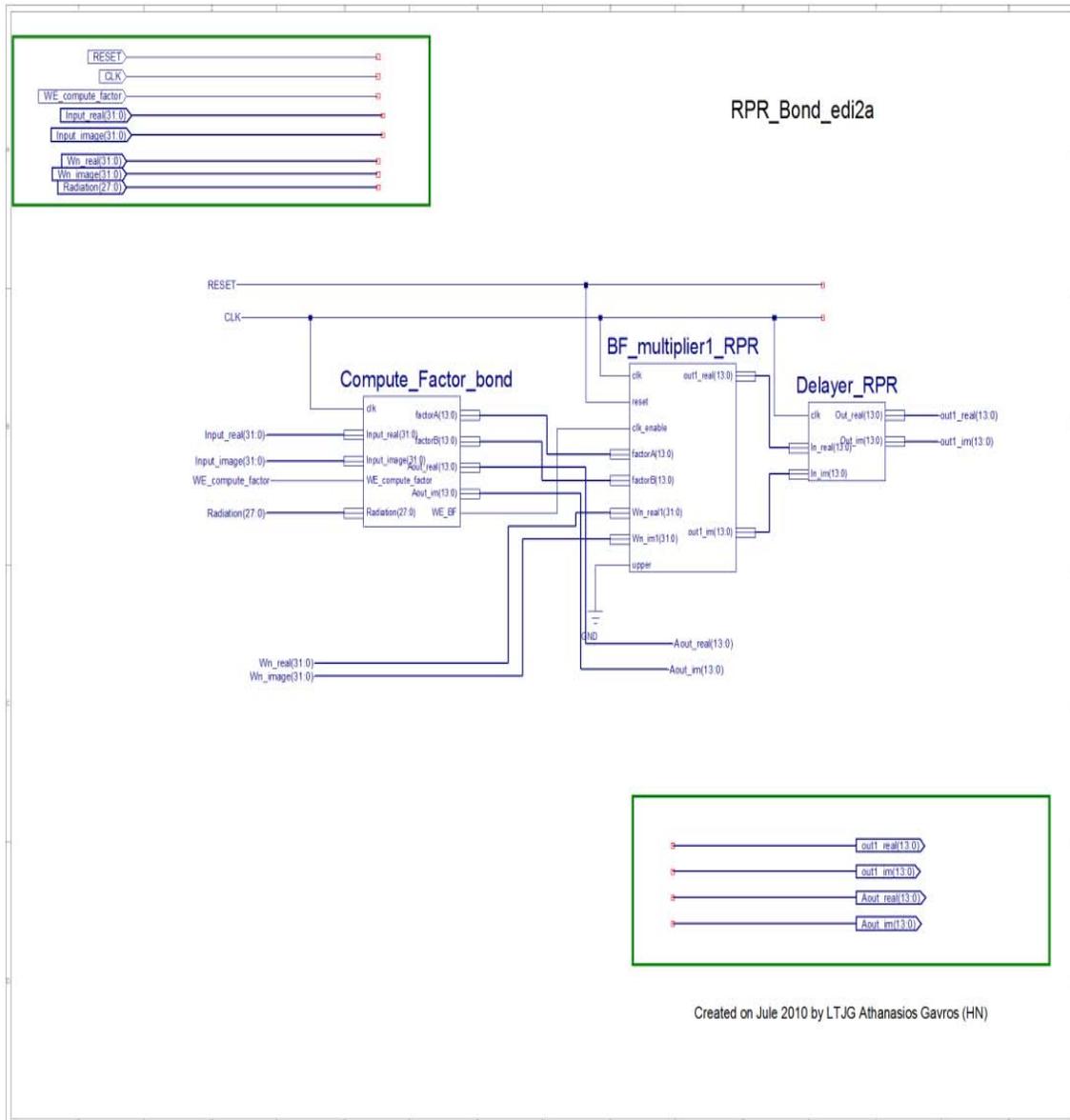


Figure 28. RPR—Butterfly

C2. RPR MODULE—XILINX STRUCTURAL AND BEHAVIORAL DESIGN

C2.I. Radiation Module

```

`timescale 1ns / 1ps
module Radiation(clk,Radiation);
input clk;
output [359:0] Radiation;
//Radiation Guide

```

```

// Radiation [359:0]
// RadiationA[119:0] = Radiation [359:240] radiation of first stage
// RadiationB[119:0] = Radiation [239:120] radiation of second stage
// RadiationC[119:0] = Radiation [119:0] radiation of third stage
reg [13:0] RadiationAr_upper=14'b0;
reg [13:0] RadiationBr_upper=14'b0;
reg [13:0] RadiationCr_upper=14'b0;
reg [13:0] RadiationAr_lower=14'b0;
reg [13:0] RadiationBr_lower=14'b0;
reg [13:0] RadiationCr_lower=14'b0;
reg [31:0] RadiationAr_TMR=32'b0;
reg [31:0] RadiationBr_TMR=32'b0;
reg [31:0] RadiationCr_TMR=32'b0;
reg [13:0] RadiationAi_upper=14'b0;
reg [13:0] RadiationBi_upper=14'b0;
reg [13:0] RadiationCi_upper=14'b0;
reg [13:0] RadiationAi_lower=14'b0;
reg [13:0] RadiationBi_lower=14'b0;
reg [13:0] RadiationCi_lower=14'b0;
reg [31:0] RadiationAi_TMR=32'b0;
reg [31:0] RadiationBi_TMR=32'b0;
reg [31:0] RadiationCi_TMR=32'b0;
wire [119:0] RadiationA,RadiationB,RadiationC;
reg [1:0] counter=2'b00;
reg [2:0] random=3'b001;
parameter High=14'b00111000000000;
parameter Low=      14'b000000000000001;
reg [6:0] mm=0;
wire [13:0] Value1;
assign Value1=High;
always @(posedge clk)
begin
counter<=2;//counter+1;
random<=random+3'b001;
case(counter)
0:
begin
RadiationAr_upper<=Value1+random;
RadiationBr_upper<=Value1+random;
RadiationCr_upper<=Value1+random;
RadiationAi_upper<=Value1+random;
RadiationBi_upper<=Value1+random;
RadiationCi_upper<=Value1+random;
RadiationAr_lower<=0;
RadiationBr_lower<=0;

```

```

RadiationCr_lower<=0;
RadiationAi_lower<=0;
RadiationBi_lower<=0;
RadiationCi_lower<=0;
RadiationAr_TMR<=0;
RadiationBr_TMR<=0;
RadiationCr_TMR<=0;
RadiationAi_TMR<=0;
RadiationBi_TMR<=0;
RadiationCi_TMR<=0;
end
1:
begin
RadiationAr_lower<=Value1+random;
RadiationBr_lower<=Value1+random;
RadiationCr_lower<=Value1+random;
RadiationAi_lower<=Value1+random;
RadiationBi_lower<=Value1+random;
RadiationCi_lower<=Value1+random;
RadiationAr_upper<=0;
RadiationBr_upper<=0;
RadiationCr_upper<=0;
RadiationAi_upper<=0;
RadiationBi_upper<=0;
RadiationCi_upper<=0;
RadiationAr_TMR<=0;
RadiationBr_TMR<=0;
RadiationCr_TMR<=0;
RadiationAi_TMR<=0;
RadiationBi_TMR<=0;
RadiationCi_TMR<=0;
end
2:
begin
mm=mm+1;
if ((mm==4)||((mm==6)||(mm==7)))
    begin
        RadiationBr_TMR<={{(Value1[13:11]+random),Value1[10:0],18'b0}};
    end
    else
        begin
            RadiationCr_TMR<=0;/{{(Value1+random),18'b0}};
        end
        RadiationCr_TMR<=0;/{{(Value1+random),18'b0}};
        RadiationAr_TMR<=0;/{{(Value1+random),18'b0}};
    end
end

```

```

        RadiationAi_TMR<=0;//{(Value1+random),18'b0};
        RadiationBi_TMR<=0;//{(Value1+random),18'b0};
        RadiationCi_TMR<=0;//{(Value1+random),18'b0};
        RadiationAr_upper<=0;
        RadiationBr_upper<=0;
        RadiationCr_upper<=0;
        RadiationAi_upper<=0;
        RadiationBi_upper<=0;
        RadiationCi_upper<=0;
        RadiationAr_lower<=0;
        RadiationBr_lower<=0;
        RadiationCr_lower<=0;
        RadiationAi_lower<=0;
        RadiationBi_lower<=0;
        RadiationCi_lower<=0;
    end

3:
begin
    RadiationAr_upper<=Value1+random;
    RadiationBr_TMR<={(Value1+random),18'b0};
    RadiationCr_lower<=Value1+random;
    RadiationAi_upper<=Value1+random;
    RadiationBi_TMR<={(Value1+random),18'b0};
    RadiationCi_lower<=Value1+random;
    RadiationBr_upper<=0;
    RadiationCr_upper<=0;
    RadiationBi_upper<=0;
    RadiationCi_upper<=0;
    RadiationAr_lower<=0;
    RadiationBr_lower<=0;
    RadiationAi_lower<=0;
    RadiationBi_lower<=0;
    RadiationAr_TMR<=0;
    RadiationCr_TMR<=0;
    RadiationAi_TMR<=0;
    RadiationCi_TMR<=0;
end
endcase
end

assign
RadiationA[119:0]={RadiationAr_upper,RadiationAi_upper,RadiationAr_TMR,RadiationBi_TMR,RadiationAr_lower,RadiationAi_lower};

```

```

assign
RadiationB[119:0]={RadiationBr_upper,RadiationBi_upper,RadiationBr_TMR,RadiationBi_TMR,RadiationBr_lower,RadiationBi_lower};
assign
RadiationC[119:0]={RadiationCr_upper,RadiationCi_upper,RadiationCr_TMR,RadiationCi_TMR,RadiationCr_lower,RadiationCi_lower};
assign Radiation[359:0]={RadiationA,RadiationB,RadiationC};
endmodule

```

C2.II. Multiplexer and Voter of Each Stage

```

`timescale 1ns / 1ps
module
Multiplexer_RPR(clk,Input1_realA,Input1_imageA,Input1_realB,Input1_imageB,Input2_realA,Input2_imageA,Input2_realB,Input2_imageB,Input3_realA,Input3_imageA,Input3_realB,Input3_imageB,WE_BF_for_multiplexer,Out_real,Out_im);

input [13:0] Input1_realA,Input1_imageA,Input1_realB,Input1_imageB;
input [31:0] Input2_realA,Input2_imageA,Input2_realB,Input2_imageB;
input [13:0] Input3_realA,Input3_imageA,Input3_realB,Input3_imageB;
input clk,WE_BF_for_multiplexer;
output [31:0] Out_real;
output [31:0] Out_im;
reg signed [31:0] Before_Voter_real;
reg signed [31:0] Before_Voter_im;
reg signed [13:0] Upper_bond_real,Upper_bond_im,Lower_bond_real,Lower_bond_im;
reg signed [31:0] Median_real,Median_im;
reg signed [13:0] Bottom_real;
reg signed [13:0] Top_real;
reg signed [13:0] Bottom_im;
reg signed [13:0] Top_im;
reg signed [13:0] Precise_real;
reg signed [13:0] Precise_image;

wire [31:0] Out_real0,Out_im0;
wire K1_r,K2_r,K3_r;
wire K4_r,K5_r;
wire Omega_real;
wire K1_i,K2_i,K3_i;
wire K4_i,K5_i;
wire Omega_im;
always @(posedge clk)
begin
    if (WE_BF_for_multiplexer)
        begin

```

```

        Upper_bond_real          =Input1_realA;
        Upper_bond_im           =Input1_imageA;
        Before_Voter_real       =Input2_realA;
        Before_Voter_im         =Input2_imageA;
        Lower_bond_real         =Input3_realA;
        Lower_bond_im           =Input3_imageA;
    end
else
    begin

        Upper_bond_real          =Input1_realB;
        Upper_bond_im           =Input1_imageB;
        Before_Voter_real       =Input2_realB;
        Before_Voter_im         =Input2_imageB;
        Lower_bond_real         =Input3_realB;
        Lower_bond_im           =Input3_imageB;
    end

```

```

Median_real={{(Upper_bond_real[13],Upper_bond_real)}+{Lower_bond_real[13],Lower_bond_real}}>>1,18'b0};// means /2
Median_im={{(Upper_bond_im[13],Upper_bond_im)}+{Lower_bond_im[13],Lower_bond_im}}>>1,18'b0};//means /2
Top_real=Upper_bond_real+14'b000000000000100;
Bottom_real=Lower_bond_real+14'b111111111111100;
Top_im=Upper_bond_im+14'b000000000000100;
Bottom_im=Lower_bond_im+14'b111111111111100;
Precise_real=Before_Voter_real[31:18];
Precise_image=Before_Voter_im[31:18];
end

```

```

//RPR Voter
assign K1_r=(Top_real>=Bottom_real)?1'b1:1'b0;
assign K2_r=(Precise_real>Top_real)?1'b1:1'b0;
assign K3_r=(Precise_real<Bottom_real)?1'b1:1'b0;
and P1 (K4_r,K1_r,K2_r);
and P2 (K5_r,K1_r,K3_r);
or P3 (Omega_real,K4_r,K5_r);
assign Out_real=(Omega_real)?Median_real:Before_Voter_real;
assign K1_i=(Top_im>=Bottom_im)?1'b1:1'b0;
assign K2_i=(Precise_image>Top_im)?1'b1:1'b0;
assign K3_i=(Precise_image<Bottom_im)?1'b1:1'b0;
and P11 (K4_i,K1_i,K2_i);
and P21 (K5_i,K1_i,K3_i);
or P31 (Omega_im,K4_i,K5_i);
assign Out_im=(Omega_im)?Median_im:Before_Voter_im;

```

```

// The following example is a behavioral voter of RPR
// But due to decrease in the speed I preferred
// the solution of a structural design as above.
////real voter
//if (Upper_bond_real>Lower_bond_real)
//    begin
//    if (Before_Voter_real>Top_real)
//        begin
//            Out_real={Median_real,24'b0};
//        end
//    else if (Before_Voter_real<Bottom_real)
//        begin
//            Out_real={Median_real,24'b0};
//        end
//    else
//        begin
//            Out_real=Before_Voter_real;
//        end
//    end
//else
//    begin
//        Out_real=Before_Voter_real;
//    end
//
////image voter
//if (Upper_bond_im>Lower_bond_im)
//    begin
//    if (Before_Voter_im>Top_im)
//        begin
//            Out_im={Median_im,24'b0};
//        end
//    else if (Before_Voter_im<Bottom_im)
//        begin
//            Out_im={Median_im,24'b0};
//        end
//    else
//        begin
//            Out_im=Before_Voter_im;
//        end
//    end
//else
//    begin
//        Out_im<=Before_Voter_im;

```

```

//      end
//
//end
Out0_real=(Before_Voter_real[0]==Before_Voter_real[1])?Before_Voter_real[0]:Before
_Voter_real[1];
//assign Out_real=(Out0_real==Before_Voter_real[2])?Out0_real:Before_Voter_real[0];
//assign
Out0_im=(Before_Voter_im[0]==Before_Voter_im[1])?Before_Voter_im[0]:Before_Vo
ter_im[1];
//assign Out_im=(Out0_im==Before_Voter_im[2])?Out0_im:Before_Voter_im[0];
endmodule

```

C2.III. BF—Delayer

```

`timescale 1ns / 1ps
module Delayer_RPR(In_real,In_im,Out_real,Out_im,clk);
input clk;
input      [13:0] In_real,In_im;
output reg  [13:0] Out_real,Out_im;

reg [13:0] Real_reg_0,Im_reg_0;
reg [13:0] Real_reg_1,Im_reg_1;
reg [13:0] Real_reg_2,Im_reg_2;
reg [13:0] Real_reg_3,Im_reg_3;
reg [13:0] Real_reg_4,Im_reg_4;
reg [13:0] Real_reg_5,Im_reg_5;
reg [13:0] Real_reg_6,Im_reg_6;

// This delayer is for the BF outputs due to the fact that the use of 8/32 or 14/32 degree of
RPR result a decrease in use of multipliers, CSA and CLAH and in the decrease of the
needed pipelines. So I have to replace the loss of the 6 "cutted" pipelines with 6 registers
in order my FFT to be synchronous.
always @(posedge clk)
begin
    Real_reg_0      <= In_real;
    Im_reg_0        <= In_im;

end

always @(posedge clk)
begin
    Real_reg_1      <= Real_reg_0;
    Im_reg_1        <= Im_reg_0;

end
end

```

```

always @(posedge clk)
begin
    Real_reg_2      <= Real_reg_1;
    Im_reg_2       <= Im_reg_1;

end

always @(posedge clk)
begin
    Real_reg_3      <= Real_reg_2;
    Im_reg_3       <= Im_reg_2;

end

always @(posedge clk)
begin
    Real_reg_4      <= Real_reg_3;
    Im_reg_4       <= Im_reg_3;

end

always @(posedge clk)
begin
    Real_reg_5      <= Real_reg_4;
    Im_reg_5       <= Im_reg_4;

end

always @(posedge clk)
begin
    Out_real       <= Real_reg_5;
    Out_im        <= Im_reg_5;

end
endmodule

```

C2.IV. BF——Multiplier

```

`timescale 1ns/100ps
// multiplier RPR Edition 1
module mult_14x14_edit1_RPR(clk,reset,clk_enable,Ainput,Binput,Sumfinal);

input  [13:0] Ainput,Binput;
input  clk,clk_enable,reset;
output reg [13:0] Sumfinal;
reg    [17:0] Ain,Bin;
reg    [17:0] Ain1,Bin1;

```

```

wire [17:0] Ain0,Bin0;

assign Ain0=Ainput[13]?{4'b1111,Ainput}:{4'b0000,Ainput};
assign Bin0=Bininput[13]?{4'b1111,Bininput}:{4'b0000,Bininput};
wire [35:0] telos;

always @(posedge clk)
begin
Ain1<=Ain0;
Bin1<=Bin0;
end

MULT18X18S A0B0 (.A(Ain1),
.B(Bin1),
.C(clk),
.CE(clk_enable),
.R(reset),
.P(telos));

always @(posedge clk)
begin
Sumfinal<=telos[25:12];
end
endmodule

```

APPENDIX D. MATLAB FILE

This appendix contains the MATLAB file created to verify the results from Xilinx's RPR module. This file has multiple functions such as:

- Generates input signals, converts them into two's complement fixed-point binary numbers and exports them into files for use in Modelsim.
- Imports the generated output files from Modelsim and converts them into decimal.
- Generates two different algorithms for calculating the FFT. It uses the MATLAB's build-in FFT function and a behavioral simulation in MATLAB, based on the implemented Xilinx design. Simultaneously, it loads the data from Xilinx that is generated by the Modelsim simulator program.
- Generates three different graphical figures based on the three set of data.
- Calculates the bounds for the expected precise calculations error and for the radiation imported errors.
- Shows the output results for every stage of the FFT and compares the three sets of data, enabling the user to inspect the impact of radiation in the whole process of the FFT algorithm.

D1. MAIN FILE

```
% It works perfectly for |Inputs|<2^(-9).
% It works with the later RPR designs due to the 3bit scale down of
% the inputs at the beginning of each stage.

% Radix 4 fft N=64 point
% programma creates twiddle for N=64 bit
% fixed point two's complement
% 30 LSB are the floating.
% sample_signal_real and sample_signal_image contains the input
equation

% this program computes the FFT Radix4 N=64point in shape geometry,
% reversing bit in every stage,compares the result of the calculations
to
% between my Matlab FFT design and Verilog Design.
% Athanasios Gavros 15/07/2010
% edition 1.3a
% As an input signal : real part : 'equationA', imaginary part
: 'equationB'.
% In Phase Compare between my FFT and Verilog, you are going to observe
```

```

% sometimes huge difference. But that is not entirely truth. Due to
small
% numbers (10(-13) etc ..) that should be accepted equal to zero, any
% fluctuation can cause wrong results. That why we compare the image
and
% real part of the two outputs and if the difference is smaller than
% 9.3*(10(-7)) , Matlab is printing the message
Phase_Checking='Correct'. "

%Beware that the results of Verilog are not Bit Reversed on the final
%stage. We are reversing the Address of the Output , but here in
Matlab, we
%are doing this final bit reversal in order to compare our final
results
%with the results of build in fft.
clear all;
close all;
colordef white;
format long
pinakas0=cell(64,64);
pinakas=cell(64,64);
Binary_to_decimal=cell(1,64);
clear cc;
adder=0;
N=64; %point
F1=[0:N-1]/N;

ert1=0;
ert2=0;
%%-----
% Input signal
for n=0:1:63
equationA(n+1)=0.5*cos(2*pi*n/64)+0.3*cos(2*pi*n/9);
equationB(n+1)=0.4*sin(2*pi*n/16)+0.1*sin(2*pi*n/14)-0.3*cos(2*pi*n/3);
end

% RANDOM signal
equationA(48)=0.15;
equationA(10)=0.11;
equationA(14)=0.05;
equationA(17)=0.5;
equationA(27)=-0.15;
equationA(20)=0.5;
equationA(21)=0.15;
equationA(31)=0.17;
equationA(41)=0.19;
equationA(51)=0;
equationB(54)=0.16;
equationB(45)=-0.2;
equationB(41)=-0.7;
equationB(11)=0.9;
equationB(12)=0.9;
equationB(24)=0.25;
%---
if (n>58)

```

```

    equationA(n+1)=0.5;%end
end
for counter=1:1:64
    sample_signal_real(counter)= double( equationA(counter));%0.0001;
    sample_signal_image(counter)= double( equationB(counter));%0.0001;
end
%-----

%%twiddle
for mm = 0:1:63
    %radix-4
    twiddl(mm+1)=double(exp(-2*pi*j*mm*1/64));
    Wn_real(mm+1)=(real(twiddl(mm+1)));%cos(2*pi*mm/64);
    Wn_im(mm+1)=(-imag(twiddl(mm+1)));%sin(2*pi*mm/64);

end

%% Stage1
for p=1:1:16
    for i=0:16:48
        if (i==0)
            Ain_real=sample_signal_real(i+p);
            Ain_im= sample_signal_image(i+p);
        end
        if (i==16)
            Bin_real=sample_signal_real(i+p);
            Bin_im= sample_signal_image(i+p);
        end
        if (i==32)
            Cin_real=sample_signal_real(i+p);
            Cin_im= sample_signal_image(i+p);
        end
        if (i==48)
            Din_real=sample_signal_real(i+p);
            Din_im= sample_signal_image(i+p);
        end
        ertl=ertl+1;
        check1(ertl)=i+p;
    end
    factor1      = Ain_real+Bin_im-Cin_real-Din_im;
    factor2      = Ain_real-Bin_real+Cin_real-Din_real;
    factor3      = Ain_real-Bin_im-Cin_real+Din_im;
    factor4      = Ain_im-Bin_real-Cin_im+Din_real;
    factor5      = Ain_im-Bin_im+Cin_im-Din_im;
    factor6      = Ain_im+Bin_real-Cin_im-Din_real;

    Areal_stage1(0+p)      =Ain_real+Bin_real+Cin_real+Din_real;
    Aim_stage1(0+p)       =Ain_im+Bin_im+Cin_im+Din_im;

    Areal_stage1(16+p)    = double(factor1*Wn_real(p) +
factor4*Wn_im(p));% ena error sto Wn (p+1)!!! pantoy xaos
    Aim_stage1(16+p)     = double(factor4*Wn_real(p) -
factor1*Wn_im(p));

```

```

        Areal_stagel(32+p) = double(factor2*Wn_real(2*(p-1)+1) +
factor5*Wn_im(2*(p-1)+1));
        Aim_stagel(32+p) = double(factor5*Wn_real(2*(p-1)+1) -
factor2*Wn_im(2*(p-1)+1));
        Areal_stagel(48+p) = double(factor3*Wn_real(3*(p-1)+1) +
factor6*Wn_im(3*(p-1)+1));
        Aim_stagel(48+p) = double(factor6*Wn_real(3*(p-1)+1) -
factor3*Wn_im(3*(p-1)+1));

end
%-----
% bit reversal of the Radix4
% example 003-> 300

ttt=0;
e2=0;
for v2=1:1:4
for p2=0:4:12
    for i2=0:16:48
        ttt=ttt+1;

        k22(ttt)=v2+p2+i2;
    end
end
end

for i=1:1:64
Aim_stagel_r(k22(i))=Aim_stagel(i);
Areal_stagel_r(k22(i))=Areal_stagel(i);
end

ttt=0;
e2=0;
for v2=0:16:48
for p2=1:1:4
    for i2=0:4:12
        ttt=ttt+1;

        k22a(ttt)=v2+p2+i2;
    end
end
end

for qw=1:1:64
Aim_stagel_rr(k22a(qw))=Aim_stagel((qw));
Areal_stagel_rr(k22a(qw))=Areal_stagel((qw));
end
%%-----

```

```

%% Stage 2
% Input signal
e2=0;
for v2=0:16:48
for p2=1:1:4
    for i2=0:4:12
        if (i2==0)
            Ain_real2=Areal_stagel_r(i2+p2+v2);
            Ain_im2= Aim_stagel_r(i2+p2+v2);
        end
        if (i2==4)
            Bin_real2=Areal_stagel_r(i2+p2+v2);
            Bin_im2= Aim_stagel_r(i2+p2+v2);
        end
        if (i2==8)
            Cin_real2=Areal_stagel_r(i2+p2+v2);
            Cin_im2= Aim_stagel_r(i2+p2+v2);
        end
        if (i2==12)
            Din_real2=Areal_stagel_r(i2+p2+v2);
            Din_im2= Aim_stagel_r(i2+p2+v2);
        end
        v2+p2+i2; % test milestone
    end

        factor1_stage2      = Ain_real2+Bin_im2-Cin_real2-Din_im2;
        factor2_stage2      = Ain_real2-Bin_real2+Cin_real2-Din_real2;
        factor3_stage2      = Ain_real2-Bin_im2-Cin_real2+Din_im2;
        factor4_stage2      = Ain_im2-Bin_real2-Cin_im2+Din_real2;
        factor5_stage2      = Ain_im2-Bin_im2+Cin_im2-Din_im2;
        factor6_stage2      = Ain_im2+Bin_real2-Cin_im2-Din_real2;

        Areal_stage2(v2+p2)
=Ain_real2+Bin_real2+Cin_real2+Din_real2;
        Aim_stage2(v2+p2)      =Ain_im2+Bin_im2+Cin_im2+Din_im2;

        Areal_stage2(v2+p2+4)  =
double(factor1_stage2*Wn_real(1+e2*4) + factor4_stage2*Wn_im(1+e2*4));
        Aim_stage2(v2+p2+4)    =
double(factor4_stage2*Wn_real(1+e2*4) - factor1_stage2*Wn_im(1+e2*4));
        Areal_stage2(v2+p2+8)  =
double(factor2_stage2*Wn_real(1+e2*8) + factor5_stage2*Wn_im(1+e2*8));
        Aim_stage2(v2+p2+8)    =
double(factor5_stage2*Wn_real(1+e2*8) - factor2_stage2*Wn_im(1+e2*8));
        Areal_stage2(v2+p2+12) =
double(factor3_stage2*Wn_real(1+e2*12) +
factor6_stage2*Wn_im(1+e2*12));
        Aim_stage2(v2+p2+12)  =
double(factor6_stage2*Wn_real(1+e2*12) -
factor3_stage2*Wn_im(1+e2*12));

        1+e2*4;1+e2*8;1+e2*12 ;%test milestone
    end
end

```

```

e2=e2+1;
end
%-----
% bit reversal of the Radix4
% example 003-> 300

ttt=0;
e2=0;
for v2=1:1:4
for p2=0:4:12
for i2=0:16:48
ttt=ttt+1;

k22(ttt)=v2+p2+i2;
end
end
end
for i=1:1:64
Aim_stage2_r(k22(i))=Aim_stage2(i);
Areal_stage2_r(k22(i))=Areal_stage2(i);
end
%%-----
%% Stage 3
% Input signal

for p3=0:4:60
for i3=1:1:4
if (i3==1)
Ain_real3=Areal_stage2_r(i3+p3);
Ain_im3= Aim_stage2_r(i3+p3);
end
if (i3==2)
Bin_real3=Areal_stage2_r(i3+p3);
Bin_im3= Aim_stage2_r(i3+p3);
end
if (i3==3)
Cin_real3=Areal_stage2_r(i3+p3);
Cin_im3= Aim_stage2_r(i3+p3);
end
if (i3==4)
Din_real3=Areal_stage2_r(i3+p3);
Din_im3= Aim_stage2_r(i3+p3);
end
ool=p3+i3; % test milestone
end

factor1_stage3 = Ain_real3+Bin_im3-Cin_real3-Din_im3;
factor2_stage3 = Ain_real3-Bin_real3+Cin_real3-Din_real3;
factor3_stage3 = Ain_real3-Bin_im3-Cin_real3+Din_im3;
factor4_stage3 = Ain_im3-Bin_real3-Cin_im3+Din_real3;
factor5_stage3 = Ain_im3-Bin_im3+Cin_im3-Din_im3;
factor6_stage3 = Ain_im3+Bin_real3-Cin_im3-Din_real3;
Areal_stage3(p3+1)
=Ain_real3+Bin_real3+Cin_real3+Din_real3;
Aim_stage3(p3+1) =Ain_im3+Bin_im3+Cin_im3+Din_im3;

```

```

        Areal_stage3(p3+2) = double(factor1_stage3*Wn_real(1) +
factor4_stage3*Wn_im(1));
        Aim_stage3(p3+2) = double(factor4_stage3*Wn_real(1) -
factor1_stage3*Wn_im(1));
        Areal_stage3(p3+3) = double(factor2_stage3*Wn_real(1) +
factor5_stage3*Wn_im(1));
        Aim_stage3(p3+3) = double(factor5_stage3*Wn_real(1) -
factor2_stage3*Wn_im(1));
        Areal_stage3(p3+4) = double(factor3_stage3*Wn_real(1) +
factor6_stage3*Wn_im(1));
        Aim_stage3(p3+4) = double(factor6_stage3*Wn_real(1) -
factor3_stage3*Wn_im(1));

end
%-----
% bit reversal of the Radix4
% example 003-> 300

ttt=0;
e2=0;
for v2=1:1:4
for p2=0:4:12
    for i2=0:16:48
        ttt=ttt+1;

        k22(ttt)=v2+p2+i2;
    end
end
end

for i=1:1:64
%if (Aim_stage1(i)<10^(-10))
%    Aim_stage1_r(k22(i))=0;
%else
Aim_stage3_r(k22(i))=Aim_stage3(i);

Areal_stage3_r(k22(i))=Areal_stage3(i);

end
%% -----
figure (1);

X1(1:N)=sqrt(Aim_stage3(1:N).^2+Areal_stage3(1:N).^2);
subplot (8,1,7)
plot (F1,X1, '-x')
grid
ylabel('Magnitude')
title ('Radix 4 FFT N=64 my design (BEFORE REVERSED BIT)');

X1(1:N)=sqrt(Aim_stage3_r(1:N).^2+Areal_stage3_r(1:N).^2);
subplot (8,1,8)

```

```

plot (F1,X1, '-x')
grid
ylabel('Magnitude')
title ('Radix 4 FFT N=64 my design (AFTER REVERSED BIT)');

subplot (8,1,1)
plot ([0:63],Areal_stagel, '-o');
title ('My Matlab FFT design - Radix 4, N=64, in shape geometry,
reversing bit in each stage - The following plot is Stagel real');
axis auto ;

subplot (8,1,2)
plot ([0:63],Aim_stagel, '-o');
title ('Stagel im');
axis auto ;

axis auto ;
subplot (8,1,3)
plot ([0:63],Areal_stage2, '-o');
title ('Stage2 real');
axis auto ;

subplot (8,1,4)
plot ([0:63],Aim_stage2, '-o');
title ('Stage2 im');
axis auto ;

subplot (8,1,5)
plot ([0:63],Areal_stage3, '-o');
title ('Stage3 real');
axis auto ;

subplot (8,1,6)
plot ([0:63],Aim_stage3, '-o');
title ('Stage3 im');
axis auto ;
%-----

figure (2)
X31(1:N)=sqrt(Aim_stagel_r(1:N).^2+Areal_stagel_r(1:N).^2);
subplot (3,1,1)
plot (F1,X31, '-x')
ylabel('Magnitude');
grid
title ('The magnitude result of each stage of my FFT desing in Matlab-
This plot is for Magnitude of stagel');

X32(1:N)=sqrt(Aim_stage2(1:N).^2+Areal_stage2(1:N).^2);
subplot (3,1,2)
plot (F1,X32, '-x')
grid
ylabel('Magnitude')
title ('Magnitude of stage2');

```

```

X33(1:N)=sqrt(Aim_stage3(1:N).^2+Areal_stage3(1:N).^2);
subplot (3,1,3)
plot (F1,X33, '-x')
grid
ylabel('Magnitude')
title ('Magnitude of stage3');

%% saving the choosen equation into binary file named
Input_from_Matlab_real.in & Input_from_Matlab_image.in

%-----
% decimal to binary
% Accepts decimal numbers between -1 < Number < 1
%and transform them into (2bit integer and 30 bit floating)
% equationA is the input decimal number variable.
% pinakas is the output binary number
% handles 64 decimal number- for the 64 point data of FFT real and
imagery

for number3=1:1:2
    if (number3==1)
        Realoo(1:64)=sample_signal_image(1:64);
    else
        Realoo(1:64)=sample_signal_real(1:64);
    end

for ax=1:1:64
    if (Realoo(ax)==1)
        pinakas{ax,1}=0;
        pinakas{ax,2}=1;
        for tr=3:1:64
            pinakas{ax,tr}=0;
        end
    end
    if (Realoo(ax)==0)
        for i=1:1:64
            pinakas{ax,i}=0;
            telos=1;
        end
    end
    if (Realoo(ax)==-1)
        pinakas{ax,1}=1;
        pinakas{ax,2}=1;
        for tr=3:1:64
            pinakas{ax,tr}=0;
        end
    end
    if (Realoo(ax)<1)&(Realoo(ax)>0)
        pinakas{ax,1}=0;
        pinakas{ax,2}=0;
        zhtoymeno5=Realoo(ax);
    end
end

```

```

    for tr=3:1:64
        zhtoymeno5=zhtoymeno5*2;
        if (zhtoymeno5>=1)
            zhtoymeno5=zhtoymeno5-1;
            pinakas{ax,tr}=1;
        else
            pinakas{ax,tr}=0;
        end
    end
end

end
if (Realoo(ax)<0)&(Realoo(ax)>-1)
    pinakas{ax,1}=1;
    pinakas{ax,2}=1;
    zhtoymeno5=1+Realoo(ax);
    for tr=3:1:64
        zhtoymeno5=zhtoymeno5*2;
        if (zhtoymeno5>=1)
            zhtoymeno5=zhtoymeno5-1;
            pinakas{ax,tr}=1;
        else
            pinakas{ax,tr}=0;
        end
    end
end

end
end
Decimal_to_binary=[];
for i=1:1:64
    for e=1:1:32
        Decimal_to_binary=[Decimal_to_binary pinakas{i,e}]; %concatenate
    end

    if (number3==1) binary_result_to_Verilog_image = Decimal_to_binary ;
    % contains the decimal result of our Verilog Design (image)
    else binary_result_to_Verilog_real = Decimal_to_binary ; % contains
the decimal result of our Verilog Design (real)
    end

    end
end
% it stores the 32bit number in line without any ; or any other
distinction
fid22 = fopen('Input_from_Matlab_image.in','w');
fprintf(fid22,'%u',binary_result_to_Verilog_image);
fclose(fid22)
fid122 = fopen('Input_from_Matlab_real.in','w');
fprintf(fid122,'%u',binary_result_to_Verilog_real);
fclose(fid122)
%% Converting binary results from verilog modelsim files into decimal
[decimal_result_of_Verilog_real,decimal_result_of_Verilog_image]=binary
_to_decimal('TB_data_file_Output_image.out','TB_data_file_Output_real.o
ut',2^9);

```

```

[decimal_result_of_Verilog_real_stage1,decimal_result_of_Verilog_image_
stage1]=binary_to_decimal('TB_data_file_stage1_image.out','TB_data_file
_stage1_real.out',2^3);
[decimal_result_of_Verilog_real_stage2,decimal_result_of_Verilog_image_
stage2]=binary_to_decimal('TB_data_file_stage2_image.out','TB_data_file
_stage2_real.out',2^6);
%% figure that contains the fft and my Verilog fft
figure (5);
as(1:64)=0.0001;
Xin_real=(sample_signal_real);
subplot (4,1,1)
plot (F1,Xin_real,'-o')
title ('Matlabs Input Function Real');
axis auto ;

Xin_image=(sample_signal_image);
subplot (4,1,2)
plot (F1,Xin_image,'-o')
title ('Matlabs Input Function Image');
axis auto ;
X2=abs(fft(sample_signal_real+j*sample_signal_image,N));
subplot (4,1,3)
plot (F1,X2,'-x')
title ('FFT N=64 - Matlabs Ready Function');
axis auto ;

Result_Verilog(1:N)=double
(sqrt(decimal_result_of_Verilog_real(1:N).^2+decimal_result_of_Verilog_
image(1:N).^2));

subplot (4,1,4)
plot (F1,Result_Verilog,'-x')
grid
title ('Radix 4 FFT N=64 VERILOG output before the proper reverse of
the final stage - As it comes from the file');
axis auto;
%-----
%% Helpfull reversing of verilog files stages

%-----
% bit reversal of the Radix4
% example 003-> 300

ttt=0;
e2=0;
for v2=1:1:4
for p2=0:4:12
for i2=0:16:48
ttt=ttt+1;

k22(ttt)=v2+p2+i2;
end
end
end

```

```

end
for i=1:1:64
%if (Aim_stagel(i)<10^(-10))
%   Aim_stagel_r(k22(i))=0;
%else
koykoyi(k22(i))=decimal_result_of_Verilog_image(i);

koykoyr(k22(i))=decimal_result_of_Verilog_real(i);
end

eel(1:64,3)=koykoyr+j*koykoyi;%metalaksh
FFT_Verilog_Final_Magn(1:N)= sqrt(koykoyr(1:N).^2+koykoyi(1:N).^2);
FFT_Verilog_Final_Angle(1:N)=unwrap(angle(koykoyr+j*koykoyi));

figure (12)
plot (F1,FFT_Verilog_Final_Magn, '-x');
title ('My Verilog FFT Stage Final Output');
ylabel('Magnititude');
axis auto ;
%% Helpfull reversing of verilog files stage 2

tttl=0;
e21=0;

for p21=0:16:48
    for v21=1:1:4
        for i21=0:4:12
            tttl=tttl+1;

            k221(tttl)=v21+p21+i21;
        end
    end
end
end

for i1=1:1:64
%if (Aim_stagel(i)<10^(-10))
%   Aim_stagel_r(k22(i))=0;
%else
Aim_stage2_rr(k221(i1))=Aim_stage2(i1);
Areal_stage2_rr(k221(i1))=Areal_stage2(i1);
end
%% The magnitude result of each stage of my FFT desing in Matlab
%%compared to the magnitude result of each stage from Verilog(after
being
%radiated) (not be reversed)

for qw=1:1:64
decimal_result_of_Verilog_image_stagel_r(k22(qw))=decimal_result_of_Verilog_image_stagel((qw));
decimal_result_of_Verilog_real_stagel_r(k22(qw))=decimal_result_of_Verilog_real_stagel((qw));
end
%for qw=1:1:64
%decimal_result_of_Verilog_image_stage2_r((qw))=Aim_stage2(k22(qw));

```

```

%decimal_result_of_Verilog_real_stage2_r((qw))=Areal_stage2(k22(qw));
%end

%for qw=1:1:64
%decimal_result_of_Verilog_image_r((qw))=Aim_stage3(k22(qw));
%decimal_result_of_Verilog_real_r((qw))=Areal_stage3(k22(qw));
%end
figure (12)
X31(1:N)=sqrt(Aim_stagel_rr(1:N).^2+Areal_stagel_rr(1:N).^2);
AA21(1:N)=sqrt(decimal_result_of_Verilog_real_stagel_r(1:N).^2+decimal_
result_of_Verilog_image_stagel_r(1:N).^2);
subplot (3,1,1)
plot (F1,X31, '-x', F1,AA21, 'ro')
ylabel('Magnitude');
grid
title ('The magnitude result of each stage of my FFT desing in Matlab
compared to the magnitude result of each stage from Verilog(after being
radiated)(not reversed)- This plot is for Magnitude of stagel');
legend ('FFT expected value', 'FFT Radiated Value');

X32(1:N)=sqrt(Aim_stage2_rr(1:N).^2+Areal_stage2_rr(1:N).^2);
AA22(1:N)=sqrt(decimal_result_of_Verilog_real_stage2(1:N).^2+decimal_re
sult_of_Verilog_image_stage2(1:N).^2);
subplot (3,1,2)
plot (F1,X32, '-x', F1,AA22, 'ro')
grid
ylabel('Magnitude')
title ('Magnitude of stage2');

X33(1:N)=sqrt(Aim_stage3(1:N).^2+Areal_stage3(1:N).^2);
AA23(1:N)=sqrt(decimal_result_of_Verilog_image(1:N).^2+decimal_result_o
f_Verilog_real(1:N).^2);
subplot (3,1,3)
plot (F1,X33, '-x', F1,AA23, 'ro')
grid
ylabel('Magnitude')
title ('Magnitude of stage3');

figure (14)
%%colordef black
difference_stagel_values=X31(1:N)-AA21(1:N);
subplot (3,1,1)
plot (F1,difference_stagel_values, '-x')
hold on
plot (F1,2^(-24.5), '+r', F1,-2^(-24.5), '+r')
hold off
ylabel('Difference in stagel values of figure 12');
grid
legend ('Difference', 'Non Radiated Expected Limits');
difference_stage2_values=X32(1:N)-AA22(1:N);
subplot (3,1,2)
plot (F1,difference_stage2_values, '-x')
hold on
plot (F1,2^(-21.5), '+r', F1,-2^(-21.5), '+r')
hold off

```

```

grid
ylabel('Difference in stage2 values of figure 12');
difference_stage3_values=X33(1:N)-AA23(1:N);
subplot (3,1,3)
plot (F1,difference_stage3_values, '-x')
hold on
plot (F1,2^(-18.5), '+r',F1,-2^(-18.5), '+r')
hold off
grid
ylabel('Difference in stage3 values of figure 12');

figure (15)
%%colordef black
difference_stagel_values=X31(1:N)-AA21(1:N);
subplot (3,1,1)
plot (F1,difference_stagel_values, '-x')
hold on
plot (F1,2^(-6.5), '+r',F1,-2^(-6.5), '+r')
hold off
ylabel('Difference in stagel values of figure 12');
grid
legend ('Difference', 'Radiated Expected Limits');
difference_stage2_values=X32(1:N)-AA22(1:N);
subplot (3,1,2)
plot (F1,difference_stage2_values, '-x')
hold on
plot (F1,2^(-3.5), '+r',F1,-2^(-3.5), '+r')
hold off
grid
ylabel('Difference in stage2 values of figure 12');
difference_stage3_values=X33(1:N)-AA23(1:N);
subplot (3,1,3)
plot (F1,difference_stage3_values, '-x')
hold on
plot (F1,2^(-.5), '+r',F1,-2^(-.5), '+r')
hold off
grid
ylabel('Difference in stage3 values of figure 12');
%-----
%% Checking the Phase
my_Matlab_fft(1:N)=sqrt(Aim_stage3_r(1:N).^2+Areal_stage3_r(1:N).^2);
%metalaksh
Angle_of_my_Matlab_fft=
unwrap(angle(Areal_stage3_r+j*Aim_stage3_r));%metalaksh
fft_build_in_matlab=abs(fft(sample_signal_real+j*sample_signal_image,N)
);
Angle_of_build_in_fft=
unwrap(angle(fft(sample_signal_real+j*sample_signal_image,N)));
eel(1:64,1)=fft(sample_signal_real(1:64)+j*sample_signal_image(1:64));
eel(1:64,2)=Areal_stage3_r+j*Aim_stage3_r;%metalaksh
pinakasA1=real (eel(:,1));
pinakasA2=imag (eel(:,1));
pinakasB1=real (eel(:,3));
pinakasB2=imag (eel(:,3));
Check_phasel=abs(pinakasA1-pinakasB1);

```

```

Check_phase2=abs(pinakasA2-pinakasB2);
figure(9)
plot (real(eel(:,1)),[1:64], 'xr',real(eel(:,3)),[1:64], 'ob');
title ('Real results Difference');
figure (10)
plot (imag(eel(:,1)),[1:64], 'xr',imag(eel(:,3)),[1:64], 'ob');
title ('Image results Difference');

figure (3)
subplot (6,1,1)
plot (F1,FFT_Verilog_Final_Magn, '-x');
title ('Radix 4 FFT N=64 - My Verilog FFT');
ylabel('Magnititude');
axis auto ;
subplot (6,1,2)
plot (F1,my_Matlab_fft, '-x')
grid
title ('Radix 4 FFT N=64 my Matlab design');
ylabel('Magnititude')
subplot (6,1,4)
plot (F1,FFT_Verilog_Final_Angle, '-x');
title ('Angle of the my Verilog fft ');
ylabel('Phase')
subplot (6,1,5)
plot (F1,Angle_of_my_Matlab_fft, '-x');
title ('Angle of my Matlab fft');
ylabel('Phase')
subplot (6,1,3)
plot (F1,fft_build_in_matlab, '-x');
title ('FFT N=64 - Matlabs Build In Function');
ylabel('Magnititude');
axis auto ;
subplot (6,1,6)
plot (F1,Angle_of_build_in_fft, '-x');
title ('Angle of the build in fft ');
ylabel('Phase')
%-----
%comparing My Matlab Function FFT with FFT Verilog (comparing
%the magnitude and the phase)
figure (4)
for i=1:1:64
    if (my_Matlab_fft<10^12)
        end
end
Amagn=abs(my_Matlab_fft-FFT_Verilog_Final_Magn);
Aphase=abs(Angle_of_my_Matlab_fft-FFT_Verilog_Final_Angle);
if (single(Check_phase1)<=(2^-1))&(single(Check_phase2)<=(2^-
1))%metalaksh
    Phase_and_Magnitude_Checking='Correct !!!'
else
    Phase_and_Magnitude_Checking='Failed !!!'
end

```

```

plot (F1,Amagn, '+g',F1,Aphase, 'or');
title ('Difference in Magnitude or in Phase between build in FFT and my
Matlab FFT ');
axis auto ;
%axis ([0 1 0 (10^(-8))]);%metalaksh
xlabel('N=64')
ylabel('Absolute value')
legend('Magnitude Difference', 'Phase Difference');
figure (13)
subplot(3,2,1),plot (F1,Xin_real, '-o')
title ('Matlabs Input Function Real');
axis auto ;
subplot (3,2,2),plot (F1,Xin_image, '-o')
title ('Matlabs Input Function Image');
axis auto ;
subplot (3,2,[5 6])
difference_in_output_a=fft_build_in_matlab-FFT_Verilog_Final_Magn;
plot (F1,difference_in_output_a, 'xr');
ylabel('Detailed Difference in Magnitude of the Output')

subplot (3,2,[3 4])
plot (F1,fft_build_in_matlab, '--gs',F1,FFT_Verilog_Final_Magn, '---+');
ylabel('Magnitude of the Output')
xlabel('N=64 samples')
legend ('fft build in matlab', 'FFT Verilog Final Magn (Post-Route
Simulation)');
hold on
for i=1:1:64
    if (abs(difference_in_output_a(i))>3.8*10^(-6))
        plot ((i-1)/64,FFT_Verilog_Final_Magn(i), 'ro')
        legend ('fft build in matlab', 'FFT Verilog Final Magn (Post-
Route Simulation)', 'Identified Errors');
    end
end
hold off

Real_diff=Check_phase1(1);
Image_diff=Check_phase2(i);
for i=1:1:64
    if (Real_diff<Check_phase1(i))
        Real_diff=Check_phase1(i);
    end
    if (Image_diff<Check_phase2(i))
        Image_diff=Check_phase2(i);
    end

end
if (Real_diff>Image_diff)
    Major_diff=Real_diff;
else
    Major_diff=Image_diff;
end

```

```

if (single(Major_diff)<=(2^(-1)))
    title ('FFT Comparing Check =Correct! Your Verilog FFT works at 167
MHz');
else
    title ('FFT Comparing Check =Wrong! Your Verilog FFT does not
work. ');
end
Major_diff

%-----
%-----
% binary to decimal
% Accepts binary numbers between 01.000..00 - 11.111...11
%(2bit integer and 30 bit floating)
% ap is the input binary number variable.(be careful, there is no
decimal
% point in the input binary numbers!
% result1 is the output decimal number

% the following part loads the 32bit number from Verilog that are in
line without any ;
% or any other distinction (only a space between the numbers)
% "transformed_bin_to_dec" is a 32bit 64 cell matrix that contains the
binary
% output of verilog simulation file
function
[decimal_result_of_Verilog_real,decimal_result_of_Verilog_image]=binary
_to_decimal(TB_data_file_Output_image,TB_data_file_Output_real,multipl)

btd = fopen(TB_data_file_Output_image,'r');
Binary_to_decimal_image=fscanf(btd,'%c');
%Binary_to_decimal=fread(btd, '*uint');%[32, 32], '*uint');
%a33=Binary_to_decimal.';
fclose(btd)
for i=1:1:64
%for eqq=0:1:31
    [token, Binary_to_decimal_image] = strtok(Binary_to_decimal_image);
    %transformed_bin_to_dec{i,1:32}=sscanf (token,'%c');
    transformed_bin_to_dec_image{i}=token;
    %Binary_to_decimal{1,1:32};
    % transformed_bin_to_dec(i)=Binary_to_decimal(1+31*eqq:32+31*eqq);
%end
end

btd1 = fopen(TB_data_file_Output_real,'r');
Binary_to_decimal_real=fscanf(btd1,'%c');

fclose(btd1)
for i=1:1:64
%for eqq=0:1:31
    [token, Binary_to_decimal_real] = strtok(Binary_to_decimal_real);
    %transformed_bin_to_dec{i,1:32}=sscanf (token,'%c');

```

```

    transformed_bin_to_dec_real{i}=token;
    %Binary_to_decimal{1,1:32};
    % transformed_bin_to_dec(i)=Binary_to_decimal(1+31*eqq:32+31*eqq);
%end
end
%-----

for number2=1:1:2
for oeo=1:1:64
    if (number2==1)
        ap=transformed_bin_to_dec_image{oeo};
    else
        ap=transformed_bin_to_dec_real{oeo};
    end

%ap='00111000000000000000000000000000';
%ap='00000010001000011001011001010010';
%bin2dec (ap{3:32})/2^30
zhtoymenol=1;
if (ap(1:2)=='01')
    result1=1;
end

if (ap(1:2)=='00')
    result1=0;
    for e1=3:1:32
        if (ap(e1)=='1')
            num1=1;
        else
            num1=0;
        end

        zhtoymenol=(zhtoymenol)/2;
        result1=num1*zhtoymenol+result1;
    end
end

if (ap(1:2)=='11')
    result1=-1;
    for e1=3:1:32
        if (ap(e1)=='1')
            num1=1;
        else
            num1=0;
        end
        zhtoymenol=(zhtoymenol)/2;
        result1=result1+num1*zhtoymenol;
    end
end
end
% in the following calculation I multiply the final result by 2^9 due
to
% 3bit step down on each stage of the FFT =>(2^3)*(2^3)*(2^3)=2^9

```

```
    if (number2==1) decimal_result_of_Verilog_image (oeo)= result1*multipl
; % contains the decimal result of our Verilog Design (image)
    else decimal_result_of_Verilog_real (oeo)= result1*multipl ; %
contains the decimal result of our Verilog Design (real)
    end

    end

end
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [1] Joshua D. Snodgrass, “Low-Power Tolerance for Spacecraft FPGA-based Numerical Computing”, Dissertation, Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, California, September 2006.
- [2] L. D. Edmonds, “An Introduction to Space Radiation Effects on Microelectronics” JPL publication 00-06, Jet Propulsion Laboratory, California Institute of Technology Pasadena, California, May 2000.
- [3] Herschel Loomis and Alan Ross, “Configurable Fault-Tolerant Architectures for Reliable Space-Based Computing (CFTP),” Naval Postgraduate School Research Proposal to the Secretary of the Air Force, March 2010.
- [4] Margaret A. Sullivan, “Reduced Precision Redundancy applied to arithmetic operations in field Programmable Gate Arrays for satellite control and sensor systems”, Master’s Thesis, Department of Electrical and Computer Engineering, Department of Mechanical and Astronautical Engineering, Naval Postgraduate School, Monterey, California, December 2008.
- [5] Nikolaos Gkikas, “Architecture for the Creation in Software, FFT /IFFT for Wireless Networks”, Master’s Thesis, Department of Physics, University of Ioannina, Patra, Greece, December 2005.
- [6] J. W. Cooley and J.W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series”, Math of Comp., Vol.19, No.90, pp 297-301, April 1965.
- [7] Charles Wu, “Implementing the Radix-4 Decimation in Frequency (DIF) Fast Fourier Transform (FFT) Algorithm Using a TMS320C80 DSP”, Application report: SPRA152, SC Sales & Marketing – TI Taiwan, Digital signal Processing Solutions, January 1998, Texas Instruments.
- [8] Marianne Delphin, “Implementing the Radix-4 FFT Algorithm Using the ST120 DSP”, application note AN1381, October 2001, STMicroelectronics.
- [9] Zhijan Sun, Xuemei Liu and Zhongxing Ji, “The Design of Radix-4 FFT by FPGA”, College of Science, Qingdao Technological University , Qingdao, Shandong, 266033, China.
- [10] Saad Bouguezel, M. Omair Ahmad, “Improved Radix-4 and Radix-8 FFT Algorithm”, Department of Electrical and Computer Engineering Concordia University.

- [11] Chu Chao, Zhang Qin, Xie Yingke and Han Chengde, “Design of a High Performance FFT Processor Based on FPGA.”
- [12] John G. Proakis, Dimitris G. Manolakis, “Digital Signal Processing Principles, Algorithms and Applications”, Third Edition, Prentice-Hall International, INC, 1996.
- [13] Douglas F. Elliot, “Handbook of Digital Signal Processing Engineering Applications”, Rockwell International Corporation, Anaheim, California, Academic Press, INC.
- [14] Siva Kumar Palaniappan & Tun Zainal Azni Zulkifli, “Design of 16-point Radix-4 Fast Fourier Transform in 0.18 μ m CMOS Technology”, American Journal of Applied Science 4(8), pp. 570-575, ISSN 1546–9239, 2007.
- [15] Douglas Lee Jones, “Decimation-in-Frequency (DIF) Radix-2 FFT”, <http://cnx.org/content/m12018/latest/>. (Accessed May 20, 2010).
- [16] CMLab, DSP Research Group, Taiwan, “Fast Fourier Transform,” <http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html>. (Accessed May 20, 2010).
- [17] Vijay Madisetti, Douglas Bennett Williams, “The Digital Signal Processing Handbook”, IEEE Press, CRC Press LLC, 1997.
- [18] Raymond F. Bernstein, Jr. “A Pipelined Vector Processor and Memory Architecture for Cyclostationary Processing.”, Dissertation, Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, California, December 1995.
- [19] Herschel H. Loomis, Jr., Notes from EC4830, Naval Postgraduate School, Monterey, CA, Week1, March 2, 2009, (unpublished).
- [20] Yao-Ting Cheng, “Autoscaling Radix-4 FFT for TMS320C6000”, application report SPRA654, March 2000, Texas Instrument.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Herschel H. Loomis, Jr.
Naval Postgraduate School
Monterey, California
4. Alan A. Ross
Naval Postgraduate School
Monterey, California