



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**REDUCED PRECISION REDUNDANCY APPLIED TO
ARITHMETIC OPERATIONS IN FIELD
PROGRAMMABLE GATE ARRAYS FOR SATELLITE
CONTROL AND SENSOR SYSTEMS**

by

Margaret A. Sullivan

December 2008

Thesis Co-Advisors:

Brij Agrawal

Herschel H. Loomis, Jr.

Second Reader:

Alan A. 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 December 2008	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE Reduced Precision Redundancy Applied to Arithmetic Operations in field Programmable Gate Arrays for Satellite Control and Sensor Systems			5. FUNDING NUMBERS	
6. AUTHOR(S) Margaret A. Sullivan				
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.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) This thesis examines two problems in on-board computing for space vehicles and develops rules for applying Reduced Precision Redundancy (RPR) as a new method of fault tolerance in Field Programmable Gate Arrays against Single Event Effects due to radiation on orbit. RPR was discovered by Snodgrass in 2006 and was first demonstrated using the single-input CORDIC algorithm. This research applies RPR to elementary multiple-input arithmetic operations (addition, subtraction, multiplication, division) and extends applications to multi-level combinations of these operations as they appear in spacecraft subsystems, specifically communication and attitude determination and control. Further modeling and simulation work explores the impact of varying levels of reduction in precision on the performance of communication and control systems using RPR. Finally, a higher-fidelity dynamics model and control system are developed for the NPS Bifocal Relay Mirror Spacecraft simulator, and potential application points for selective redundancy using RPR are identified.				
14. SUBJECT TERMS Reduced Precision Redundancy, RPR, Fault Tolerance, SEU, SEE, FPGA, Reprogrammable Computers, On-Board Processing, Radiation Effects, Software Defined Radio, DFT, FFT, Butterfly Operator, Satellite Attitude Control, ADCS, Flexible Structure, LQR, liner-quadratic-Gaussian, Bifocal Relay Mirror Satellite, BRMS			15. NUMBER OF PAGES 175	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**REDUCED PRECISION REDUNDANCY APPLIED TO ARITHMETIC
OPERATIONS IN FIELD PROGRAMMABLE GATE ARRAYS FOR
SATELLITE CONTROL AND SENSOR SYSTEMS**

Margaret A. Sullivan
Captain, United States Air Force
B.S., Massachusetts Institute of Technology, 2003

Submitted in partial fulfillment of the
requirements for the degrees of

**MASTER OF SCIENCE IN ASTRONAUTICAL ENGINEERING
and
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING**

from the

**NAVAL POSTGRADUATE SCHOOL
December 2008**

Author: Margaret A. Sullivan

Approved by:

Brij Agrawal
Thesis Co-Advisor

Herschel H. Loomis, Jr.
Thesis Co-Advisor

Knox T. Millsaps
Chairman, Department of
Mechanical and Astronautical
Engineering

Alan A. Ross
Second Reader

Jeffrey B. Knorr
Chairman, Department of
Electrical and Computer
Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis examines two problems in on-board computing for space vehicles and develops rules for applying Reduced Precision Redundancy (RPR) as a new method of fault tolerance in Field Programmable Gate Arrays against Single Event Effects due to radiation on orbit. RPR was discovered by Snodgrass in 2006 and was first demonstrated using the single-input CORDIC algorithm. This research applies RPR to elementary multiple-input arithmetic operations (addition, subtraction, multiplication, division) and extends applications to multi-level combinations of these operations as they appear in spacecraft subsystems, specifically communication and attitude determination and control. Further modeling and simulation work explores the impact of varying levels of reduction in precision on the performance of communication and control systems using RPR. Finally, a higher-fidelity dynamics model and control system are developed for the NPS Bifocal Relay Mirror Spacecraft simulator, and potential application points for selective redundancy using RPR are identified.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	OBJECTIVE	2
B.	BACKGROUND	2
	1. Flexible Computing in Space Applications.....	2
	2. Reprogrammable Computers in the Space Environment.....	3
	3. Fault Tolerance Methods	4
	a. Error Correction Coding	4
	b. Triple Modular Redundancy	5
	c. Reduced Precision Redundancy.....	6
	4. The Configurable Fault Tolerant Processor.....	6
C.	ORGANIZATION OF THIS THESIS.....	7
II.	PROBLEM DISCUSSION.....	9
A.	RECOGNIZING A SUITABLE OPERATION.....	9
B.	SPACECRAFT ATTITUDE DETERMINATION AND CONTROL.....	10
	1. Purpose and Requirements of a Spacecraft Attitude Determination and Control System (ADCS)	10
	2. General ADCS Overview	12
	3. Example Control Algorithm: Proportional-Derivative Control....	14
C.	SPACECRAFT SIGNAL PROCESSING: SOFTWARE-DEFINED RADIOS.....	17
	1. Purpose and Requirements of a Spacecraft Signal Processor	17
	2. General SDR Overview	18
	3. Example SDR Function: Fast Fourier Transform (FFT)	19
D.	COMMON ELEMENTARY OPERATIONS.....	24
III.	REDUCED-PRECISION REDUNDANCY FOR COMMON ELEMENTS	27
A.	ASSUMPTIONS, TERMINOLOGY AND GENERAL RULES.....	27
B.	ADDITION AND SUBTRACTION	30
	1. Computation Error in Addition and Subtraction.....	30
	2. Upper and Lower Bound Determination for Addition and Subtraction	31
	a. Special Cases Where $ a = b $	35
	b. Special Cases Where Precise and Bound Values Are the Same	35
	c. Overflow Cases.....	36
	d. Special Cases Involving Zero	37
	3. Demonstration of RPR in Addition and Subtraction	40
	4. Comparing RPR and TMR Implementations	45
C.	MULTIPLICATION	47
	1. Computation Error in Multiplication	47
	2. Upper and Lower Bound Determination for Multiplication	49
	a. Special Cases Involving Zero	51

	b.	<i>Special Cases Involving Small Numbers</i>	51
	3.	Demonstration of RPR in Multiplication.....	53
	4.	Comparing RPR and TMR Implementations	58
D.		DIVISION.....	60
	1.	Computation Error in Division.....	60
	2.	Determining RPR Bounds for Different Division Implementations.....	61
	a.	<i>Convergence by Repeated Multiplication</i>	61
	b.	<i>Division by Reciprocation</i>	62
	3.	Comparing RPR and TMR Implementations	63
E.		COMPOUND OPERATIONS	64
	1.	Matrix multiplication.....	67
	a.	<i>Determining Bounds for Matrix Multiplication</i>	67
	b.	<i>Comparing RPR and TMR Implementations</i>	68
	2.	Discrete Fourier Transform and Fast Fourier Transform	69
	a.	<i>Determining Bounds for a DFT</i>	69
	b.	<i>Determining Bounds for an FFT Butterfly Operator</i>	70
	c.	<i>Comparing RPR and TMR Implementations</i>	71
F.		FURTHER DISCUSSION OF ERRORS AND THE RPR VOTER.....	73
	1.	Additional Considerations for RPR Voters.....	73
	2.	Error Detection with RPR.....	74
IV.		EVALUATING RPR PERFORMANCE.....	77
	A.	MODELING ERRORS DUE TO SINGLE EVENT UPSETS	77
	1.	Classes of Errors in FPGAs	77
	2.	Modeling Errors as Noise.....	78
	3.	Experiment Details.....	79
	B.	EVALUATING PERFORMANCE IN SPACECRAFT ADCS.....	80
	C.	EVALUATING PERFORMANCE IN SOFT RADIO SYSTEMS: FFT	90
	D.	GENERAL NOTES ON RPR-PROTECTED SYSTEM PERFORMANCE	94
V.		APPLYING RPR TO A COMPLEX SYSTEM.....	95
	A.	THE NPS SPACECRAFT SIMULATOR.....	95
	B.	DYNAMICS OF THE BRMS SIMULATOR	96
	1.	Current Model: Rigid-Body Dynamics.....	96
	2.	Modeling the Flexible Appendages	98
	C.	CONTROL OF THE BRMS SIMULATOR	106
	1.	State-Space System Representation	106
	2.	Linear-Quadratic-Gaussian Controller	108
	3.	Demonstrating LQG Control of the Flexible System	112
	a.	<i>Rigid-body model with classical control (original system)</i> ..	112
	b.	<i>Flexible structure model with classical control</i>	114
	c.	<i>Rigid-body model with LQR control</i>	115
	d.	<i>Flexible structure with LQR control</i>	117
	e.	<i>Flexible Structure with LQG control</i>	119

f.	Summary.....	120
D.	APPLYING RPR TO THE BRMSS CONTROL SYSTEM.....	121
VI.	CONCLUSIONS AND RECOMMENDATIONS.....	123
A.	SUMMARY	123
B.	CONCLUSIONS	124
C.	RECOMMENDATIONS FOR FUTURE STUDY	126
1.	Investigating Internal vs. External RPR.....	126
2.	Fault Detection and Location Methods.....	126
3.	Standard Degrees of RPR	127
4.	Implementing RPR in Floating Point Representation.....	127
5.	Performance Evaluation Using Hardware	127
6.	Comparing RPR to Other Fault-Tolerance Methods.....	127
APPENDIX A.	RPR MODULE DESIGN	129
A.1	REPRESENTATIVE RPR AND TMR ADDER OPERATION MODULES	129
A.2	REPRESENTATIVE RPR AND TMR ADDER VOTER MODULES ..	131
A.3	REPRESENTATIVE RPR AND TMR MULTIPLIER OPERATION MODULES	133
A.4	REPRESENTATIVE RPR AND TMR MULTIPLIER VOTER MODULES	135
APPENDIX B.	COMPOUND OPERATION BOUND TESTING	137
B.1	TESTING UPPER/LOWER BOUNDS ON MATRIX MULTIPLICATION	137
B.2	CODE TESTING UPPER/LOWER BOUNDS ON FFT OPERATION	138
APPENDIX C.	SYSTEM SIMULATION CODE	141
C.1	ADCS ERROR INJECTION MODEL SCRIPT FOR TESTING	141
C.2	FFT ERROR INJECTION MODEL SCRIPT FOR TESTING	142
APPENDIX D.	BRMSS CODE FOR DYNAMICS MODEL AND CONTROLLER	143
D.1	EFFECT OF APPENDAGES ON SYSTEM MOI (J).....	143
D.2	STATE-SPACE SYSTEM DESCRIPTION/CONTROL USING (J)	144
D.3	SYSTEM AND CONTROLLER A,B,C,D MATRICES (OUTPUT).....	145
	LIST OF REFERENCES	149
	INITIAL DISTRIBUTION LIST.....	153

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. Bitwise Majority Voter With Single Error Correction and Voter Error Detection.	5
Figure 2. Definition of Attitude Angles and Torque Components in Spacecraft Reference Frame.	10
Figure 3. A Typical Attitude Control System.	13
Figure 4. Schematic of Fundamental Control Problems.	14
Figure 5. PD controller in ideal three-axis-stabilized spacecraft ADCS.	16
Figure 6. Extract Position Angles function in ADCS.	16
Figure 7. Model of a Software Radio.	19
Figure 8. Graphical representation of FFT BFM.	20
Figure 9. Flow graph of an Eight-Point DFT using three levels of.	22
Figure 10. Xilinx FFT v4.1 Pipelined, Streaming I/O Architecture.	23
Figure 11. Sixteen Bit Fixed-Point Two's Complement Representation of $(-\pi/4)$	29
Figure 12. Sixteen Bit Fixed-Point Sign-Magnitude Representation of $(-\pi/4)$	29
Figure 13. Number line showing maximum error in one addition operation.	31
Figure 14. Upper and Lower Bounds of Fixed-Point Two's Complement Numbers.	32
Figure 15. Cases 1 through 4 for $c = a + b$ and $d = a - b$	34
Figure 16. Cases 5 through 8 for $c = a + b$ and $d = a - b$	34
Figure 17. RPR Adder Top-Level Block Diagram (OV = overflow).	40
Figure 18. RPR Adder - Operation Block Diagram.	41
Figure 19. TMR Adder - Operation Block Diagram (compare to Figure 18).	42
Figure 20. RPR Adder - Voter Block Diagram.	43
Figure 21. TMR Adder - Voter Block Diagram (compare to Figure 20).	43
Figure 22. Number Line for Multiplication in Fractional Fixed-Point.	47
Figure 23. Upper and Lower Bounds of Fixed-Point Sign-Magnitude Numbers.	49
Figure 24. Erroneous Bounds Due To Rounding Products.	51
Figure 25. RPR Multiplier Top-Level Block Diagram.	53
Figure 26. RPR Multiplier - Operation Block Diagram.	54
Figure 27. TMR Multiplier - Operation Block Diagram (compare to Figure 26).	55
Figure 28. RPR Multiplier - Voter Block Diagram.	56
Figure 29. TMR Multiplier - Voter Block Diagram (compare to Figure 28).	57
Figure 30. Radix-Two FFT Butterfly Operation (from Figure 8).	70
Figure 31. Possible Error Scenarios in RPR.	75
Figure 32. BRMS Simulator System Model.	80
Figure 33. BRMSS Model Control and CMG Allocation with Error Injection.	82
Figure 34. ADCS Reference Maneuver with No Error - Angular Position.	83
Figure 35. ADCS Reference Maneuver with No Error - Commanded Control.	83
Figure 36. Unbounded Transient Error Effect on BRMSS Reference Maneuver.	84
Figure 37. Transient RPR Result Effect on BRMSS Reference Maneuver ($r = 8$).	85
Figure 38. Unbounded Persistent Error Effect on BRMSS Reference Maneuver.	86
Figure 39. Extended Simulation of Unbounded Persistent Error.	87
Figure 40. Persistent RPR Result Effect on BRMSS Reference Maneuver ($r = 8$).	87

Figure 41. Persistent RPR Result Effect on BRMSS Reference Maneuver ($r = 16$).....	88
Figure 42. Small Timestep.....	89
Figure 43. FFT Error Simulation Model.....	90
Figure 44. FFT Constructed With Four Levels of Fixed-Point Complex Butterfly Machines, $N = 16$ (Error Injected at Level 2).....	91
Figure 45. Relative Error in FFT Representative Output for RPR ($N_{FFT} = 8$).....	93
Figure 46. Relative Error in FFT Representative Output for RPR ($N_{FFT} = 16$).....	93
Figure 47. NPS Bifocal Relay Mirror Spacecraft Simulator.	95
Figure 48. BRMS Simulator System Model (Copy of Figure 32).....	96
Figure 49. Rigid-body model of BRMSS, showing principal body axes.	97
Figure 50. BRMSS Flexible Appendage* Model (Not To Scale).	98
Figure 51. BRMSS Appendage Attachment Points and Orientation (Top View).	100
Figure 52. Flexible Appendage – Local Body Axes Redefined for Appendage 3 Only.	102
Figure 53. Block diagram of state-space BRMSS system with LQG controller.	111
Figure 54. Attitude of Main Body (Rigid-Body Model with PD Control).	113
Figure 55. Control history (Rigid-Body Model with PD Control).	113
Figure 56. Attitude Angles (Flexible Structure with PD Control).....	114
Figure 57. Control History (Flexible Structure with PD Control).....	115
Figure 58. Attitude Angles (Rigid-Body Model with LQR Control).	116
Figure 59. Control History (Rigid-Body Model with LQR Control).....	116
Figure 60. Attitude Angles (Flexible Structure with LQR Control).....	118
Figure 61. Control History (Flexible Structure with LQR Control).	118
Figure 62. Attitude Angles (Flexible Structure with LQG Control).....	119
Figure 63. Control History (Flexible Structure with LQG Control).....	120
Figure 64. Updated BRMSS control system.....	122
Figure 65. RPR 8/64 Addition - Operation Module.	129
Figure 66. TMR 64-bit Addition - Operation Module.....	130
Figure 67. RPR 8/64 Addition - Voter Module.	131
Figure 68. TMR 64-bit Addition - Voter Module.....	132
Figure 69. RPR 16/64 Multiplication - Operation Module.....	133
Figure 70. TMR 64-bit Multiplication - Operation Module.	134
Figure 71. RPR 16/64 Multiplication - Voter Module with $2r$ -bit Comparators.....	135
Figure 72. TMR 64-bit Multiplication - Voter Module (Bitwise Majority).	135

LIST OF TABLES

Table 1. Determining Operations Not Suitable for RPR.	24
Table 2. Eight Combinations of Input Properties for Addition and Subtraction.	33
Table 3. Upper and Lower Bounds for 4/6 RPR Showing CO, OV and Modified x_U	39
Table 4. RPR Result Selection Logic.	44
Table 5. Area Required By TMR and Representative RPR Addition Experiments.	45
Table 6. FPGA Area Comparison for RPR and TMR Adders.	46
Table 7. Area Required By TMR and RPR Multiplication Experiments.	58
Table 8. FPGA Area Comparison for RPR and TMR Multipliers.	59
Table 9. Projected FPGA Area Required for Matrix Multiplication.	68
Table 10. Projected FPGA Area Required for Radix-Two FFT Butterfly Operation.	72
Table 11. Area Required By TMR and RPR Multiplication Experiments.	73
Table 12. Error in FFT with No Fault Tolerance and RPR Fault Tolerance.	92
Table 13. BRMSS Control System Simulation Results.	120

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ADC	Analog-to-Digital Converter
AFRL	Air Force Research Laboratory
ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BFM	Butterfly Machine
BRMS	Bifocal Relay Mirror Satellite
BRMSS	BRMS Simulator
CARE	Continuous-time Algebraic Riccati Equation
CFTP	Configurable Fault Tolerant Processor
CM	Center of Mass
CMG	Control Moment Gyroscope
DAC	Digital-to-Analog Converter
DFT	Discrete Fourier Transform
DOD	Department of Defense
DSP	Digital Signal Processor/Processing
ECC	Error Correction/Control Coding
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
LEO	Low-Earth Orbiting
NPS	Naval Postgraduate School
ORS	Operationally Responsive Space
RAM	Random-Access Memory
R&D	Research and Development
RF	Radio Frequency
RPR	Reduced Precision Redundancy
SECDED	Single Error Correcting, Double Error Detecting
SEE	Single Event Effects
SEU	Single Event Upset
SNR	Signal-to-Noise (Power) Ratio
SRAM	Static Random-Access Memory
TAS	Three-Axis-Stabilized
TASS	Three-Axis-Stabilized Simulator
TMR	Triple Modular Redundancy

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to the members past and present of the CFTP team at NPS for bringing me into their world of bitflips and fault tolerance (or “fault ignorance”): Professor Hersch Loomis, Professor Alan Ross, Ron Aikins, Rita Painter, Mindy Surratt, Major Jerry Caldwell, USMC, and Major Josh Snodgrass, USAF, to name a few. In particular, I owe any future success of mine in this area to Professors Loomis and Ross for their guidance, instruction, wisdom, interest, enthusiasm, and unending support as I struggled toward the end product of this thesis and my degree in electrical engineering. Without them, I surely would not have achieved so much at this institution, nor would I be so inspired to continue in this line of work.

I would like to thank Professor Brij Agrawal for providing the chance to work in the Spacecraft Research and Design Center with the world-class resources he has assembled there, and for his direction and advice on the culmination of my work in the astronautical engineering department. I also express my gratitude to Dr. Jae Jun Kim, who spent many patient hours working with me on dynamics and controls.

Finally, I thank my parents, Patrick and Mary, and my sister Lucy for the tremendous appreciation they have instilled in me for the beauty of nature, life and culture as seen through the language of science, mathematics and engineering. Without them I would never have come to learn all of the amazing things about this world that I continue to discover and experience with wonder every day.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The harsh radiation environment of space generates faults in FPGAs that affect both data and configuration memory. The Configurable Fault Tolerant Processor at the Naval Postgraduate School is a platform for testing methods of fault tolerance that guard against the single-event effects of radiation in FPGAs. In 2006 Snodgrass introduced a new method of fault tolerance, Reduced Precision Redundancy (RPR), as a power-saving alternative to traditional Triple Modular Redundancy (TMR). This research focuses on the details of implementing RPR and the effect of RPR fault tolerance on the performance of spacecraft systems.

Two categories of system architectures are discussed: recursive data management, found in feedback control systems; and flow-through data management, found in signal processing tools such as the fast Fourier transform. Examples of the two architectures are broken down into their elementary operations, and the common operations are chosen as the subjects of experiments in RPR implementation. The “degree of RPR” is defined as a measure of reduction in precision. Detailed RPR designs for addition/subtraction and multiplication are programmed, simulated and mapped to the Virtex™ XQVR600 FPGA using the Xilinx Integrated Software Environment. Versions of each operation are built in TMR and several degrees of RPR, and the FPGA resources required for each degree of RPR are compared to the resources used by the corresponding TMR experiments. The results obtained from the detailed designs are extrapolated to estimate the resources required to implement RPR division and the compound operations of matrix multiplication and the fast Fourier transform butterfly machine.

An evaluation of RPR-protected system performance is conducted on models of recursive and flow-through data architecture systems using MATLAB and Simulink computational tools. Transient and persistent errors are modeled as delta and step functions of additive noise in the signal data flow, and RPR error correction is modeled as an increase in signal-to-noise ratio whose magnitude depends on the degree of RPR.

The improvement in system response and reduction in output error between “no fault tolerance” and “RPR fault tolerance” are measured to determine the impact of RPR on system performance.

One example system is also studied further, in order to improve it and assess it as a complicated candidate system for RPR. A new dynamics model is developed for the Bifocal Relay Mirror Satellite Simulator test bed at NPS. It describes the effects of three flexible appendages, which expands the model to a twelve-state system with limited observable output. A linear-quadratic Gaussian controller is developed for the flexible structure model by combining a Kalman filter state estimator with a cost-optimal linear quadratic regulator (LQR). The new dynamics and control system is tested using a reference maneuver, and control cost is compared to the cost of using a simple PD controller. Finally, the enhanced BRMSS model is examined as a candidate system for RPR, and operations suitable for applying RPR are identified.

Through the research presented in this thesis, it is shown that RPR can be implemented for arithmetic operations using standard rules for upper and lower bound determination. It is shown that the FPGA area and power savings gained by using RPR instead of TMR increase with the complexity of the module being protected, and that the complexity of an RPR voter compared to a TMR voter makes RPR application a more efficient choice for large multi-part operations. It is also demonstrated that transient errors are less damaging than persistent errors to systems of recursive and flow-through data architectures. An example system simulation shows that RPR of a degree less than 16/52 provides satisfactory performance in the presence of either transient or persistent errors. It is concluded that the trade space defined by FPGA capacity, speed of operation, and error tolerance requirements must be examined by a system developer in order to determine the optimal level and degree at which to apply RPR fault tolerance.

I. INTRODUCTION

As the electronics of space systems have evolved since the 1960s from masses of analog and digital circuits to principally digital systems, the responsibility placed on computers operating in the harsh radiation environment of space has greatly increased. Most modern spacecraft have passive redundancy in the computer systems that conduct centralized control of the satellite and run communications tasks, attitude control tasks, and often on-board payload data processing tasks. The combination of the inaccessibility of most spacecraft after launch and the mass and power restrictions imposed on spacebound payloads generates a strong argument for choosing reconfigurable digital hardware for some applications in space vehicle computers. Specifically, field programmable gate arrays (FPGA) offer a high degree of flexibility for supporting multiple applications through reprogramming as well as the speed necessary to operate complex architectures in real time [1].

The challenges faced by the developer of an FPGA-based system in space are not trivial. In addition to the damage of long-term radiation exposure, which can be minimized using standard electronics shielding techniques [2], FPGAs are susceptible to errors in both data and architecture configuration caused by single event effects (SEE). Over the past decade, the computer engineering group at the Naval Postgraduate School has been modeling SEE, developing fault tolerance methods for spaceborne reprogrammable computers, and testing them using the FPGA-based Configurable Fault Tolerant Processor experiment. In 2006, a new method of fault tolerance was discovered called Reduced Precision Redundancy (RPR) that offers a trade between precision in calculation and power savings on an FPGA. RPR can potentially reduce power consumption in an FPGA up to 70% over traditional redundant fault tolerance techniques [3]. However, the concept of RPR is in its infancy – the method still needs to be evaluated as applied to operations commonly found in computers on space vehicles.

A. OBJECTIVE

The objective of this thesis is twofold: to define further the implementation of fault tolerance using RPR, and to investigate the effect of using RPR on some major space vehicle processing problems. Algorithms for many common tasks on spacecraft can be broken down into a set of elementary operations; this thesis provides rules and considerations for applying RPR to several elementary operations as well as a model for assessing the impact of error propagation through a system to which RPR has been applied.

B. BACKGROUND

1. Flexible Computing in Space Applications

The recent advent of the Operationally Responsive Space (ORS) concept and Department of Defense (DOD) office of the same name have brought great publicity to the concept of using reprogrammable and otherwise flexible computers and processors in satellite systems. The benefits of having a reprogrammable asset in space are many: transceivers may be updated with new communications or compression algorithms; satellite control may be saved in the event of a primary computer failure (or partial failure); entire new mission applications may be uploaded for experimental purposes.

Reprogrammable hardware options for digital computers include microprocessors, field-programmable gate arrays (FPGA), and digital signal processor (DSP) chips. Each type of hardware is optimally suited to provide a different combination of function, flexibility and processing speed. FPGAs are different from microprocessors and DSPs because the *device configuration* is actually programmable. This gives the FPGA greater operating speed for most tasks than a microprocessor, which must load applications only in software, and far greater flexibility than most DSPs, which have basic functions permanently implemented in hardware for speed.

In general an FPGA will operate more slowly than a custom-designed application-specific integrated circuit (ASIC) that performs the same function, but the benefits of the

after-market *programmability* of an FPGA often make it a more attractive choice. Specifically, an FPGA requires much less development time than an ASIC, enables post-production re-programming to fix bugs or problems discovered after implementation, and incurs lower design and development (non-recurring) costs.

The ability to change or reprogram a system remotely after it has been activated is of particular interest to the space community, where the only unmanned satellite ever visited for repeated post-launch repairs has been the Hubble Space Telescope. The potential insertion points for new or updated algorithms, missions, and applications are numerous – but along with the benefits of reprogrammable computers in space come the limitations and special considerations associated with their unique operating environment.

2. Reprogrammable Computers in the Space Environment

It is a well-known fact in the satellite development and operation community that the space environment is unforgiving of its man-made trespassers [4]. Space vehicles experience erosion, thermal imbalance and other surface damage due to impact from micrometeorites traveling at relative velocities up to 8 km/s (in low-earth orbit). As orbit altitude increases, the percentage of ionized molecules in the particles surrounding the spacecraft also increases. This “plasma environment” causes charge to build up both on the surface of and deep inside spacecraft; the charge accumulation can overcome electric fields or trigger severely damaging arcing across the vehicle. In addition to general charging from the plasma environment, individual heavy ions from the Van Allen belts of the earth, solar events or cosmic radiation cause Single Event Effects (SEE) when they interact with a space vehicle. SEE include permanent hardware failures such as circuit burnout or gate latch-up as well as temporary failures due to single-event upsets (SEU). SEU occur when a highly energetic (MeV or higher) charged particle strikes one or more memory bits in a spacecraft computer, changing the bit’s energy level and thus also changing the value stored in that memory location. This is called a *fault*. Depending on the location and importance of the affected bit, a fault can cause an entire processor to fail temporarily until it can be stopped, its memory cleared, and restarted.

The special quality of an FPGA relative to other electronic devices or processors is that an FPGA can be reconfigured, even after being launched in a spacecraft. RAM-based FPGAs are particularly flexible, since their entire configuration is set in memory. However, this creates a unique vulnerability in RAM-based FPGAs: both the application data *and the configuration* of the device – how it performs its intended mission – are susceptible to SEUs. When a fault occurs, it may not always cause a complete failure of the affected FPGA; it may merely change part of the FPGA function such that it continually generates errors in the output data. In this case, the fault will go undetected unless the configuration of the FPGA is checked periodically for errors.

There are currently several versions of radiation-hardened processors and other components available from commercial electronics developers [5]. All are built to withstand the total radiation dose levels and accumulation of charge experienced over the lifetime of the satellite. However, even radiation-hardened version of FPGAs currently available *do not* inherently guard against SEU. Instead, the algorithms implemented on the FPGAs must have fault tolerance designed into them, using one of several possible approaches.

3. Fault Tolerance Methods

The two most common approaches taken by designers to minimize the impact of single-event radiation effects on FPGAs are error correction coding (ECC) and redundancy – specifically Triple Modular Redundancy (TMR). A new variant on TMR, Reduced Precision Redundancy (RPR), is explored in this research as another viable fault tolerance approach.

a. Error Correction Coding

Error correction coding is applied to binary data in a communication or computer system to protect the integrity of the bitstream as it is moved over some spatial distance or stored for some length of time. Depending on the requirements of a system, error correction codes may correct errors automatically or merely detect them in order to alert an operator that errors have occurred. Practical error correction codes, such as the

TMR is generally applied at the lowest level in a circuit where it can still effectively correct errors [8]. It provides highly reliable error correction for single errors, but the cost of that reliability is a circuit that requires over three times the programmable logic space on an FPGA as that required by the unprotected circuit. This thesis addresses the theory that Reduced Precision Redundancy may alleviate the limitations imposed by the size of a TMR circuit on an FPGA.

*c. **Reduced Precision Redundancy***

The concept of Reduced Precision Redundancy (RPR) allows the sacrifice of some level of precision in calculation, in the event that an error occurs, in return for space and power savings on an FPGA. The theory as developed by Snodgrass [3] at the Naval Postgraduate School (NPS) in 2006 suggests that instead of generating three identical copies of a circuit and voting on the outcome as with TMR, some functions lend themselves to operating in a single thread at full precision, and in two additional threads at reduced precision. The two reduced-precision operations generate an upper and lower bound on the correct function output. The precise calculation result is then compared to the upper and lower bounds, and voting logic determines whether the precise result may be used, or if an error has occurred in the precise solution and the average of the bounds must be used as a less-precise result.

Snodgrass demonstrated the concept of RPR in a COordinate Rotation Digital Computer (CORDIC) algorithm built for the Xilinx Virtex™ XQVR600 devices on the Configurable Fault Tolerant Processor experiment platform at NPS.

4. The Configurable Fault Tolerant Processor

The Configurable Fault Tolerant Processor (CFTP) is a research experiment in the NPS Space Systems Academic Group. It is a Xilinx Virtex™ FPGA-based platform for developing and testing new fault tolerance methods on a spacecraft computer system in both the laboratory and the space environment. On the ground, CFTP has been subjected to controlled radiation experiments using the cyclotron at the Crocker Nuclear Laboratory, University of California-Davis, CA [9]. Detailed descriptions of the CFTP

architecture are available in [10] and [11]; instructions and examples of CFTP experiments can be found in [12].

The fault tolerance method currently used in most CFTP architecture and experiments is TMR [11], [12]. However, future generations of CFTP will incorporate more complicated experiments that, when implemented using TMR, will be limited by the capacity of the FPGA. Implementing new experiments using RPR is one method of alleviating the space limitation of the FPGA – but in order to do this, RPR must be defined and its effects understood completely for a wide range of possible applications. That definition and understanding is the intended goal of this thesis.

C. ORGANIZATION OF THIS THESIS

Chapter II describes the conditions necessary to apply RPR successfully to a problem, and presents two examples of systems that contain processes meeting the necessary criteria. The first example system type, spacecraft attitude determination and control, operates on a matrix of states in a feedback loop. The second example system, software-defined radio, processes data in a flow-through manner. Chapter II concludes with a list of elementary operations common to many processes that have been collected from the example systems and also meet the criteria for good RPR candidates.

Chapter III discusses the common elementary operations in detail and describes how to apply RPR to each one. Included in each discussion are rules governing upper and lower bound selection for the operation and an example implementation in FPGA schematic design. The RPR implementations are compared to corresponding TMR implementations to determine the space and power savings of RPR.

Chapter IV addresses the impact of errors due to single-event effects on the performance of a system using RPR. Performance is evaluated using a set of metrics based on modeling RPR contingency operations as spikes or increases in the system noise level. Both control and soft radio systems are examined via modeling in Simulink®.

Chapter V explores a more complicated practical scenario in spacecraft attitude determination and control: the NPS Bifocal Relay Mirror Satellite (BRMS) Simulator

(BRMSS). The BRMSS dynamics model is improved by adding the effects due to flexible appendages on the BRMSS structure. A more accurate linear-quadratic-Gaussian (LQG) controller, incorporating uncertainty in modeling and measurement due to unobservable states and the effects of additive white Gaussian noise, is developed as an alternative to the current proportional-derivative (PD) controller. Chapter V concludes by demonstrating connections between the more sophisticated BRMSS control system and the elementary operations described in Chapter III, as well as the performance evaluation methods proposed in Chapter IV.

Chapter VI contains a summary, conclusion, and recommendations for future work.

II. PROBLEM DISCUSSION

A. RECOGNIZING A SUITABLE OPERATION

In developing RPR, Snodgrass divides problems into two types: Class A, suitable for RPR implementation; and Class B, not suitable for RPR implementation. The most significant distinction between Class A and Class B problems is in the organization and representation of data [3].

Class A problems generally manipulate numbers or other data represented by blocks of bits ordered in increasing or decreasing importance. In a Class A problem, it is possible to repeat or duplicate processes using only a subset of the data bits – the most important bits – in order to produce a reduced-precision redundant result. Fixed-point numerical problems are a good example of Class A problems, as they have clearly-defined most significant bits (MSB) and least significant bits (LSB) for every item of data.

Class B problems may represent data using single- or multi-valued (Boolean) logic functions where the importance of each data bit is the same. The output of such functions cannot be generated using fewer bits without greatly changing the meaning of the result. Problems also may be categorized as Class B when any less precise representation of the problem is significantly more complicated than the full-precision operation, as this would eliminate the benefits of RPR (less space and power required for reduced-precision calculations). Furthermore, some problems have no solution algorithm that can be executed in multiple ways; the full-precision method may be the only possible method.

This study examines two “real world” applications of reprogrammable computers in space vehicles from the perspective of implementing RPR for each application. The two applications are attitude determination and control, an essential subsystem for most spacecraft; and signal processing, part of the spacecraft’s communications subsystem or payload.

B. SPACECRAFT ATTITUDE DETERMINATION AND CONTROL

1. Purpose and Requirements of a Spacecraft Attitude Determination and Control System (ADCS)

Almost any modern space-based mission requires knowledge and control of the spacecraft's orientation – usually to point an antenna or telescope for transmitting or receiving information [13]. Even if the primary mission of a satellite does not require precise pointing, communication between the space vehicle and its ground control segment often requires some basic attitude control of the vehicle in order to compensate for various disturbance torques that affect spacecraft in orbit [14]. Typical requirements levied on the ADCS of a space vehicle include accuracy and range for both attitude knowledge and attitude control, as well as jitter, drift and settling-time constraints on the control system [14]. A diagram of a notional rigid body with torque components and attitude angles is shown in Figure 2.

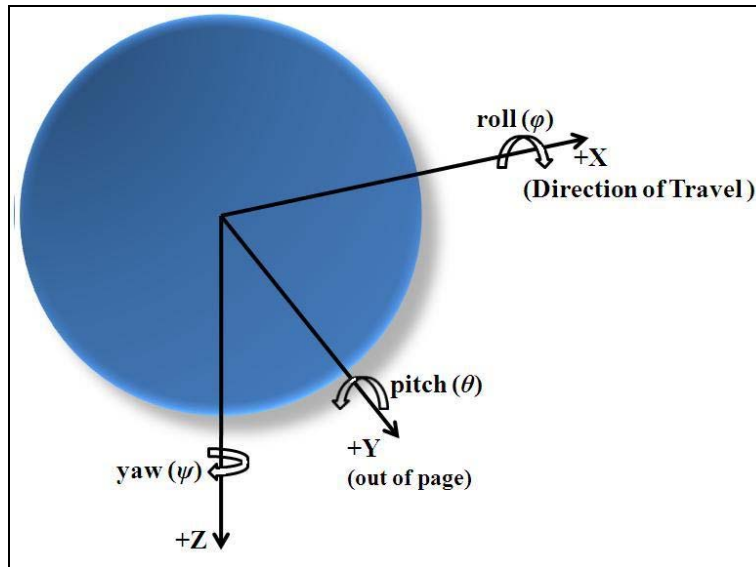


Figure 2. Definition of Attitude Angles and Torque Components in Spacecraft Reference Frame.

During its operational life, a space vehicle may undergo many changes – either in operational mission or in physical form or function. Although a space vehicle is designed with parts made to last beyond its mission lifetime, on some occasions events during

launch or on orbit may cause certain parts to break partially or completely. In many cases, satellites are launched to fulfill a certain mission need, but after some period of operation they are re-tasked to accomplish a different mission instead of or in addition to their original purpose. In cases like these, it is imperative that the space vehicle subsystems be able to adapt to the physical or operational changes to the vehicle. Flexible computers for subsystems such as the ADCS are advantageous in these situations, because they allow changes in not only commanded trajectory (a standard input), but also stored mass and momentum properties of the vehicle, the importance of each sensor, or the capability of each actuator. Furthermore, if the ADCS of a satellite has been implemented on an FPGA, it can be reprogrammed to take advantage of new and more efficient control techniques developed after it has been launched. In addition, if the disturbance environment for a given satellite is more accurately modeled over time, the control algorithm or entire ADCS can be reprogrammed to utilize the knowledge gained from the new models.

A space vehicle is generally subject to two types of external disturbance torques in its orbital environment: cyclic disturbances, which vary periodically as the spacecraft travels around its orbit, and secular disturbances, which are continuously additive and do not cancel themselves out over the course of an orbit [14]. One example of cyclic torque on an earth-oriented vehicle is solar radiation pressure, whose direction is always radially out from the sun (and therefore constant within any given revolution of an earth-orbiting spacecraft). An example of a constant (secular) torque on an earth-oriented vehicle is aerodynamic drag due to the earth's upper atmosphere. This is particularly notable in LEO satellites. In addition to cyclic or secular external torques, some disturbance torques on a spacecraft are internal, e.g., liquid sloshing in fuel tanks [15].

It is important to distinguish between the concept of torque, which acts on only the attitude and orientation of a vehicle, as opposed to a force, which controls the position of the center of mass of a vehicle with respect to an external reference frame. This research examines only the part of a spacecraft ADCS that controls the torques on the orientation of a spacecraft through the storage and use of angular momentum, and does not look at the kinematics of the entire vehicle's trajectory through space.

The effects of any disturbance torques on the dynamics of a space vehicle can be modeled mathematically, as in [15], [16], [17]. These dynamic models are in turn used to build attitude determination and feedback control system models for space vehicles, which often run autonomously on either a dedicated ADCS processor or the main space vehicle computer. Each operation in the ADCS processor must be executed at some minimum level of precision in order to maintain the tolerances required by a given mission for pointing accuracy, jitter control, drift, and settling time.

2. General ADCS Overview

A spacecraft ADCS is typically a feedback control system with two basic components: attitude knowledge or determination, and control. Attitude determination is achieved by processing data from sensors (e.g. earth or sun sensors, magnetometers) and control is executed using actuators. Actuators can be either passive (e.g., gravity booms, magnetic systems) or active (e.g., reaction wheels, momentum wheels, control moment gyroscopes (CMG)).

Regardless of the type of sensors or actuators used, the ADCS processor must manipulate the measurements taken by the sensors, update the state matrix describing the orientation and rate of the vehicle, compute the necessary control to maintain or change the state of the vehicle, and allocate the prescribed control among the actuators. In most space vehicles built today, the ADCS processor is a digital computer that samples sensor values at discrete intervals determined by a system clock. Along with the external and internal disturbance torques on the satellite, additional error and uncertainty is introduced into the control system as noise from sampling and quantizing the attitude determination sensor measurements. A high-level block diagram of a feedback control system is shown in Figure 3, where the Vehicle Dynamics block represents the system being controlled (the plant) and the sensor measurements, control algorithm, and torque generation are all parts of the controller.

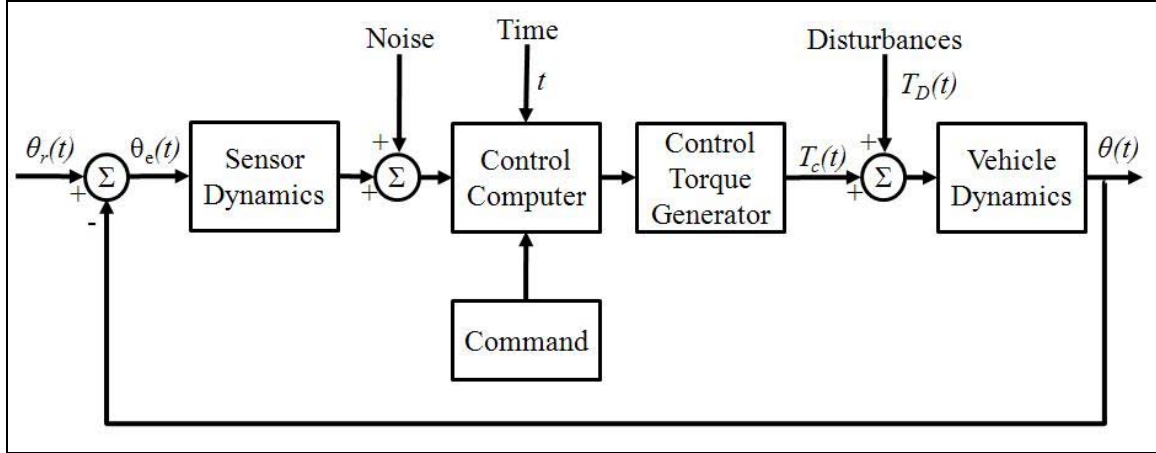


Figure 3. A Typical Attitude Control System (From [14]).

Within the Control Computer block in Figure 3, there are separate processes for estimating the vehicle state based on sensor measurements, running the control algorithm (which operates on the difference between the estimated states and a commanded state), and allocating the resulting control torque appropriately among actuators [18]. This is depicted in Figure 4, with the functions of the control computer enclosed within the dotted line.

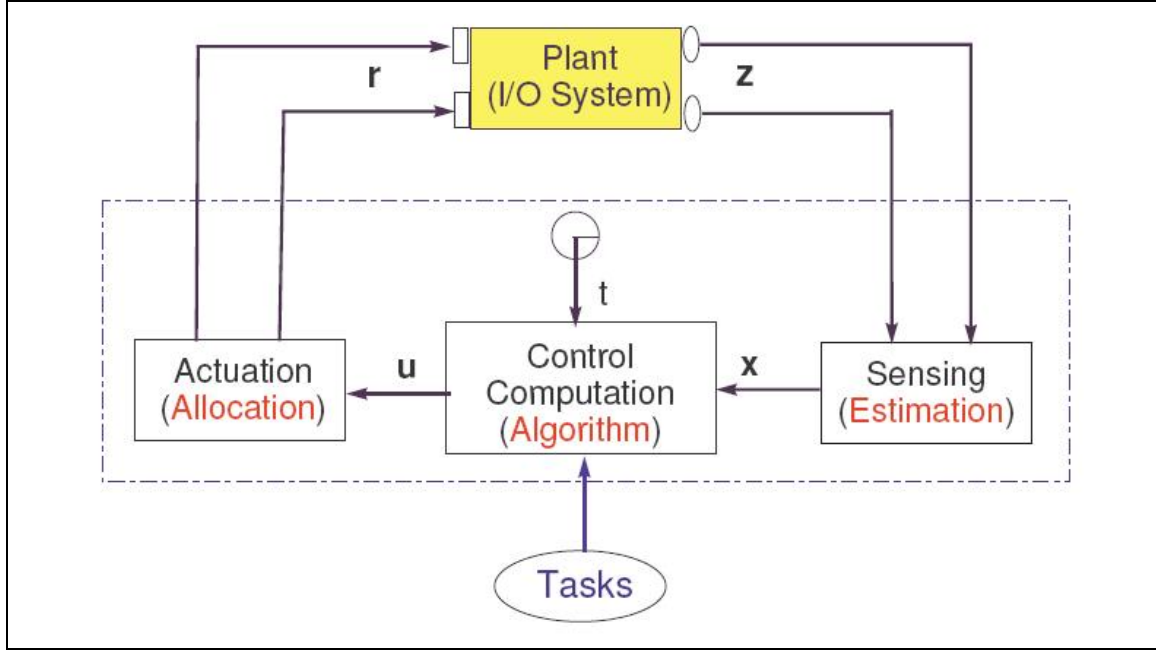


Figure 4. Schematic of Fundamental Control Problems (From [18]).

This research examines only the control algorithm portion of the ADCS as a candidate for RPR. The estimation problem associated with sensing attitude and the allocation problem associated with operating actuators are beyond the scope of the RPR problem at this time.

3. Example Control Algorithm: Proportional-Derivative Control

A basic control algorithm consists of comparing the current state of a system (position and rate) to its desired state, and calculating the correctional force or torque necessary to reduce or eliminate the difference between the actual and desired states. A simple mathematical representation of spacecraft dynamics and control uses linear ordinary differential equations (ODE) to describe the relationships among angular position, rate, and acceleration, and applied torque (from disturbances or the controller) in the roll, pitch, and yaw directions (refer to Figure 2). It is most straightforward to consider the case of pitch-only dynamics and control, as the pitch motion of an earth-orbiting spacecraft can often be decoupled from the roll and yaw motion for small-

disturbance cases. Simple proportional-derivative (PD) control of the pitch motion for a three-axis-stabilized spacecraft can be modeled using a linear ordinary differential equation of motion (EOM):

$$T_D = J_y \ddot{\theta} + K_d \dot{\theta} + K_p \theta \quad (1)$$

where T_D is the disturbance torque, J_y is the principle moment of inertia of the spacecraft about the pitch axis, and K_d and K_p are the gains associated with the PD controller. In the simplest case, the K_d and K_p gains are constants that are initially chosen using a control design approach (e.g., the root locus method) based on the characteristics of the system. The goals of the control system are to achieve stability, and to minimize steady-state error and system settling time.

The pitch control function is nominally implemented in a model of an ideal three-axis-stabilized spacecraft. The model includes the effect of environmental disturbance torques, and takes into account the coordinate transformations necessary to calculate the dynamics of the system (Figure 5). The PD controller itself contains only the simple multiplication and addition operations on the constant gains and state variables in Equation (1). However, if the control is extended to address disturbances in all three directions (roll, pitch and yaw) as well as coupled relationships among the angles, the operations become two-dimensional matrix multiplications. Other operations in the control model include storing and repeatedly accessing several constants or sets of constants (vectors of fixed or floating-point numbers), reshaping and selecting or multiplexing matrix elements, multiplying and dividing direction cosines to extract position angles, and calculating trigonometric functions.

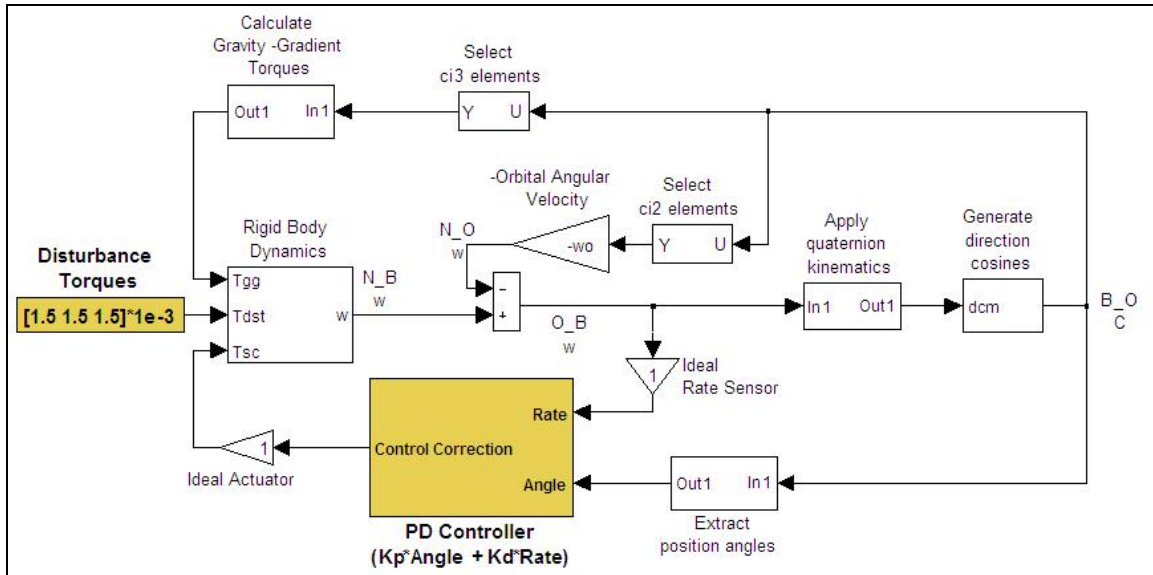


Figure 5. PD controller in ideal three-axis-stabilized spacecraft ADCS (After [19]).

Although most of the blocks in Figure 5 model the dynamics and kinematics of a spacecraft and would not be present in the ADCS processor on the actual platform, the Extract Position Angles function is one example of additional manipulations that need to be performed in the control computer. The detailed operation of this function is shown in Figure 6. Within this subsystem are both elementary and more complicated operations.

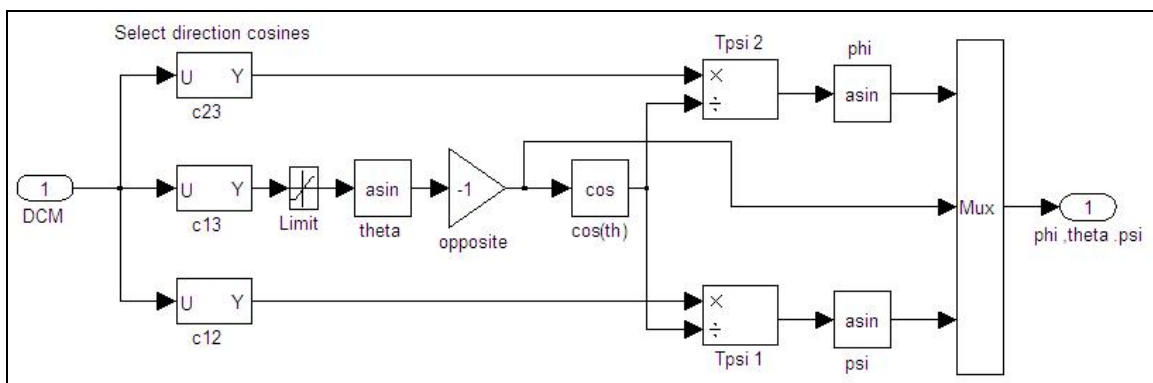


Figure 6. Extract Position Angles function in ADCS (After [19]).

Elementary operations include multiplication, division, signal limiting (ceiling or floor function based on specified constant bounds), and selecting and multiplexing signals. More complicated operations include the inverse trigonometric functions used to convert the direction cosine data into Euler angles of spacecraft position.

Although the control algorithm in the ADCS processor is essential on-orbit processing for most space vehicles, it is not the only computing application that must function correctly and reliably on board a satellite. Another subsystem that is necessary for every spacecraft to carry out its mission is the communication system.

C. SPACECRAFT SIGNAL PROCESSING: SOFTWARE-DEFINED RADIOS

1. Purpose and Requirements of a Spacecraft Signal Processor

One of the most computationally intensive parts of a spacecraft communication subsystem is the signal processor. Almost all modern spacecraft use digital computers, which operate on data that have been extracted from samples of a signal received by the communications antenna. Between the baseband information processing in the main spacecraft computer and the RF signal transmitted from or received by the antenna is a series of signal processing operations that can be implemented in hardware or in some combination of hardware and software. Reed [20] defines a software radio as “a radio that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software.” Many software radio designers use SRAM-based FPGAs for digital data processing because they can handle more complex circuits than programmable digital signal processing (DSP) chips, and can execute signal processing operations faster than standard microprocessors. This is particularly evident when designers take advantage of parallel processing architectures allowed by FPGA logic block configurations. FPGAs also offer great flexibility because they can be reprogrammed to handle different architectures or algorithms, even “on the fly” during system operation.

A very important requirement for a software radio processor is computation speed: it must be able to handle data at the sample rate necessary to receive and/or

transmit all required information. In addition to speed, a software radio must be able to handle some level of complexity in its circuits, for algorithm implementation. The greatest environmental requirements on a space vehicle for any part or subsystem (as opposed to functional requirements) are limitations on mass, power consumption, and heat dissipation – all due to on-board resource availability. All these must be taken into account when deciding how to allocate functions to hardware and software in a spaceborne software-defined radio.

2. General SDR Overview

In general, a radio is meant to process streams of signals and data. Signals are received through an antenna, decoded, and presented to an observer or listener through an output. The data can also be internally generated or input through a device, coded, and transmitted as an RF signal through the antenna. In either direction, data is rarely saved or retransmitted, except in the case of error-checking protocols such as automatic repeat request (ARQ). Compared to the ADCS, which has a set of system states and state errors that are continually updated using a feedback loop, a radio can be considered a “flow-through” process. Data may be processed in a continuous stream or broken into batches, but either way data are brought in, manipulated, and put out in a minimal number of clock cycles of the system. Within the communications system family, the characteristics distinguishing a software radio from a traditional radio depend on how much of the radio’s functionality can be changed without replacing hardware, and how flexible the radio is overall.

Any software radio is identified by the presence of flexibility throughout its architecture. A generic block diagram of a software radio is shown in Figure 7. The “smart antenna” and RF hardware are ideally able to cover multiple spectrum bands within a broad range. The analog-to-digital converter (ADC) and digital-to-analog converter (DAC) may have multiple settings for fidelity of information (e.g., 8-, 12-, or 16-bit converters). The processor(s) may implement communications algorithms in some combination of software and hardware, depending on requirements for speed and reprogrammability.

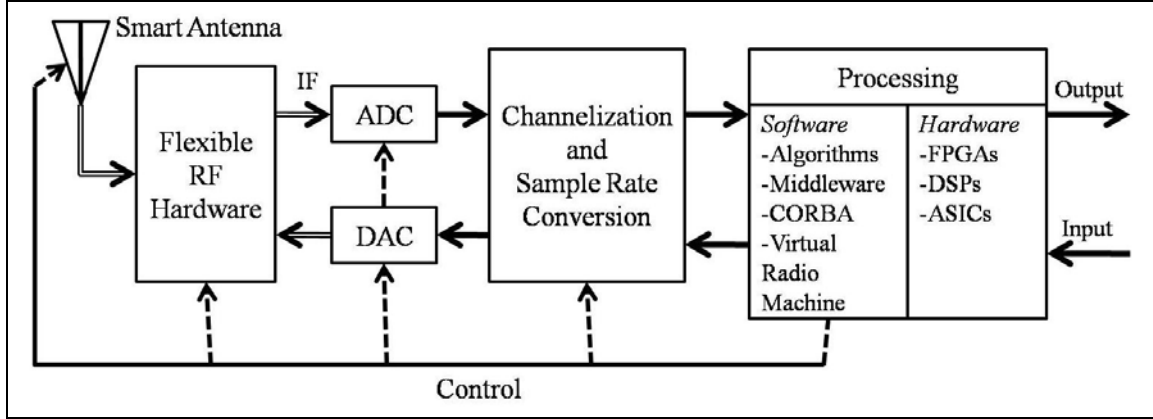


Figure 7. Model of a Software Radio (From [20]).

Although software radios can use any combination of application-specific integrated circuits (ASICs), DSPs, or FPGAs as processing hardware, this research focuses on implementing software radio processes on RAM-based FPGAs (such as those on CFTP) and how to make those processes fault-tolerant using RPR. Within the “processing” block, there are functions that break down or divide the signal into batches, decode the signal, reconstruct information from decoded data, communicate with other processes, or report to a higher level processor or to an overall operating system. This exploration focuses on one of the most common operations found in the signal processing block of any radio: the Discrete Fourier Transform (DFT), and as implemented in digital computers, the Fast Fourier Transform (FFT).

3. Example SDR Function: Fast Fourier Transform (FFT)

The operation used to translate a sampled signal represented by a set of N data points in the time domain to a second set of N points in the frequency domain is the Discrete Fourier Transform (DFT), given by

$$X[k] = \text{DFT}\{x[n]\} = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}, \text{ for } k = 0, \dots, N-1 \quad (2)$$

where $x[n]$ are the time-domain input samples, $X[k]$ are the frequency-domain output samples, and $j = \sqrt{-1}$. The complex exponential can also be represented as the phase factor w_N , as in

$$X_N[k] = \sum_{n=0}^{N-1} x[n] w_N^{kn}, \quad k = 0, \dots, N-1 \quad (3)$$

where $w_N = e^{-j2\pi/N}$ and the subscript N indicates that there is a unique phase factor for any FFT of size N . When the DFT is implemented in digital computers to process sets of data where N is a power of 2, the transform operation can be continually broken down into the sum of two half-size transforms, as in

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

until the lowest level is reached, where Equation (4) simplifies to the addition of two input elements with a coefficient $w_{N/2}^{kn}$. When $N = 2$, the coefficient $w_{N/2}^{kn} = (1, -1)$, and so the fundamental operation for each pair of output points becomes the addition and subtraction of two corresponding input points – known as a butterfly operation or butterfly machine (BFM). A graphical representation of this operation, where the crisscross “winged” shape is evident, is shown in Figure 8.

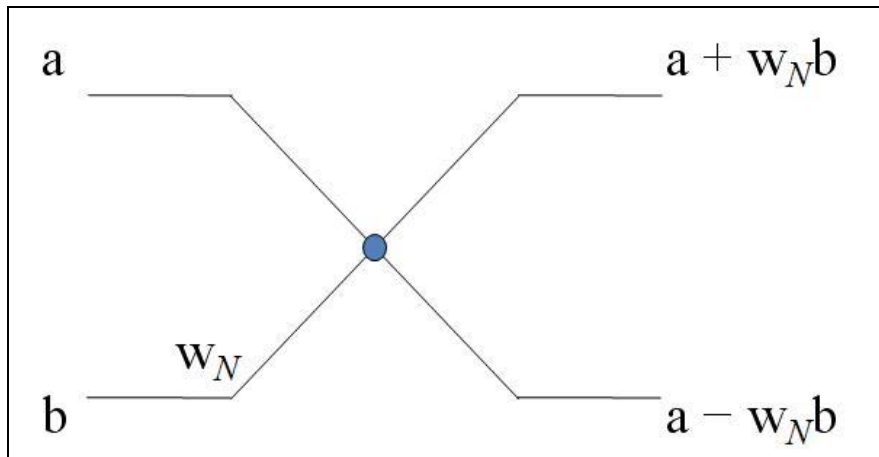


Figure 8. Graphical representation of FFT BFM.

Computing a DFT as a series of butterfly operations is known as the Radix-2 Fast Fourier Transform (FFT). The complexity of the FFT increases as $N \log_2(N)$, while the complexity of the DFT operation grows as N^2 . Additional details on the FFT, its complexity relative to that of the DFT, and this derivation are available in [21].

One BFM is used to compute a two-point FFT ($N = 2$). In a two-point FFT, the phase factor or twiddle factor w_N used to multiply the second operand (b in Figure 8) prior to the addition and subtraction is equal to 1. To compute FFTs of practical lengths (e.g., $N = 8$ or higher), multiple BFMs are implemented in levels and pipelined. The number of levels of BFMs in an FFT is $l = \lceil \log_2 N \rceil$, with each stage containing $N/2$ radix-2 butterfly operators. The only differences from one butterfly operation to the next, or among levels of butterflies, are: the memory locations of the input, memory locations of the output, and the factors w_N . The nets (connections) between levels or individual operations are mapped differently depending on the algorithm of the FFT; the algorithm is chosen based on design implementation (e.g., pipelined or iterative) and also affects the order of the input and output points. A diagram of an eight-point DFT calculation using an in-place algorithm FFT with three levels of butterfly operations is shown in Figure 9.

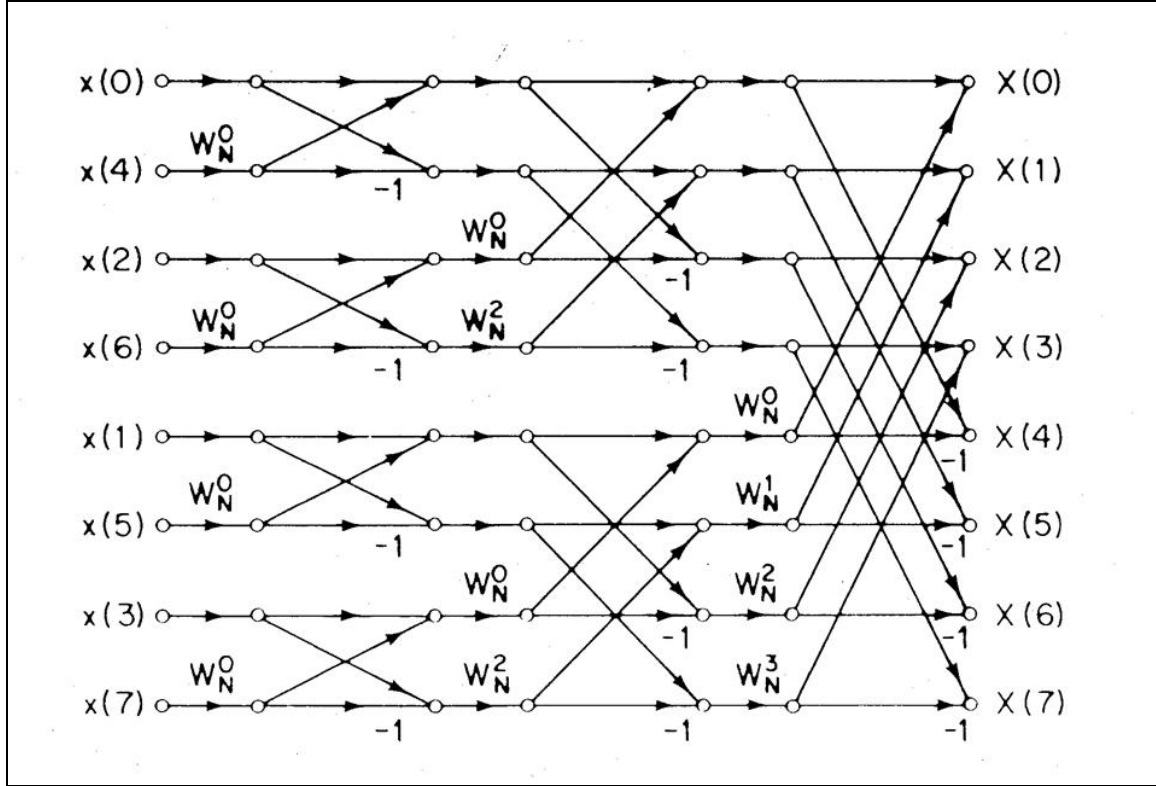


Figure 9. Flow graph of an Eight-Point DFT using three levels of (From [22]).

The FFT is among the most commonly-found functions implemented in pipelined form in SDRs [20]. The pipelining and parallel data processing possible in an FPGA enables a computation speedup proportional to the number of pipeline stages used in the implementation.

The prefabricated FFT used in a software radio design developed at the Naval Postgraduate School in 2008 [23] is the Xilinx Fast Fourier Transform v4.1, which computes the FFT with the Cooley-Tukey algorithm [24]. Wright [23] uses the FFT v4.1 in pipelined, streaming input/output (I/O) mode, depicted in Figure 10. In this mode, which is computed using radix-2 BFM's with either bit-reversed or naturally-ordered addressing, each stage of the FFT has its own memory for storing input and (intermediate) output. Both the input data and the phase factors may be expressed as fixed-point numbers with width between 8 and 24 bits, inclusive.

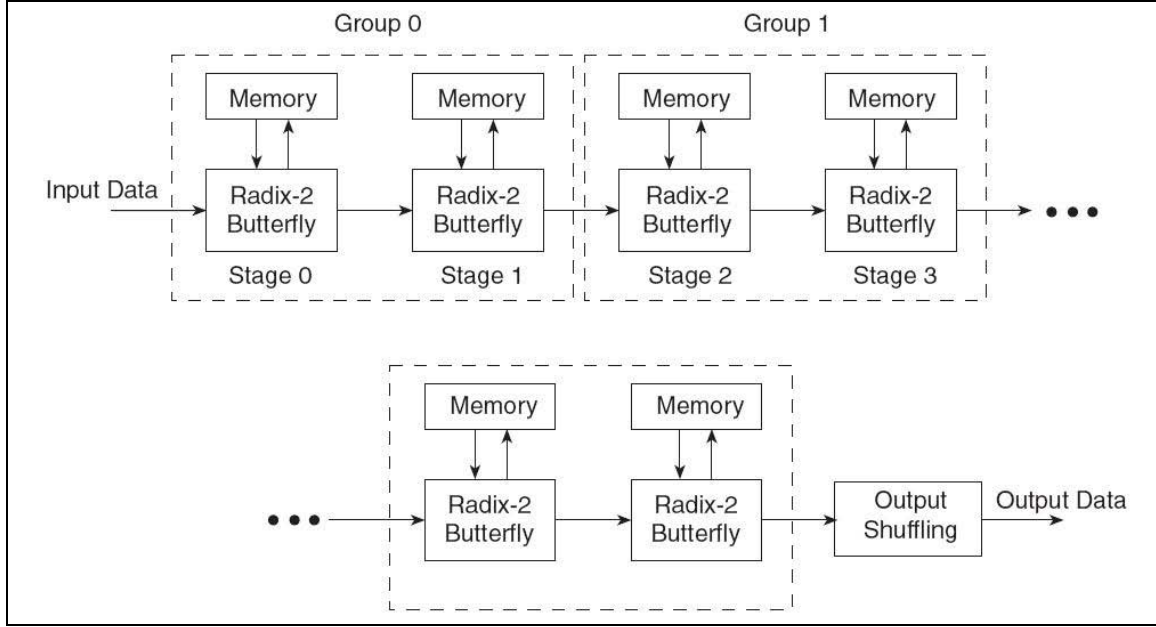


Figure 10. Xilinx FFT v4.1 Pipelined, Streaming I/O Architecture (From [24]).

In the Xilinx v4.1 pipelined streaming I/O configuration, each stage may begin computation when the first pair of results is available from the previous stage. Output is available continuously after the latency period of the pipeline. Detailed timing information on the streaming I/O mode of the Xilinx FFT v4.1 is available in [24].

The fundamental processes that make up a butterfly operation are addition, subtraction, and multiplication. These are somewhat complicated by the fact that in any FFT with $N > 2$, the phase factor w_N is complex – therefore most intermediate and final results are also complex. However, in most FFT implementations (including the Xilinx FFT v4.1), the real and imaginary parts of complex values are handled separately; supplementary multiplication, addition and subtraction logic is included to handle the recombination of real, imaginary and complex values as needed. The complex phase factors are calculated and stored in advance. An additional task for running an FFT made up of layers of BFM's is the memory indexing required for input and output at each stage. It is also worth considering the butterfly as a single operation – and investigating whether it, as a collection of more elementary operations, is a suitable candidate for RPR.

D. COMMON ELEMENTARY OPERATIONS

In order to apply RPR to a PD controller or a pipelined FFT, it is necessary to determine how to apply RPR to the elementary operations used in these systems. The elementary arithmetic operations used in these systems include addition, subtraction, multiplication, and division. In principle, each of these operations is a Class A problem: results may be calculated at full precision or with less precision. Chapter III investigates rules for applying RPR to each function individually, and whether any groups of these operations are similar enough for a designer to adhere to a common set of rules for RPR.

In addition to the arithmetic operations, other fundamental processes are necessary to execute either the controller or the FFT. These include memory indexing (e.g., “output shuffling” at the end of the FFT), element selection or concatenation from a set of stored or transmitted data (e.g., values from a direction cosine), and maintaining constants in memory (e.g., FFT phase factors or controller gains). These processes and requirements are all more appropriately Class B problems. Justification for these decisions is recorded in **Error! Reference source not found..**

Operation	Key Activity	Justification for Class B
FFT output shuffling	Reassigning memory index	Requires exact value for unique memory location address.
ADCS matrix element selection or vector concatenation	Forwarding a single variable or subset of variables from a stored or transmitted block of data	From memory: requires exact address. From data stream: <i>may</i> be modified to handle data represented by fewer bits – but will almost always be stored in intermediate RAM location regardless.
Accessing stored constants	Reading data from memory locations	Requires exact value for unique memory location address.* *Benefit of coding vs. copying stored values needs further study – storing copies, even with reduced precision, requires more memory than implementing parity checks. Both approaches require additional decision logic, for voting or for correction.

Table 1. Determining Operations Not Suitable for RPR (Justification from [3]).

Finally, there are two significant compound processes used in these systems: matrix multiplication in the controller, and the butterfly operation in the FFT. If either or both of these operations can be made fault-tolerant by applying RPR at the compound-process level rather than at the elementary operation level, there is potential to save additional space and power on an FPGA.

Whether in an ADCS or an SDR, there are basic arithmetic operations that form the heart of the computational processes required of a space vehicle computer. Chapter III explores the nature of each suitable operation and provides rules of applying RPR in each case.

THIS PAGE INTENTIONALLY LEFT BLANK

III. REDUCED-PRECISION REDUNDANCY FOR COMMON ELEMENTS

A. ASSUMPTIONS, TERMINOLOGY AND GENERAL RULES

Any operation protected by RPR has two main processes: (1) executing the operation (in both full- and reduced-precision), and (2) error detection, result selection and reporting. In some cases the upper and lower bounds of one or more operands must be computed prior to beginning the RPR operation (one might call this “process (0)”); in other cases the operands are already available in full- and reduced-precision (e.g., if generated by the last operation). In this investigation of RPR applications, all arithmetic operation sections include the following subsections: an expression of error in the computation of the fundamental operation, the approach to determining RPR bounds for input (operands) and output (result), and a demonstration either in tabular form or as programmed using Xilinx Integrated Simulation Environment (ISE) Release 6.3.03i. The Xilinx ISE modules were generated for the Xilinx Virtex™ XQVR600 radiation-hardened device, which is the FPGA used on the NPS CFTP, using a combination of VHDL and schematic entry. Module functionality was tested using the simulation environment ModelSim SE version 6.3c. The RPR demonstrations implemented in ISE contained both arithmetic operations and voters, and were compared to analogous TMR demonstrations using FPGA area occupied as a metric for evaluation. The FPGA area was reported during the mapping process in ISE as a *slice count*, where a slice consists of two 4-input LUTs, two D flip-flops, and two carry and control units [25].

In order to discuss different methods and effects of applying RPR, it is necessary to define a metric that represents the “degree” of RPR – that is, the amount by which precision is reduced from the original high-precision calculation to the redundant lower-precision calculations. An easily-accessible quantity is the ratio of number of non-sign-bits r in the variables of the redundant calculations to the number of non-sign-bits n in the variables of the precise calculation. For example, if the precise operands and result are represented in eight bits of precision and the upper and lower bounds have five bits of

precision, then the *degree of RPR* is $r/n = 5/8$. If the precise calculation is made using 32 bits of precision and the redundant calculations are made using 16 bits of precision, the degree of RPR is $r/n = 16/32$. This is of course mathematically equivalent to $1/2$ (or $8/16$) – however, the value of the denominator is important¹, so the fraction is not reduced. When comparing different degrees of RPR for a single original calculation of a given n , the ratios may be expressed as decimals to indicate the relative precision retained. For example, RPR may be applied to a 64-bit number as $32/64$, $16/64$, $8/64$, or any other ratio r/n . To make the metric easily comparable when considering performance of different RPR approaches for this fixed n , the degree of RPR may be expressed as 0.50, 0.25, 0.125, etc. instead of as fractions.

In each section of this chapter, an RPR protection approach is introduced. For the purposes of illustrating the approaches with experiments, the following conventions are used for number representation and digital computation.

For addition and subtraction, numbers in digital computation are represented in fixed-point two's complement (generally of 8-, 16-, 32-, or 64-bit width), with the radix point one place to the right of the MSB (i.e., after the sign bit). This implies 7, 15, 31 or 63 bits of precision, respectively. In practical applications, inputs a and b would be scaled prior to the operation such that they fall within the range $-1 \leq (a, b) < 1$. An example of this representation is seen in Figure 11, where the decimal number -0.785398_{10} ($\pi/4$ to six decimal places) is represented in two's complement format as a fifteen-bit fraction with a leading sign bit. The value of the binary number in Figure 11, when converted back to decimal notation, is actually $-0.78536..._{10}$, because the smallest representable value in fifteen places of fixed-width fractional precision is $\pm 2^{-15} = (1/32768)$ or $0.00003..._{10}$.

¹ Consider the degree of RPR to be similar to the time signature in a musical piece: $4/4$ is mathematically equivalent to $2/2$, but in fact there is a different number of beats per measure in each scheme, as well as a different note representing the fundamental beat. The “feel” of the rhythm in $4/4$ vs. $2/2$ is quite different.

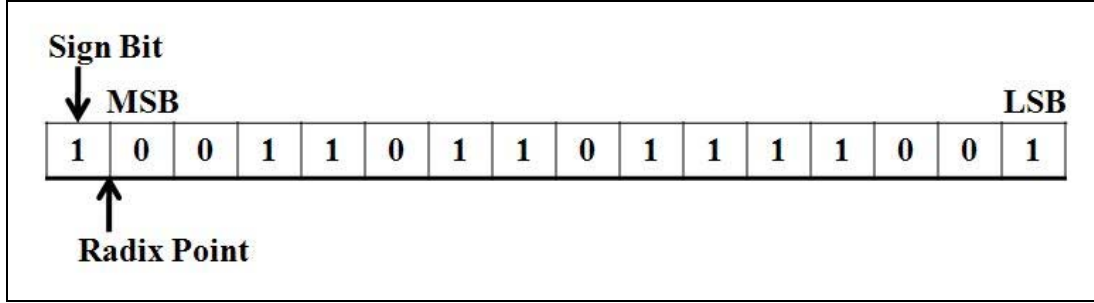


Figure 11. Sixteen Bit Fixed-Point Two's Complement Representation of $(-\pi/4)$.

For multiplication and division, numbers are also represented as fractional fixed-point values, but are in sign-magnitude form (as opposed to two's complement). The additional processing required to multiply two's complement numbers instead of sign-magnitude numbers makes two's complement a poor choice of representation for multiplication. This is explained further section C, on multiplication. Figure 12 shows the sign-magnitude format for -0.785398_{10} (compare to Figure 11). The absolute value of the number is represented in the fifteen fractional bits, and the MSB is the sign bit (1 for negative numbers).

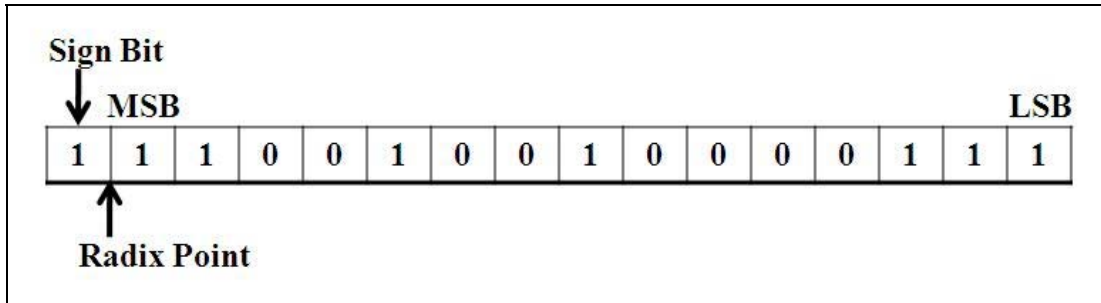


Figure 12. Sixteen Bit Fixed-Point Sign-Magnitude Representation of $(-\pi/4)$.

Additional characteristics, constraints and rules for implementing RPR in each operation are expounded in the operation sections. The final section of this chapter discusses the logic required to fabricate an RPR voter, because regardless of the method required to determine the upper and lower bounds on a result, much of the voting process is the same once the bounds are obtained. At the conclusion of the chapter, RPR is

viewed from the perspective of error detection and correction in terms of the trades a designer must make in order to use RPR effectively on the operations most suitable for its application.

B. ADDITION AND SUBTRACTION

1. Computation Error in Addition and Subtraction

Given two fixed-point binary numbers a and b of same precision n with the radix point in the same location, there is no significant round-off error due to a single execution of the fundamental operation of addition or subtraction [26]. That is, the mathematical operation $c = a \pm b$ is computed accurately to the precision of a and b . The accuracy of each binary operand is limited by its precision, that is, if a is a fractional binary number with n places after the radix point, it can only be accurate to within $\pm 2^{-(n+1)}$ of its actual value. The error magnitude in any fixed-point fractional number of n places is therefore $\varepsilon \leq 2^{-(n+1)}$. When two numbers are added, the error in each operand may be additive or subtractive, i.e., in the same or different directions on a number line (Figure 13). The maximum error in the sum occurs when the errors in the operands are additive, and turns out to be

$$\begin{aligned} \varepsilon_{\text{sum}} &\leq \varepsilon_{a_{\text{max}}} + \varepsilon_{b_{\text{max}}} = 2^{-(n+1)} + 2^{-(n+1)} = 2\left(2^{-(n+1)}\right) = 2^{-n} \\ \varepsilon_{\text{sum}} &\leq 2^{-n} \end{aligned} \tag{5}$$

which is equal to the value represented by the LSB of the result. This is shown by the potential range $c - 2\varepsilon$ to $c + 2\varepsilon$ for the result c in Figure 13. Although the largest possible ε_{sum} is still only the smallest number that may be represented using the chosen precision, an error in that LSB can propagate to the next level of computation and therefore may become significant in later operations.

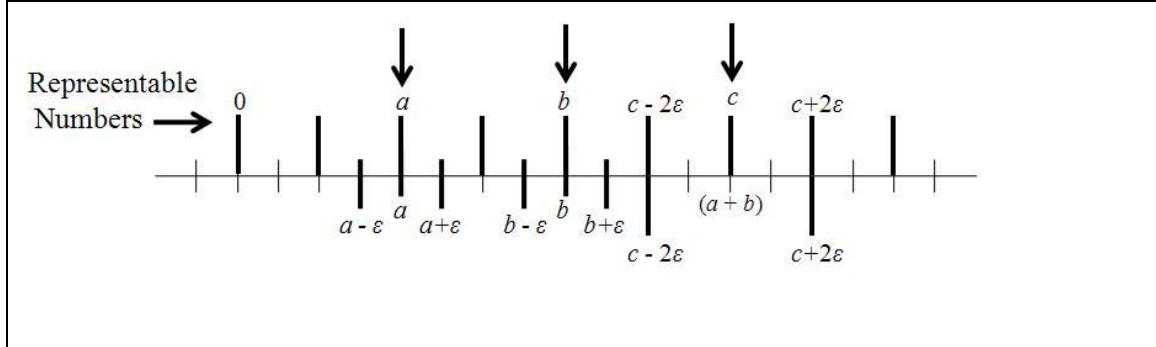


Figure 13. Number line showing maximum error in one addition operation.

Since the extreme values $c - 2\varepsilon$ and $c + 2\varepsilon$ represent the maximum *range* of the result c of an addition or subtraction operation, these extrema also define the *bounds* on the error in the result c . This concept of bounding error in results can be extended to problems of different levels of precision when applying RPR to addition and subtraction.

2. Upper and Lower Bound Determination for Addition and Subtraction

To apply RPR successfully to a given operation, it is necessary to determine the reduced-precision upper and lower limits, or bounds, of the result (the output) as functions of the operands (the input). Using two's complement notation for addition and subtraction enables identical treatment of the two operations, provided the transitions between the negative number closest to zero ($1.111..._2$) and zero ($0.000..._2$) are carried out successfully. In this scheme, the upper bound x_U of any operand or result x must lie to the right of x on a number line; the lower bound x_L must lie to the left of x , as shown in Figure 14. This means that if x is a *negative* number, the *magnitude* of x_L is larger than the *magnitude* of x , whose magnitude is in turn larger than the magnitude of x_U .

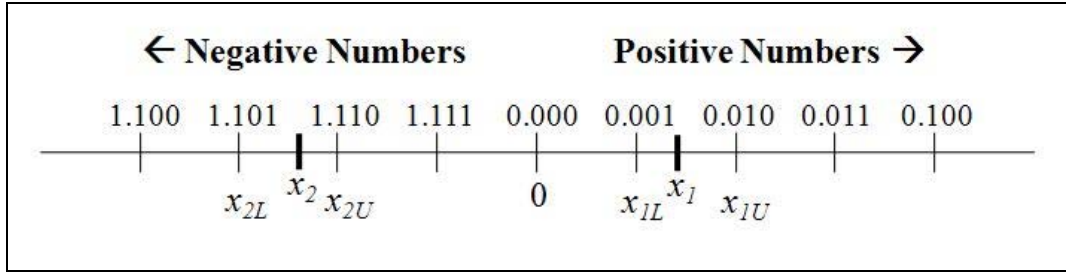


Figure 14. Upper and Lower Bounds of Fixed-Point Two's Complement Numbers.

When operands a and b are numbers of precision n and the degree of RPR desired is r/n , the lower bounds a_L and b_L are defined as the highest (rightmost on number line) numbers of precision r such that $a_L \leq a$ and $b_L \leq b$. The upper bounds a_U and b_U are defined as the lowest (leftmost on number line) numbers of precision r such that $a < a_U$ and $b < b_U$. For example: if $n = 5$, $r = 3$ and $x_1 = 0.00101_2$ as in Figure 14, then

$$x_{1L} = 0.001_2 \leq 0.00101_2 \quad \text{and} \quad 0.00101_2 < 0.010_2 = x_{1U}. \quad (6)$$

Using operand upper and lower bounds as demonstrated in Equation (6), the sum or difference of two fixed-point fractional two's complement operands will always be contained within a result range bounded as described in Equation (7).

$$\begin{aligned} \text{For } c = a + b \text{ and } d = a - b, \text{ given } a_L \leq a < a_U, \quad b_L \leq b < b_U : \\ c_L \leq c < c_U, \quad d_L < d < d_U, \text{ where} \\ c_L = a_L + b_L, \quad c_U = a_U + b_U, \quad d_L = a_L - b_U, \quad d_U = a_U - b_L. \end{aligned} \quad (7)$$

Subtraction can be accomplished in two ways. One way is to create a distinct RPR subtraction module that executes only subtraction and calculates the bounds as defined for differences d in Equation (7). The other way is to prepend a two's-complementer to an RPR addition module, which calculates bounds as defined for sums c in Equation (7), and then to add the two's complement of the minuend to the subtrahend. The two methods generate the same result; the two's complement/sign change method will be used from this point forward because it generates the fewest additional special cases.

Equation (7) is true regardless of sign and magnitude of a and b , as is shown in the following exercise. In a two-input addition or subtraction operation, one may characterize the problem using three Boolean variables that express properties of the input set: the sign S_a of the first addend (or subtrahend) a , the sign S_b of the second addend (or minuend) b , and the magnitude of a compared to the magnitude of b (or $|a| \stackrel{?}{>} |b|$). In the simplest case, there are $2^3 = 8$ possible arrangements on a number line for the operands and result of the problem $c = a \pm b$ as described by these properties. Special cases, such as where $|a| = |b|$, will be addressed separately. Table 1 lists the eight simple scenarios; Figure 15 and Figure 16 show all eight cases on number lines. In each scenario, the sum c or difference d is wholly bounded by the sum or difference of some combination of the upper and lower bounds of the operands, as dictated by Equation (7).

Case	S_a	S_b	$ a > b $	$S(a + b)$	$S(a - b)$
1	0	0	1	0	0
2	0	0	0	0	1
3	0	1	1	0	0
4	0	1	0	1	0
5	1	0	1	1	1
6	1	0	0	0	1
7	1	1	1	1	1
8	1	1	0	1	0

Table 1. Eight Combinations of Input Properties for Addition and Subtraction.

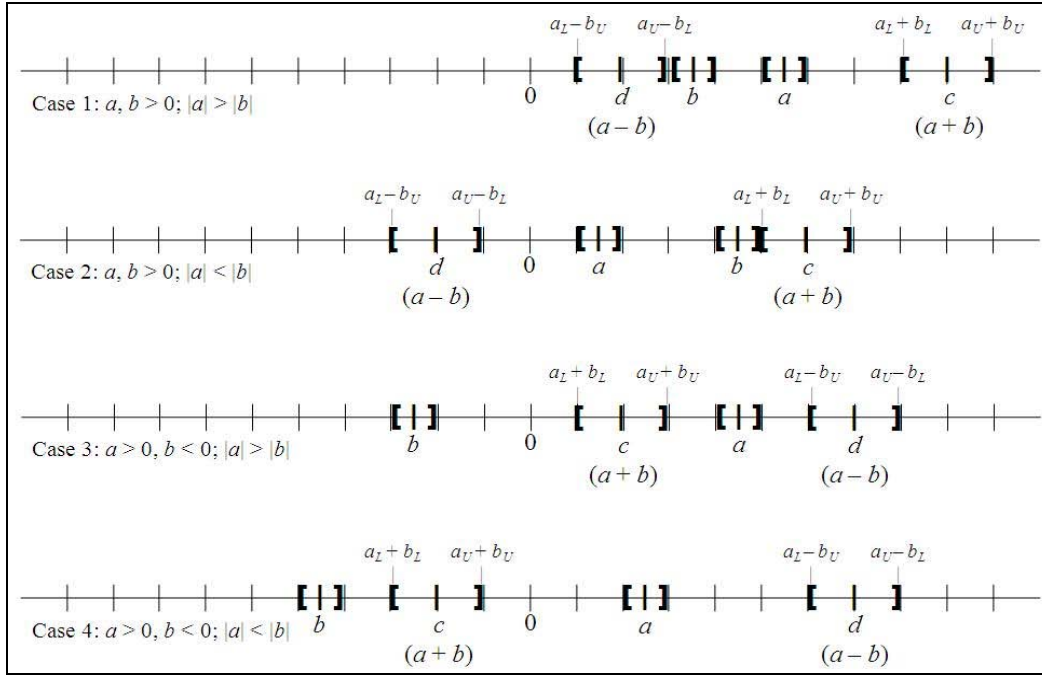


Figure 15. Cases 1 through 4 for $c = a + b$ and $d = a - b$.

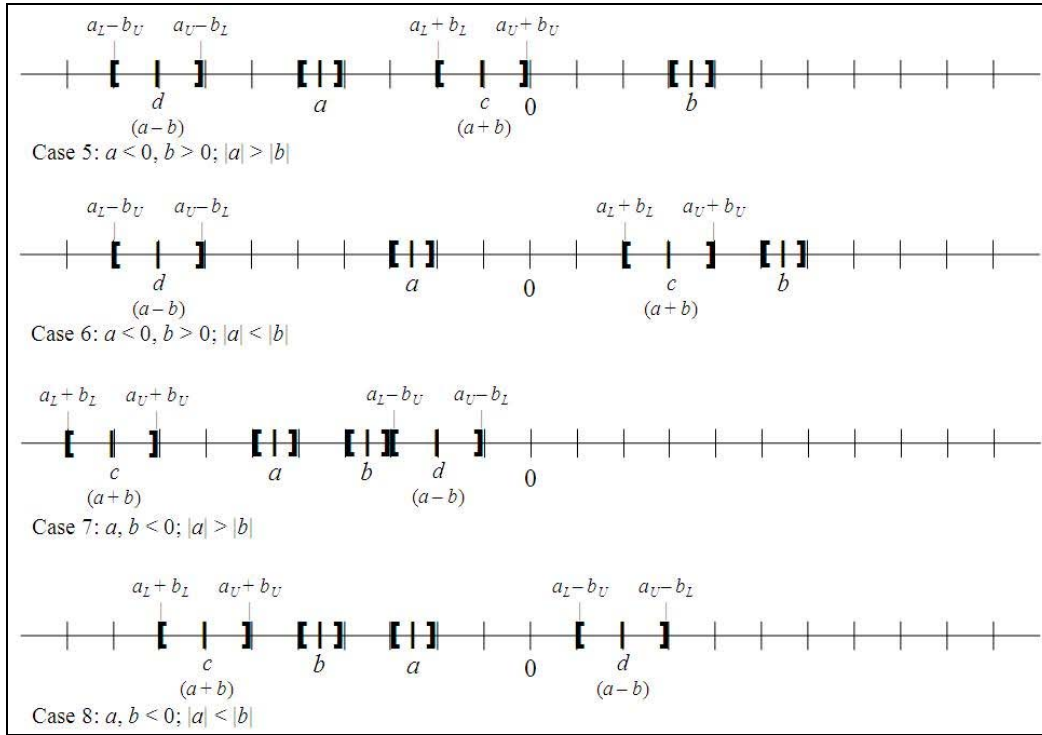


Figure 16. Cases 5 through 8 for $c = a + b$ and $d = a - b$.

There are a few additional considerations and special cases worthy of discussion when working with addition and subtraction. None invalidates the rules for determining bounds; however, each needs to be investigated to confirm its conformity.

a. Special Cases Where $|a| = |b|$

There are four special cases where the operands are of equal magnitude: (1) $a = b$ and $a, b > 0$; (2) $a = b$ and $a, b < 0$; (3) $a = -b$ and $a > 0$; (4) $a = -b$ and $a < 0$. In cases (1) and (2) the operands subtract to give zero; in (3) and (4) they add to give zero. Although they cross the boundary between positive and negative numbers, these cases follow the same bound convention as those with unequal operands. These are special cases for two reasons: first, because they operate in the transition between $1.111..._2$ and $0.000..._2$, and therefore at least one case should be included to test the carry-out/carry-in operation to execute that transition. Second, it is notable that the upper and lower of the zero results are necessarily of opposite sign; in circuit design and operation this should not be a problem, but the designer must be aware of the crossover.

Finally, it may seem that if an error were to occur in the precise calculation in these cases, some operations that *should* result in true zero would accrue some nonzero value at the output. However, this may be avoided with appropriate error correction techniques, which are discussed in the section on RPR voter logic.

b. Special Cases Where Precise and Bound Values Are the Same

When an operand a or b contains only r significant digits – i.e., all the least significant $(n - r)$ bits are zero – then the condition $a_L = a$ or $b_L = b$ applies. In this case, the correct precise results c and d will always fall within a range that is only half the maximum error allowed by Equation (7). It is possible to tighten the bounds on the result in this case by adding logic to test operands for zeros in the $(n - r)$ LSB: if an operand a with only r significant digits is discovered, then a_U may be set equal to a_L (which is already equal to the precise value a). This will reduce the maximum error of the results ($c_U - c_L$ and $d_U - d_L$) to half their nominal range, restoring the effective precision of c

and d to r bits. This is equally applicable to the subtrahend a or minuend b . However, the benefit gained from the relatively unlikely occurrence of this case may not be worth the additional logic circuitry required to detect the LSB condition and reset the bounds. Candidate systems would need to be examined on a case-by-case basis to determine the merit of including this option.

c. Overflow Cases

Because the upper bound of the operands is determined by rounding up, any full-precision number greater than the largest representable positive number of reduced precision r (i.e., $a > 1 - 2^{-r}$) will cause an overflow condition in its upper bound. In two's complement, the most-negative number is 2^{-r} , so the analogous overflow condition need not occur for lower bounds. However, if a particular application requires the number "1.000..." to represent a value other than 2^{-r} , then there also may be overflow in the most-negative bound, i.e., when $a < -1 + 2^{-r}$. In either overflow case, the designer may choose to use the overflow indicators as control signals to activate an alternate RPR result (such as using only the non-overflow bound instead of a combination of the two). The alternate RPR result may not be as accurate as the default RPR result, but in the case of an SEU-induced error the alternate RPR result is still closer to the correct result than the unprotected (erroneous) precise result would be. A second option is to scale the operands such that they lie in the range $1 - 2^{-r} > a > -1 + 2^{-r}$ before determining bounds and beginning the operation, so that the overflow condition on the bounds is avoided altogether.

When the addition operation itself results in overflow in the precise result, the overflow flag is activated as it would be in a normal (non-RPR) addition operation. In this case, the overflow will also occur in at least one of the upper or lower bound results, and may possibly occur in both. This property enables the designer to use a bitwise-majority voter (such as is used in TMR) to determine whether overflow occurs in the addition operation.

d. *Special Cases Involving Zero*

When one or more of the operands is identically zero, the operand upper bounds determined as in Equation (6) will be nonzero values. Similarly, when an operand is between zero and 2^{-r} (or zero and -2^{-r}), the lower (or upper) bound will be zero when the value is nonzero. In general, this case has negligible impact in addition and subtraction for two reasons: (1) every addition or subtraction operation involving zero is defined, and (2) since addition and subtraction are linear operations, adding or subtracting zero has almost the same effect as adding or subtracting a quantity very close to zero. The complication arises when the result of an addition or subtraction operation must be tested for equality to another result (or to its input values), or when a quantity that should (or should not) be zero is used in a subsequent multiplication (or division) operation. In these cases, the output from the RPR voter that tells whether the final result is the (correct) precise result or the RPR result is very important; it may be an important input condition for the next operation.

To provide numerical illustration of some of the special cases, Table 2 demonstrates a nearly-exhaustive treatment of 4/6 RPR. It lists exact values, lower and upper bounds for each representable number in the range $(-1, 1)$. The optional modified upper bounds for the special cases when $x = x_L$ are also included.

The maximum error in any addition or subtraction calculation protected using RPR is determined by the precision of the reduced-precision operation. As implied by Figure 13, the range between the upper and lower bounds on sum c or difference d is

$$(c + 2\varepsilon) - (c - 2\varepsilon) = 4\varepsilon, \quad \varepsilon \leq 2^{-(r+1)} \quad (8)$$

where r is the precision of the bound operations. However, Equation (5) states that the maximum error in a calculation of precision r is 2ε . In order to ensure the minimum difference 2ε between the correct precise result and the RPR result, the RPR result must actually be the *average* of the upper and lower bounds

$$d_{RPR} = \frac{d_L + d_U}{2} = (d_L + d_U) \times 2^{-1}. \quad (9)$$

In addition and subtraction, Equation (9) is also equivalent to adding $2^{-(r+1)}$ to the lower bound result d_{RPR} .

The manipulation of the bound results required to produce the RPR result is one component of an interesting problem concerning the use of RPR protection for small circuits. This problem is explored further in the demonstration and voter sections.

FULL PRECISION (6)		REDUCED PRECISION (4)		
x_{10}	x_2	x_L	x_U	MOD x_U
0.96875	0.11111	0.111	OV	OV
0.93750	0.11110	0.111	OV	OV
0.90625	0.11101	0.111	OV	OV
0.87500	0.11100	0.111	OV	0.111
0.84375	0.11011	0.110	0.111	0.111
0.81250	0.11010	0.110	0.111	0.111
0.78125	0.11001	0.110	0.111	0.111
0.75000	0.11000	0.110	0.111	0.110
0.71875	0.10111	0.101	0.110	0.110
0.68750	0.10110	0.101	0.110	0.110
0.65625	0.10101	0.101	0.110	0.110
0.62500	0.10100	0.101	0.110	0.101
0.59375	0.10011	0.100	0.101	0.101
0.56250	0.10010	0.100	0.101	0.101
0.53125	0.10001	0.100	0.101	0.101
0.50000	0.10000	0.100	0.101	0.101
0.46875	0.01111	0.011	0.100	0.100
0.43750	0.01110	0.011	0.100	0.100
0.40625	0.01101	0.011	0.100	0.100
0.37500	0.01100	0.011	0.100	0.011
0.34375	0.01011	0.010	0.011	0.011
0.31250	0.01010	0.010	0.011	0.011
0.28125	0.01001	0.010	0.011	0.011
0.25000	0.01000	0.010	0.011	0.010
0.21875	0.00111	0.001	0.010	0.010
0.18750	0.00110	0.001	0.010	0.010
0.15625	0.00101	0.001	0.010	0.010
0.12500	0.00100	0.001	0.010	0.001
0.09375	0.00011	0.000	0.001	0.001
0.06250	0.00010	0.000	0.001	0.001
0.03125	0.00001	0.000	0.001	0.001
0.00000	0.00000	0.000	0.001	0.000
-0.03125	1.11111	1.111	0.000	0.000
-0.06250	1.11110	1.111	0.000	0.000
-0.09375	1.11101	1.111	0.000	0.000
-0.12500	1.11100	1.111	0.000	1.111
-0.15625	1.11011	1.110	1.111	1.111
-0.18750	1.11010	1.110	1.111	1.111
-0.21875	1.11001	1.110	1.111	1.111
-0.25000	1.11000	1.110	1.111	1.110
-0.28125	1.10111	1.101	1.110	1.110
-0.31250	1.10110	1.101	1.110	1.110
...	...	...	...	...
-0.71875	1.01001	1.010	1.011	1.011
-0.75000	1.01000	1.010	1.011	1.010
-0.78125	1.00111	1.001	1.010	1.010
-0.81250	1.00110	1.001	1.010	1.010
-0.84375	1.00101	1.001	1.010	1.010
-0.87500	1.00100	1.001	1.010	1.001
-0.90625	1.00011	1.000	1.001	1.001
-0.93750	1.00010	1.000	1.001	1.001
-0.96875	1.00001	1.000	1.001	1.001

Table 2. Upper and Lower Bounds for 4/6 RPR Showing CO, OV and Modified x_U .

3. Demonstration of RPR in Addition and Subtraction

The overall structure of an RPR operation contains two modules: (1) the high- and low-precision copies of the operation itself, and (2) the RPR voter that contains logic to select the final result and report any errors. This two-module configuration is shown in Figure 17. The incoming clock signal is necessary in order to synchronize the operations, particularly when the degree of RPR is small (significant reduction), because in that case the reduced-precision calculations have significantly less delay than the full-precision calculation. The difference in delay can cause errors if the output availability is not synchronized. There are six functional inputs to an RPR adder: the two operands a and b (of precision n), and each operand's upper and lower bounds (of lower precision r). There is also a clock signal input. There is at least one output from an RPR adder: the chosen result. In Figure 17 there are many possible outputs shown; these are discussed in greater detail in the section on voter logic. The remaining signals in the top-level RPR module are the intermediate results of the precise, upper bound and lower bound addition operations. These intermediate results are fed into the RPR voter, which determines what the final result should be.

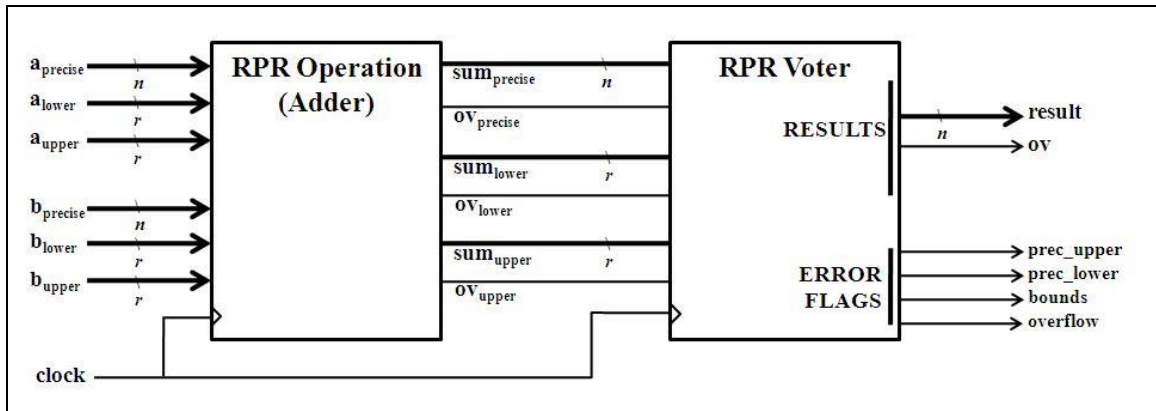


Figure 17. RPR Adder Top-Level Block Diagram (OV = overflow).

The first module, the operation, is shown in Figure 18 for addition only. It consists of three adders and clocked registers to synchronize the inputs and outputs. There is one full-precision (n -bit) adder and two reduced-precision (r -bit) operators (one

for each bound calculation). These three adders are analogous to the three identical copies of a full-precision operation that exist in a TMR-protected operation, as depicted in Figure 19. The architecture shown in Figure 18 can be used only for addition because the incoming bounds (a_L, a_U, b_L, b_U) are arranged in the configuration that produces upper and lower bounds on a sum, not a difference (see Equation (7)). In a subtraction operation, the upper and lower bounds of a are paired with the opposite bounds of b (i.e., lower and upper, respectively). However, if a designer wanted to use only one RPR implementation for both addition and subtraction, he could find the two's complement of the minuend and then use this RPR operation module to add the subtrahend and complemented minuend – that is the alternative to building the equivalent subtraction operation module.

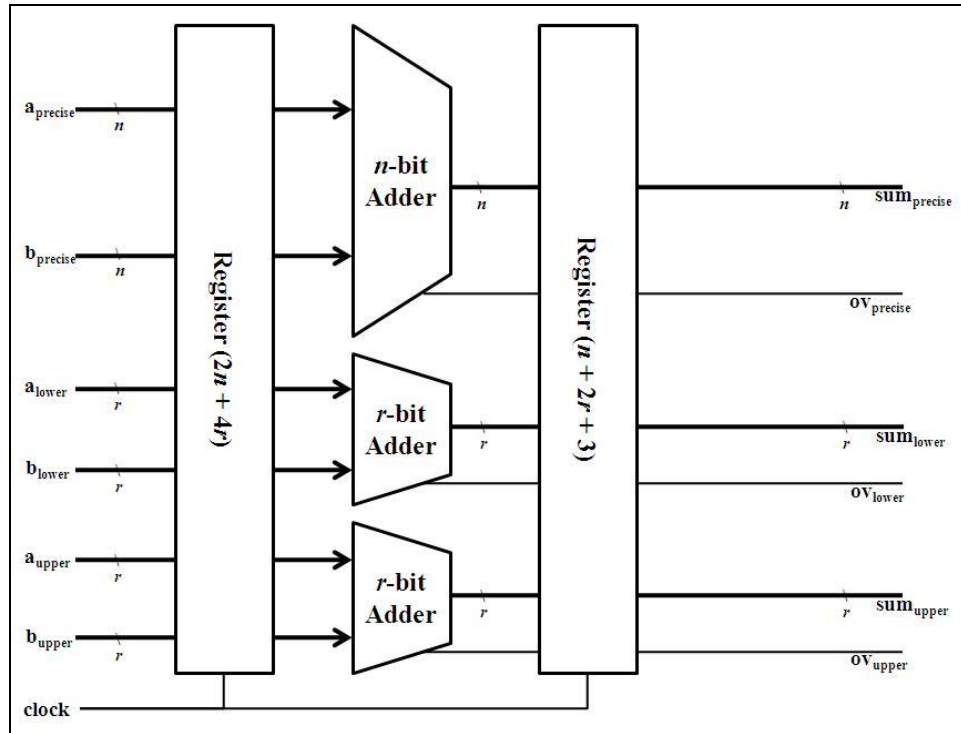


Figure 18. RPR Adder - Operation Block Diagram.

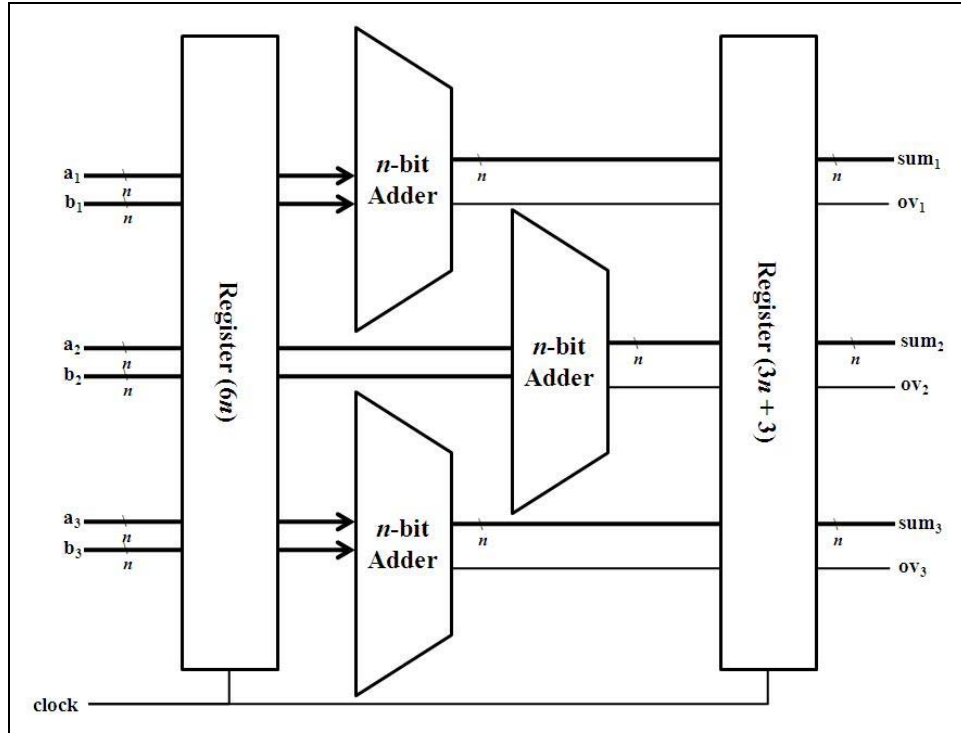


Figure 19. TMR Adder - Operation Block Diagram (compare to Figure 18).

The architecture of the second module, the RPR voter, is shown in Figure 20. In an adder or subtractor, the “Generate RPR Result” block may be replaced by a block that adds $2^{-(r+1)}$ to the lower bound, as noted after Equation (9). For comparison, a traditional (TMR) voter is depicted in Figure 21.

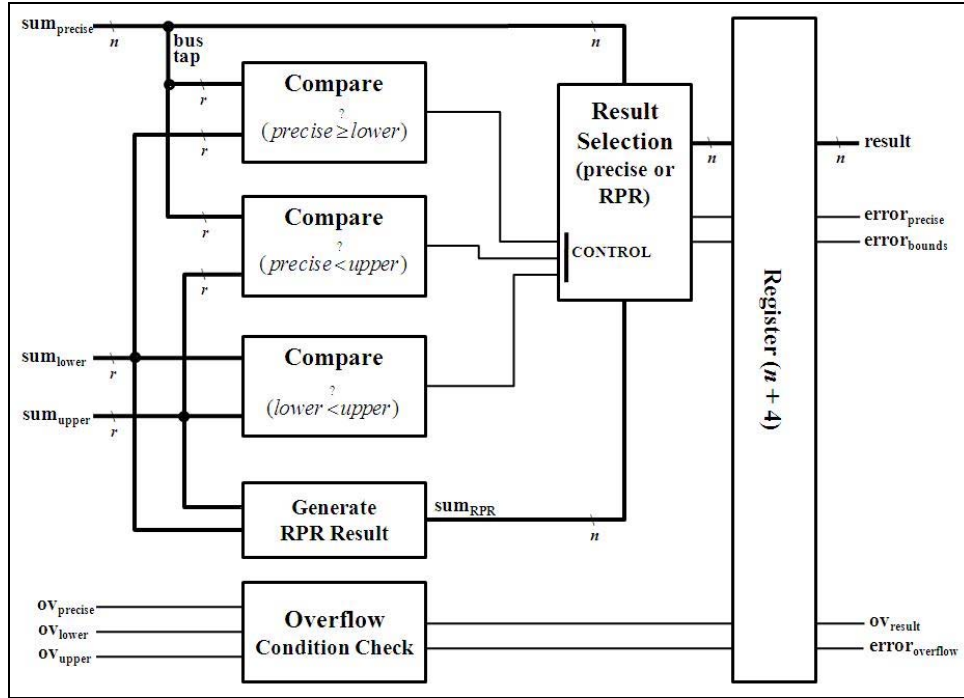


Figure 20. RPR Adder - Voter Block Diagram.

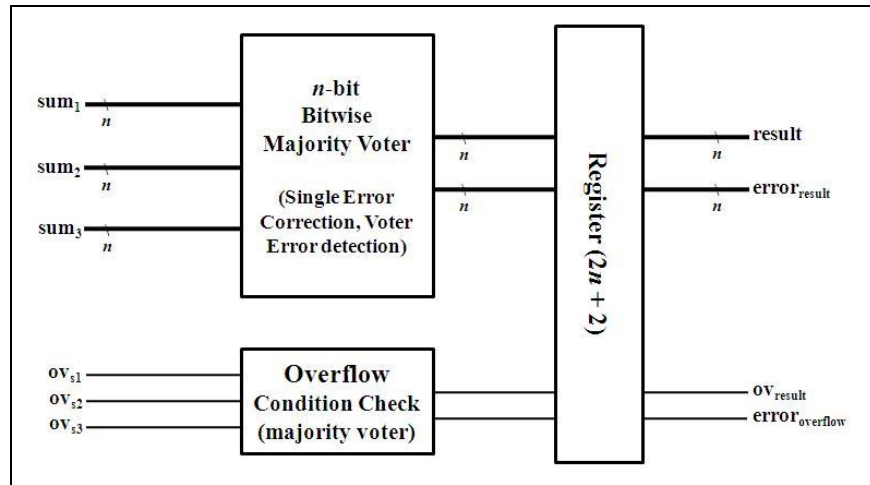


Figure 21. TMR Adder - Voter Block Diagram (compare to Figure 20).

The three comparators in Figure 20 compare two r -bit numeric values; they are not bitwise operations. The comparators must compare numeric values because there is no way to guarantee any individual bits in the upper or lower bound results should be the same as in the precise result. A carry-out generated between the lower bound and precise

result or precise result and upper bound may propagate anywhere from one to r bits toward the MSB of the result; therefore there is no way to predict how many bits (zero to r) of the bound results and precise results should match.

As defined in this work, the correct precise sum will always be greater than or equal to the sum of the lower bounds of the operands, and it will always be less than the sum of the upper bounds. The output of each comparator is a control signal that tells the RPR result selection block whether there was an error in the comparison result (e.g., ‘1’ if the precise result was less than the lower bound). The RPR result selection block checks the output from each compare operation, and selects the best final result to report based on Table 3. The benefit of comparing the upper bound to the lower bound in addition to comparing the precise result to each bound is that if there is an error in one of the precise-result comparisons *and* in the upper-lower bound comparison, then assuming a single-error scenario, the error is in the bound calculation and the precise solution is in fact correct. The RPR selection logic output can also be modified to report the result and a single flag, instead of two. The single flag indicates whether the final output is the precise result or the RPR result (since both final result options have precision n , they are indistinguishable without a separate report). Reporting errors is important both for locating areas of the processor that need to be reconfigured due to SEU, and for identifying the sources of imprecise calculations that may affect system performance.

Precise < Lower	Precise $\geq$ Upper	Lower $\geq$ Upper	Result Reported
0	0	0	Precise
0	0	1	X*
0	1	0	RPR
0	1	1	Precise
1	0	0	RPR
1	0	1	Precise
1	1	0	X*
1	1	1	X*

Table 3. RPR Result Selection Logic (* indicates multiple error condition).

The overflow condition check in Figure 20 is a standard bitwise majority voter with single-error correction and voter-error detection, commonly used in TMR (see

Figure 1). This voter is sufficient for the overflow check because if there is no error in the entire RPR operation, then the majority overflow report will be correct. If there is an error in the RPR operation (as detected in the comparators), then overflow reporting is ignored. Existence of an overflow condition in the sum may be recomputed based on the source of the erroneous result calculation – e.g., if the sum lower bound is determined to be in error and the overflow checker has only two out of three signals high, the precise result most likely has *not* caused overflow, and may be used without issue. The exact nature of the error conditions is revisited in the section on general RPR voter logic.

4. Comparing RPR and TMR Implementations

The main advantage of RPR over TMR is that it requires less space on an FPGA while still providing the same accuracy of computation in a no-error situation, and accuracy within a certain tolerance when errors do occur. This reduction in area means that using RPR instead of TMR to protect a circuit requires less power – or, conversely, more functionality may be obtained using an FPGA of the same area operating with the same amount of power. Snodgrass showed in [3] that an implementation of the CORDIC algorithm protected using TMR requires two to three times the power required by the same process protected using RPR. The FPGA area required by the components and complete circuits for several RPR and TMR implementations of a simple addition operation are presented in Table 4.

Redundancy Type	Precision (or Degree)	Slice Count		
		Operation	Voter	Complete*
TMR	64	99	65	163
RPR	32/64 (0.5)	67	115	181
RPR	16/64 (0.25)	51	95	146
RPR	8/64 (0.125)	43	58	100
TMR	32	51	33	83
RPR	16/32 (0.5)	35	78	114
RPR	8/32 (0.25)	27	42	68
TMR	16	27	17	43
RPR	8/16 (0.5)	19	34	52

*Complete circuit area is computed independently of “Operation + Voter”

Table 4. Area Required By TMR and Representative RPR Addition Experiments.

The first significant result of these experiments is that using RPR to protect an operation does *not* guarantee less area (and lower power) than TMR in every case. The slice count for the complete implementations of 0.5 RPR at *any* original precision are greater than the slice count for the TMR implementation of the same precision. Furthermore, it is evident that this is due entirely to the large amount of logic required for the RPR voter versus the logic required for the TMR voter. The RPR addition *operation* module is always smaller than the TMR addition *operation* module by a significant amount. Some calculated relative FPGA area requirements of TMR and RPR for various n and r are listed in Table 5.

Redundancy Type	Precision (or Degree)	Percent of Circuit Occupied by Voter	Ratio of RPR/TMR Operation Size	Ratio of RPR/TMR Voter Size	Ratio of RPR/TMR Total Size
TMR	64	39.9%	(1.00)	(1.00)	(1.00)
RPR	32/64 (0.5)	63.5%	0.68	1.77	1.11
RPR	16/64 (0.25)	65.1%	0.52	1.46	0.90
RPR	8/64 (0.125)	58.0%	0.43	0.89	0.61
TMR	32	39.8%	(1.00)	(1.00)	(1.00)
RPR	16/32 (0.5)	68.4%	0.69	2.36	1.37
RPR	8/32 (0.25)	61.8%	0.53	1.27	0.82
TMR	16	39.5%	(1.00)	(1.00)	(1.00)
RPR	8/16 (0.5)	65.4%	0.70	1.26	1.21

Table 5. FPGA Area Comparison for RPR and TMR Adders.

Table 5 shows 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 an RPR addition process to be smaller than the analogous TMR modules, the degree of RPR must be less than 0.25. The performance impact of using reduced-precision results in systems protected with very small degrees of RPR (e.g., 8/64) are investigated in Chapter IV.

C. MULTIPLICATION

1. Computation Error in Multiplication

Multiplication in a digital computer is fundamentally made up of a series of addition operations. It is frequently implemented as an accumulating process where a computer begins with a register set to zero, then shifts and adds the multiplicand a to the register some number of times, where the addition and shift amount are dictated by the digits in the multiplier b [27]. Because performing multiplication is mathematically equivalent to performing addition repeatedly, an initial hypothesis may be that the magnitude of the total error ε accrued in a multiplication operation depends on the number of times addition is performed – i.e., the value of the multiplier. However, scaling the operands prior to performing the multiplication minimizes this error accumulation. Without knowing the exact values of the multiplier and multiplicand, it is still possible to bound the error ε on an RPR multiplication operation based on the precisions n and r of the operands, as will be shown in this section.

Scaling multiplication operands such that $-1 < (a, b) < 1$ guarantees that the product c will always have absolute value equal to or smaller than both operands – that is, c will be closer to zero on a number line than either of the operands. To understand this point, it is helpful to view the entire set of representable numbers in a fractional fixed-point binary system as depicted in Figure 22. There is a finite set of representable numbers between -1 and 1 (non-inclusive), and the smallest distance between any two numbers on this number line is 2^{-n} . The number line in Figure 22 is logarithmic in base 2 to show that the *relative* error between any two numbers of precision n is greater when those numbers are close to zero.

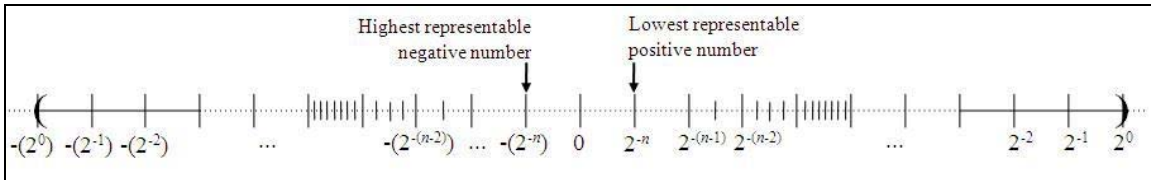


Figure 22. Number Line for Multiplication in Fractional Fixed-Point.

Multiplying any two numbers of precision n in a digital computer generates a product of precision $2n$. When multiplying binary integers, the product extends toward infinity (or negative infinity), increasing the order of the most significant bit (MSB). The advantage of scaling operands such that they fall between -1 and 1 is that the result always falls within this range as well: the product extends toward zero, *decreasing* the order of the *least* significant bit (LSB). To obtain a final product with original precision n (versus $2n$), the exact product p is approximated by adding $\frac{1}{2} \times 2^{-n} = 2^{-(n+1)}$ to it and throwing away the n LSB (bits $n-1$ down to 0). The bound on error acquired in fraction multiplication using this approximation is, as specified in [26],

$$p = (a * b) + \varepsilon, \quad |\varepsilon| \leq 2^{-(n+1)}. \quad (10)$$

Multiplication generates computational error of the same maximum magnitude regardless of the sign of the operands. In fact, multiplication is mathematically the same operation on any combination of negative or positive numbers; the only difference is the sign of the product. The sign of the product is determined by the XOR function of the signs of the operands: if one operand is negative and one is positive, the product is negative; if the signs of the operands are the same, the product is positive.

Although multiplication is mathematically the same for positive and negative numbers, in digital computers the operation is more complicated due to the common representation of numbers in two's complement form. When numbers in two's complement need to be multiplied, they are often transformed to sign-magnitude form and multiplied as positive numbers with separate sign determination. Alternatively, numbers may be multiplied in two's complement form – but corrective factors must be added to counter the effects of the complement representation. One approach is to multiply the two's complement operands of precision n without the sign bit (MSB), and to examine the sign bit separately. If one of the operands is negative, add the two's complement of the *other* operand, shifted left by n places (i.e., multiplied by 2^n). If both operands are negative, no change need be applied. Details on this and other more complicated methods of correcting multiplication in two's complement are available in [27]. In all cases, the amount of logic required to find the correction factor when one operand is negative – some expression of the two's complement of the positive operand –

is at least as much if not more logic than that which is required to convert the negative operand to sign-magnitude form prior to the multiplication. Therefore, for the purposes of this research, it is assumed that multiplication operations receive input in sign-magnitude form. The magnitude multiplication is then performed as dictated before Equation (10), and the sign S_p of the result is determined using an XOR gate with the signs S_a and S_b of the operands as input. The magnitude and the sign are checked separately for errors in the voter, and the correct sign is prepended to the final result.

2. Upper and Lower Bound Determination for Multiplication

The same mathematical convention is used to define upper and lower bounds of multiplication operands as is used for addition/subtraction operands: the upper bound is the next number to the right on a number line, and the lower bound is the next number to the left on a number line (Figure 23).

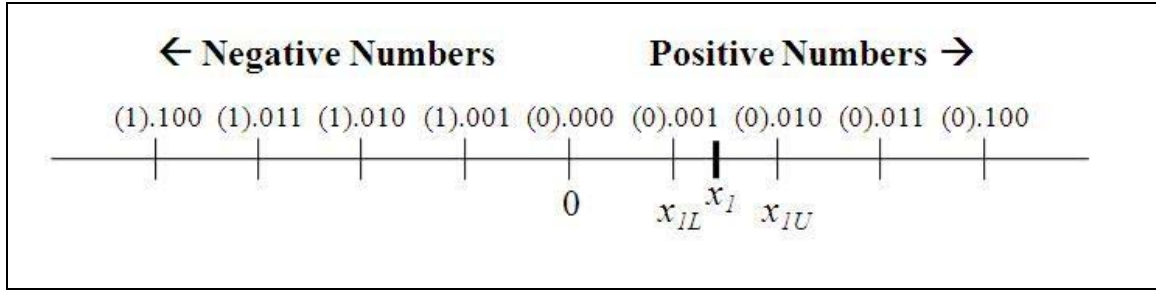


Figure 23. Upper and Lower Bounds of Fixed-Point Sign-Magnitude Numbers.

The relationship of the products of operands' upper and lower bounds to the precise product is explained in the following paragraph for the cases of positive and negative numbers. However, since multiplication of numbers in sign-magnitude representation is performed on only the magnitude of the operands, one need only consider the positive cases (numbers to the right of zero, inclusive) for implementation purposes.

Given a multiplication problem of precision n , the operand lower bounds (a_L, b_L) and upper bounds (a_U, b_U) are found in the same manner as they are for addition and

subtraction. That is, the operand a of precision n will always have RPR lower and upper that are the next lower and higher representable numbers of precision r such that

$$\begin{aligned}
& a_L < a < a_U, \text{ where} \\
& a - a_L = \varepsilon_{a,L}, \quad 0 \leq \varepsilon_{a,L} < 2^{-r}, \\
& a_U - a = \varepsilon_{a,U}, \quad 0 < \varepsilon_{a,U} < 2^{-r}, \text{ and} \\
& |\varepsilon_{a,L} - \varepsilon_{a,U}| = 2^{-(r+1)}.
\end{aligned} \tag{11}$$

As discussed in the previous section, the nature of multiplying fixed-point fractional numbers is such that the product p is always smaller (closer to zero) than either of the operands. This property can be applied to the RPR operation: multiplying the smaller (lower, if positive) bounds of two operands will always produce a result that is smaller than the precise result. Likewise, the product of the (positive) upper bounds will always be larger in magnitude than the precise result. The relationships are described mathematically as

$$\begin{aligned}
& p_L \leq p < p_U, \text{ where} \\
& \text{IF } a, b > 0: \quad p_L = a_L \times b_L \text{ and } p_U = a_U \times b_U \\
& \text{IF } a > 0, b < 0: \quad p_L = a_U \times b_L \text{ and } p_U = a_L \times b_U \\
& \text{IF } a < 0, b > 0: \quad p_L = a_L \times b_U \text{ and } p_U = a_U \times b_L \\
& \text{IF } a, b < 0: \quad p_L = a_U \times b_U \text{ and } p_U = a_L \times b_L
\end{aligned} \tag{12}$$

The magnitude of the difference between the upper and lower bounds of an RPR product will always be on the order of 2^{-r} . Therefore, in multiplication the product may be tested the same way as the sum is tested in addition: the first r bits of the precise product should always be a number that is greater than or equal to the product of the lower bounds, and less than the product of the upper bounds. It is important to note that this is only true if the comparison is done *before* the RPR bound products are rounded as described above Equation (10). This is because the addition of $2^{-(r+1)}$ to the lower bound product sometimes produces a carry that makes the first r bits of $p_L (= a_L \times b_L)$ equal to the first r bits of $p_U (= a_U \times b_U)$, making *both* bound results greater than the precise product. One example of this failure is demonstrated in Figure 24.

$ \begin{array}{r} .0011 \text{ (a}_L\text{)} \\ \times .0100 \text{ (b}_L\text{)} \\ \hline 0000 \\ 0000 \\ 0011 \\ + 0000 \\ \hline .00001100 \text{ (2r)} \\ \\ + \quad (1) \\ \hline .0001 \text{ (p}_L, r\text{)} \end{array} $	$ \begin{array}{r} .001110010 \text{ (a)} \\ \times .010000110 \text{ (b)} \\ \hline 000000000 \\ 001110010 \\ 001110010 \\ 000000000 \\ 000000000 \\ 000000000 \\ 001110010 \\ + 000000000 \\ \hline .000011101110101100 \text{ (2n)} \\ + \quad (1) \\ \hline .00001111 \text{ (p, n)} \end{array} $	$ \begin{array}{r} .0100 \text{ (a}_U\text{)} \\ \times .0101 \text{ (b}_U\text{)} \\ \hline 0100 \\ 0000 \\ 0100 \\ + 0000 \\ \hline .00010100 \text{ (2r)} \\ \\ + \quad (1) \\ \hline .0001 \text{ (p}_U, r\text{)} \end{array} $
--	---	--

Figure 24. Erroneous Bounds Due To Rounding Products.

As with addition, there are special cases in multiplication that need to be discussed. In multiplication these are small operands (those that are close to zero, or less than 2^{-r}) and cases involving true zero.

a. Special Cases Involving Zero

The zero property of multiplication states that if the product of two numbers is zero, then at least one of the operands must be zero. Conversely, in arithmetic zero times any number is zero. Particularly in systems where repetitive multiplication is performed on values that may equal zero (e.g., gain multiplication in a control feedback loop), it is preferable to test the operands for zero first in order to avoid incorrect accrual of nonzero values. If either operand is identically zero, then zero (to precision n and r) should be supplied directly as both the precise result and its bounds, rather than executing the operation.

b. Special Cases Involving Small Numbers

A very small number in RPR multiplication is defined as magnitude less than 2^{-r} but greater than 2^{-n} . There are two main concerns that arise when dealing with

very small numbers in RPR multiplication: (1) the presence of zero in bound calculations when neither operand is truly equal to zero, and (2) the generation of an RPR result that is several orders of magnitude larger than the precise result when both precise operands are very close to zero (i.e., when the precise result has a value much smaller than the smallest representable RPR value 2^{-r}).

When one (or both) operands has magnitude less than 2^{-r} , that operand's lower bound will be zero. When either lower bound is equal to zero, the lower bound p_L of the product is necessarily also zero. If the RPR product (used when an error is found in the precise product) is obtained by finding the arithmetic mean of the upper and lower bound products, then the RPR result will be greater than or equal to $2^{-(r+1)}$ and the false zero will not propagate. However, the lower bound on the new product will also be zero (since the product is always smaller in magnitude than either factor); in this case the next operation must continue to avoid obliterating the very small product by ensuring that true zero is not propagated as the result.

When both operands are of magnitude $2^{-(r+1)}$ or smaller, the first $2r$ digits of the product will all be zero. The RPR result in this case will be equal to $2^{-(r+1)}$, which may be up to $(r - n)$ orders of magnitude larger than the precise result. However, the precision of fixed-point multiplication is such that if there is an error in the precise result that is detectable through comparison with the upper bound 2^{-r} and lower bound 0, the RPR result will have less error than the incorrect precise result and it will be the best possible solution in that case. A value of “true” on a status signal indicating that the RPR result was used (i.e., there was an error in the precise result) in multiplication also alerts a designer that if the operands were smaller than 2^{-r} , the RPR product supplied may be orders of magnitude larger than the (correct) precise product.

There is some discussion necessary on the generation of the RPR result. The RPR result is found after the computation of the precise, upper and lower bound products. The two principal ways to obtain an RPR product are to find the arithmetic mean of the full upper and lower bound products (giving a result of precision $2r$), and to find the arithmetic mean of the rounded upper and lower bound products (giving a result

of precision r). The first of these methods gives the best result – i.e., the result with the least error relative to the correct precise result – but it requires more area to implement since it involves operands of size $2r$ versus size r . The impact of obtaining the best RPR result is shown in the demonstration of RPR multiplication.

3. Demonstration of RPR in Multiplication

A complete RPR multiplication implementation contains an operation module and a voter module, as with addition or subtraction. The main difference between multiplication and addition/subtraction is the set of signals passed from the operation to the voter. The multiplication voter uses the intermediate results of the upper and lower bound computation – of precision $2r$ – to compare with the first $2r$ bits of the precise product and/or to generate the RPR result. This is depicted in Figure 25.

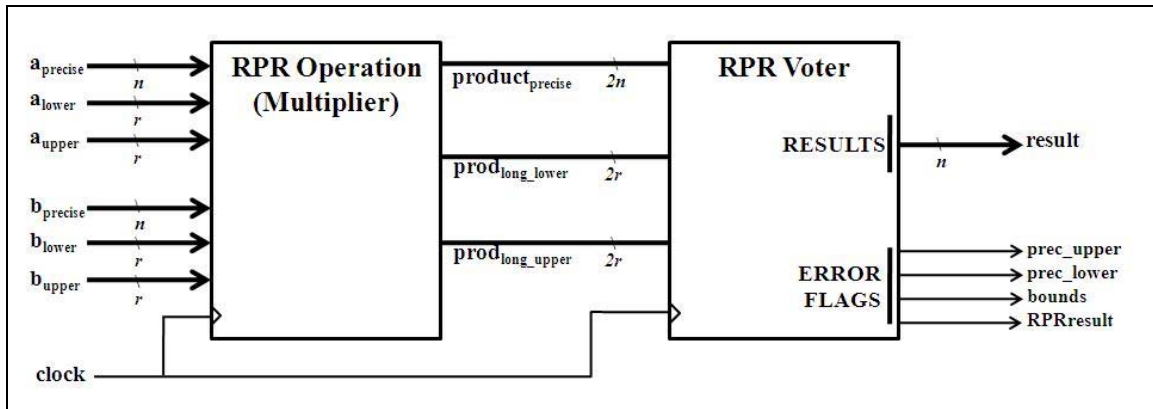


Figure 25. RPR Multiplier Top-Level Block Diagram.

Using $2r$ bits instead of r bits to generate the RPR result p_{RPR} increases the size of the voter, the impact of which is shown in the following section on comparing RPR multiplication with TMR multiplication. However, this allows the most accurate RPR result to be generated and eliminates the comparison issues associated with rounding the product lower bound.

The operation module structure for multiplication is essentially the same as the operation module for addition. The differences between addition and multiplication designs include the lack of any overflow flags (since the fractional product always falls

within the allowed range $-1 < c < 1$) and the increase in output ports required due to the propagation of the $2r$ -width bound results. A block diagram showing these details of the multiplication operation module structure is drawn in Figure 26. For comparison, a TMR multiplication operation module is shown in Figure 27.

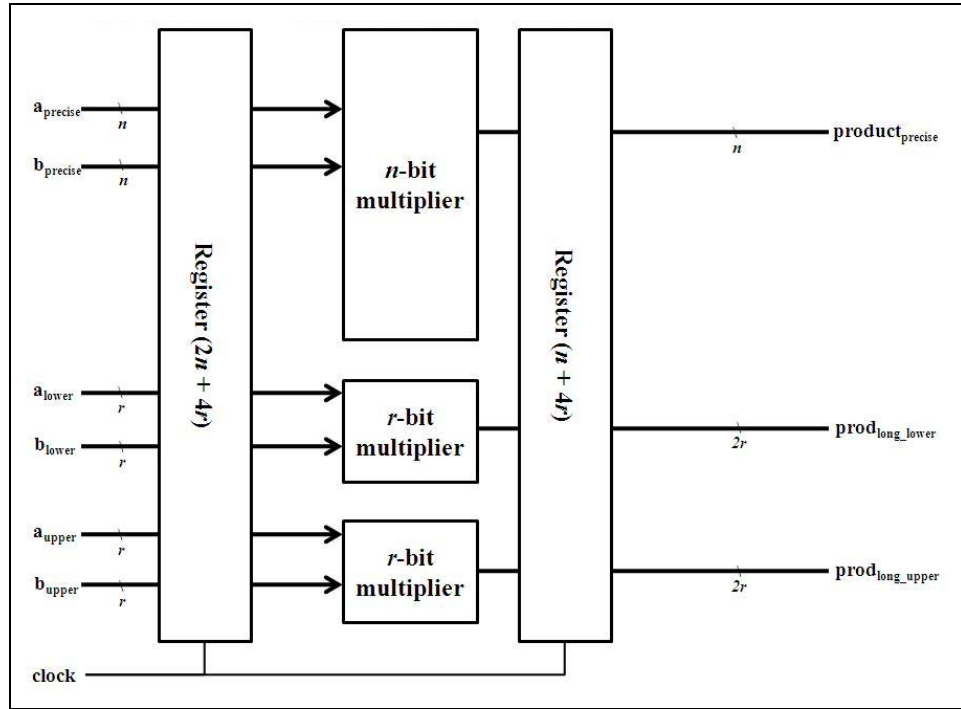


Figure 26. RPR Multiplier - Operation Block Diagram.

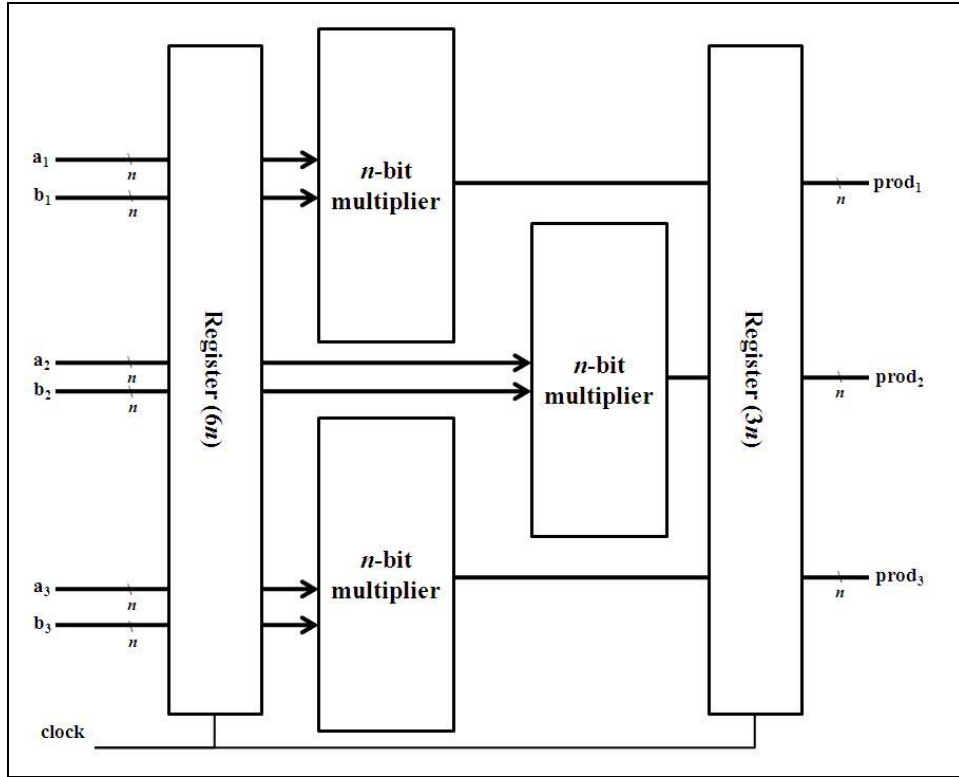


Figure 27. TMR Multiplier - Operation Block Diagram (compare to Figure 26).

The second RPR module, the voter, is almost the same in structure as the RPR voter for addition. The single bitwise majority voter checks the sign bit of the product instead of the overflow flag, so that circuitry remains the same. The RPR voters for addition and multiplication differ primarily in the comparators and in the generation of the RPR result. Since multiplication of two r -bit numbers generates an intermediate product of precision $2r$, it is easy and facilitates obtaining the most accurate RPR result to use the long intermediate products of the upper and lower bounds in the RPR voter. The comparators may be either r or $2r$ -bit operations; $2r$ -precision comparators detect more errors (down to 2^{-2r} in magnitude). However, a comparator occupies roughly as many FPGA slices as its precision (i.e., a 32-bit comparator maps to 33 slices on an FPGA), so if area is at a premium and other processes in the system are detecting errors only to r bits of precision, it may be preferable to use only the first r bits of the bound products for testing the precise result in multiplication. This approach is shown in Figure 28.

Regardless of the bound values used for comparison, the long bound products may be used to compute the most accurate possible RPR result p_{RPR} .

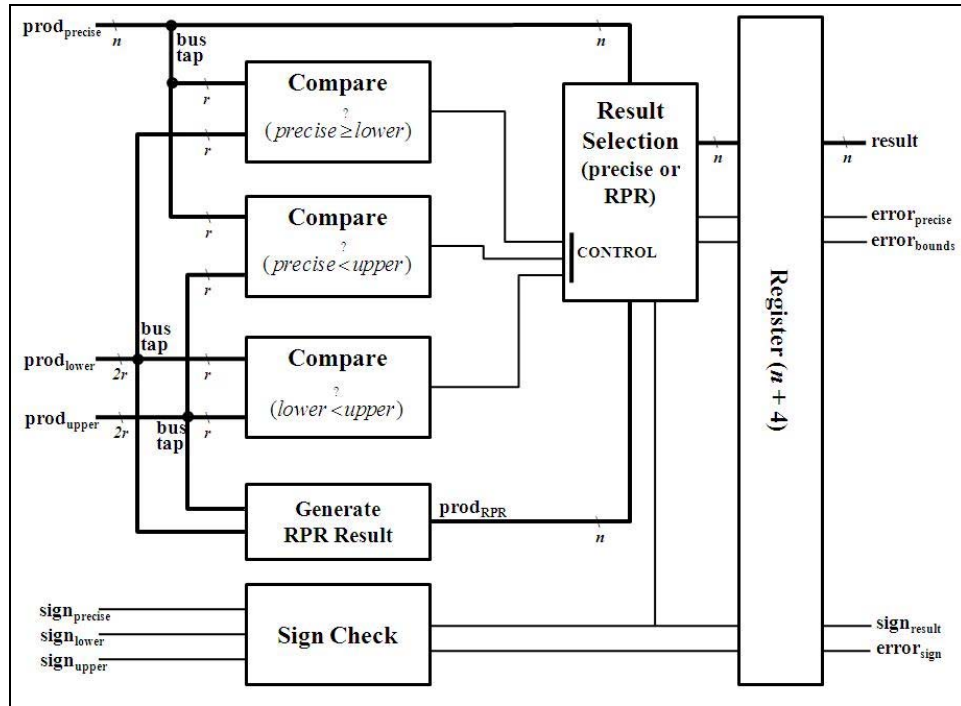


Figure 28. RPR Multiplier - Voter Block Diagram.

The voter module for a TMR operation is the same regardless of the operation being performed, since the voter simply takes three copies of the same result and bitwise-compares them, correcting any single error. The overflow bit check can be used as a sign bit check, although it is not necessary if the sign bit is included in the n -bit product. For comparison with the RPR voter, a TMR voter is shown in Figure 29.

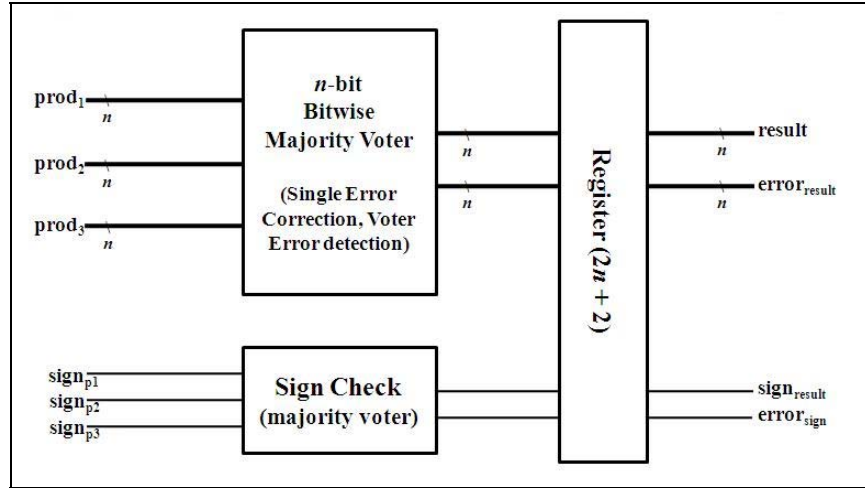


Figure 29. TMR Multiplier - Voter Block Diagram (compare to Figure 28).

The RPR result selection block in the multiplication voter is the same as in the addition voter, both in operation and input (data and control signals), with one exception: the checked/corrected product sign bit is fed into the RPR result selection block separately because it is not checked as part of the comparison process. The bound tests are executed on only the magnitudes of the precise and bound results; the sign bits are checked using a bitwise majority voter, as in TMR.

The other logic block that is different between the RPR voter for multiplication and for addition is the generation of the RPR result. In RPR addition, this block can simply extend the lower bound sum from precision r to precision n , with ‘1’ in the $2^{-(r+1)}$ position and ‘0’ in the remaining $n - (r + 1)$ positions, as noted after Equation (9). However, in RPR multiplication the $2r$ -bit bound products rarely differ by only 1×2^{-2r} . Therefore the RPR result generation block must add the two bound results and shift the sum right one bit to get their arithmetic mean, which is then concatenated with additional 0’s to obtain the n -bit RPR result. This process includes an entire arithmetic operation (addition) that itself was investigated in the previous section as a potential object of RPR application. Using addition in the voter for multiplication both increases the size of the voter and introduces the question of “trusted” operations: if a designer is protecting low-level arithmetic operations with RPR, is it necessary to protect the addition in the RPR “overhead” (the voter) as well? Or is that a calculated risk that a designer will accept?

This is a potential drawback of RPR as compared to TMR as well, which should be considered in addition to the quantitative comparison expounded in the next section.

4. Comparing RPR and TMR Implementations

Since multiplication is essentially a collection of many addition operations, it is reasonable that mapping a multiplication operation to an FPGA should require significantly greater area than an addition operation requires. This is indeed the case; in fact, the difference is so great that the size of the multiplication operation modules in either TMR or RPR implementations fairly dwarfs the size of their respective voter modules. This makes the space and power savings of RPR over TMR a more significant benefit. The FPGA area required by the components and complete circuits for several RPR and TMR implementations of a simple addition operation are presented in Table 6.

Redundancy Type	Precision (or Degree)	Slice Count		
		Operation	Voter	Complete*
TMR	64	6378	64	6436
RPR	32/64 (0.5)	3208	226	3429
RPR	16/64 (0.25)	2400	130	2526
RPR	8/64 (0.125)	2194	101	2285
TMR	32	1622	32	1112
RPR	16/32 (0.5)	816	114	926
RPR	8/32 (0.25)	610	85	684
TMR	16	413	16	427
RPR	8/16 (0.5)	205	77	274

*Complete circuit area is computed independently of "Operation + Voter"

Table 6. Area Required By TMR and RPR Multiplication Experiments.

The size of the TMR voter modules for multiplication is the same as for addition; this makes sense because the TMR voting operation is the same regardless of the type of process that produces the results the voter is operating. In the multiplication experiment, the addition required to generate the RPR result was considered to be a trusted operation, so RPR or other protection was not applied to it. In practice this assumption would need to be revisited. Unlike the voter, a TMR multiplier operation module is ten to fifty times larger than a TMR adder module for inputs of the same size. In RPR, the multiplication operation modules are also many times larger than the addition modules of the same-size

operands, but the space saved by RPR multiplication over TMR multiplication is greater: an RPR multiplication module requires 1/3 to 1/2 the FPGA slices of a TMR multiplication module, depending on the degree of RPR. The RPR addition operation requires 40%-70% of the area of the TMR addition operation. This implies that multiplication, being a more complicated *operation*, is better suited for RPR fault-tolerance methods than is addition.

In order to examine better the relative impact of the operation and voter modules, an RPR/TMR area comparison by percent and ratio is presented in Table 7.

Redundancy Type	Precision (or Degree)	Percent of Circuit Occupied by Voter	Ratio of RPR/TMR Operation Size	Ratio of RPR/TMR Voter Size	Ratio of RPR/TMR Total Size
TMR	64	1.0%	(1.00)	(1.00)	(1.00)
RPR	32/64 (0.5)	6.6%	0.50	3.53	0.53
RPR	16/64 (0.25)	5.1%	0.38	2.03	0.39
RPR	8/64 (0.125)	4.4%	0.34	1.58	0.36
TMR	32	2.8%	(1.00)	(1.00)	(1.00)
RPR	16/32 (0.5)	12.3%	0.50	3.56	0.83
RPR	8/32 (0.25)	12.4%	0.38	2.65	0.62
TMR	16	3.7%	(1.00)	(1.00)	(1.00)
RPR	8/16 (0.5)	28.1%	0.50	4.81	0.64

Table 7. FPGA Area Comparison for RPR and TMR Multipliers.

Of particular interest in Table 7 are the “percent of circuit occupied by voter” values. Because the multiplication operation is so large, the least and greatest relative sizes of the voter in any TMR or RPR operation range from 1% to 30%. All cases of RPR or TMR multiplication have less of the circuit devoted to the voter than any of the cases of addition (in which the portion of the circuit occupied by the voter ranges from 40% to 68%). This means that even in 8/16 RPR, which would not be used in most practical applications, the RPR-protected circuit saves significant area compared to the TMR-protected circuit ($1.00 - 0.64 \approx 35\%$).

Although RPR appears to give the expected benefit of 1/3 to 1/2 space and power savings when applied to multiplication, it is still worth noting that the *voter* modules in RPR multiplication are extremely large compared to the TMR voter modules. One of the

drawbacks to RPR is that not only do the voter modules require more thought and design than the simple bitwise majority-voting of TMR, but also they almost always take up more space. A TMR voter needs only to compare each bit of three inputs with a few logical gates; an RPR voter must execute numerical (value) comparisons of three r -bit numbers, generate the RPR solution (which may include additional arithmetic operations), and choose the best output based on the result of the bound value tests. This complexity is evident in multiplication, where every RPR voter is larger than its analogous TMR voter –sometimes the RPR voter is three to five *times* greater. Since all the voter circuits are still very small and simple compared to the multiplication operation circuits, this complexity of the RPR voters is not as critical as is in the simpler operation of addition. However, it cannot be overlooked. It may be possible to optimize further and/or standardize the design of RPR voters in order to make them more comparable to the TMR voter, which operates on each result bit independently. This is another point discussed in the additional notes on RPR voter design at the conclusion of this chapter.

D. DIVISION

1. Computation Error in Division

Division is the most complicated arithmetic operation, and often presents a special problem in digital computing. Most importantly for this discussion, there is no synthesizable “divide” operator (e.g., $a / b = q$) in the standard VHDL arithmetic libraries analogous to add (+), subtract (–), or multiply (*). The division operation must instead be synthesized as a clocked process made up of successive subtractions or additions, or as multiplication of a reciprocal of the divisor ($a * 1/b$). This is explored further in the division implementation discussion.

Another significant point regarding division in computers is that in fractional fixed-point arithmetic – the convention chosen for this research – division operations are only meaningful when $a < b$ (i.e., the divisor is greater than the dividend). This is because in cases where the divisor is smaller than dividend, the resulting quotient q

would fall outside the permitted range (i.e., $q < -1$ or $q > 1$). Therefore to execute the operation at all, the constraint $a < b$ must be applied to the domain of possible operands [28].

Assuming that the constraints are met and a suitable implementation is found, the computation error in division is similar to that of multiplication. The quotient q of a dividend a and a divisor b is traditionally calculated one digit at a time, beginning with the MSB. The computation of a n -bit quotient is complete when $n+1$ digits have been calculated and rounded to n bits by adding ‘1’ to the LSB, which is the same rounding process that is applied to multiplication. This gives the computation equation

$$c = \frac{a}{b} + \varepsilon, \quad \varepsilon \leq 2^{-(n+1)} \quad (13)$$

where ε is the rounding error. As derived in [26], the upper bound on the absolute value of the difference between the computed product $b*q$ and the dividend a is therefore

$$|bq - a| \leq |b| 2^{-(n+1)} \quad (14)$$

where b is the fractional divisor.

2. Determining RPR Bounds for Different Division Implementations

The major methods of implementing division in a computer include basic shift-and-subtract schemes (including restoring and nonrestoring solutions); variations on the basic schemes such as modular, high-radix (>2) or array dividers; and division by convergence. Shift-and-subtract division is difficult to pipeline for quick computation due to the need for a conditional subtractor for each quotient bit [29] and therefore convergence schemes are more often utilized. Methods of division by convergence comprise successive approximation by repeated multiplication, Newton-Raphson iteration, and division by reciprocation [27], [28], [29].

a. Convergence by Repeated Multiplication

In division by convergence through repeated multiplication, the quantity a/b is multiplied by a sequence of factors such that the denominator ($b*x_1*x_2*...*x_f$) converges to 1. This causes the numerator to converge to the quotient q . The next factor

(x_{i+1}) is the 2's complement of the current denominator $(b * x_1 * \dots * x_i)$. The repeated multiplication method has quadratic convergence: the number of multiplications required to obtain precision n in the quotient is m , where

$$m = \lceil \log_2 n \rceil. \quad (15)$$

Because this method consists of several multiplications, it may seem that the total error accumulated is an aggregate of the computation error accrued in each multiplication. However, if the entire process is carried out as a block operation in both precise and RPR versions at the double precision ($2n$ or $2r$, respectively) of the intermediate products, then the computation error will only occur in the rounding of the final result. The final result is obtained after m_n or m_r iterations, and the precise and bound results may be compared as in regular multiplication. This leads to a suggestion that RPR may be applicable to block operations as well as to individual elements – in fact that RPR may be applied at the process level with greater efficiency than at the individual operation level. This is discussed further in the section on compound operations (matrix multiplication and FFT computation).

The added value of several multiplication operations and only one voter operation at the end of the multiplication increases the relative space and power savings of RPR over TMR. The quantitative benefit of using RPR depends on the degree of RPR, but will approach the ratios found in column six of Table 7, the comparison of multiplication operation sizes.

b. Division by Reciprocation

Division by reciprocation consists of finding the reciprocal $1/b$ of the divisor b , and then multiplying that reciprocal by the dividend a to obtain the quotient q . The reciprocal is commonly found using Newton-Raphson iteration [29], which begins with an initial estimate $x^{(0)}$ of some precision k and successively applying the formula

$$x^{(i+1)} = x^{(i)} (2 - x^{(i)} b) \quad (16)$$

from [28]. The initial estimate $x^{(0)}$ is either stored as a constant or computed based on b , to precision r or n depending on whether the operation is precise or RPR. Finding the

reciprocal requires the same basic operations as those that the repeated multiplication approach requires: finding 2's complement and performing multiplication. The Newton-Raphson method converges quadratically (the same degree as the repeated-multiplication approach), and the computation error is determined in a fashion similar to that of the repeated-multiplication approach as well. The main differences between division by reciprocation and repeated multiplication are in the initial estimate and final multiplication (by the dividend) in the reciprocation method. The number of multiplications required in each approach is the same, although in the reciprocal method they are performed successively and in the repeated multiplication method they are performed independently on the numerator and denominator. The successive multiplication in the reciprocal method also increases the potential cumulative error of the process – however performing the multiplication at double precision keeps the intermediate results accurate and alleviates this concern. Ultimately, division by reciprocation is another process that is more accurately computed using parallel block operations in precision n and r .

3. Comparing RPR and TMR Implementations

If the desired implementation approach for division is any method of convergence (as opposed to a shift-and-subtract method), then the number of operations required to complete the division can be estimated as a function of the precision required. For example, if the required precision of a quotient is $n = 32$, the number of iterations m of the repeated multiplication method is $\log_2(32) = 5$, as determined by Equation (15). This means that the number of multiplications is $2 \times 5 - 1 = 9$ (numerator and denominator in iterations 1-4 and numerator only in iteration 5), and the 2's complement of the denominator must be found five times. Using the operation module size ratios in Table 7 as a guide, it follows that for 8/32 RPR, the TMR implementation would be about 2.5 times the size of the RPR implementation on an FPGA. Depending on the capacity of the FPGA and the desired computation speed, the division operation may be implemented using an architecture with fewer multiplication modules; the relative savings of RPR would be maintained regardless of how many multipliers are used.

The appearance of several successive subtraction, multiplication or 2's complement operations as part of a division operation highlights the question of whether to treat blocks of arithmetic operations as a single module to which one applies RPR. Scenarios so far have indicated that as RPR is applied to more complicated operations, its benefit increases (i.e., size compared to TMR decreases). However, the complicating factor of applying RPR to block operations is the determination of upper and lower bounds on the results of processes that involve multiple inputs and/or multiple iterations. The following section on compound operations addresses this and other questions.

E. COMPOUND OPERATIONS

A single adder or multiplier is rarely found in a configuration where it is completely isolated from all other arithmetic operations in a computer. In fact, computers were developed to execute many successive arithmetic operations – whether in a recursive or an iterative architecture. However, examining a many-part operation from the perspective of applying RPR generates questions: what are the upper and lower bounds for the result of the overall operation? Is there a performance benefit to be gained from testing intermediate results for errors? Can the chosen degree of RPR be maintained throughout the operation? Conversely, what is the impact of error (loss of precision) accumulated over the entire operation?

The first question to address is whether upper and lower bounds can be computed directly for the results of operations more complicated than single instances of addition, subtraction, multiplication, or division. The most straightforward cases are when all operands are nonnegative. If an entire problem can be scaled such that it both operates on and produces only nonnegative numbers, then the case becomes simple: upper and lower bounds of the result may be obtained by performing the given operation on the upper and lower bounds of the operands, respectively. In unsigned fractional representation, any combination of addition, subtraction, multiplication and (constrained) division can be computed on precise operands to produce the precise result, on operands' lower bounds to generate the lower bound of the result, and on operands' upper bounds to generate the upper bound of the result.

Unfortunately, this rule does not hold for multiplication when one or more operands is negative. When negative numbers are introduced, addition and subtraction may proceed normally since the relation among those operands' and results' upper and lower bounds does not change with sign. However, when dealing with negative numbers in multiplication the upper and lower bounds of the operands must be rearranged as dictated in Equation (12) before being fed into reduced-precision multipliers. This rearranging ensures that the reduced-precision operations generate appropriate upper and lower bounds for the correct full-precision product. In a process that contains several successive multiplications, the signs of all operands must be tested before the operands are used so that their bounds are assigned to the appropriate reduced-precision operation inputs. In software implementations, this results in a series of statements conditioned on the signs of the original (full-precision) operands. In hardware implementations, the sign bits of the operands can be used to control a selector at the entry to each reduced-precision multiplier (upper and lower bound) that chooses the appropriate inputs for the multiplier.

The second question to answer when considering compound operations for RPR is whether there is benefit in testing intermediate precise and bound results, or if only the final result should be tested. As shown previously, adding a voter to test an RPR addition result generally doubles the number of FPGA slices required to execute the addition operation. Multiplication is a larger operation and therefore the relative significance of its RPR voter size is smaller, but the absolute RPR voter size in multiplication is even larger than in addition. Any benefit of testing intermediate results in RPR processes must be considered against the additional space it requires.

The most significant benefit of testing intermediate RPR results is in locating the source of an error more quickly, thereby enabling faster recovery to error-free status. Particularly in parallel-processing designs, where many arithmetic modules are used to reduce computation time, it may be both desirable and possible to reconfigure part of the FPGA that has been affected by an SEU while processing continues on the unaffected partitions of the hardware. However, in order (1) to confirm that an incorrect result was

caused by a configuration error (as opposed to a data error) and (2) to identify which partition is affected in the case of a configuration error, it is necessary to locate the origin of the computation error.

If a compound operation is executed using modules in multiple partitions of an FPGA, testing consecutive intermediate results is the only indirect method of narrowing the range of possible error locations. The primary method of discovering, locating and correcting configuration errors on an FPGA is by comparing the FPGA operational configuration directly to a stored (protected) copy of the configuration at regular time intervals, and reprogramming the FPGA with the stored copy of the configuration when a discrepancy is found. If intermediate result checking can be used as a passive way to locate whether, when and where configuration errors occur in a circuit, it may enable more efficient operation of the FPGA. RPR voters may be inserted at the output of any single operation where a precise result and two bound results are generated, and additional signals may be set to trigger reconfiguration if two consecutive errors are encountered. However, the quantitative benefit (e.g., computation speedup) of detecting and locating errors in this manner needs to be explored through further study that focuses specifically on that issue.

The third question generated while designing RPR protection for compound operations is how to quantify the error – or loss of precision – accrued over multiple calculations. This is particularly important in RPR-protected systems because the degree of RPR represents the greatest acceptable loss of precision; any further reduction in precision due to accumulated error may cause a significant decline in system performance. The amount of error accumulated in compound operations depends primarily on two things: the number of addition operations in the set, and the internal precision of the operation – i.e., the number of guard bits in each calculation. Examples of the error accumulated in two types of compound operations are shown in the following subsections on matrix multiplication and FFT computation.

1. Matrix multiplication

Matrix multiplication consists of computing the *inner product* of two vectors for each matrix element. The inner product is defined as

$$p = \sum_{i=1}^N x_i y_i \quad (17)$$

where x_i and y_i are corresponding elements of the two vectors (or row and column elements of two matricial operands) and have precision n . Assuming each addition adds a maximum possible error $\varepsilon_{\max} = 2^{-n}$, the maximum aggregate error in each element of a product matrix of size N is $(\varepsilon_{\max})_{\text{total}} = N\varepsilon_{\max} = N \times (2^{-n})$, or N times the smallest representable number of precision n . This maximum total error reduces the precision of the solution by $m = \log_2(N)$ bits. Therefore, in order to maintain n significant bits of precision in the solution to an inner product or matrix multiplication of size N , at least m *guard bits* – extra bits of precision – must be carried through the operation. The final result is then rounded to n bits as with simple addition or multiplication. Executing the combination multiplication-addition operation at double precision ($2n$), as described previously for RPR multiplication, provides enough guard bits to ensure that the only significant error accumulated is the $\varepsilon \leq 2^{-n}$ generated in the rounding of the final result.

a. Determining Bounds for Matrix Multiplication

Determining proper bounds on the result of each inner product depends on the signs of the input vector (or matrix) elements x_i and y_i . The arrangement of bounds for each multiplication part of the operation is described in Equation (12). The addition parts of the operation require no special treatment – provided the addends are already in two's complement form, or converted to that form, before performing the addition. A complete matrix multiplication process developed using MATLAB 2007 Release A is included in Appendix B. The treatment in Appendix B multiplies numbers of arbitrary sign and random fractional values. It concludes by comparing upper and lower bound results to the precise result and showing that the

algorithm for choosing RPR operation inputs generates the correct upper and lower bounds on the inner product result.

b. Comparing RPR and TMR Implementations

Finding the inner product of two vectors of length l requires l multiplications and $l - 1$ additions. (Alternatively there may be l additions if the first product is added to a zero register, as in an accumulator.) Matrix multiplication is the computation of m inner products, where m is the number of elements in the product matrix. If the implementation design contains one multiplier and one adder, meant to be used many times, the space savings of RPR over TMR is on the order of the space savings of one multiplication operation, since multiplication is a much larger operation than addition. This is shown in Table 8.

Redundancy Type	Precision (or Degree)	Approximate Slice Count Projection (Implementation: multipliers/adders/voters)			
		1/1/1	1/1/2	l/l/1	l/l/l
TMR	64	6542	6599	$6477l + 65$	$6542l$
RPR	32/64 (0.5)	3390	3610	$3275l + 115$	$3390l$
RPR	16/64 (0.25)	2546	2672	$2451l + 95$	$2546l$
RPR	8/64 (0.125)	2295	2385	$2237l + 58$	$2295l$
TMR	32	1706	1195	$1673l + 33$	$1706l$
RPR	16/32 (0.5)	929	1040	$851l + 78$	$929l$
RPR	8/32 (0.25)	679	752	$637l + 42$	$679l$
TMR	16	457	470	$440l + 17$	$457l$
RPR	8/16 (0.5)	258	326	$224l + 34$	$258l$

Table 8. Projected FPGA Area Required for Matrix Multiplication.

Depending on the amount of error checking desired, matrix multiplication may be implemented with a voter after each calculation (one multiplier, one adder, two voters) or after the compound operation (one multiplier, one adder, one voter). Bounds of the result in either case (one operation or two in sequence) are easily determined. If the operation is implemented with l multipliers and one or more adders, the space savings of applying RPR over TMR increases with l . As discussed previously, the number of intermediate results checked for errors – i.e., the number of voters – is a design decision.

2. Discrete Fourier Transform and Fast Fourier Transform

The DFT is expressed as

$$X(k) = \sum_{j=1}^N x(j) w_N^{(j-1)(k-1)} \quad (18)$$

where X is the output sequence of frequency terms corresponding to the set x of time-domain input samples. Implementing a DFT generally includes generating a lookup table of twiddle factors w_N and then completing successive multiplication and addition or subtraction operations to obtain the output frequency-domain terms from the input time-domain terms. In MATLAB® the DFT is calculated using the Cooley-Tukey FFT algorithm, which reduces the series calculation to repetitive butterfly operations with different twiddle factors, as described in Chapter II. In this research, RPR bound determination was considered for both the DFT block operation and the FFT butterfly operation.

a. Determining Bounds for a DFT

To find each output point of the N -point DFT shown in Equation (18), the tasks are: retrieve a multiplication factor from a lookup table, multiply an input value by the factor from the table, and add the result to the accumulating sum. In practice, twice as many operations are performed as are represented symbolically because the w_N factors are complex; this requires that the real and imaginary parts of each result be calculated separately. Like in matrix multiplication, the arrangement of upper and lower bounds at the input to each multiply operation depends on the sign of the operands. In this case, the three possible operands are the j th input sample, the real part of the associated twiddle factor w_N , and the imaginary part of w_N . A complete DFT process generated using MATLAB 2007 Release A that takes a sequence of arbitrary length, determines input upper and lower bounds, performs the transforms and checks the validity of the precise, upper and lower bound output values is included in Appendix B.

b. Determining Bounds for an FFT Butterfly Operator

The butterfly operator is the smallest repeating unit of operation in an FFT. The structure for a radix-2 butterfly, shown first in Chapter II, is repeated here for convenience (Figure 30). The twiddle factors w_N are obtained from a lookup table as with the full DFT.

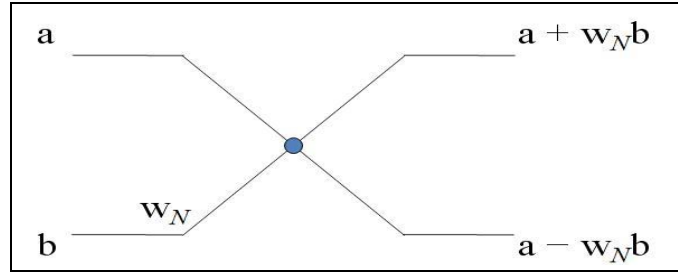


Figure 30. Radix-Two FFT Butterfly Operation (from Figure 8).

The butterfly operation comprises one multiplication, one addition and one subtraction. The subtraction may be executed by adding the two's complement of the multiplication result $w_N b$ to the addend a . The real and imaginary parts of the results $a + w_N b$ and $a - w_N b$ must be computed separately from the real and imaginary parts of the three inputs a , b and w_N . The bounds on the results are calculated from the bounds of the inputs, arranged depending on the signs of each input as stated in Equation (12) for multiplication and Equation (7) for addition/subtraction. The factor w_N must also be rounded for the reduced-precision calculations; either the rounded value may be used as both “upper” and “lower” bounds, or upper and lower bounds on w_N may be computed and used in the product bound computations. If the single rounded w_N value is used, the resulting reduced-precision products will still be upper and lower bounds on the precise product as long as the upper and lower bounds of input b are used correctly. Equation (19) is a set of conditional equations that demonstrates the relationship between the signs of the real and imaginary parts of the operands b and w_N and the upper and lower bounds of the product $w_N b$. For the sign bits s_x , ‘0’ implies positive or zero and ‘1’ implies negative.

				$(w_N b)_r = w_r b_r - w_i b_i$		$(w_N b)_i = w_r b_i + w_i b_r$	
s_{b_r}	s_{b_i}	s_{w_r}	s_{w_i}	$(wb)_{rL} =$	$(wb)_{rU} =$	$(wb)_{iL} =$	$(wb)_{iU} =$
0	0	0	0	$b_{rL} w_{rL} - b_{iL} w_{iL}$	$b_{rU} w_{rU} - b_{iU} w_{iU}$	$b_{rL} w_{iL} + b_{iL} w_{rL}$	$b_{rU} w_{iU} + b_{iU} w_{rU}$
0	0	0	1	$b_{rL} w_{rL} - b_{iU} w_{iL}$	$b_{rU} w_{rU} - b_{iL} w_{iU}$	$b_{rL} w_{iL} + b_{iU} w_{rL}$	$b_{rU} w_{iU} + b_{iL} w_{rU}$
0	0	1	0	$b_{rU} w_{rL} - b_{iL} w_{iL}$	$b_{rL} w_{rU} - b_{iU} w_{iU}$	$b_{rU} w_{iL} + b_{iL} w_{rL}$	$b_{rL} w_{iU} + b_{iU} w_{rU}$
0	0	1	1	$b_{rU} w_{rL} - b_{iU} w_{iL}$	$b_{rL} w_{rU} - b_{iL} w_{iU}$	$b_{rU} w_{iL} + b_{iU} w_{rL}$	$b_{rL} w_{iU} + b_{iL} w_{rU}$
0	1	0	0	$b_{rL} w_{rL} - b_{iL} w_{iU}$	$b_{rU} w_{rU} - b_{iU} w_{iL}$	$b_{rL} w_{iL} + b_{iL} w_{rU}$	$b_{rU} w_{iU} + b_{iU} w_{rL}$
0	1	0	1	$b_{rL} w_{rL} - b_{iU} w_{iU}$	$b_{rU} w_{rU} - b_{iL} w_{iL}$	$b_{rL} w_{iL} + b_{iU} w_{rU}$	$b_{rU} w_{iU} + b_{iL} w_{rL}$
0	1	1	0	$b_{rU} w_{rL} - b_{iL} w_{iU}$	$b_{rL} w_{rU} - b_{iU} w_{iL}$	$b_{rU} w_{iL} + b_{iL} w_{rU}$	$b_{rL} w_{iU} + b_{iU} w_{rL}$
0	1	1	1	$b_{rU} w_{rL} - b_{iU} w_{iU}$	$b_{rL} w_{rU} - b_{iL} w_{iL}$	$b_{rU} w_{iL} + b_{iU} w_{rU}$	$b_{rL} w_{iU} + b_{iL} w_{rL}$
1	0	0	0	$b_{rL} w_{rU} - b_{iL} w_{iL}$	$b_{rU} w_{rL} - b_{iU} w_{iU}$	$b_{rL} w_{iU} + b_{iL} w_{rL}$	$b_{rU} w_{iL} + b_{iU} w_{rU}$
1	0	0	1	$b_{rL} w_{rU} - b_{iU} w_{iL}$	$b_{rU} w_{rL} - b_{iL} w_{iU}$	$b_{rL} w_{iU} + b_{iU} w_{rL}$	$b_{rU} w_{iL} + b_{iL} w_{rU}$
1	0	1	0	$b_{rU} w_{rU} - b_{iL} w_{iL}$	$b_{rL} w_{rL} - b_{iU} w_{iU}$	$b_{rU} w_{iU} + b_{iL} w_{rL}$	$b_{rL} w_{iL} + b_{iU} w_{rU}$
1	0	1	1	$b_{rU} w_{rU} - b_{iU} w_{iL}$	$b_{rL} w_{rL} - b_{iL} w_{iU}$	$b_{rU} w_{iU} + b_{iU} w_{rL}$	$b_{rL} w_{iL} + b_{iL} w_{rU}$
1	1	0	0	$b_{rL} w_{rU} - b_{iL} w_{iU}$	$b_{rU} w_{rL} - b_{iU} w_{iL}$	$b_{rL} w_{iU} + b_{iL} w_{rU}$	$b_{rU} w_{iL} + b_{iU} w_{rL}$
1	1	0	1	$b_{rL} w_{rU} - b_{iU} w_{iU}$	$b_{rU} w_{rL} - b_{iL} w_{iL}$	$b_{rL} w_{iU} + b_{iU} w_{rU}$	$b_{rU} w_{iL} + b_{iL} w_{rL}$
1	1	1	0	$b_{rU} w_{rU} - b_{iL} w_{iU}$	$b_{rL} w_{rL} - b_{iU} w_{iL}$	$b_{rU} w_{iU} + b_{iL} w_{rU}$	$b_{rL} w_{iL} + b_{iU} w_{rL}$
1	1	1	1	$b_{rU} w_{rU} - b_{iU} w_{iU}$	$b_{rL} w_{rL} - b_{iL} w_{iL}$	$b_{rU} w_{iU} + b_{iU} w_{rU}$	$b_{rL} w_{iL} + b_{iL} w_{rL}$

(19)

After computing the bounds on the product $w_N b$, the results are used to compute the upper and lower bounds on the operations $a + w_N b$ and $a - w_N b$ (or the two's complement operation, mathematically $a + (2^n - w_N b)$) as determined by Equation (7). Because the operands are complex, the two operations are actually four arithmetic operations (adding real and imaginary parts separately for each addition). The bounds for each addition are calculated in the same way.

c. Comparing RPR and TMR Implementations

A fast Fourier transform butterfly operation contains one complex multiplication and two complex additions. Separating the real and imaginary parts of a , b , and w for computation yields four multiplications, three additions and three subtractions (or four multiplications, six additions and two 2's complement operations) as shown in Equation (20).

$$\begin{aligned}
(w_r + w_i i)(b_r + b_i i)(w_r + w_i i) &= (w_r b_r - w_i b_i) + (w_r b_i + w_i b_r) i \rightarrow 4 \text{ mult, 1 add, 1 sub} \\
(a_r + a_i i) + ((wb)_r + (wb)_i i) &= (a_r + (wb)_r) + (a_i + (wb)_i) i \rightarrow 2 \text{ add} \\
(a_r + a_i i) - ((wb)_r + (wb)_i i) &= (a_r - (wb)_r) + (a_i - (wb)_i) i \rightarrow 2 \text{ sub (complement/add)}
\end{aligned} \tag{20}$$

Depending on the amount of error checking required, a butterfly operation may include one or more voters on the final or intermediate results. Since the butterfly operator is the basic building block that is copied many times to build an FFT calculator, one obvious approach is to treat the butterfly as a block operation, and simply append two voters (one each for the real and imaginary parts of the result) to each RPR butterfly machine. However, it is possible to include a pair of voters after the complex products $(w_N b)_{r,i}$ as well, or even after each individual multiplication $w_x b_x$. An option at the other extreme is to check only the final result for each FFT output point, after the data have passed through all levels of butterflies. Based on the area requirements for multiplication and addition determined in Table 4 and Table 6, a comparison of projected requirements for a few butterfly operation designs is shown in Table 9.

Redundancy Type	Precision (or Degree)	Approximate Slice Count Projection (Implementation: multipliers/adders/voters)			
		4/6/2	4/6/4	4/6/8	4/6/0
TMR	64	26,236	26,366	26,622	26,106
RPR	32/64 (0.5)	13,464	13,694	14,598	13,234
RPR	16/64 (0.25)	10,096	10,286	10,806	9,906
RPR	8/64 (0.125)	9,150	9,266	9,670	9,034
TMR	32	6,860	6,926	7,054	6,794
RPR	16/32 (0.5)	3,630	3,786	4,242	3,474
RPR	8/32 (0.25)	2,686	2,770	3,110	2,602
TMR	16	1,848	1,882	1,946	1,814
RPR	8/16 (0.5)	1,002	1,070	1,378	934

Table 9. Projected FPGA Area Required for Radix-Two FFT Butterfly Operation.

Essentially, the space savings for RPR as compared to TMR in an FFT butterfly operation are on the order of 50% for high degrees of RPR (32/64 or 16/32). To gain greater advantage in size, choosing a smaller degree of RPR (such as 8/32 or even 8/64) gives a space savings approaching 66% (that is, the RPR operation takes only 1/3 the space of the TMR operation). This is consistent with the findings of Snodgrass in [3].

The presence of many voters in the design does not significantly change the area requirement in cases of high precision in the main operation (i.e., $n = 32$ or greater).

F. FURTHER DISCUSSION OF ERRORS AND THE RPR VOTER

1. Additional Considerations for RPR Voters

One of the results of the experiments conducted in this chapter was a table of the area required by the voters/error checkers for TMR and various degrees of RPR for multiplication and addition. For the purpose of comparing across operations, the voter area requirements are shown in Table 10, *without* the requirements for the corresponding operation modules.

Redundancy Type	Precision (or Degree)	Slice Count	
		Addition Voter	Multiplication Voter
TMR	64	65	64
RPR	32/64 (0.5)	115	226
RPR	16/64 (0.25)	95	130
RPR	8/64 (0.125)	58	101
TMR	32	33	32
RPR	16/32 (0.5)	78	114
RPR	8/32 (0.25)	42	85
TMR	16	17	16
RPR	8/16 (0.5)	34	77

Table 10. Area Required By TMR and RPR Multiplication Experiments.

The voters constructed for the multiplication operations in this research compared the full intermediate product of precision $2r$ with the value in the first $2r$ places of the precise result. This required the comparators in the RPR multiplication voters to be twice the size of the comparators in the RPR addition voters, which in turn made the multiplication voters approximately two times the size of the addition voters. Much of this increase in size can be improved by comparing only the first r bits of the products, as discussed in the multiplication section.

Another requirement for RPR implementation that is not represented in these results is the logic to test the signs of the operands and choose the correct input for the upper and lower bound calculations. This applies to multiplication and, if a separate subtractor is desired (instead of adding the 2's complement of the subtrahend), to subtraction. The selectors implemented to choose the final result (precise or RPR) in each RPR voter occupy as many slices of FPGA logic as there are bits in the output product – therefore it is likely that the area required by each input selector is as many slices as the precision of the input value (e.g., 8 for 8/64 RPR). In a small degree of RPR, this number is trivial compared to the total size of the operation (over 2000 slices for $r/64$ multiplication).

2. Error Detection with RPR

An SEU causes an *error* in an FPGA if and only if the *fault* it produces – by changing a logic value in configuration or data memory – directly or indirectly changes one or more output values. When a fault occurs in an RPR-protected addition or subtraction operation, there are eight possible scenarios: the error-free case and seven different types of errors. Figure 31 shows the eight scenarios as first defined by Snodgrass [3], emphasizing the differences among them.

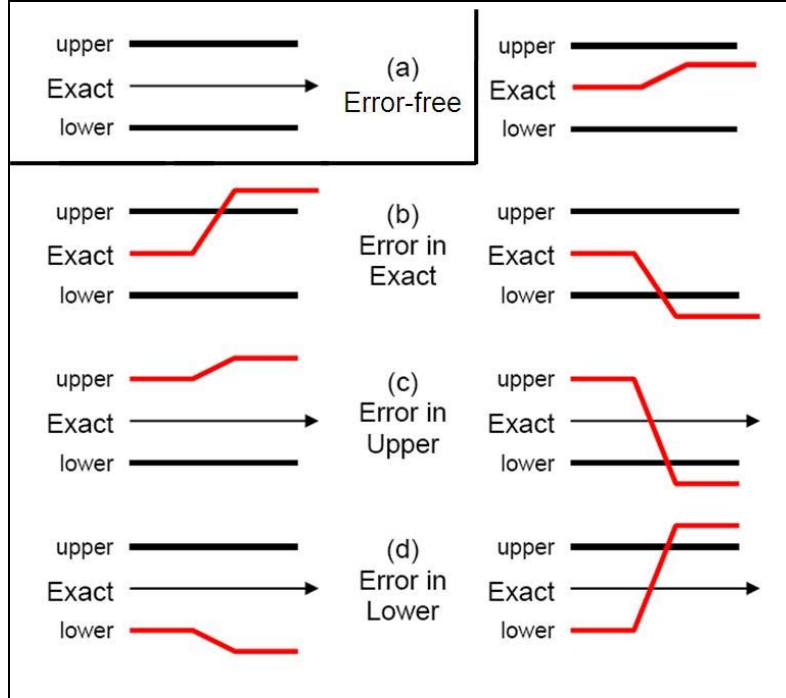


Figure 31. Possible Error Scenarios in RPR (After [3]).

Of the seven types of errors depicted in Figure 31, only four (the four cases where any two lines cross) are detectable using an RPR voter with the comparison methods described in this chapter. Of the remaining three cases, two are trivial: when there is an error in either bound result that maintains the correct relative magnitude of the upper, exact and lower results, the (correct) exact result will be used and no error will be propagated. The third undetected error, which occurs in the exact result but is small enough not to disturb the upper/exact/lower relative magnitudes, is by definition still within the tolerance described by the degree of RPR in the system.

In order to determine whether an error occurred in *data* or *configuration* memory, it is necessary either to check the FPGA configuration directly (if no discrepancies are found, the error was in data), or to keep track of errors in the operation output for more than one cycle. In general, an SEU occurring in data memory will only affect a single final output, although the error will appear in all intermediate results involving that particular value after the error occurred. In contrast, an SEU occurring in FPGA

configuration memory may or may not cause an error in the operation output. There are usually “extra” bits in FPGA configuration memory whose state is irrelevant to the proper operation of the circuit in most or all cases. Most of these are unused logic – if an SEU affects an unused FPGA slice, no error occurs. However, if a fault occurs in a slice of programmed logic and does cause an error, *every* value that flows through the faulty operation is affected. These error classes and the effects of the different errors are reiterated and discussed further in Chapter IV.

Whether the method of redundancy chosen is TMR or RPR, the designer of a system must decide whether checking output for successive errors is the preferred method of detecting configuration errors. If it is, the output checks may be implemented in an RPR system by using the three “comparison error” output signals set by any RPR voter. These three signals may control or trigger a scrub and reconfiguration of all or part of the FPGA. The additional logic needed to execute the conditional scrub must be traded against the time required to do configuration scrubs at regular intervals without being called by an error signal.

Regardless of the time required to detect and correct FPGA configuration errors in a system, the minimization of the errors’ impact to system performance is the true measure of success in fault tolerance. Chapter IV investigates some of the effects of using RPR on the performance of satellite control and sensor systems.

IV. EVALUATING RPR PERFORMANCE

The RPR operations in Chapter III demonstrate that sometimes a degree of RPR of 0.125 or smaller is needed to achieve significant FPGA area and power savings. This is a notable reduction in precision. To illustrate, consider that the smallest representable value in signed 64-bit fixed-point format is on the order of 1×10^{-19} . The smallest representable value in signed 8-bit fixed point format (which is the precision of the bounds in 8/64 RPR) is 0.0039, or 4×10^{-3} . Because the reduction in precision may be great, it is important to explore its impact on the performance of a system protected using RPR. This can be done using numerical simulations, with the occurrence of errors (triggering subsequent use of RPR results) modeled as an increase in system noise.

A. MODELING ERRORS DUE TO SINGLE EVENT UPSETS

1. Classes of Errors in FPGAs

It is important to remember that there are two distinct classes of faults that may cause errors in an FPGA-based system. A *data fault* occurs when a charged particle changes the value or one or more bits in data memory. This causes a *transient* error, so named because it only lasts as long as that particular piece of data is in the system. In a process like the FFT, where data continually flows through the system in a single direction, a transient error affects one or more output values depending on how early in the computation process it occurs. In a recursive process like an ADCS, a data error may have greater impact because some data is “remembered” – for example, the system state vector is maintained in memory and updated with every execution of the control loop. However, the state vector is also updated every cycle with fresh input data from the system attitude sensors. This could overwrite any errors in the stored system state vector depending on the ADCS architecture.

The second class of fault that can occur in an FPGA is in the configuration of the FPGA. A *configuration fault* occurs when a charged particle changes the value of one or more bits in the memory that holds the FPGA configuration. In the Xilinx® Virtex™

XCV600 FPGAs used on CFTP, there are 6,127,744 configuration bits and only 98,304 total block SelectRAM bits [25], so it is much more likely that a randomly incident charged particle will affect a configuration bit than a data bit. A configuration memory fault has the potential to inflict much more damage on a reconfigurable system than a transient fault could cause, because a configuration fault has the potential to generate an error in every value that it touches – it is *persistent*. A configuration fault may also create situations that are not even possible within the mathematical or logical rules governing the system operation: since it changes a random bit in a random lookup table (LUT) or other functional element of the FPGA, it may eliminate a potential output state, change the role of an input/output (I/O) buffer, or create other hazardous conditions outside the “logical” possibilities.

Both classes of faults manifest themselves as errors in the output values of a reconfigurable computer. However, the propagation and extent of the errors caused by data and configuration faults are different. When inspecting the instantaneous output of a system, redundant results (TMR or RPR) may be compared to detect and correct an SEU-related error. However, whether the SEU caused a data error or a configuration error is not evident until two or more successive sets of output are examined. A transient error due to a data fault may propagate through multiple steps in a processor, but within a finite number of clock cycles the error becomes obsolete and no longer affects the computation results. A persistent error due to a configuration fault is not corrected until part or all of the FPGA is reconfigured. Detecting the presence of either fault may be done by flagging errors in the system output; *distinguishing* between errors caused by a configuration fault and those caused by a data fault is much more difficult, but may be accomplished by integrating output errors over a number of clock cycles.

2. Modeling Errors as Noise

Consider a signal that is digitally sampled at regular intervals with precision $n = 32$. Ideally the smallest representable energy level in the sampling regime, assuming fixed-point fractional representation, is $2^{-32} = 2.328 \times 10^{-10}$ volts. If the signal processor has no fault tolerance and a data or configuration error occurs, a given output could be

incorrect in any or all digits: the magnitude of the error ranges from 2^{-32} to 2^{-1} , with maximum error magnitude $|\varepsilon| \leq O(2^{-1}) = (0.1)_2 = (0.5)_{10}$.

If the same signal is sampled with reduced precision $r=8$, the smallest representable energy level is $2^{-8} = 3.91 \times 10^{-3}$ volts. If a signal processor has RPR fault tolerance and an error occurs, the RPR result will be used. The RPR result is guaranteed to be correct to the r th bit, i.e., maximum error is $|\varepsilon| \leq O(2^{-(r+1)}) = 1 \times 2^{-9} = (0.001953)_{10}$ volts. The r -bit RPR result sacrifices $r - n$ bits of precision when it must be used instead of a correct full-precision result. However, the RPR result *preserves* r bits of precision when the alternative is no fault tolerance.

The difference in magnitude of the error in a system with no fault tolerance and the error in a system with RPR fault tolerance can be expressed using the signal-to-noise ratio (SNR) concept. When there is no fault tolerance in a system, the “noise” that corrupts a result due to an error has potentially the same magnitude and power as the signal itself. The maximum possible error $|\varepsilon| = (0.5)_{10}$ translates to $SNR_{\text{worst}} = 3$ dB for worst-case error scenarios. When an error is detected in an RPR system and the RPR result is used, the loss of precision in the RPR result is analogous to noise at a lower power. Low-power noise essentially randomizes values in the LSB of the result, making fine resolution of the signal impossible; however it does not affect coarse signal resolution, which is found by interpreting the MSB of the result. In the example system (8/32 RPR), the relative noise power represented by the RPR result gives $SNR_{\text{RPR}} = 10 \log_{10}(1/0.001953) = 27.1$ dB. This procedure can be applied to any degree of RPR to determine the equivalent noise introduced by using the RPR result.

3. Experiment Details

The experiments in this chapter were conducted using MATLAB Release 2007a with Simulink version 6.6. Machine epsilon ε for the system configuration used was

2.2204×10^{-16} , approximately equivalent to $n = 52$ in fixed-point representation². The degrees of RPR simulated in this experiment therefore follow the form $r/52$. Errors were simulated using either a random number generator with output scaling (to represent an uncorrected error or an RPR result), or an additive white Gaussian noise (AWGN) channel simulator with SNR corresponding to the desired reduction in precision. In the AWGN channel simulator, the representative input signal power was calculated using Xilinx Virtex™ XQVR600 FPGA power information [30] as a guide: this experiment was based on signal strength of 2.5 V at 10 mA for a total of 25mW simulated input signal power.

B. EVALUATING PERFORMANCE IN SPACECRAFT ADCS

The example attitude control system chosen for RPR performance evaluation was the NPS Bifocal Relay Mirror Satellite (BRMS) Simulator (BRMSS) model, developed in 2005 by Kim [31] and described in detail in Chapter V. Figure 32 shows the block diagram of the complete system model.

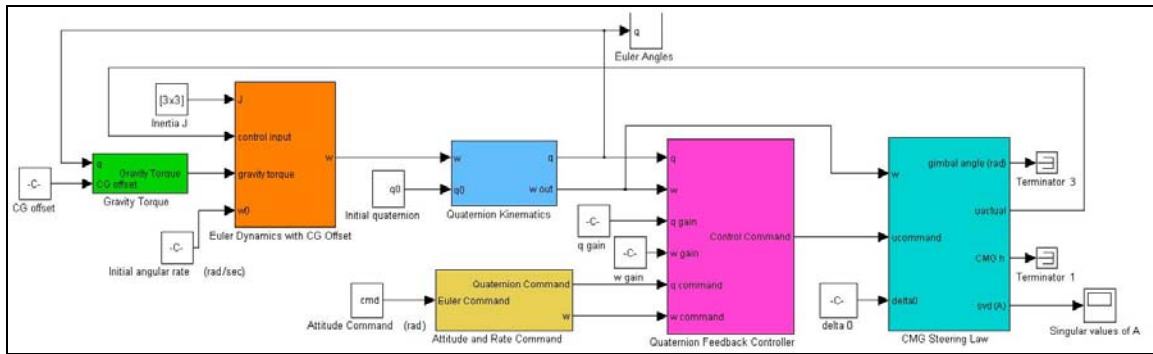


Figure 32. BRMS Simulator System Model (From [31]).

The shaded blocks in Figure 32 represent key subsystems of the ADCS model. From left to right, they are (1) the gravity torque model, (2) the Euler dynamics model, (3, top) the quaternion kinematics conversion, (4, bottom) the attitude and rate command

² Although MATLAB actually uses double-precision floating-point representation, these experiments are meant to *simulate* performance impact by examining numeric results at a level far above machine implementation. Therefore it is not a concern that the operations are not performed using the fixed-point designs specified in the previous chapter.

processing, (5) the quaternion feedback controller, and (6) the control moment gyro (CMG) steering law. When an experiment is run with the BRMSS hardware, the physics of the hardware replaces blocks (1) through (3) and the operator commands a trajectory through block (4). The heart of the ADCS is in blocks (5) and (6) – the controller and the control allocation (CMG steering law).

One of the most sensitive points in a satellite ADCS is allocation of the control command to the actuators – in this case, CMGs. CMGs are heavy, delicate machinery that apply very high torque to a spacecraft relative to the power they require to operate. Commanding a set of CMGs in a manner unsuitable for their operation can cause both temporary and permanent damage to the actuators or their housing, as well as loss of control of the spacecraft. For this reason, the worst-case scenario chosen for injecting error into the BRMSS ADCS model was between the controller and the CMG steering law: the control command.

The control command in the BRMSS model is a three-element vector generated by the controller that represents torque in the roll, pitch and yaw directions. The control command is converted by the CMG steering law into power levels supplied to each actuator that sum to produce the desired total torque. If any single component of the control command is in error, that component of the correction torque generated by the actuators will force the spacecraft to point or rotate away from the desired state. If the error is large, it is possible to drive the system past the limit of controllable error. At this point, control of the spacecraft is lost and cannot be recovered without contingency operations.

In this experiment, the three components of the control command were passed through independent additive white Gaussian noise (AWGN) channels. Activation, level and timing of the noise on each channel was controlled by variables set in a MATLAB script and called by the functions in the model. Since the experiment was meant to simulate a *single* event effect, only one noise channel was activated for any given trial. A diagram of the error injection system is shown in Figure 33.

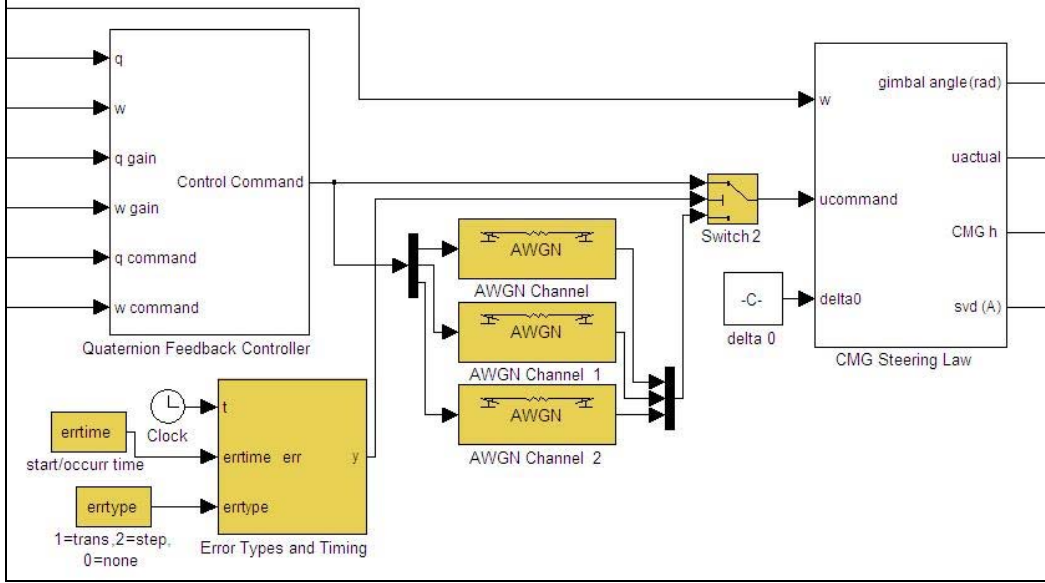


Figure 33. BRMSS Model Control and CMG Allocation with Error Injection.

The scenario used for the RPR performance evaluation was a standard “reference maneuver,” where the spacecraft began at rest in orientation $(\phi, \theta, \psi) = (0^\circ, 0^\circ, 0^\circ)$, and was commanded to move to orientation $(1^\circ, 1^\circ, 1^\circ)$ and stop. The maneuver with no fault-induced errors is executed in less than twenty seconds, which includes the time required for the system to settle to within two percent of the target orientation (Figure 34). The control command with no error over the course of the maneuver is shown in Figure 35. In Figure 34 and Figure 35, the roll (X) and pitch (Y) trajectories are coincident; only the yaw (Z) trajectory is different. This is due to the moment of inertia (MOI) of the simulated spacecraft, which is larger about the Z axis (J_{zz}) than about the X or Y axes.

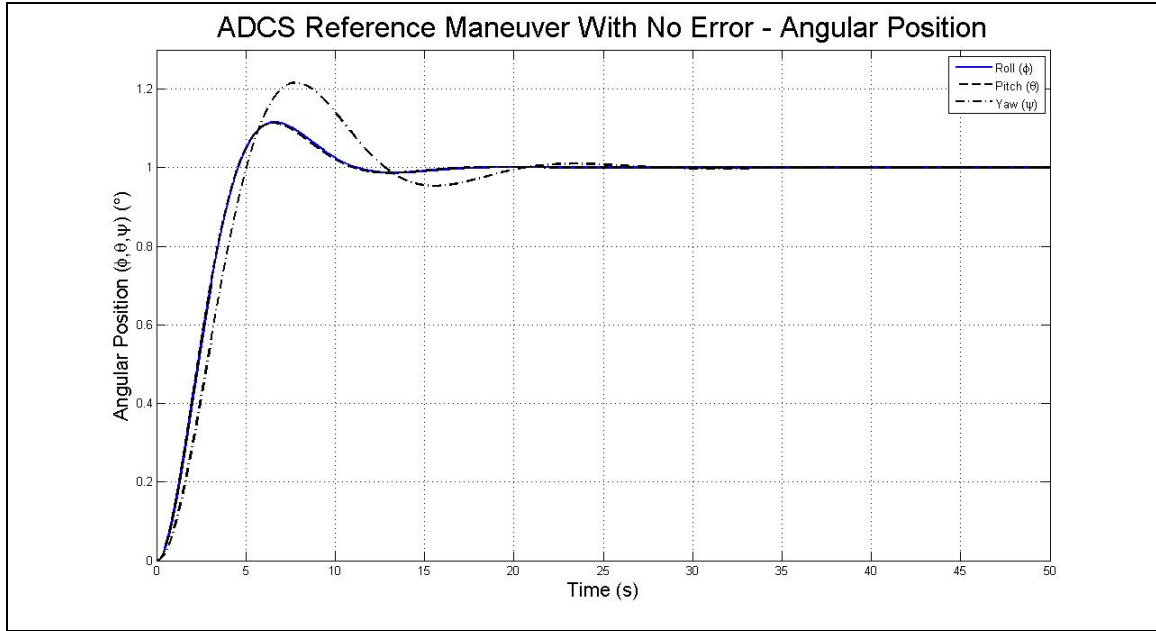


Figure 34. ADCS Reference Maneuver with No Error - Angular Position.

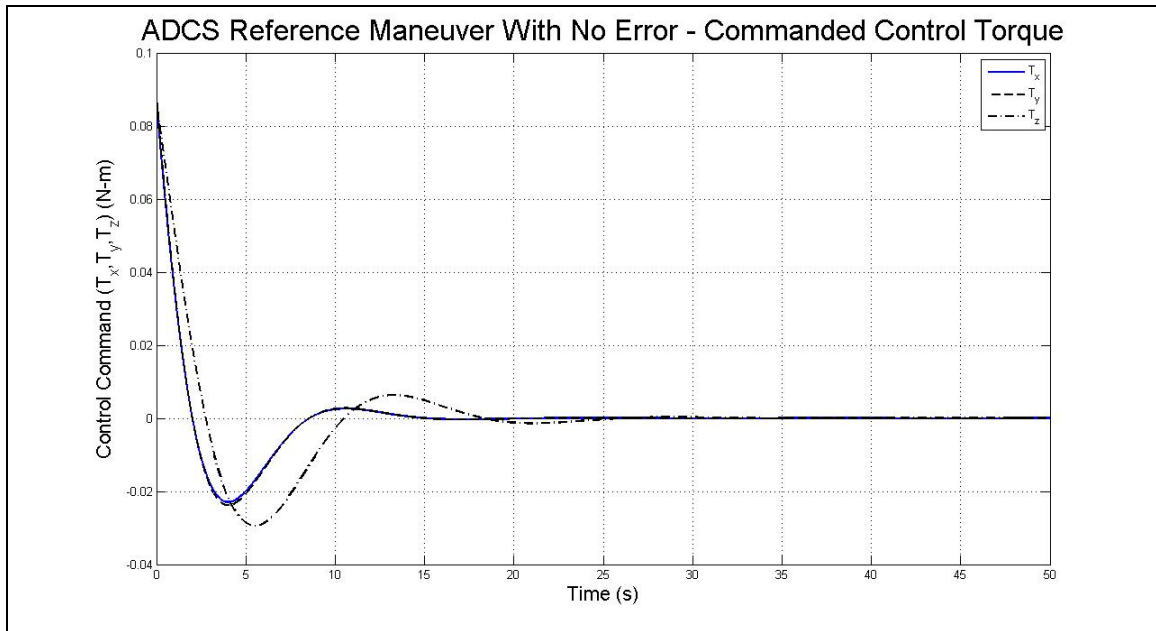


Figure 35. ADCS Reference Maneuver with No Error - Commanded Control.

The error scenarios tested with the ADCS model included both transient (discrete delta function δ) and persistent (step function) error models. Both types of errors

occurred at five seconds into the simulation ($t = 5$). For each type of error, the reference maneuver was executed with SNR levels equivalent to no fault tolerance ($SNR = 3$ dB), and RPR fault tolerance for $r = 8, 16, 24$, and 32 ($SNR = 27, 51, 75$, and 99 dB, respectively). The third independent variable in the tests was the element of the control command into which the error was introduced: T_x, T_y , or T_z . Corrupting the X, Y, or Z element of the control command generated highly correlated effects in the response of the simulated satellite through roll, pitch and yaw angles, respectively.

Rather than present an exhaustive collection of data from the thirty scenarios tested, the most significant results are included here. The first notable outcome of the tests is the conclusion that at the low operating power level assumed for this ADCS (25 mW), a transient data fault can generate an error only great enough to change the magnitude of the system transient response – it cannot change the steady-state properties of the system. For example, in Figure 36 an instantaneous unbounded error has been injected into the X element of the commanded control vector at 5 seconds (shown by the sharp spike in T_x control at $t = 5$).

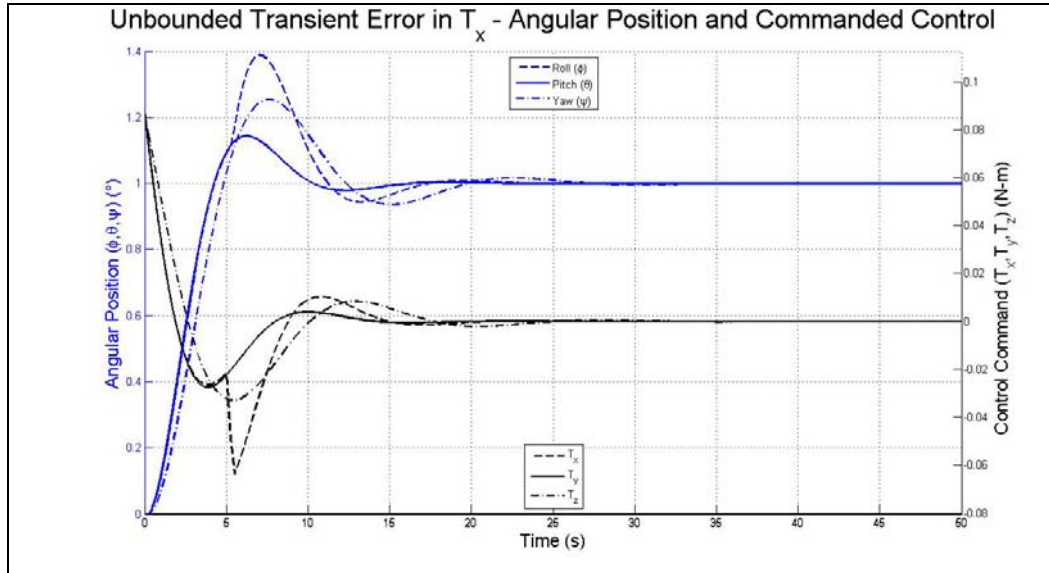


Figure 36. Unbounded Transient Error Effect on BRMSS Reference Maneuver.

The effect of this spike is a steep rise and increased overshoot in the roll of the spacecraft – the peak roll is $\phi=1.4^\circ$ instead of the nominal 1.1° , which is a 300% increase in percent overshoot and 27% increase in maximum attitude change of the spacecraft. However, the position settles to its steady state at $(1^\circ, 1^\circ, 1^\circ)$ in the same amount of time that it took to settle when there was no error to counteract. This result was the same in all tests run with transient errors. The worst-case transient error causes the spacecraft to experience greater torques, faster motion and more total rotation than in the maneuver with no error. Raising the simulated input signal power above the nominal 25 mW increases the overshoot of the system still further, but does not change the timing of the maneuver. Since the error is transient, the feedback control system corrects the error in the next cycle of the feedback loop. There is no lasting effect, but the spacecraft parts and assembly must be rated to withstand effects due to an envelope of higher torques that may occur when the ADCS corrects for transient errors in its data path.

If components rated for high torques are not practical, using RPR is one way to reduce the effect of a transient error. Figure 37 shows the response of the system attitude and control when RPR of degree 8/52 is applied: the difference between the RPR case and the case with no error injected (Figure 34 and Figure 35) is almost imperceptible.

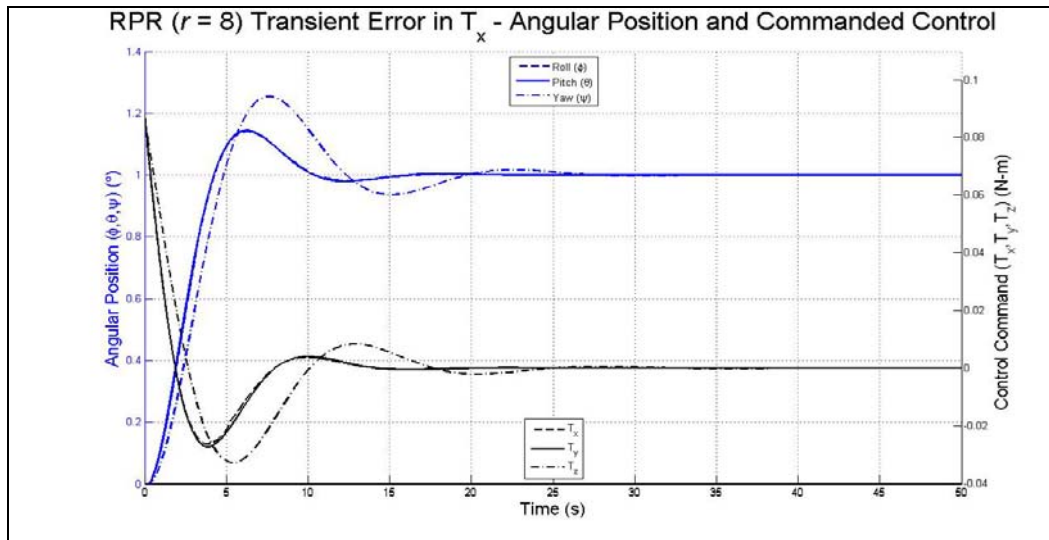


Figure 37. Transient RPR Result Effect on BRMSS Reference Maneuver ($r = 8$).

When a persistent error is introduced to a control system, the effect is much more significant than that of a transient error. Since the source of a persistent error is in the system configuration, a persistent error corrupts data in every execution of the feedback loop and disturbs every control command sent to the actuators. A representative example of the effect of a persistent error in the X element of the control command is shown in Figure 38. To ensure that the system did not eventually converge, the simulation time was extended from 50 seconds to 500 seconds for some trials of the persistent error scenario. The system did not settle: the noisy oscillation in the control and corresponding motion in roll angle maintained their average magnitudes over time (Figure 39).

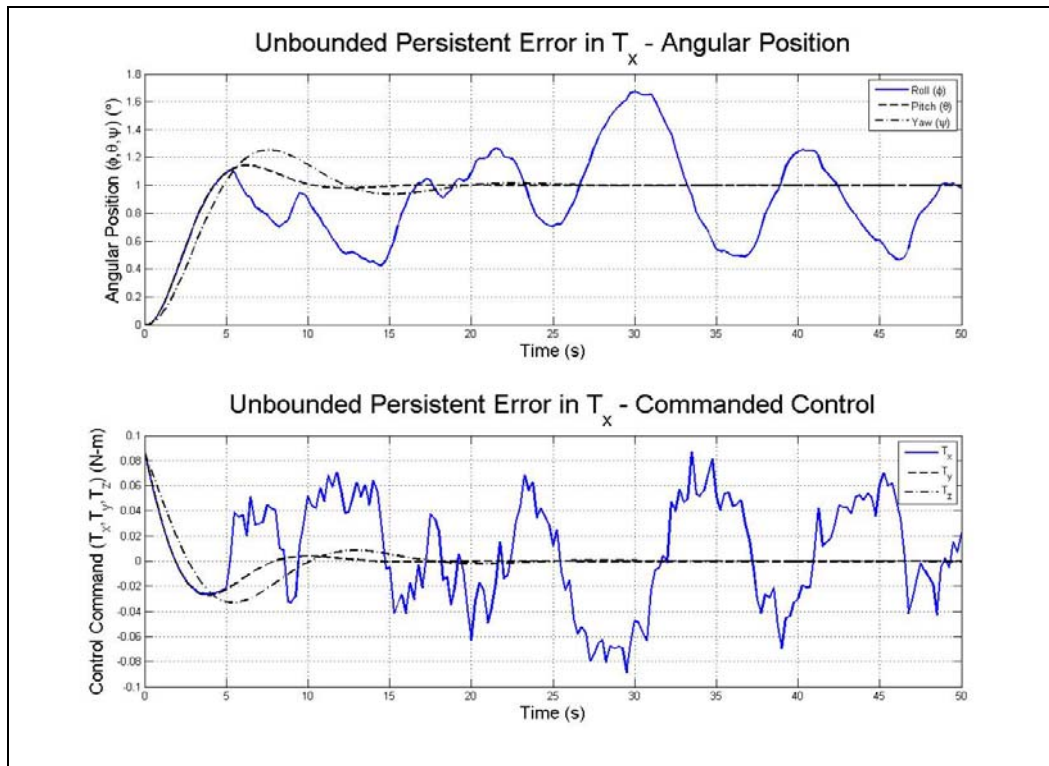


Figure 38. Unbounded Persistent Error Effect on BRMSS Reference Maneuver.

When RPR is applied to the system, the magnitude of the error is dramatically reduced. While a configuration fault in the unprotected system causes unbounded error that makes the system completely unusable, a configuration fault in the system protected with RPR generates errors whose magnitude is strictly bounded, and even a small degree of RPR ($r = 8$) shows marked improvement in the system trajectory (Figure 40).

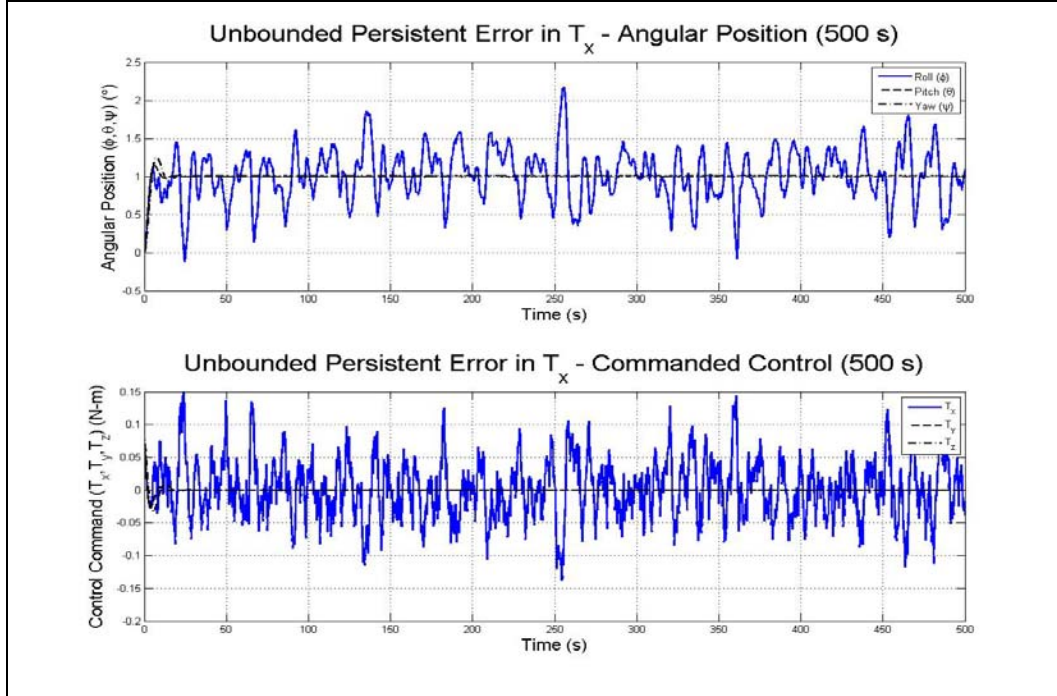


Figure 39. Extended Simulation of Unbounded Persistent Error.

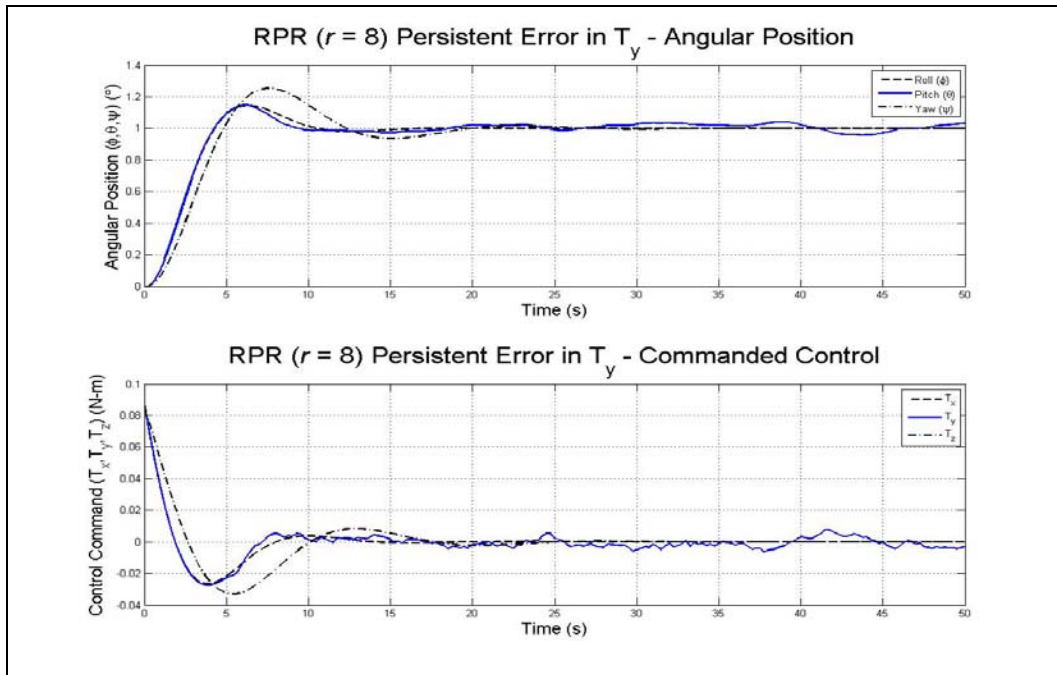


Figure 40. Persistent RPR Result Effect on BRMSS Reference Maneuver ($r = 8$).

The scenario in Figure 40 was run with SNR equivalent to RPR of degree 8/52 (SNR = 27 dB) applied to the Y component of the control command. (The effect was similar when RPR was applied to the X and Z components, so those results are not shown.) The response for RPR 8/52 is still not satisfactory for the fine pointing requirements of the BRMS, but could potentially provide an adequately steady state to operate a spacecraft with less stringent attitude control mission requirements (e.g., an RF communications satellite). However, when the scenario was run with SNR equivalent to RPR of degree 16/52 (SNR = 51 dB) both control and angle trajectories were virtually indistinguishable from the error-free case. This is shown in Figure 41 (compare to Figure 34 and Figure 35).

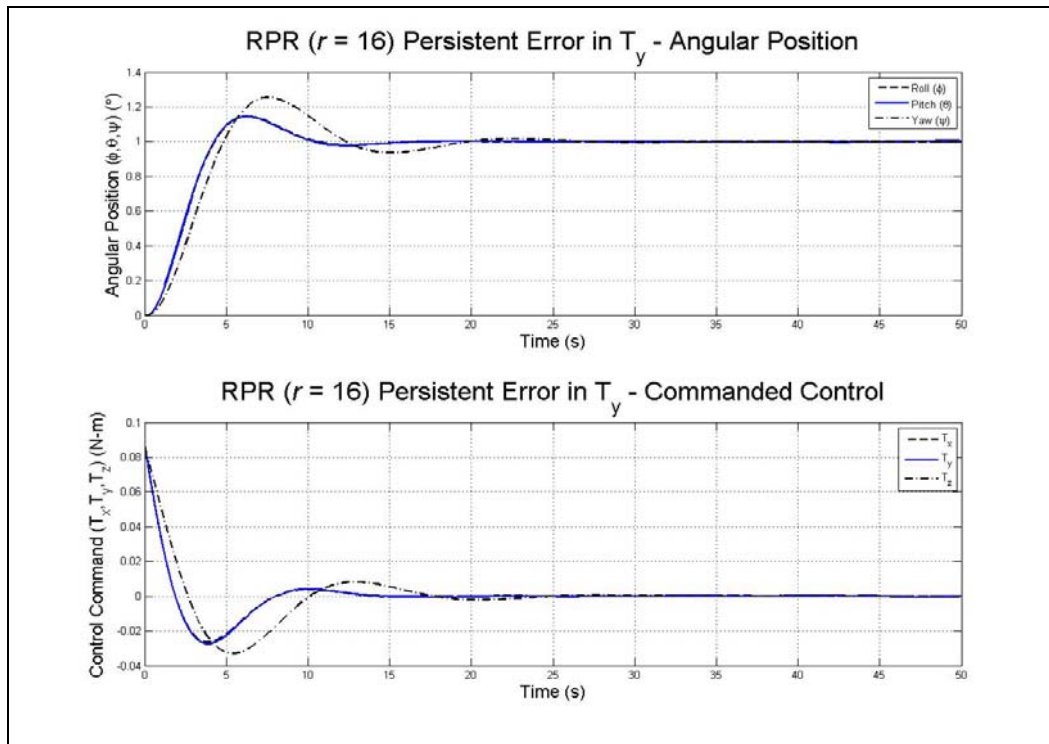


Figure 41. Persistent RPR Result Effect on BRMSS Reference Maneuver ($r = 16$).

These results show that RPR has the potential to supply enough precision in a reduced-precision solution to allow continuous operation through both transient and persistent errors, even in finely-controlled satellites.

One factor that should be addressed in the next treatment of this subject is the bandwidth of the control system – i.e., the speed with which the system state is updated. The scenarios in this chapter were designed to model a “worst case” scenario, so the simulations were first run with a fixed sample time of 0.25 seconds. This translates to a state update four times per second, which is slower than most ADCS for three-axis-stabilized spacecraft. Therefore, several simulations were run again with a fixed sample time of 0.025 seconds, which is more realistic (and even optimistic) for modern ADCS designs on reconfigurable computers. The overall effect of the faster system was better control responses, noticeable in both the transient and persistent error cases. With the 0.025 sample time, the RPR 8/52 scenario was markedly smoother due to the higher state update rate. However, some oscillation was still present (see Figure 42), so RPR 16/52 would still be wise for systems like the BRMS that require fine pointing accuracy or jitter control.

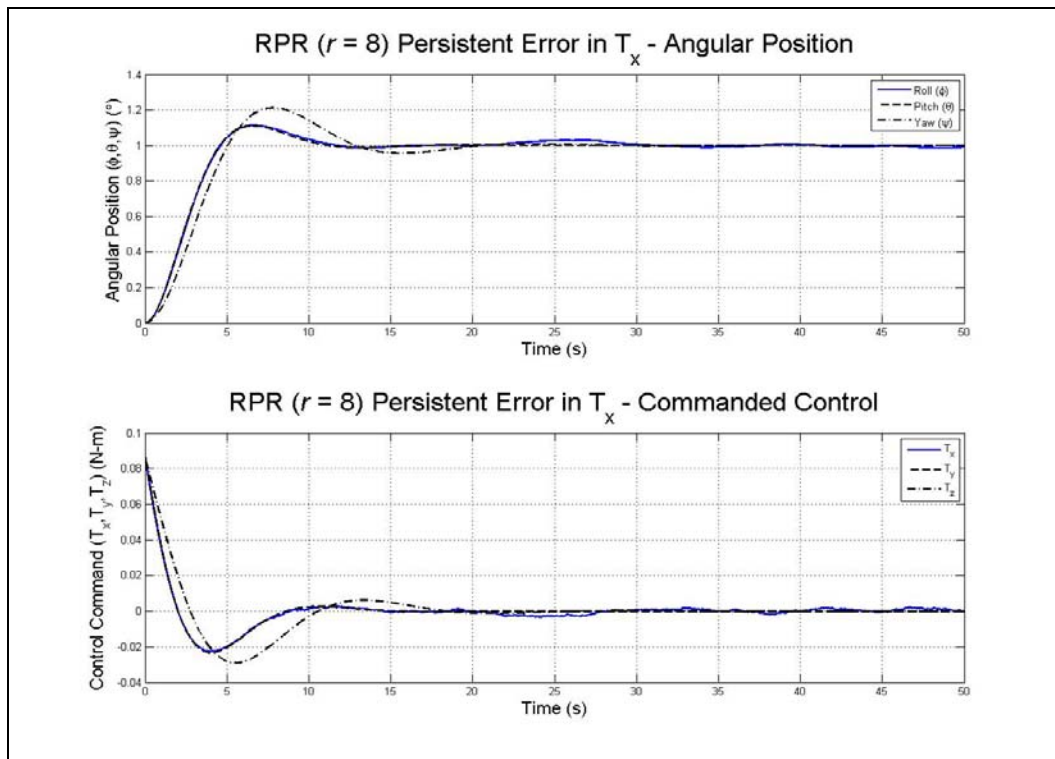


Figure 42. Small Timestep.

C. EVALUATING PERFORMANCE IN SOFT RADIO SYSTEMS: FFT

The level of complexity chosen for evaluating system performance using RPR in a soft radio system was the FFT example function described in Chapter II. For numeric simulation, the function chosen was the magnitude-FFT, whose final result is the magnitude of the complex FFT output. In order to model single error injection, the input sources needed to be managed individually. Therefore the point sizes chosen for the FFT (number N of samples in a computation batch) were small: for this experiment, $N = 8$ and $N = 16$. The model constructed for the 8-point FFT is shown in Figure 43; the script developed to run the model and conduct analysis on the data is included in Appendix C. The 16-point model (not shown) was identical in architecture, but had 16 input channels (including AWGN channel emulators and switches) instead of eight.

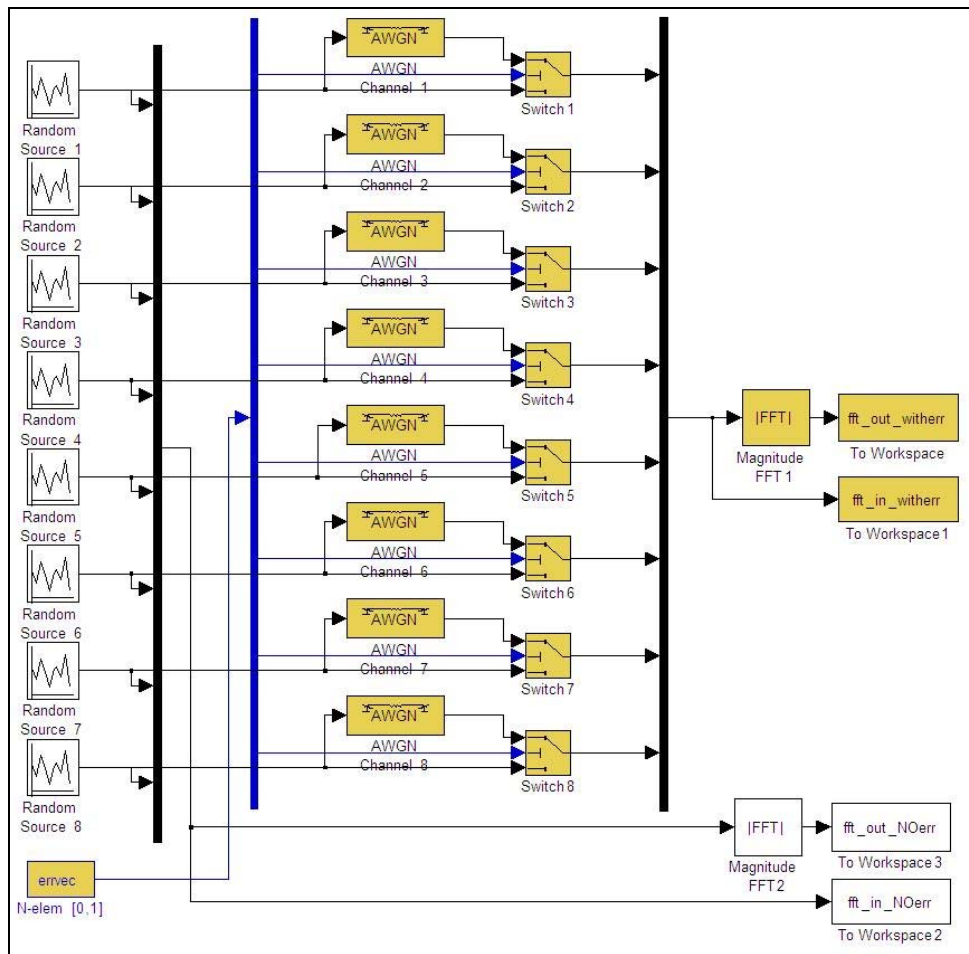


Figure 43. FFT Error Simulation Model.

Since the best method of executing the main experiment was to test the entire FFT as a block operation, the propagation of a single error through an FFT needed to be confirmed before the main experiment. Error propagation through one FFT was tested separately using a model constructed of four levels of butterfly machines (Figure 44).

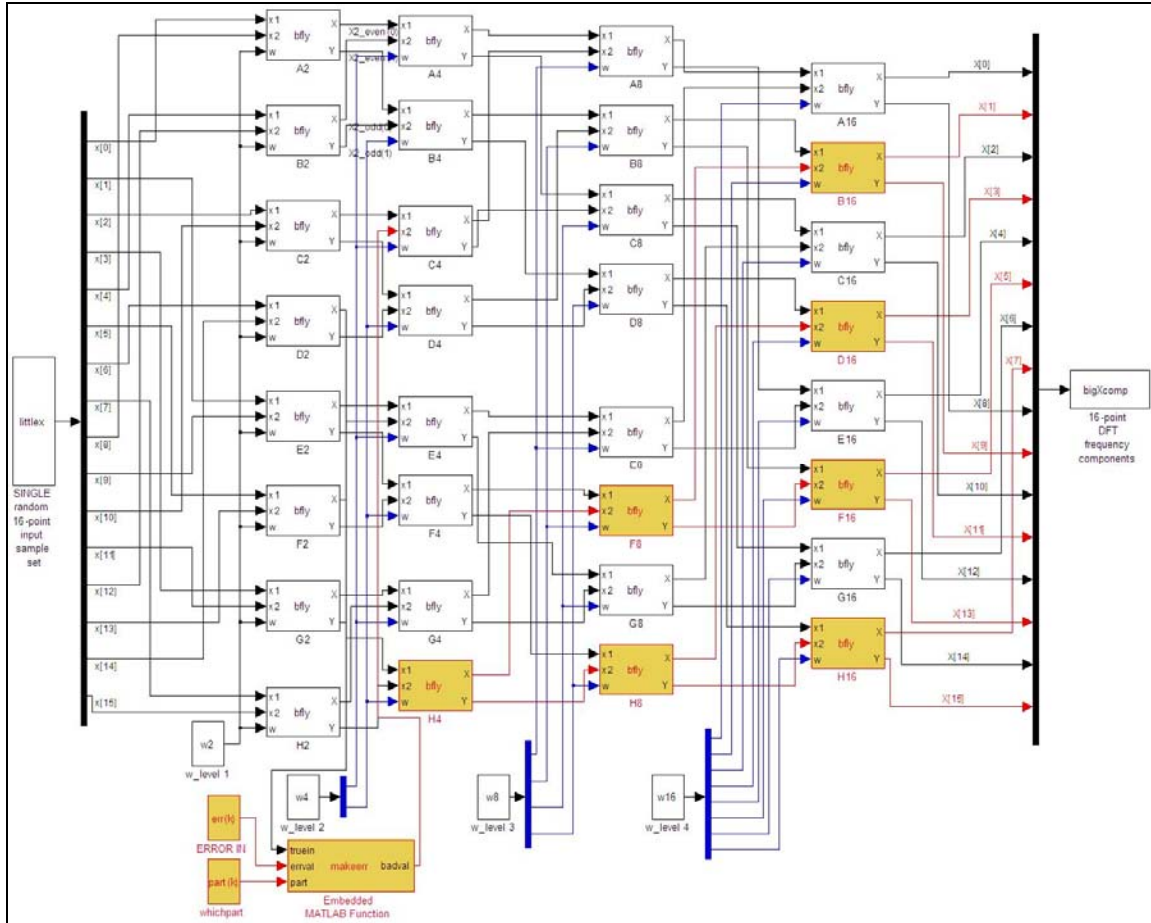


Figure 44. FFT Constructed With Four Levels of Fixed-Point Complex Butterfly Machines, $N = 16$ (Error Injected at Level 2).

Figure 44 depicts a 16-point FFT constructed of complex butterfly operations that follow the rules described in Chapter III. This FFT was tested using sets of 16 random fixed-point inputs on the interval $(-1, 1)$ with an “error” applied to one input. The system was tested with the single error in different locations in each level to understand the propagation of error through the FFT. In summary, the closer an error is injected to the beginning of the FFT, the more output values it will affect. This is logical, since in

each level of an FFT the BFM's operate on different combinations of the inputs until each output is a summation of all the other inputs. In order to model the worst-case scenario for system performance, the error must be introduced as early as possible in the FFT, i.e., at one of the original inputs. This also aligns with the model of applying RPR at the block operation level: if one of the inputs to this FFT comes from another operation that was deemed to be faulty, it will already have effective precision r and will therefore affect the precision of the entire FFT. The degree to which a single erroneous input affects the FFT output is shown by the main FFT experiment.

Testing for the main experiment included repeated execution of 100,000 trials (sets) of 8- and 16-point FFTs with random input sets (see Figure 43). A series of trials was run with SNR set equivalent to each of $r = 0$ (no fault tolerance), 8, 16, 24, and 32. A single error was injected by selecting the AWGN channel for one of the input values instead of the direct channel. In this model, one full FFT was computed at each timestep – this reduced the difference between the effect of a transient (data) fault and persistent (configuration) fault to whether the noise was injected for multiple FFT trials. In other words, the equivalent error in one FFT was the same whether it was a single data error due to a data memory fault, or one of several data errors appearing in succession as they would in the case of a configuration memory fault.

The results of this experiment are presented in two ways: Table 11 shows the relative error in a representative set of FFT output points for the full range of SNR settings. Figure 45 and Figure 46 show this data as a series of points for each SNR setting r whose y-axis values are the relative error in the representative output points.

FFT size (N)	r	SNR (dB)	Err In (%)	Error Out (%)					
				$X[0]$	$X[1]$	$X[2]$	$X[3]$	$X[4]$	$X[5]$
8	0	3.00	187.0%	46.68%	5.99%	6.03%	5.96%	46.01%	N/A
8	8	27.09	9.03%	5.09%	0.37%	0.37%	0.38%	5.82%	N/A
8	16	51.18	0.47%	0.19%	0.02%	0.02%	0.02%	0.25%	N/A
8	24	75.26	0.027%	0.0173%	0.0015%	0.0015%	0.0014%	0.0125%	N/A
8	32	99.34	0.002%	0.0007%	0.0001%	0.0001%	0.0001%	0.0006%	N/A
16	0	3.01	127.7%	39.61%	4.33%	4.34%	4.25%	4.34%	4.41%
16	8	27.09	5.10%	2.20%	0.28%	0.27%	0.27%	0.26%	0.26%
16	16	51.18	0.31%	0.12%	0.02%	0.02%	0.02%	0.02%	0.02%
16	24	75.26	0.022%	0.0056%	0.0010%	0.0010%	0.0010%	0.0010%	0.0010%
16	32	99.34	0.001%	0.0005%	0.0001%	0.0001%	0.0001%	0.0001%	0.0001%

Table 11. Error in FFT with No Fault Tolerance and RPR Fault Tolerance.

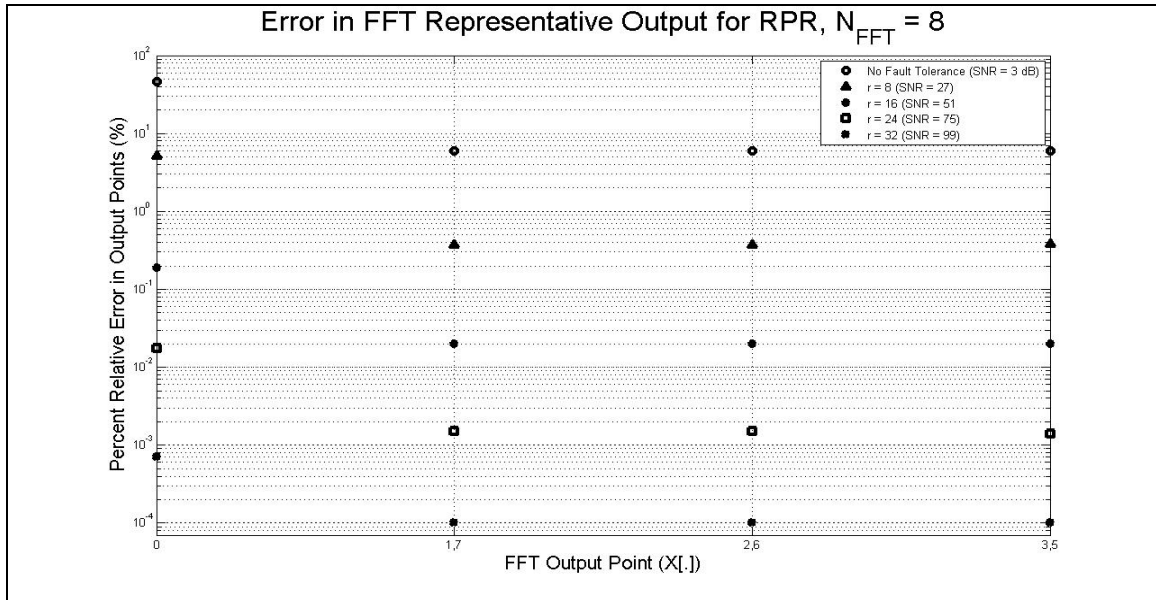


Figure 45. Relative Error in FFT Representative Output for RPR ($N_{\text{FFT}} = 8$).

The multiple X-axis tick labels represent the fact that the error in the FFT output was symmetric, e.g., the error in $X[3]$ and $X[13]$ for the 16-point FFT had the same magnitude.

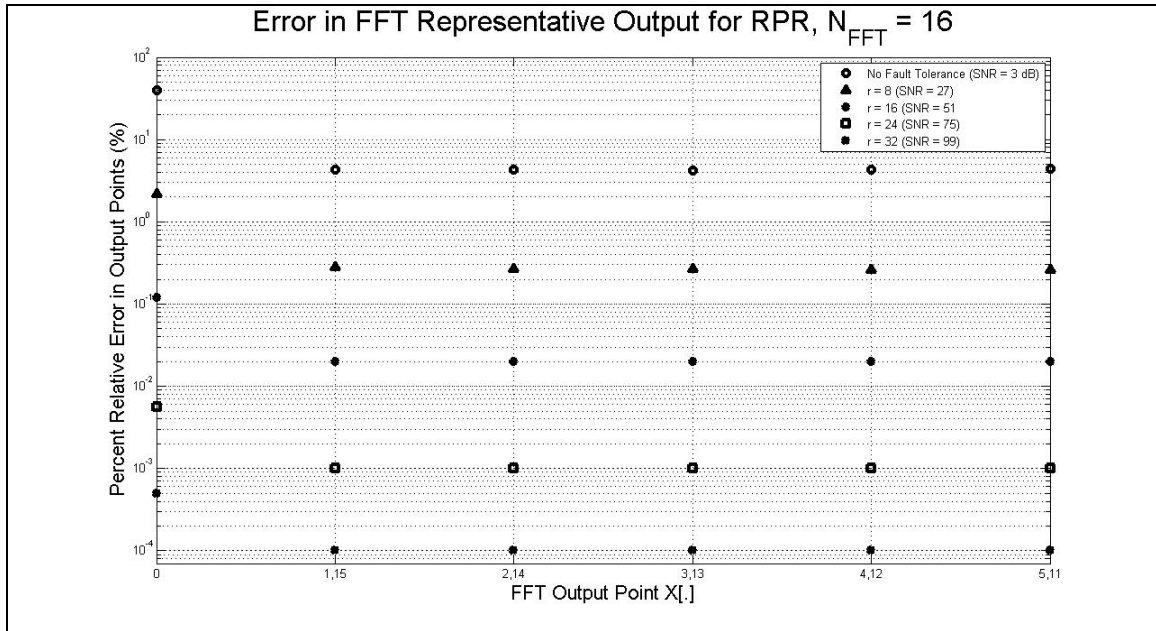


Figure 46. Relative Error in FFT Representative Output for RPR ($N_{\text{FFT}} = 16$).

Overall, the data gathered from the FFT experiment shows a marked increase in reliable precision of the output – and therefore improvement in accuracy – with higher degrees of RPR. Although the smallest tested degree of RPR ($r = 8$) still allowed an average of five percent relative error in the principal FFT output values, the next larger degree of RPR ($r = 16$) allowed less than one-quarter of one percent maximum error in any result. In every case, *any* degree of RPR afforded orders of magnitude less error in all final output values than the error in the results obtained using the unprotected systems.

Also, the relationship between relative error and degree of RPR is comparable across different sizes of FFT. Before implementing a very large FFT, additional study should be done to confirm that the precision of the output is maintained over many (e.g., 10) levels of butterfly operations. However, if the operations are implemented with guard bits as discussed in Chapter III, the precision should be preserved.

Ultimately the trade between the space savings of small degrees of RPR and the more significant error allowed by small degrees of RPR is a consideration that must be evaluated by a system designer.

D. GENERAL NOTES ON RPR-PROTECTED SYSTEM PERFORMANCE

From the simulations in this chapter several points can be made. The effect of a transient error due to a data memory fault is far less damaging than the effect of a persistent error due to a configuration memory fault. Even in a worst-case scenario, 16 bits or less of precision in an RPR result for an overall system $n = 52$ (i.e., RPR degree $16/52$ or smaller) provides the accuracy necessary to maintain tight control in an ADCS or less than 0.2% error in FFT output elements. The speed with which a recursive data management system (like an ADCS) operates has a significant impact on the precision required of an acceptable RPR result. Overall, the performance of a system improves with larger degrees of RPR – but the fundamental benefit of RPR increases with smaller degrees of RPR, so any designer must evaluate carefully the trade space bounded by FPGA logic capacity, operation speed, and lowest tolerated precision.

V. APPLYING RPR TO A COMPLEX SYSTEM

A. THE NPS SPACECRAFT SIMULATOR

The Bifocal Relay Mirror Spacecraft (BRMS) simulator (BRMSS) is an experimental test bed developed at the Naval Postgraduate School and used for ground testing of spacecraft adaptive control algorithms [32]. The BRMSS is based on a structure of circular platforms supported by a spherical air bearing that allows it to rotate freely about three axes. The top platform of the simulator contains the spacecraft payload apparatus; between the middle and bottom platforms are the spacecraft computers, sensors, and actuators. Mounted around the edge of the simulator are three flexible appendages, each linked to the main body by a single torsional spring. Three Control Moment Gyroscope (CMG) actuators provide torque for the rotational motion of the spacecraft (Figure 47).

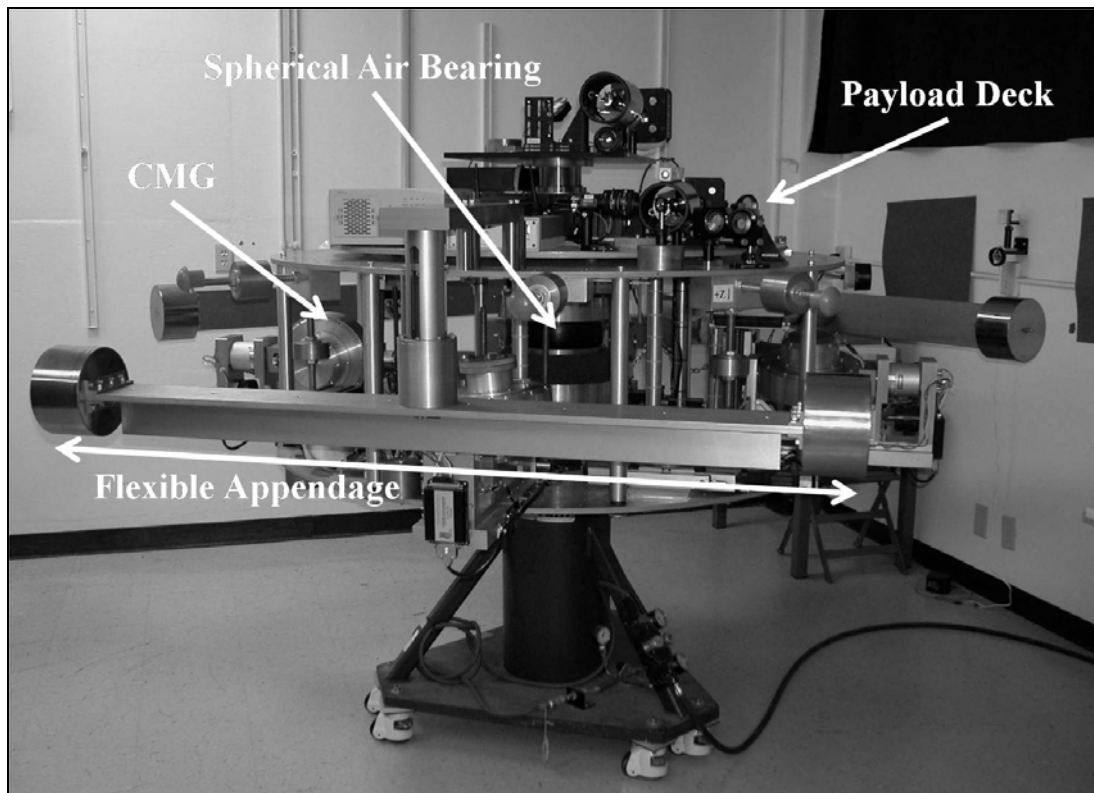


Figure 47. NPS Bifocal Relay Mirror Spacecraft Simulator (After [32]).

The appendages are meant to simulate the motion of solar arrays or antennas, which are often flexible components attached to the central body of a spacecraft.

Before new algorithms are applied to the test bed hardware, they are developed in a software simulation that contains models of the simulator dynamics, the simulated kinematics of the spacecraft, the controller, the allocation of control to the actuators, and the commanded new attitude and rate for the simulator (Figure 48). In the following sections, the dynamics and control blocks of this model are examined and improved. At the conclusion of this chapter is a brief analysis from the perspective of implementation on a reprogrammable computer, and suitable locations for applying RPR are noted.

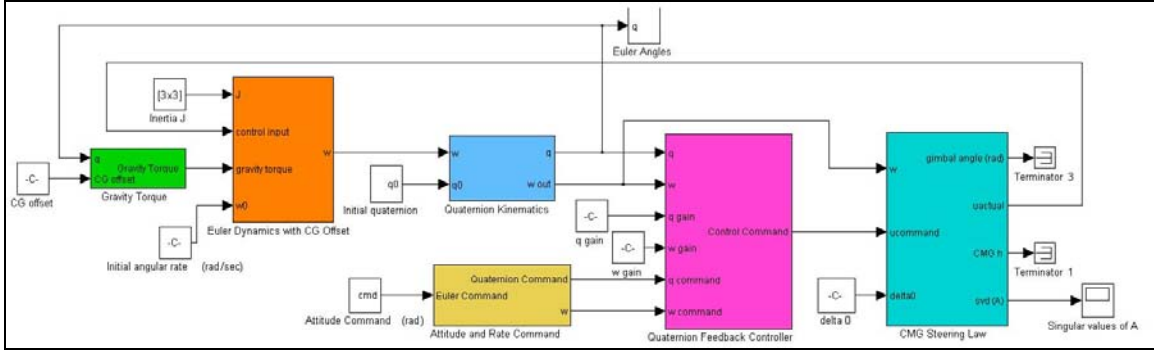


Figure 48. BRMS Simulator System Model (Copy of Figure 32).

B. DYNAMICS OF THE BRMS SIMULATOR

1. Current Model: Rigid-Body Dynamics

Currently the BRMSS is modeled as a central, rigid body with no flexible appendages and body-centered, body-fixed coordinate frame $OXYZ$ (Figure 49). There is non-linear coupling among the three axes of rotation of the simulator. The moment of inertia (MOI) matrix for the rigid-body BRMSS J_B in its body-centric reference frame $\begin{pmatrix} B \end{pmatrix}$ is represented by

$${}^B J_B = \begin{bmatrix} J_{xx} & J_{xy} & J_{xz} \\ J_{yx} & J_{yy} & J_{yz} \\ J_{zx} & J_{zy} & J_{zz} \end{bmatrix} \quad (21)$$

with principal axes (X, Y, Z) components (J_{xx}, J_{yy}, J_{zz}) and cross-coupling terms ($J_{xy}, J_{yx}, J_{xz}, J_{zx}, J_{yz}, J_{zy}$) all in the central body reference frame. The values for each inertia matrix element are determined experimentally in [32] to be

$$J = \begin{bmatrix} 130.34 & 3.01 & 10.52 \\ 3.02 & 174.64 & -0.40 \\ 10.52 & -0.40 & 181.23 \end{bmatrix}. \quad (22)$$

Equation (22) shows that the rigid-body principal axis terms dominate the overall system MOI (i.e., the principal axis terms are much larger than the cross-coupling terms). Therefore, for the purposes of this approximate model the cross-coupling terms of the rigid-body MOI are neglected, leaving

$$J = \begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix} = \begin{bmatrix} 130.34 & 0 & 0 \\ 0 & 174.64 & 0 \\ 0 & 0 & 181.23 \end{bmatrix}. \quad (23)$$

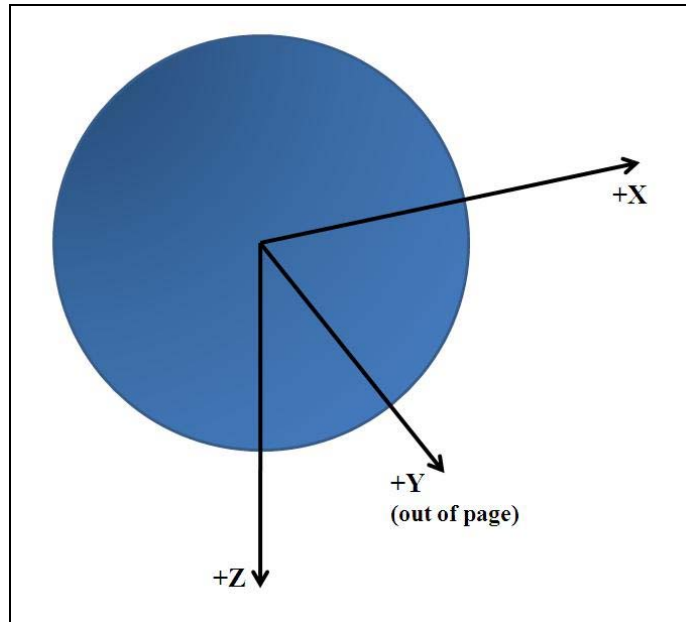


Figure 49. Rigid-body model of BRMSS, showing principal body axes.

The simplified equations of motion (EOM) of the rigid body BRMSS in matrix form are

$$\begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix} \begin{Bmatrix} \ddot{\theta}_x \\ \ddot{\theta}_y \\ \ddot{\theta}_z \end{Bmatrix} = \begin{Bmatrix} T_x \\ T_y \\ T_z \end{Bmatrix} \quad (24)$$

where $\{T_x, T_y, T_z\}^T$ are the components of the control torque applied by the actuators (the CMGs) in the direction of each of the principal inertia axes of the rigid-body model.

2. Modeling the Flexible Appendages

There are three identical flexible appendages attached to the central body of the BRMSS. Each appendage consists of a long reinforced (rigid) bar with a mass on each end; the bar is connected to the main simulator body by a torsional spring at the center of the bar (Figure 50). The rigid bar has thickness a , length b , width c and mass M . Each end mass has radius r and mass m . The center of each end mass is located a perpendicular distance R from the torsional spring. The torsional spring has spring constant k .

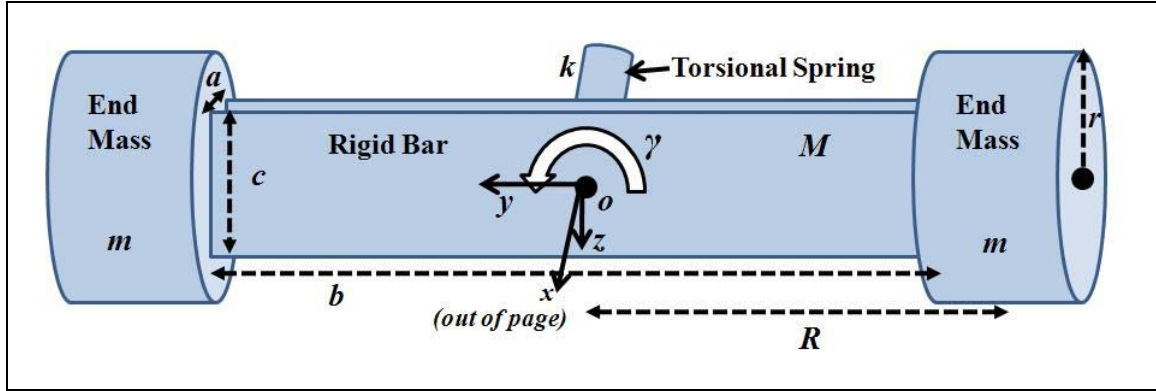


Figure 50. BRMSS Flexible Appendage* Model (Not To Scale).

*Local coordinate system (xyz) shown is used for appendages 1 and 2 only.

The principal moments of inertia of the appendage in its local body frame are calculated using simple geometry and the parallel axis theorem. Using the parameters defined in Figure 50, the principal moments of inertia of a single appendage in its local frame of reference ($oxyz$) are

$$\begin{aligned} J_{Axx} &= \frac{1}{12} M (b^2 + c^2) + 2(mR^2) \\ J_{Ayy} &= \frac{1}{12} M (a^2 + c^2) + 2\left(\frac{1}{2}mr^2\right) = \frac{1}{12} M (a^2 + c^2) + mr^2 \\ J_{Azz} &= \frac{1}{12} M (a^2 + b^2) + 2(mR^2) \end{aligned} \quad (25)$$

Each torsional spring is attached to the BRMSS main structure at a point in the XY (horizontal) plane of the central body (Figure 51). Appendages 1 and 2 are oriented such that the axes about which the appendages rotate lie in the central body XY (horizontal) plane, displaced from the central body X axis by the angles α and β , respectively. Appendage 3 is oriented such that its local axis of rotation is parallel to the central body Z axis (i.e., the appendage sweeps out local angle γ , shown in Figure 50, in the central body XY plane). The vectors $\bar{s}_i$ from the origin of the central body O to the attachment point of each torsional spring are determined to be

$$\begin{aligned} {}^B\bar{s}_1 &= S_1 \cos \alpha \hat{\mathbf{X}} + S_1 \sin \alpha \hat{\mathbf{Y}} + 0\hat{\mathbf{Z}} = S_1 (\cos \alpha \hat{\mathbf{X}} + \sin \alpha \hat{\mathbf{Y}}), \quad 0 \geq \alpha \geq -\frac{\pi}{2} \\ {}^B\bar{s}_2 &= S_2 \cos \beta \hat{\mathbf{X}} + S_2 \sin \beta \hat{\mathbf{Y}} + 0\hat{\mathbf{Z}} = S_2 (\cos \beta \hat{\mathbf{X}} + \sin \beta \hat{\mathbf{Y}}), \quad -\frac{\pi}{2} \geq \beta \geq -\pi \\ {}^B\bar{s}_3 &= 0\hat{\mathbf{X}} + S_3 \hat{\mathbf{Y}} + 0\hat{\mathbf{Z}} = S_3 \hat{\mathbf{Y}} \end{aligned} \quad (26)$$

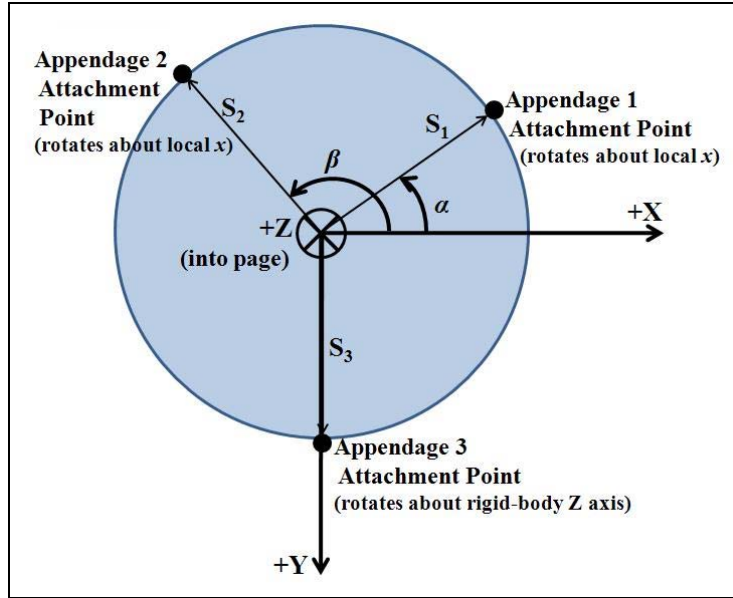


Figure 51. BRMSS Appendage Attachment Points and Orientation (Top View).

In order to determine the effect of the flexible appendages on the dynamics of the BRMSS, the MOI of each appendage must be expressed in the central body reference frame. For appendages 1 and 2 there are two transformations: from the central body frame B to the torsional spring frame S , and from the spring frame S to the local appendage body frame A . The first transformation is a rotation about the central body Z (or B_3) axis by the angle α , represented by the third Euler axis rotation matrix [33],

$${}^B C^{S_1} = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (27)$$

where α is defined as the constant angle between the central body X axis and the vector drawn from the central body origin O to the spring attachment point for appendage 1 (see Figure 51). In this scenario, the value of angle α is such that $0 \geq \alpha \geq -\frac{\pi}{2}$. The second

transformation is a rotation of the appendage about the spring axis, which is coincident with the local x axis of the appendage. This is represented by the first Euler axis rotation matrix [33],

$${}^{S_1}C^{A_1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \gamma_1 & \sin \gamma_1 \\ 0 & -\sin \gamma_1 & \cos \gamma_1 \end{bmatrix} \quad (28)$$

where the angle γ_1 is swept out by appendage 1 rotating about its local x axis.

The complete transformation needed to express vector properties of appendage 1 in the central body reference frame is the multiplication of these two rotation matrices, or

$${}^B C^{A_1} = [{}^B C^{S_1}] [{}^{S_1} C^{A_1}] = \begin{bmatrix} \cos \alpha & \sin \alpha \cos \gamma_1 & \sin \alpha \sin \gamma_1 \\ -\sin \alpha & \cos \alpha \cos \gamma_1 & \cos \alpha \sin \gamma_1 \\ 0 & -\sin \gamma_1 & \cos \gamma_1 \end{bmatrix} \quad (29)$$

By similarity, the transformation matrix needed to express vector properties of appendage 2 in the central body reference frame is

$${}^B C^{A_2} = [{}^B C^{S_2}] [{}^{S_2} C^{A_2}] = \begin{bmatrix} \cos \beta & \sin \beta \cos \gamma_2 & \sin \beta \sin \gamma_2 \\ -\sin \beta & \cos \beta \cos \gamma_2 & \cos \beta \sin \gamma_2 \\ 0 & -\sin \gamma_2 & \cos \gamma_2 \end{bmatrix} \quad (30)$$

where β is defined as the constant angle between the central body X axis and the vector drawn from the central body origin O to the spring attachment point for appendage 2 (see Figure 51 and Equation (26)), and γ_2 is the angle swept out by appendage 2 rotating about its local x axis.

Appendage 3 is oriented differently: its local axis of rotation is aligned with the principal Z axis of the central body. Because of this alignment, it is more straightforward to redefine the local body axes for appendage 3 as depicted in Figure 52 (compare to local body axes in Figure 50 for appendages 1 and 2). Using this local reference frame, the single transformation required to express vector properties of appendage 3 in the central body reference frame is the third Euler rotation matrix [33],

$${}^B C^{A_3} = \begin{bmatrix} \cos \gamma_3 & \sin \gamma_3 & 0 \\ -\sin \gamma_3 & \cos \gamma_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (31)$$

where γ_3 is the angle swept out by appendage 3 rotating about its local x axis.

$$\left[{}^B J^B \right] = \left[{}^B J^S \right] - M_A [\tilde{s}] [\tilde{s}] \quad (35)$$

from [33], where

$$[\tilde{s}] \triangleq \begin{bmatrix} 0 & -s_z & s_y \\ s_z & 0 & -s_x \\ -s_y & s_x & 0 \end{bmatrix} \quad (36)$$

using the vectors s_i as defined in Equation (26) for the spring attachment point of each appendage.

The complete BRMSS MOI is equal to the sum of the original rigid-body MOI in its central body reference frame (Equation (23)) and the parallel axis effects of the mass of each flexible appendage (applying Equation (35)). The complete system MOI represented in compact form by

$$\left[{}^B J_{BRMSS} \right] = \left[{}^B J_B \right] + \sum_{i=1}^3 \left(\left[{}^B J_{Ai} \right] - M_A [\tilde{s}_{Ai}] [\tilde{s}_{Ai}] \right). \quad (37)$$

In addition to the system MOI, the spring constant of the torsional spring in each appendage is necessary in order to find the equations of motion (EOM) for the BRMSS. The stiffness of the torsional spring for each flexible appendage is calculated using the natural frequency of the appendage (experimentally determined) f_n , as in

$$f_n = \frac{\omega_n}{2\pi} = \frac{1}{2\pi} \sqrt{\frac{k}{J_{Arot}}} \rightarrow k = J_{Arot} (2\pi f_n)^2 \quad (38)$$

where J_{Arot} is the moment of inertia of the appendage about its axis of rotation (J_{xx} for appendages 1 and 2, and J_{zz} for appendage 3). The variable ω_n is the natural frequency of the rotating body in radians, f_n is the natural frequency of the rotating body in Hertz, and the constant k is the calculated stiffness of the torsional spring.

The EOM that completely describe the BRMSS depend on a set of six state variables: $\{\theta_x, \theta_y, \theta_z, \gamma_1, \gamma_2, \gamma_3\}^T$. In compact form, these equations are

$$\begin{aligned} \left[{}^B J_{BRMSS} \right] \{\ddot{\theta}_B\} + \sum_{i=1}^3 k_i \left(\{\theta_B\} - \left[{}^B C^A \right] \{\gamma_i\} \right) &= \{T_{CMG}\} \\ \left[{}^A_i J_{A_i} \right] \{\ddot{\gamma}_{A_i}\} + k_i \left(\{\gamma_i\} - \left[{}^A_i C^B \right] \{\theta_B\} \right) &= \{0\}, \quad i = \{1, 2, 3\}. \end{aligned} \quad (39)$$

When neglecting cross-coupling as in Equation (23), the inertia matrices become diagonal. After applying the transformations ${}^B C^A$ and ${}^A C^B$, the decoupled EOM can be separated and treated as three independent control problems in the three principal central-body axes. This process is shown in Equations (40) through (46). Equations (40) through (43) are the four EOM (the central body and each of the three appendages), in matrix form (three dimensions) with no cross-coupling effects among the dimensions.

$$\begin{aligned}
& \begin{bmatrix} J_{Bxx} & 0 & 0 \\ 0 & J_{Byy} & 0 \\ 0 & 0 & J_{Bzz} \end{bmatrix} \begin{Bmatrix} \ddot{\theta}_{Bx} \\ \ddot{\theta}_{By} \\ \ddot{\theta}_{Bz} \end{Bmatrix} + k \left(\begin{Bmatrix} \theta_{Bx} \\ \theta_{By} \\ \theta_{Bz} \end{Bmatrix} - \begin{bmatrix} \cos \alpha & \sin \alpha \cos \gamma_1 & \sin \alpha \sin \gamma_1 \\ -\sin \alpha & \cos \alpha \cos \gamma_1 & \cos \alpha \sin \gamma_1 \\ 0 & -\sin \gamma_1 & \cos \gamma_1 \end{bmatrix} \begin{Bmatrix} \gamma_1 \\ 0 \\ 0 \end{Bmatrix} \right) \dots \\
& + k \left(\begin{Bmatrix} \theta_{Bx} \\ \theta_{By} \\ \theta_{Bz} \end{Bmatrix} - \begin{bmatrix} \cos \beta & \sin \beta \cos \gamma_2 & \sin \beta \sin \gamma_2 \\ -\sin \beta & \cos \beta \cos \gamma_2 & \cos \beta \sin \gamma_2 \\ 0 & -\sin \gamma_2 & \cos \gamma_2 \end{bmatrix} \begin{Bmatrix} \gamma_2 \\ 0 \\ 0 \end{Bmatrix} \right) \dots \quad (40) \\
& + k \left(\begin{Bmatrix} \theta_{Bx} \\ \theta_{By} \\ \theta_{Bz} \end{Bmatrix} - \begin{bmatrix} \cos \gamma_3 & \sin \gamma_3 & 0 \\ -\sin \gamma_3 & \cos \gamma_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} 0 \\ 0 \\ \gamma_3 \end{Bmatrix} \right) = \begin{Bmatrix} T_{CMG_x} \\ T_{CMG_y} \\ T_{CMG_z} \end{Bmatrix}
\end{aligned}$$

$$\begin{bmatrix} J_{A1xx} & 0 & 0 \\ 0 & J_{A1yy} & 0 \\ 0 & 0 & J_{A1zz} \end{bmatrix} \begin{Bmatrix} \ddot{\gamma}_1 \\ 0 \\ 0 \end{Bmatrix} + k \left(\begin{Bmatrix} \gamma_1 \\ 0 \\ 0 \end{Bmatrix} - \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha \cos \gamma_1 & \cos \alpha \cos \gamma_1 & -\sin \gamma_1 \\ \sin \alpha \sin \gamma_1 & \cos \alpha \sin \gamma_1 & \cos \gamma_1 \end{bmatrix} \begin{Bmatrix} \theta_{Bx} \\ \theta_{By} \\ \theta_{Bz} \end{Bmatrix} \right) = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \quad (41)$$

$$\begin{bmatrix} J_{A2xx} & 0 & 0 \\ 0 & J_{A2yy} & 0 \\ 0 & 0 & J_{A2zz} \end{bmatrix} \begin{Bmatrix} \ddot{\gamma}_2 \\ 0 \\ 0 \end{Bmatrix} + k \left(\begin{Bmatrix} \gamma_2 \\ 0 \\ 0 \end{Bmatrix} - \begin{bmatrix} \cos \beta & -\sin \beta & 0 \\ \sin \beta \cos \gamma_2 & \cos \beta \cos \gamma_2 & -\sin \gamma_2 \\ \sin \beta \sin \gamma_2 & \cos \beta \sin \gamma_2 & \cos \gamma_2 \end{bmatrix} \begin{Bmatrix} \theta_{Bx} \\ \theta_{By} \\ \theta_{Bz} \end{Bmatrix} \right) = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \quad (42)$$

$$\begin{bmatrix} J_{A3xx} & 0 & 0 \\ 0 & J_{A3yy} & 0 \\ 0 & 0 & J_{A3zz} \end{bmatrix} \begin{Bmatrix} 0 \\ 0 \\ \ddot{\gamma}_3 \end{Bmatrix} + k \left(\begin{Bmatrix} 0 \\ 0 \\ \gamma_3 \end{Bmatrix} - \begin{bmatrix} \cos \gamma_3 & -\sin \gamma_3 & 0 \\ \sin \gamma_3 & \cos \gamma_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} \theta_{Bx} \\ \theta_{By} \\ \theta_{Bz} \end{Bmatrix} \right) = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \quad (43)$$

Equation sets (44) through (46) show the EOM rewritten as uncoupled equations from Equations (40) through (43), regrouped to address motion in the X, Y, and Z principal axes of the central body.

$$\begin{aligned}
J_{B_{xx}} \ddot{\theta}_{B_x} + k \left[(\theta_{B_x} - \cos \theta \cdot \gamma_1) + (\theta_{B_x} - \cos \beta \cdot \gamma_2) + (\theta_{B_x} - 0) \right] &= T_{CMG_x} \\
J_{A1_{xx}} \ddot{\gamma}_1 + k \left[\gamma_1 - (\cos \alpha \cdot \theta_{B_x} - \sin \alpha \cdot \theta_{B_y}) \right] &= 0 \\
J_{A2_{xx}} \ddot{\gamma}_2 + k \left[\gamma_2 - (\cos \beta \cdot \theta_{B_x} - \sin \beta \cdot \theta_{B_y}) \right] &= 0 \\
J_{A3_{xx}} (0) + k \left[(0) - (\cos \gamma_3 \cdot \theta_{B_x} - \sin \gamma_3 \cdot \theta_{B_y}) \right] &= (\sin \gamma_3 \cdot \theta_{B_y} - \cos \gamma_3 \cdot \theta_{B_x}) = 0
\end{aligned} \tag{44}$$

$$\begin{aligned}
J_{B_{yy}} \ddot{\theta}_{B_y} + k \left[(\theta_{B_y} + \sin \alpha \cdot \gamma_1) + (\theta_{B_y} + \sin \beta \cdot \gamma_2) + (\theta_{B_y} - 0) \right] &= T_{CMG_y} \\
J_{A1_{yy}} (0) + k \left[0 - (\sin \alpha \cdot \cos \gamma_1 \cdot \theta_{B_x} + \cos \alpha \cdot \cos \gamma_1 \cdot \theta_{B_y} - \sin \gamma_1 \cdot \theta_{B_z}) \right] &= 0 \\
J_{A2_{yy}} (0) + k \left[0 - (\sin \beta \cdot \cos \gamma_2 \cdot \theta_{B_x} + \cos \beta \cdot \cos \gamma_2 \cdot \theta_{B_y} - \sin \gamma_2 \cdot \theta_{B_z}) \right] &= 0 \\
J_{A3_{yy}} (0) + k \left[0 - (\sin \gamma_3 \cdot \theta_{B_x} + \cos \gamma_3 \cdot \theta_{B_y}) \right] &= 0
\end{aligned} \tag{45}$$

$$\begin{aligned}
J_{B_{zz}} \ddot{\theta}_{B_z} + k \left[(\theta_{B_z} - 0) + (\theta_{B_z} - 0) + (\theta_{B_z} - \gamma_3) \right] &= J_{B_{zz}} \ddot{\theta}_{B_z} + k (3\theta_{B_z} - \gamma_3) = T_{CMG_z} \\
J_{A1_{zz}} (0) + k \left[0 - (\sin \alpha \cdot \sin \gamma_1 \cdot \theta_{B_x} + \cos \alpha \cdot \sin \gamma_1 \cdot \theta_{B_y} + \cos \gamma_1 \cdot \theta_{B_z}) \right] &\dots \\
&= \sin \alpha \cdot \sin \gamma_1 \cdot \theta_{B_x} + \cos \alpha \cdot \sin \gamma_1 \cdot \theta_{B_y} + \cos \gamma_1 \cdot \theta_{B_z} = 0 \\
J_{A2_{zz}} (0) + k \left[0 - (\sin \beta \cdot \sin \gamma_2 \cdot \theta_{B_x} + \cos \beta \cdot \sin \gamma_2 \cdot \theta_{B_y} + \cos \gamma_2 \cdot \theta_{B_z}) \right] &\dots \\
&= \sin \beta \cdot \sin \gamma_2 \cdot \theta_{B_x} + \cos \beta \cdot \sin \gamma_2 \cdot \theta_{B_y} + \cos \gamma_2 \cdot \theta_{B_z} = 0 \\
J_{A3_{zz}} (\dot{\gamma}_3) + k \left[\gamma_3 - (\theta_{B_z}) \right] &= J_{A3_{zz}} \dot{\gamma}_3 + k (\gamma_3 - \theta_{B_z}) = 0
\end{aligned} \tag{46}$$

Equations (44) through (46) simplify to the following set of six EOM:

$$\begin{aligned}
J_{xx} \ddot{\theta}_x + k (3\theta_x - \cos \alpha \cdot \gamma_1 - \cos \beta \cdot \gamma_2) &= T_x \\
J_{yy} \ddot{\theta}_y + k (3\theta_y + \sin \alpha \cdot \gamma_1 + \sin \beta \cdot \gamma_2) &= T_y \\
J_{zz} \ddot{\theta}_z + k (3\theta_z - \gamma_3) &= T_z \\
J_{A1_x} \ddot{\gamma}_1 + k (-\cos \alpha \cdot \theta_x + \sin \alpha \cdot \theta_y + \gamma_1) &= 0 \\
J_{A2_x} \ddot{\gamma}_2 + k (-\cos \beta \cdot \theta_x + \sin \beta \cdot \theta_y + \gamma_2) &= 0 \\
J_{A3_z} \ddot{\gamma}_3 + k (-\theta_z + \gamma_3) &= 0
\end{aligned} \tag{47}$$

where (J_{xx}, J_{yy}, J_{zz}) are the principle moments of inertia of the combined BRMSS from Equation (37). This representation neglects cross-coupling terms in order to allow independent control of the BRMSS in each principal axis. These EOM describe the BRMSS system and may be used to design a controller that takes into account both the rigid-body dynamics and the flexible appendage motion.

C. CONTROL OF THE BRMS SIMULATOR

1. State-Space System Representation

The BRMSS dynamics and control can be represented as a linear time-invariant (LTI) system with noise,

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) + \mathbf{v}(t) \\ \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) + \mathbf{w}(t),\end{aligned}\tag{48}$$

where $\mathbf{x}$ is the system state vector, $\mathbf{y}$ is the observable states (a subset of $\mathbf{x}$), and $\mathbf{u}$ is the vector of controls. The matrix $\mathbf{A}$ is the system (or plant), $\mathbf{B}$ is the input matrix, $\mathbf{C}$ is the output matrix (that selects the states observable as output), and $\mathbf{D}$ is the feed-forward matrix (nominally set to $[\mathbf{0}]$). The vectors $\mathbf{v}$ and $\mathbf{w}$ represent system noise: $\mathbf{v}$ is the noise due to model uncertainty, and $\mathbf{w}$ is the noise due to measurement uncertainty or sensor error. Although the true BRMSS system is time-varying (i.e., $\mathbf{A} = \mathbf{A}(t)$, $\mathbf{B} = \mathbf{B}(t)$, $\mathbf{C} = \mathbf{C}(t)$, $\mathbf{D} = \mathbf{D}(t)$), it may be represented as an LTI system by making two simplifications: neglecting the MOI cross-coupling of the BRMSS system and appendages, and eliminating the time-dependence of the system MOI ($\gamma = 0$ for J_{xx} , J_{yy} , J_{zz} calculations only). The angles of the flexible appendages (γ_i in Equation (47)) are still allowed to vary.

Converting the EOM of Equation (47) into state-space form gives the first system equation

$$\dot{\mathbf{x}}(t) = \mathbf{A} \cdot \mathbf{x}(t) + \mathbf{B} \cdot \mathbf{u}(t) + \mathbf{v}(t)$$

$$\Downarrow$$

$$\begin{Bmatrix} \ddot{\theta}_x \\ \ddot{\theta}_y \\ \ddot{\theta}_z \\ \ddot{\gamma}_1 \\ \ddot{\gamma}_2 \\ \ddot{\gamma}_3 \\ \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \\ \dot{\gamma}_1 \\ \dot{\gamma}_2 \\ \dot{\gamma}_3 \end{Bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \frac{-3k}{J_{xx}} & 0 & 0 & \frac{k \cos \alpha}{J_{xx}} & \frac{k \cos \beta}{J_{xx}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-3k}{J_{yy}} & 0 & \frac{-k \sin \alpha}{J_{yy}} & \frac{-k \sin \beta}{J_{yy}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-3k}{J_{zz}} & 0 & 0 & \frac{k}{J_{zz}} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{k \cos \alpha}{J_{A1x}} & \frac{-k \sin \alpha}{J_{A1x}} & 0 & \frac{-k}{J_{A1x}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{k \cos \beta}{J_{A2x}} & \frac{-k \sin \beta}{J_{A2x}} & 0 & 0 & \frac{-k}{J_{A2x}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{k}{J_{A3z}} & 0 & 0 & \frac{-k}{J_{A3z}} \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \\ \dot{\gamma}_1 \\ \dot{\gamma}_2 \\ \dot{\gamma}_3 \\ \theta_x \\ \theta_y \\ \theta_z \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{Bmatrix} + \begin{bmatrix} \frac{1}{J_{xx}} & 0 & 0 \\ 0 & \frac{1}{J_{yy}} & 0 \\ 0 & 0 & \frac{1}{J_{zz}} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} T_{CMG_x} \\ T_{CMG_y} \\ T_{CMG_z} \end{Bmatrix} + \begin{Bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \\ v_{10} \\ v_{11} \\ v_{12} \end{Bmatrix}.$$

(49)

In the BRMSS, the only directly observable states are the angular rates of the complete system about each principal axis $(\dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_z)$. The rates can be integrated to obtain the BRMSS system angles $(\theta_x, \theta_y, \theta_z)$, but the system angles are not *directly* observed. This means that the vector of **y** observable states in the BRMSS system is described as

$$\begin{aligned}
\mathbf{y}(t) &= \mathbf{C} \cdot \mathbf{x}(t) + \mathbf{D} \cdot \mathbf{u}(t) + \mathbf{w}(t) \\
&\Updownarrow \\
\begin{Bmatrix} \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \\ \dot{\gamma}_1 \\ \dot{\gamma}_2 \\ \dot{\gamma}_3 \\ \theta_x \\ \theta_y \\ \theta_z \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{Bmatrix} &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \\ \dot{\gamma}_1 \\ \dot{\gamma}_2 \\ \dot{\gamma}_3 \\ \theta_x \\ \theta_y \\ \theta_z \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{Bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} T_{CMG_x} \\ T_{CMG_y} \\ T_{CMG_z} \end{Bmatrix} + \begin{Bmatrix} w_1 \\ w_2 \\ w_3 \end{Bmatrix}. \quad (50)
\end{aligned}$$

The noise values $v_1 \dots v_{12}$ in Equation (49) and w_1, w_2, w_3 in Equation (50) are random variables with zero-mean and constant covariance. The level of measurement noise $\mathbf{w}(t)$ is determined in part using precision characteristics published by the sensor manufacturers. The level of model uncertainty $\mathbf{v}(t)$ is based on the inaccuracies of the model, e.g., assuming linearity or time-invariance.

Representing the BRMSS in state-space format facilitates the use of a more sophisticated controller for the system. A linear quadratic Gaussian controller enables both estimation of the unobserved states and compensation for the model uncertainty and measurement noise.

2. Linear-Quadratic-Gaussian Controller

The BRMSS system is currently operated using a PD controller that multiplies the measured error in central body rates $\dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_z$ and angles $\theta_x, \theta_y, \theta_z$ (determined through integration of the rates) by constant gains to calculate the required control torque. A more sophisticated controller is the linear quadratic regulator (LQR) controller, which minimizes a quadratic performance index, or cost, subject to constraints imposed by the

linear system representation of the BRMSS EOM. The LQR controller is a fundamental example of the application of Pontryagin's Minimum Principle to linear dynamical systems; for a thorough derivation see [34]. Given the LTI system described in Equation (48), the goal is to generate an optimal feedback control gain $F(t)$ that satisfies the linear relationship

$$\mathbf{u}(t) = -F(t)\mathbf{x}(t) \quad (51)$$

and minimizes the quadratic cost function

$$J = \frac{1}{2} \mathbf{x}^T(T)F(T)\mathbf{x}(T) + \int_0^T (\mathbf{x}^T Q \mathbf{x} + \mathbf{u}^T R \mathbf{u}) dt \quad (52)$$

where Q and R are weighting matrices for the state and control vectors. The control gain matrix F is given by

$$F = R^{-1}B^T P \quad (53)$$

where P is the solution to the continuous time algebraic Riccati equation (CARE)

$$A^T P + PA - PBR^{-1}B^T P + Q = 0. \quad (54)$$

Detailed derivations and discussion of numerical methods for solving these equations may be found in [18], [34] and [35].

An important requirement for using a linear quadratic regulator is that the system in question must have the full state available, i.e., all elements of the state vector must be observable. In the rigid-body BRMSS model, only three states are observable: the rates about the central body axes $(\dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_z)$. This generates significant uncertainty in the feedback system. In cases like the BRMSS where the system is not completely observable, the control design must include an estimator in addition to the LQR. The LQR and the estimator together are considered to be *Linear-quadratic-Gaussian (LQG)* control. An LQG controller combines an LQR with a Kalman filter to estimate the unobservable states. It also accounts for the effects of Gaussian noise added to the system due to model uncertainty and measurement imprecision (v_i and w_i). The LQG controller form for the system in Equation (48) that minimizes the cost function in equation (52) is

$$\begin{aligned}\dot{\hat{\mathbf{x}}}(t) &= A(t)\hat{\mathbf{x}}(t) + B(t)\mathbf{u}(t) + K(t)(\mathbf{y}(t) - C(t)\hat{\mathbf{x}}(t)), \quad \hat{\mathbf{x}}(0) = E(\mathbf{x}(0)) \\ \mathbf{u}(t) &= -L(t)\hat{\mathbf{x}}(t)\end{aligned}\tag{55}$$

where K is the Kalman gain associated with the Kalman filter used to estimate the unobservable states, and L is the feedback gain matrix. The function E represents expectation (in this case, of the state $\mathbf{x}$ at initial time $t = 0$). The Kalman gain K is equal to

$$K(t) = P(t)C^T(t)W^{-1}(t)\tag{56}$$

where $P(t)$ is the solution estimator problem posed by the matrix Riccati differential equations

$$\begin{aligned}\dot{P}(t) &= A(t)P(t) + P(t)A^T(t) - P(t)C^T(t)W^{-1}(t)C(t)P(t) + V(t), \\ P(0) &= E(\mathbf{x}(0)\mathbf{x}^T(0)).\end{aligned}\tag{57}$$

Similarly, the feedback gain matrix L is equal to

$$L(t) = R^{-1}(t)B^T(t)S(t)\tag{58}$$

where $S(t)$ is the solution of the LQR problem expressed by the matrix Riccati differential equations

$$\begin{aligned}\dot{S}(t) &= A^T(t)S(t) + S(t)A(t) - S(t)B(t)R^{-1}(t)B^T(t)S(t) + Q(t), \\ S(T) &= F.\end{aligned}\tag{59}$$

The LQG controller expressed in Equation (55) is inserted into the forward path of a feedback loop with the plant described by Equations (49) and (50) to generate the controlled system depicted in Figure 53. Although both the BRMSS model and the LQG controller are expressed as state-space systems, the generic input $\mathbf{u}$ to the controller is in fact the computer observable state error $\mathbf{y}_e(t)$, and the generic output $\mathbf{y}$ of the controller is the calculated system control torque $\mathbf{u}(t)$ that is also the input to the BRMSS dynamics.

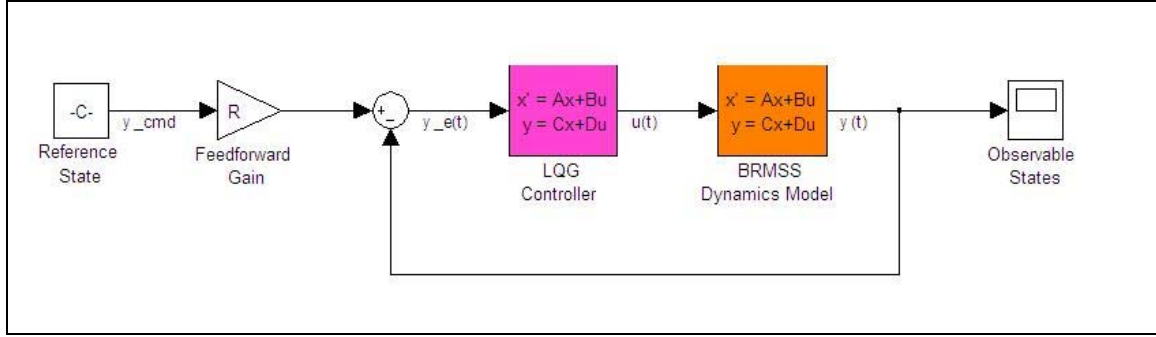


Figure 53. Block diagram of state-space BRMSS system with LQG controller.

One consequence of using an LQG controller is the presence of nonzero steady-state error e_{ss} in the solution. This error is eliminated by including a feedforward gain R in the path of the reference state that scales the reference command before it is used to compute the system error $y_e(t)$. The feedforward gain is included in Figure 53.

As implemented for the BRMSS in this research, the LQG controller requires six observable states as input: central-body rates $\dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_z$ and angles $\theta_x, \theta_y, \theta_z$. This enables the system to respond to commanded angles as well as rates. In the previous PD controller model, the central body angles were obtained by integrating the directly-observable rates. The true BRMSS uses sun sensors (with a simulated sun) as well as rate gyros to determine attitude rates and pointing angles. Therefore, the assumption for the flexible-structure LQG controller model that the central-body position and rates are observable is valid. The error accumulated in the real system due to any dependence of the angles on the rate measurements is accounted for in the covariance of the noise values v_i and w_i . The six remaining states (flexible appendage oscillation rates $\dot{\gamma}_1, \dot{\gamma}_2, \dot{\gamma}_3$ and angles $\gamma_1, \gamma_2, \gamma_3$) are truly unobservable and are therefore estimated using the Kalman filter in the LQG calculations.

3. Demonstrating LQG Control of the Flexible System

The enhancement of the BRMSS simulation was completed in phases. First the flexible structure model was developed to replace the rigid-body model and was tested with the classical (PD) controller; second an LQR controller was developed for the rigid-body model and tested in that environment. Then an LQR controller was generated for the flexible structure model, as if it were fully observable, to understand the nature of the optimized control for that system. Finally the LQG controller was developed for the flexible structure, and testing was completed on the fully updated simulation.

In each scenario, the independent variables were the gains (in PD control) or the weighting matrices (in the LQ control). There were four metrics observed that determined “good enough” control: steady-state error within one percent, settling time less than twenty seconds, overshoot less than two percent, and minimal (if any) amplitude of oscillation in steady-state solution. The cost in each case was the control effort $u(t)$, measured both by maximum value and by total effort about all axes $\left(\int_0^T |\mathbf{u}(t)| dt\right)$.

a. *Rigid-body model with classical control (original system)*

The original BRMSS system simulation used a rigid-body model with a classical PD controller operating on attitude quaternions (transformation from Euler angles) and rates (reference Figure 48). The arbitrary MOI for the rigid body was $(J_{xx}, J_{yy}, J_{zz}) = (15, 15, 25)$, and the gains were $(K_p, K_d) = (10, 10)$. Figure 54 depicts the attitude history of the system through a reference maneuver from $(0^\circ \ 0^\circ \ 0^\circ)$ to $(1^\circ \ 1^\circ \ 1^\circ)$. It shows that the settling time of the system is between 20 and 30 seconds, the overshoot is less than 20%, and there is no error (bias) or oscillation in the steady-state solution. Figure 55 shows the control history of the system through the same reference maneuver; it shows that the maximum control effort expended is approximately 0.09 N-m. The total control torque applied for the reference maneuver about all three axes is 25.3 N-m.

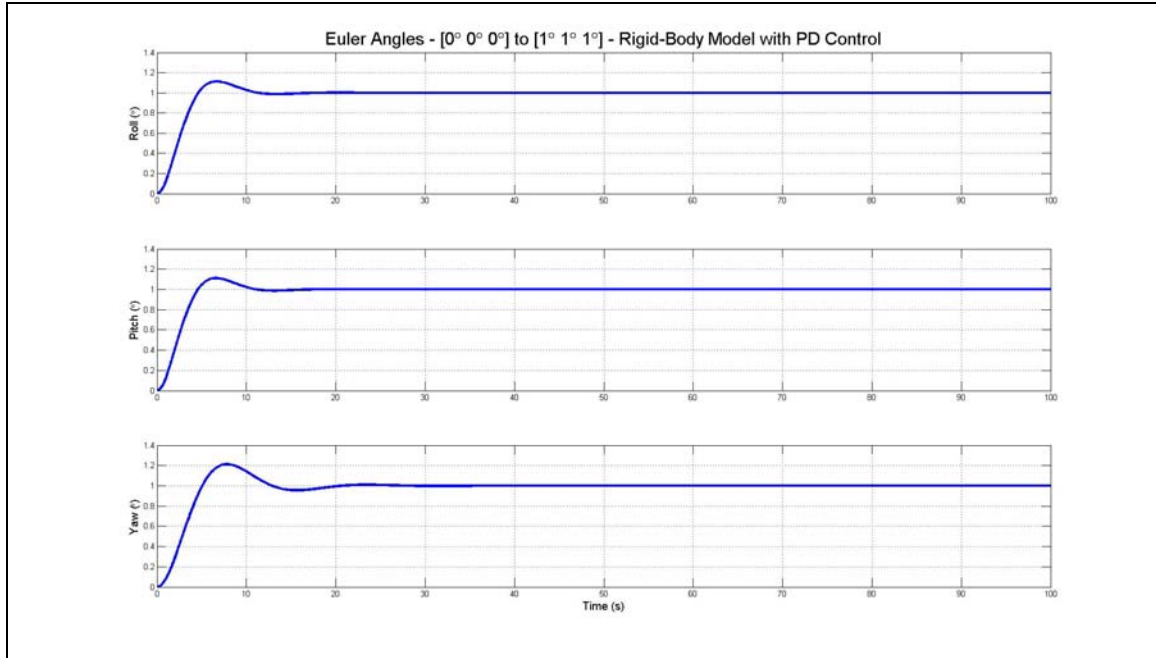


Figure 54. Attitude of Main Body (Rigid-Body Model with PD Control).

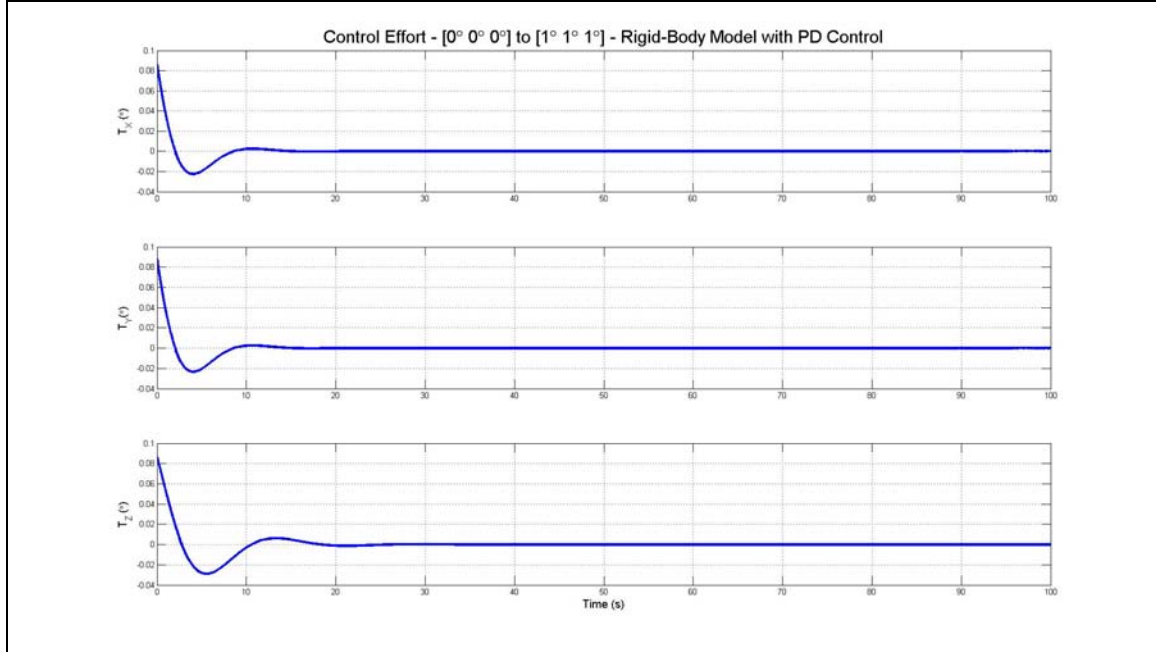


Figure 55. Control history (Rigid-Body Model with PD Control).

After determining the response of the original system to the reference maneuver, the first modification task was to replace the rigid-body BRMSS model with a new flexible structure model.

b. Flexible structure model with classical control

For the first system modification, the Euler dynamics model was replaced with the state-space model derived during this research for the flexible system. The flexible system has twelve total states with three observable states $(\dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_z)$ that are integrated to obtain the attitude angles $(\theta_x, \theta_y, \theta_z)$. Figure 56 and Figure 57 depict the attitude angles and control history, respectively, for this scenario. Figure 56 demonstrates that the settling time, overshoot and steady-state accuracy are comparable to that of the rigid-body model. Figure 57 shows that the control required for this model is considerably higher than for the rigid-body model, but this is reasonable since the MOI for the flexible-structure model is on the order of $(J_{xx}, J_{yy}, J_{zz}) = (200, 200, 300)$, which is an order of magnitude more than the rigid-body model. The total control torque required to execute the reference maneuver for the flexible structure using the PD controller is 377.2 N-m.

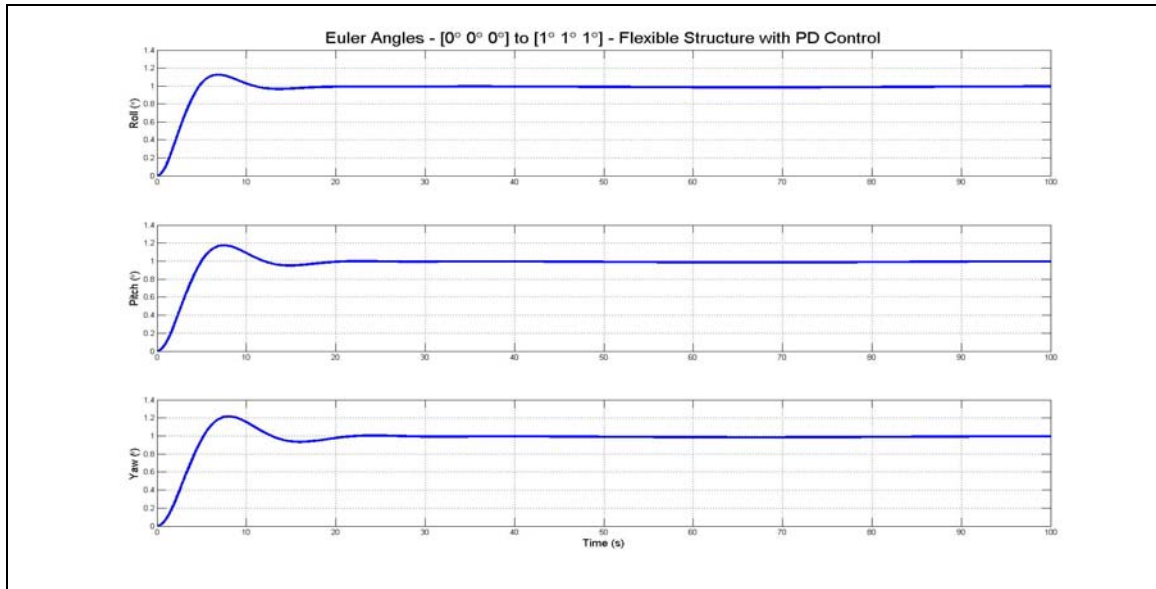


Figure 56. Attitude Angles (Flexible Structure with PD Control).

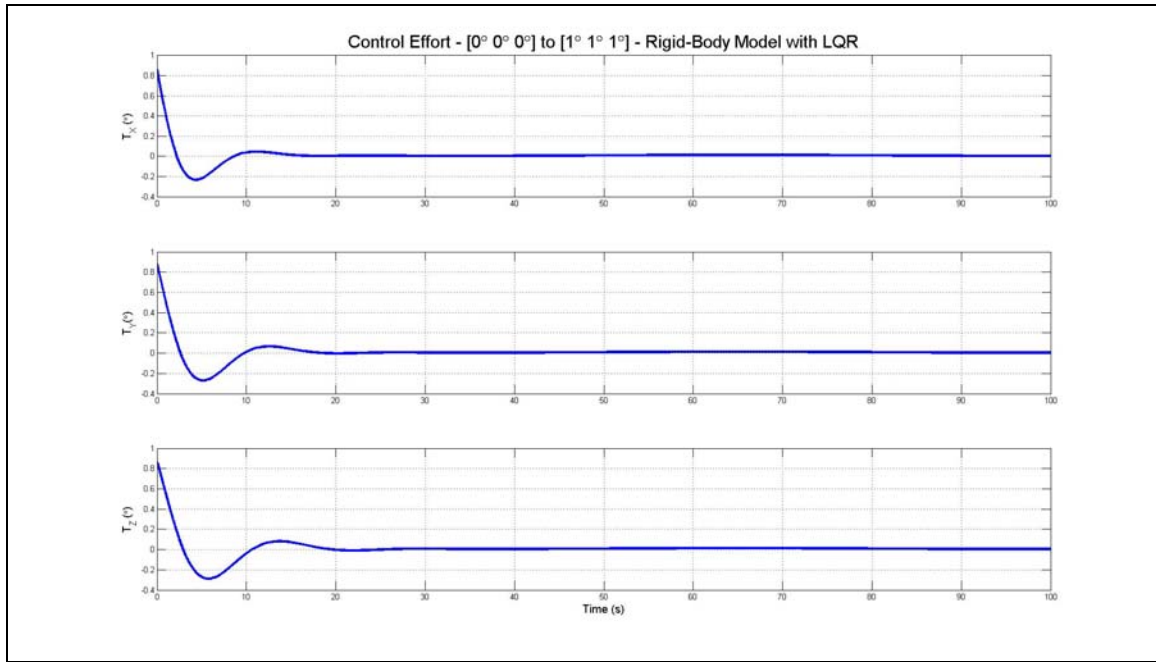


Figure 57. Control History (Flexible Structure with PD Control).

The next step in the modification process was to generate a new LQR controller for the rigid-body system using optimization methods and test its operation.

c. Rigid-body model with LQR control

For this system, the rigid-body EOM were converted into state-space form to enable the creation of a LQR controller based on the state matrix A , control matrix B , and weighting matrices Q and R . Figure 58 shows that the LQR controller generates a smoother response than the PD controller for the rigid-body model; the system also settles faster with the LQR controller. Figure 59 shows that the maximum control expended with the LQR controller is less than for the PD controller (less than 0.6 N-m vs. 0.1 N-m). The total control expended by the LQR controller to execute the reference maneuver with the rigid-body model is only 3.6 N-m – an improvement by a factor of six over the PD controller effort.

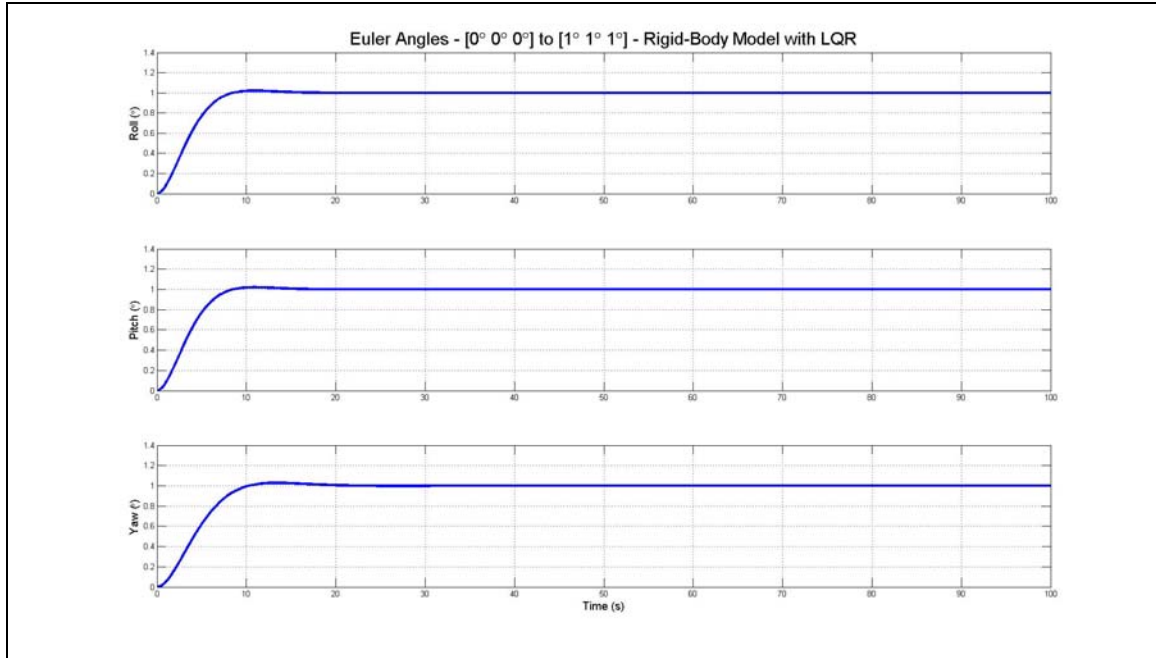


Figure 58. Attitude Angles (Rigid-Body Model with LQR Control).

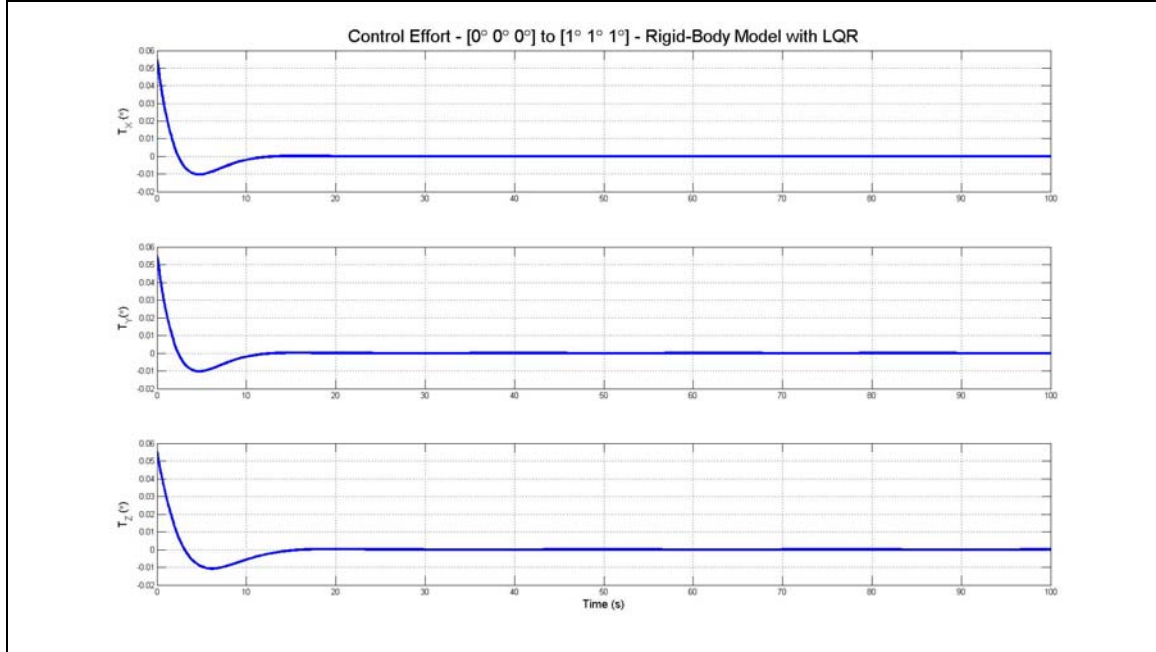


Figure 59. Control History (Rigid-Body Model with LQR Control).

After demonstrating the concept of the LQ controller improving the rigid-body system response at lower cost, the next task was to generate a LQR controller for the flexible structure model.

d. Flexible structure with LQR control

Instead of proceeding directly to the LQG controller, which combines two new functions (LQR for the flexible structure and the Kalman filter to estimate unknown states), the next system tested was a simulation of the flexible structure in full-state form – i.e., with all states directly observable. This eliminated the need for an estimator and enabled the design of an LQR controller for the flexible system.

Figure 60 shows the attitude angles through the reference maneuver for the LQR-controlled flexible structure. Although the LQR controller worked very quickly, it was necessary to increase the weighting on the main body angles $(\theta_x, \theta_y, \theta_z)$ by a factor of six in order to reduce the amplitude of a 0.01 Hz oscillation present in the system steady state. A byproduct of this was the very fast settling time of the system (under 10 seconds). Figure 61 shows the control history for this system – and indicates that the maximum torque about any axis was around 10 N-m. The total control for the maneuver was 315.4 N-m – less than for the classical controller, but still large because of the high gains required to dampen the steady-state oscillation.

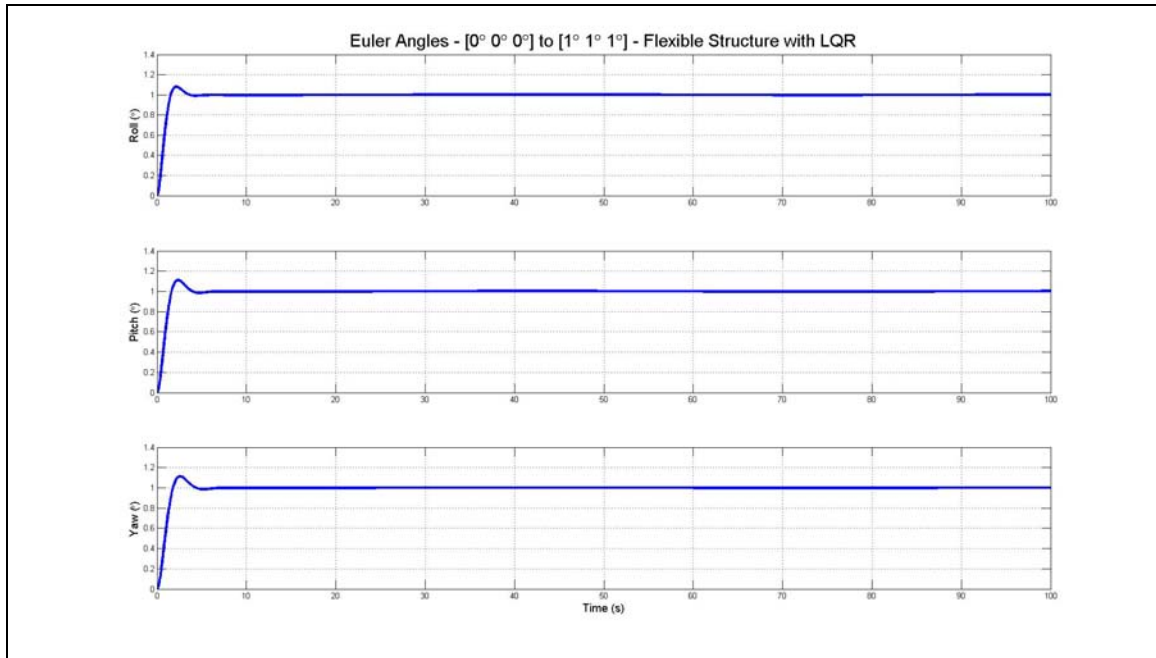


Figure 60. Attitude Angles (Flexible Structure with LQR Control).

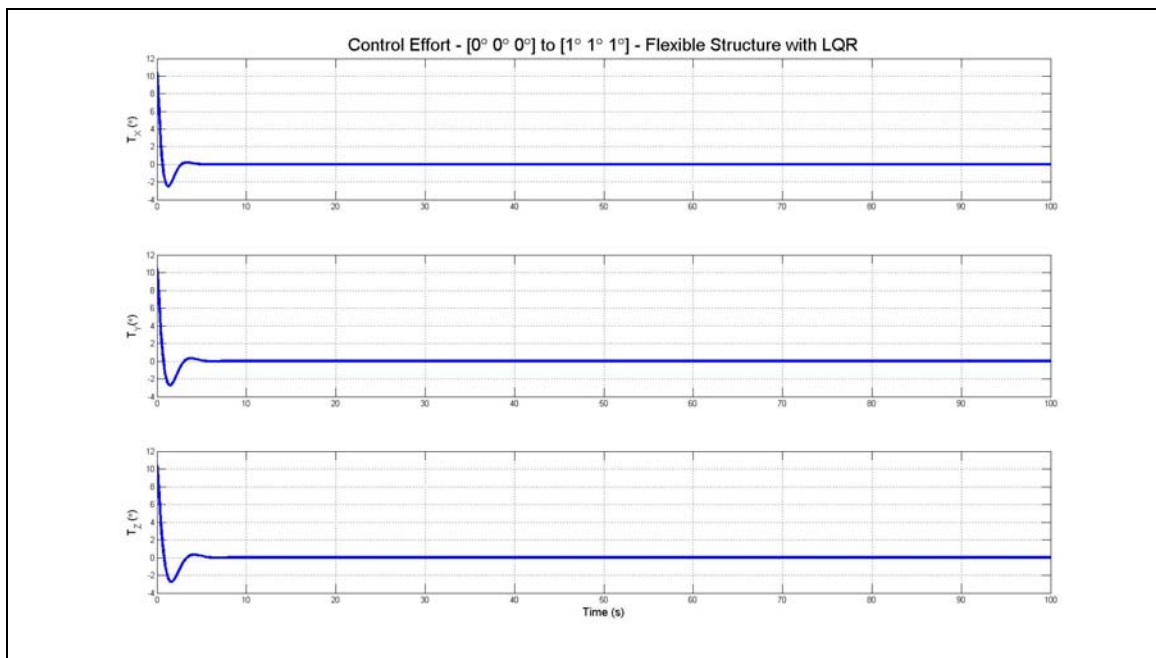


Figure 61. Control History (Flexible Structure with LQR Control).

e. Flexible Structure with LQG control

The final task in construction of the sophisticated model for the BRMSS was the generation of the LQG controller for the flexible-structure model. This controller required moderately high weighting factors on the angles $(\theta_x, \theta_y, \theta_z)$, similar to the requirement for the LQR controller, but the magnitude of the weighting factors did not need to be as high to achieve the same results. Figure 62 shows that the flexible-structure system response using the LQG controller reaches steady-state in under 10 seconds with less than 20% overshoot. It also has minimal steady-state error, with oscillations in the steady state reduced to negligible levels. Figure 63 depicts the control history for the LQG-controlled reference maneuver with the flexible-structure system; it shows that the maximum control torque in any axis is less than 6 N-m. The total control required for the reference maneuver using this system was 203.0 N-m. This is significantly lower cost than the PD or LQR-controlled systems. However, the cost is still high due to the presence of the steady-state 0.01 Hz oscillation in this system that must be controlled.

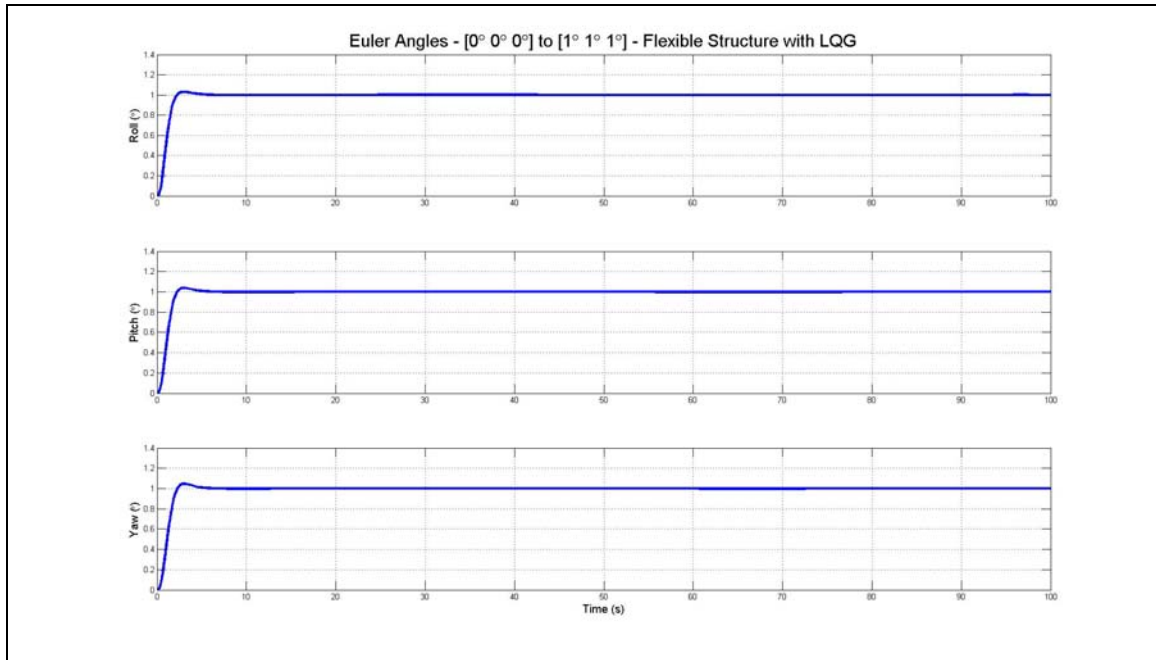


Figure 62. Attitude Angles (Flexible Structure with LQG Control).

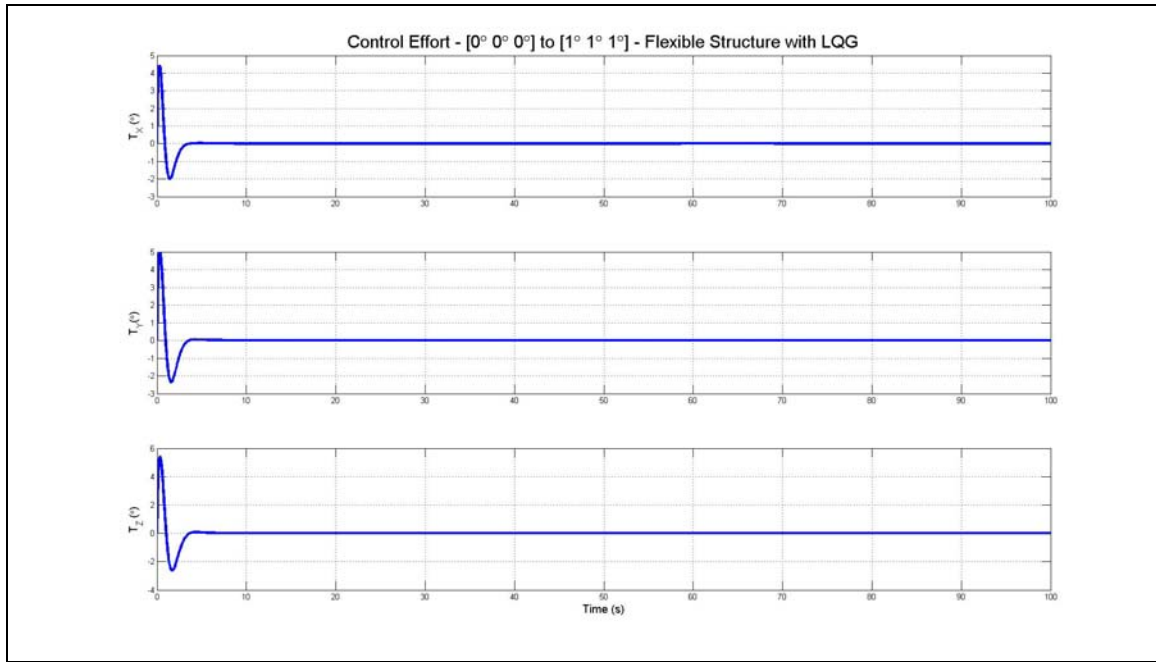


Figure 63. Control History (Flexible Structure with LQG Control).

f. Summary

A numeric summary and comparison of the results for each system is presented in Table 12. From this table, it is evident that in order to control the full flexible system at minimum cost, the linear-quadratic-Gaussian optimal control methods are desired. Further research in this area may focus on removing the oscillation entirely from the steady-state response of the flexible system.

Case	Gain/ Weight (rate, angle)	Steady-State Error (°)	θ_{ss} Oscill. Ampl. (°)	Settling Time (s)	Percent Overshoot	Control (Cost)
a	10, 10	0	0	30	12%	25.3
b	100, 100	0.01	0.012	30	20%	377.2
c	10, 10	0	0	30	3%	3.6
d	400, 600	5e-4	0.006	15	11%	315.4
e	100, 2e+6	1e-4	0.002	10	< 5%	203.0

Table 12. BRMSS Control System Simulation Results

When used in a spacecraft for attitude control, any of these systems would be implemented in either the main flight computer or an ADCS co-processor. The ADCS co-processor in particular is an ideal candidate for using reprogrammable hardware. If some of the actuator or sensor hardware were to become inoperable or unusable, the ADCS co-processor could be reprogrammed with a new algorithm that optimizes use of the sensor data or control torque that is still available. If the requirements for the ADCS were to change, e.g., the operators were to attempt a more aggressive slewing maneuver that required more expensive control than that of the original system design, the ADCS could be similarly reprogrammed with an algorithm optimized to a new cost metric developed for the more aggressive mission.

If the ADCS for a spacecraft is implemented in an FPGA, it will experience some level of SEU activity on orbit. Since spacecraft control involves multi-part systems (measurement, control, allocation) that require substantial memory in on-board processors, it is important to minimize the space and power required by the ADCS on its co-processor FPGA. In order to use the RPR techniques developed in this research with an ADCS like the BRMSS, it is necessary to associate the processes in the BRMSS control system with those previously identified operations that are most suitable for applying RPR.

D. APPLYING RPR TO THE BRMSS CONTROL SYSTEM

To apply RPR fault tolerance methods to the BRMSS control system, it is necessary to identify the fundamental operations in the control system processes. A simplified block diagram of the updated BRMSS control system (flexible structure model and LQG controller) is depicted in Figure 64. In order from left to right in Figure 64, the following nine processes occur, with associated computer operations:

1. Receive commanded state (six-element vector) – input/memory access
2. Convert degrees to radians – multiplication by a constant
3. Modify command with feed-forward gain – multiplication by a constant
4. Obtain state error (current/actual – commanded) – addition/subtraction
5. Generate control commands based on current state error – matrix multiplication, multiplication by a constant, addition/subtraction

6. Apply control to system – output to actuators (allocator problem)
7. Update system states – input from sensors (measurement problem)
8. Convert radians to degrees – multiplication by a constant
9. Report/store current state – output/memory access
- 10.

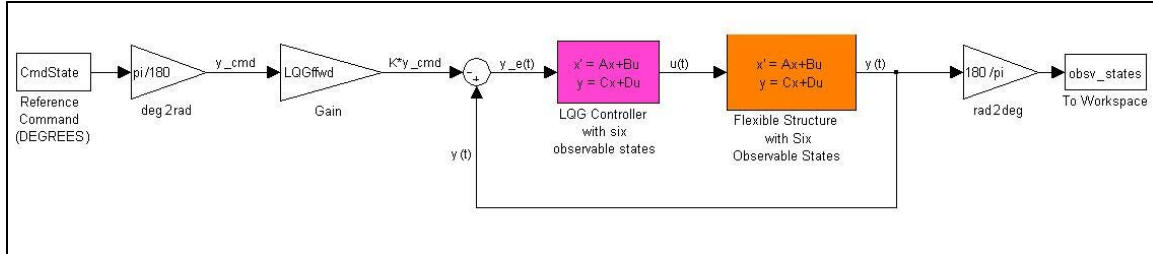


Figure 64. Updated BRMSS control system.

The most suitable operations for RPR are the arithmetic operations – namely addition/subtraction, multiplication, and combinations of these as they are found in matrix operations. The multiplications involving constants may be implemented such that the reduced-precision operations select only a subset of the MSB of the constants as they are stored in memory.

A designer of the RPR version of this ADCS would need to conduct additional performance trade studies before beginning implementation. One such trade is to compare speed and memory for storing constants. For example, the value for “ $180/\pi$ ” (radians-to-degrees) may be stored once with full precision, coded to guard against errors. It may also be stored in multiple locations – with full and reduced precision – to allow simultaneous access by all copies of a multiplication circuit implemented using RPR.

Trades between speed and memory space are only one category of complications that still remain for implementing RPR in a complicated system. Another residual issue is how often to correct configuration errors that have been detected: in a real-time system the accuracy of the configuration is highly important, but correction of the system configuration (particularly when the errors are not causing significant degradation in control performance) cannot interfere with regular operation. This and other questions are enumerated as part of the conclusion of this work.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY

The harsh radiation environment of space generates faults in FPGAs that affect both data and configuration memory. The Configurable Fault Tolerant Processor at the Naval Postgraduate School is a platform for testing methods of fault tolerance that guard against the single-event effects of radiation in FPGAs. In 2006 Snodgrass introduced a new method of fault tolerance, Reduced Precision Redundancy, as a power-saving alternative to traditional Triple Modular Redundancy. This research focused on the details of implementing RPR and the effect of RPR fault tolerance on the performance of spacecraft systems.

Two categories of system architectures were discussed: recursive data management, found in feedback control systems; and flow-through data management, found in signal processing tools such as the fast Fourier transform. Examples of the two architectures were broken down into their elementary operations, and the common operations were chosen as the subjects of experiments in RPR implementation. The “degree of RPR” was defined as a measure of reduction in precision. Detailed RPR designs for addition/subtraction and multiplication were programmed, simulated and mapped to the Virtex™ XQVR600 FPGA using the Xilinx Integrated Software Environment. Versions of each operation were built in TMR and several degrees of RPR, and the FPGA resources required for each degree of RPR were compared to the resources used by the corresponding TMR experiments. The results obtained from the detailed designs were extrapolated to estimate the resources required to implement RPR division and the compound operations of matrix multiplication and the fast Fourier transform butterfly machine.

An evaluation of RPR-protected system performance was conducted on models of recursive and flow-through data architecture systems using MATLAB and Simulink computational tools. Transient and persistent errors were modeled as delta and step functions of additive noise in the signal data flow, and RPR error correction was modeled

as an increase in signal-to-noise ratio whose magnitude depended on the degree of RPR. The improvement in system response and reduction in output error between “no fault tolerance” and “RPR fault tolerance” were measured to determine the impact of RPR on system performance.

One example system was also chosen to improve and assess as a complicated candidate system for RPR. A new dynamics model was developed for the Bifocal Relay Mirror Satellite Simulator testbed at NPS that described the effects of three flexible appendages, which expanded the model to a twelve-state system with limited observable output. A linear-quadratic Gaussian controller was developed for the flexible structure model by combining a Kalman filter state estimator with a cost-optimal linear quadratic regulator (LQR). The new dynamics and control system was tested using a reference maneuver, and control cost compared to the cost of using a simple PD controller. Finally, the enhanced BRMSS model was examined as a candidate system for RPR, and operations suitable for applying RPR were identified.

B. CONCLUSIONS

This research has shown that RPR is a viable fault tolerance approach for arithmetic operations. In order for RPR to be effective, the upper and lower bounds of the result must be generated in a certain manner depending on the operation being executed, paying particular attention to the signs of the operands. Also, the RPR voter must be constructed such that it conducts a numerical comparison of the MSB of the precise result with the bound results, as opposed to the bitwise comparison used in TMR.

Experimental results show that for the simplest operations, RPR is *not* always the most efficient fault tolerance approach. The increased complexity of an RPR voter over that of a TMR voter actually causes an addition operation protected at a high degree of RPR to be *larger* than its corresponding TMR-protected operation. For RPR to provide notable FPGA space savings over TMR in the simplest of operations, the degree of RPR must be less than 0.25. The significance of this result is that the findings by Snodgrass (that RPR provides a 50%-70% FPGA area and power savings) are largely application-dependent: if RPR is applied externally, as in the CORDIC processor Snodgrass

developed, RPR provides a much greater area and power savings (over TMR) than if RPR is applied internally. Essentially, the benefit of RPR increases with the complexity of the operation to which it is applied.

System performance simulations demonstrate that RPR provides very good recovery from errors caused by SEU in spacecraft systems. With a baseline precision n of 52 bits, even an approximate RPR result with only eight bits of precision drastically improved the transient and steady-state response of an attitude control system. Using the RPR result also reduced the relative error in frequency output values of an FFT to less than 0.2%. The performance simulations also demonstrated that the bandwidth of a feedback control system (dependent on processor speed and data I/O limitations) has significant impact on its ability to reject noise of any kind, which in turn affects the minimum acceptable degree of RPR for the system. This and other implementation considerations contribute to the design trade space of FPGA capacity and power, fault tolerance requirements and system performance metrics.

The in-depth investigation of the dynamics and control of a flexible space structure illustrate that even a complex system can be reduced to operations suitable for RPR. The time-dependent inertia, equations of motion and optimized linear-quadratic Gaussian controller developed are all different combinations of the arithmetic operations discussed in this work, interspersed with memory access and non-numerical logic functions. If a system such as the BRMSS ADCS were implemented on an FPGA-based computer, it could be made fault-tolerant using RPR.

A final word about RPR gleaned from this research is this: a combination of high- and low-level application appears to be the best use of RPR as a fault tolerance technique. Computing and propagating bounds from one low-level operation to the next allows good error control in a sequence of operations, which is necessary in a large system. However, reserving the RPR voter implementation for only a few major points within the system – or at its final output – keeps the overall cost (in FPGA area) of using RPR low compared to the cost of TMR. This combination of internal and external RPR could potentially be the most efficient RPR architecture for many types of processors, but additional experimentation is required before that can be postulated with confidence.

C. RECOMMENDATIONS FOR FUTURE STUDY

1. Investigating Internal vs. External RPR

Previous research in the CFTP group at NPS explored both external and internal TMR. In external TMR voters are placed at the output points of a processor and have no access to the intermediate results within the processor. Internal TMR designs use voters after individual operations or sub-processes within the main processor, and check intermediate outputs so that errors are never propagated far within the processor.

Snodgrass's initial implementation of RPR in a CORDIC processor applied RPR externally: the entire CORDIC algorithm was implemented in VHDL with full and reduced precision, and any voting logic operated on the output of the processors. The operations described in chapter III of this work are the building blocks for *internal* RPR – the benefit and computational cost of applying RPR internally must be evaluated using a full processor architecture that implements RPR at the single operation and/or sub-process level.

2. Fault Detection and Location Methods

The difference between data and configuration faults in FPGAs, and how to identify each with certainty, has been discussed at length in this work. Traditionally configuration faults in an FPGA are identified and corrected by “scrubbing” the FPGA at regular intervals and reloading the configuration if any bits are found to be incorrect. In order to proceed in this area of work, a dedicated quantitative study is necessary to analyze the benefits and drawbacks of using error monitoring (1) to distinguish errors due to configuration faults from errors due to data faults, (2) as a method of triggering configuration scrubs, and (3) to locate the source of a configuration error on an FPGA. These functions should be explored for their potential to increase the efficiency of an FPGA, whether it is protected by TMR or RPR.

3. Standard Degrees of RPR

In this thesis, arithmetic operations were implemented using combinations of $n \in (16, 32, 64)$ and $r \in (8, 16, 32)$. In practice, r is not limited to powers of two or even to multiples of two. Further study of more degrees of RPR is needed to generate a set of “standards” that may be used for applications with certain requirements on precision.

4. Implementing RPR in Floating Point Representation

All the VHDL or schematic FPGA design in this research was conducted using fixed-point numbers. The rules governing arithmetic and error accumulation are very different for IEEE (or other) floating-point representation – in fact, many “rules of algebra” are not even true for floating-point numbers [28]. Although an FPGA may be programmed using any numeric standard, interoperability with many general-purpose processors of today demands floating-point representation. The changes in behavior of RPR arithmetic operations when implemented using floating-point representation must be documented before attempting to build an RPR floating-point processor.

5. Performance Evaluation Using Hardware

The simulations in chapter IV suggest that the signal power level and power limits on the device or sensors used in a system have a significant effect on what degree of RPR is required to maintain control and minimize error in a system. The ADCS and/or FFT experiments should be implemented using physical devices in order to confirm the relationships between signal power, equivalent noise power and degree of RPR.

6. Comparing RPR to Other Fault-Tolerance Methods

A concept introduced but not investigated in this research was that of the benefits and drawbacks of using error correction codes vs. redundancy for preserving data integrity during storage and transmission. Two approaches to this trade merit further study: using some form of RPR as protection for data storage or FPGA configuration, and comparing RPR to coding checks such as residue arithmetic for smaller common

operations. The complexity of the RPR voter is such that it can be very powerful for checking complicated processes, but its benefit must be examined very carefully when working with small circuits. A detailed study is warranted on the viability of using RPR over a simpler check like residue addition for simple operations.

APPENDIX A. RPR MODULE DESIGN

A.1 REPRESENTATIVE RPR AND TMR ADDER OPERATION MODULES

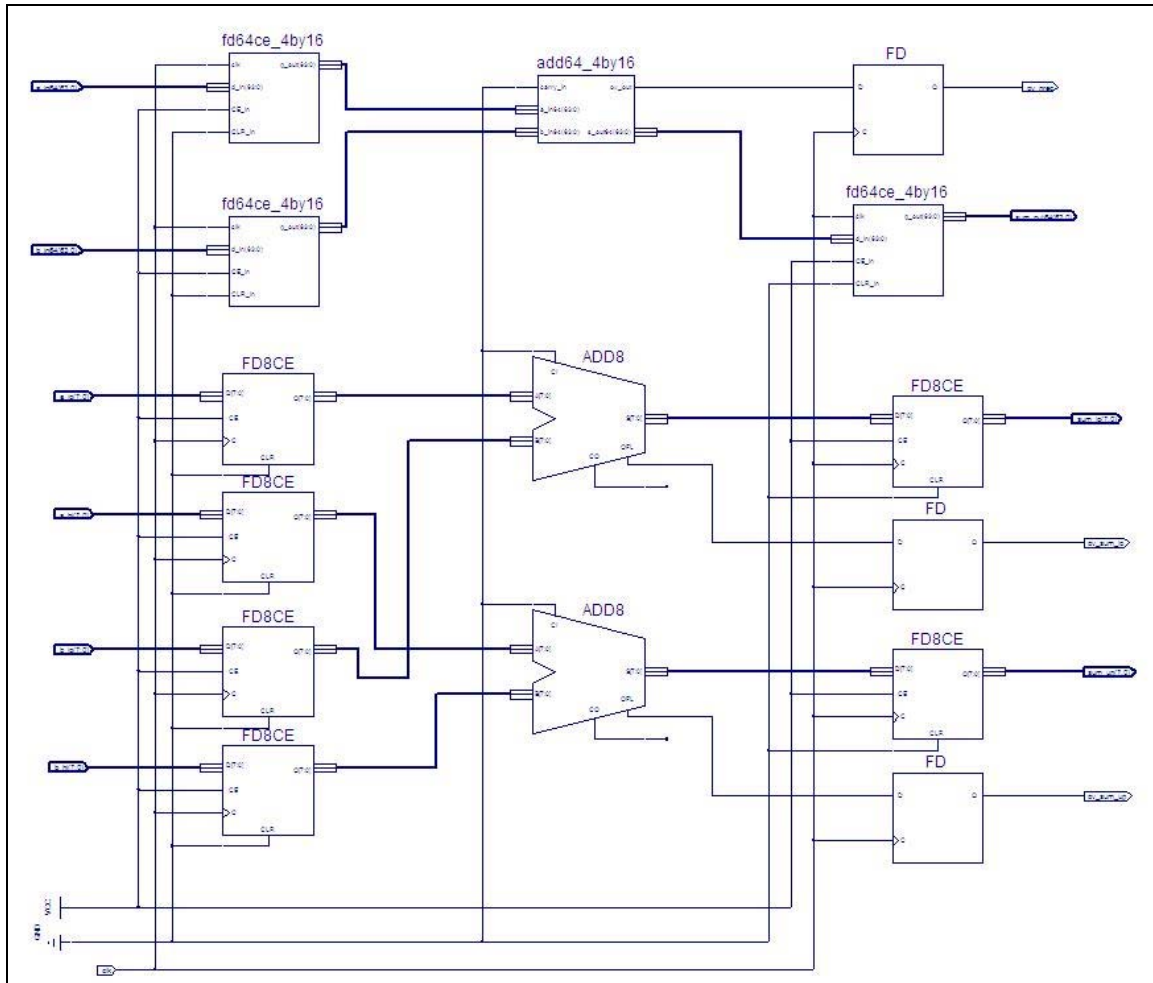


Figure 65. RPR 8/64 Addition - Operation Module.

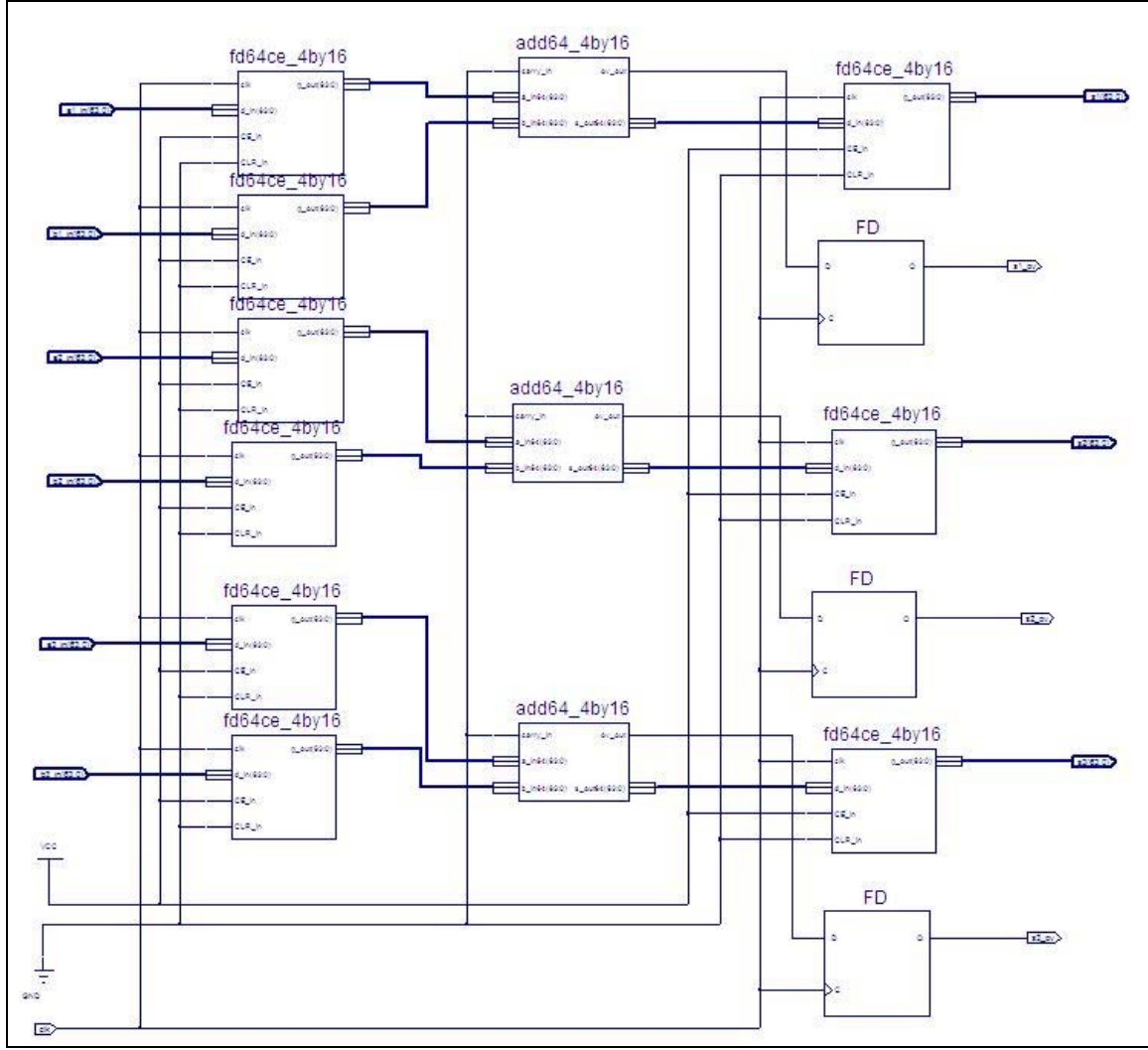


Figure 66. TMR 64-bit Addition - Operation Module.

A.2 REPRESENTATIVE RPR AND TMR ADDER VOTER MODULES

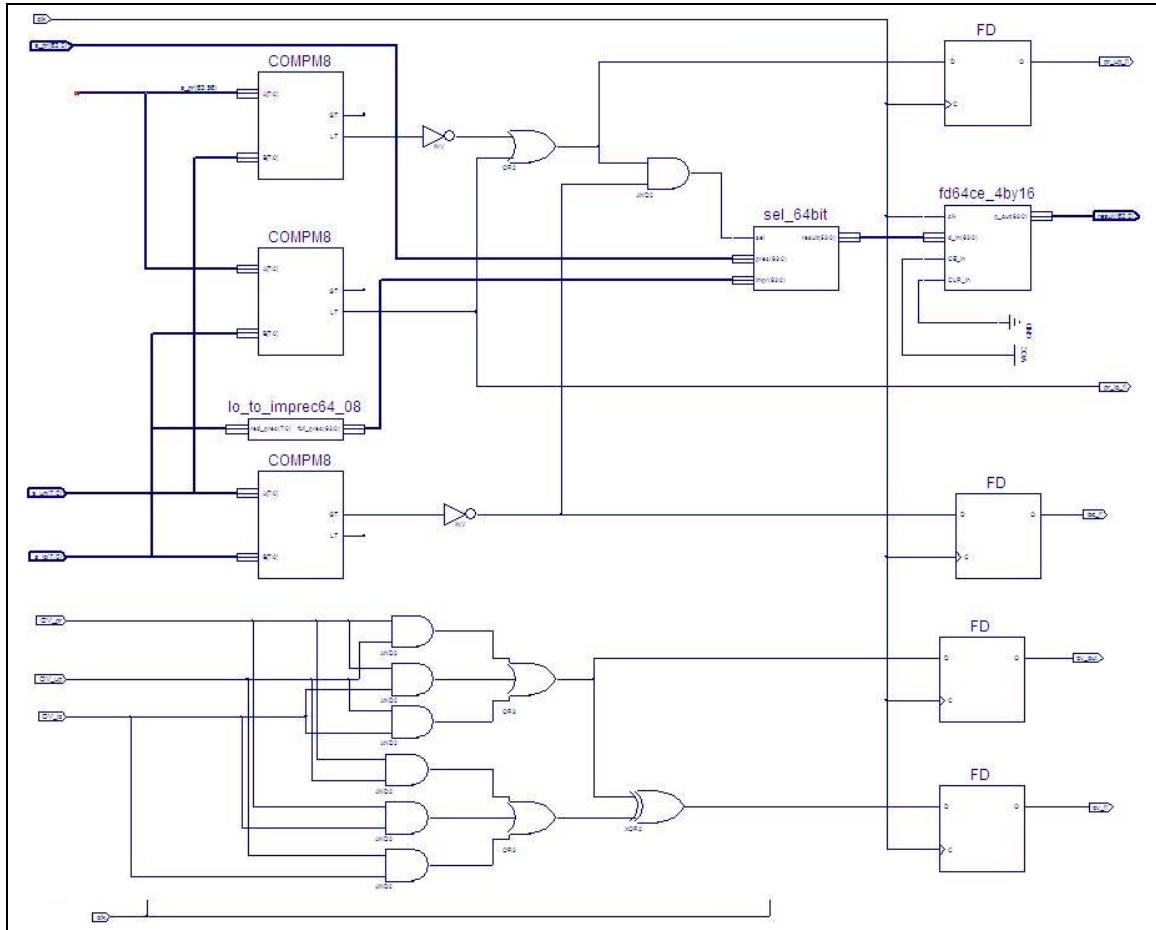


Figure 67. RPR 8/64 Addition - Voter Module.

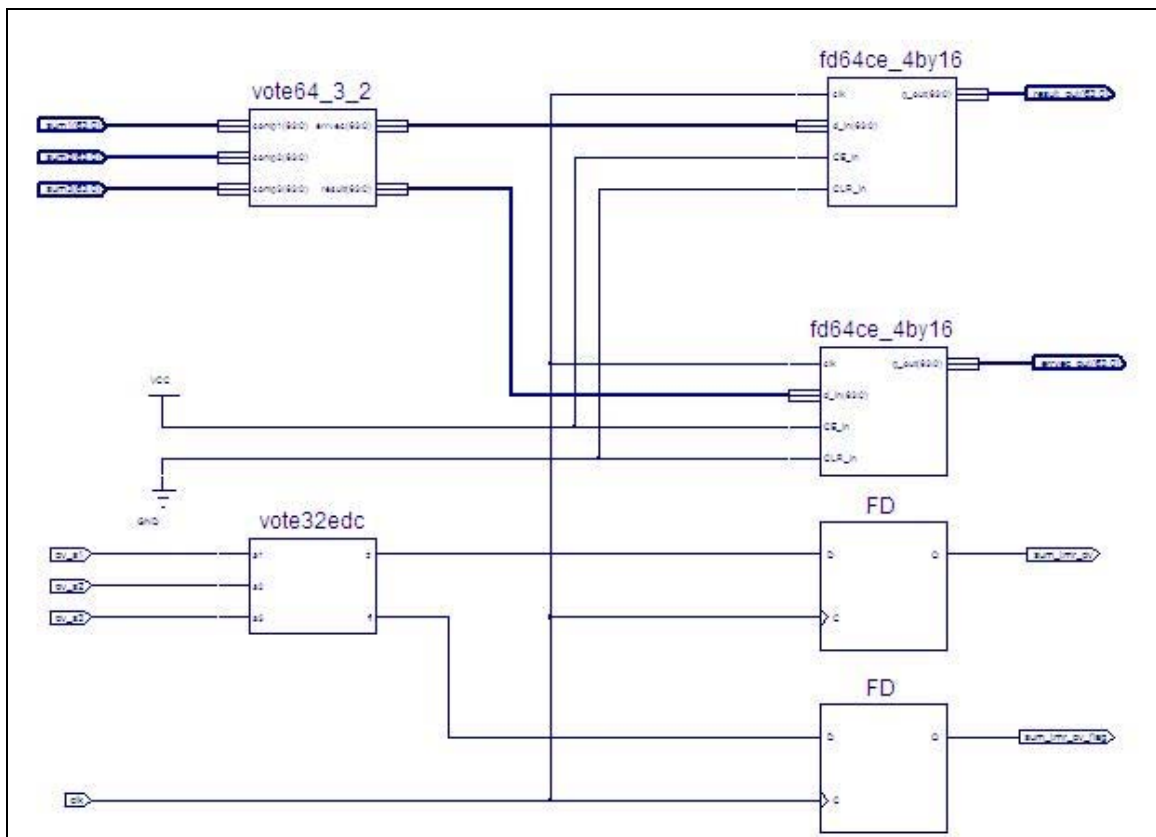


Figure 68. TMR 64-bit Addition - Voter Module.

A.3 REPRESENTATIVE RPR AND TMR MULTIPLIER OPERATION MODULES

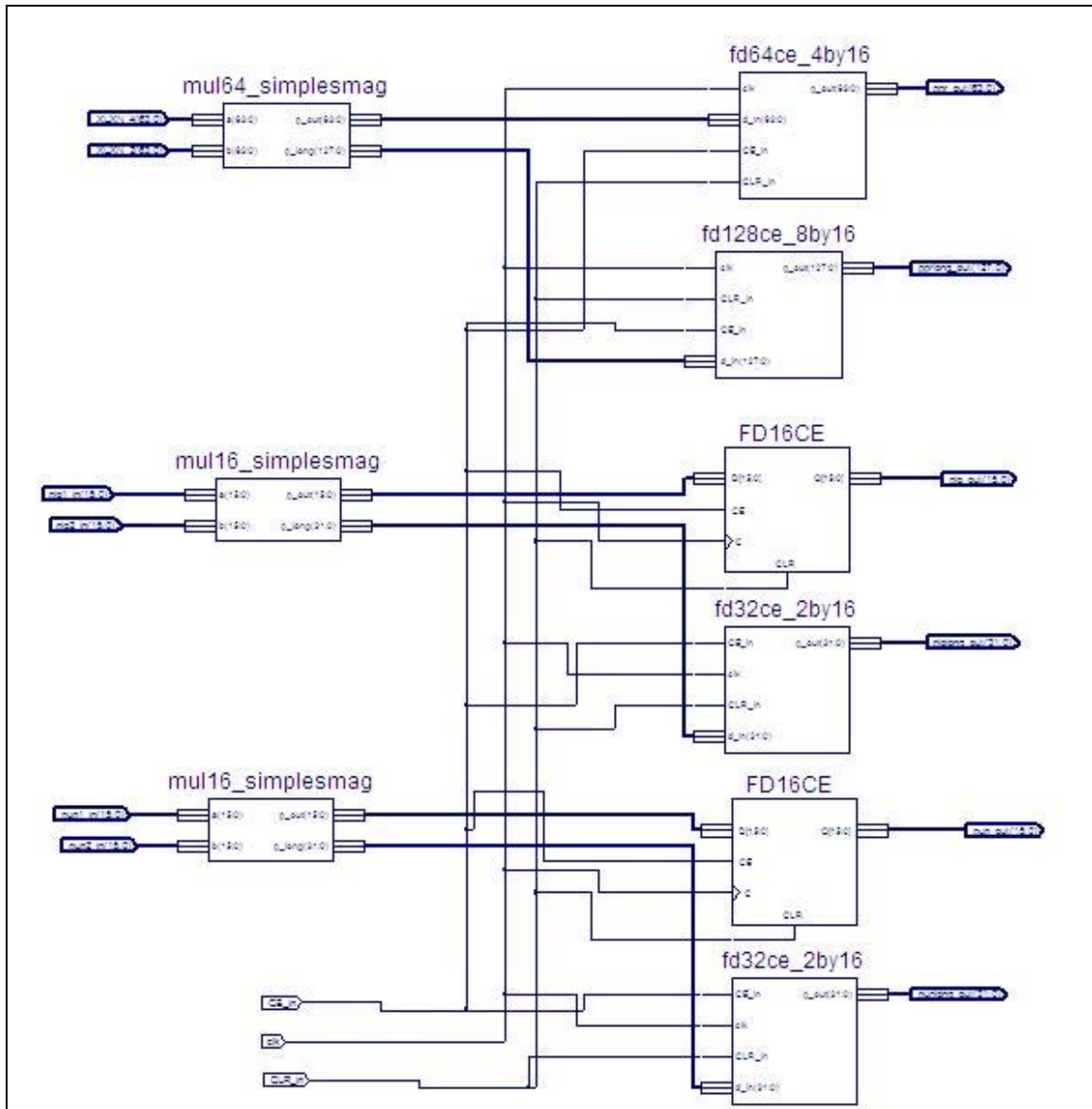


Figure 69. RPR 16/64 Multiplication - Operation Module.

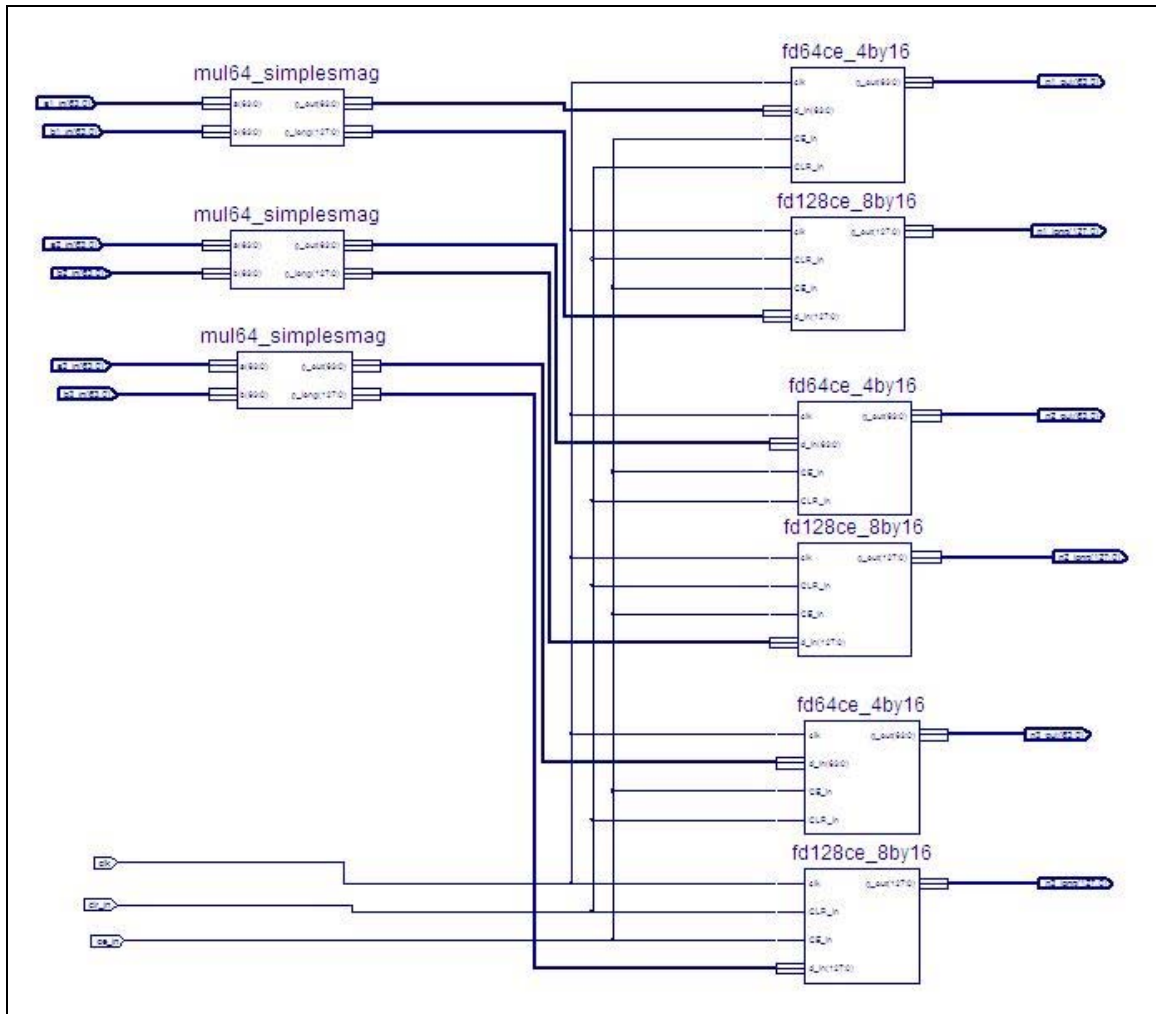


Figure 70. TMR 64-bit Multiplication - Operation Module.

A.4 REPRESENTATIVE RPR AND TMR MULTIPLIER VOTER MODULES

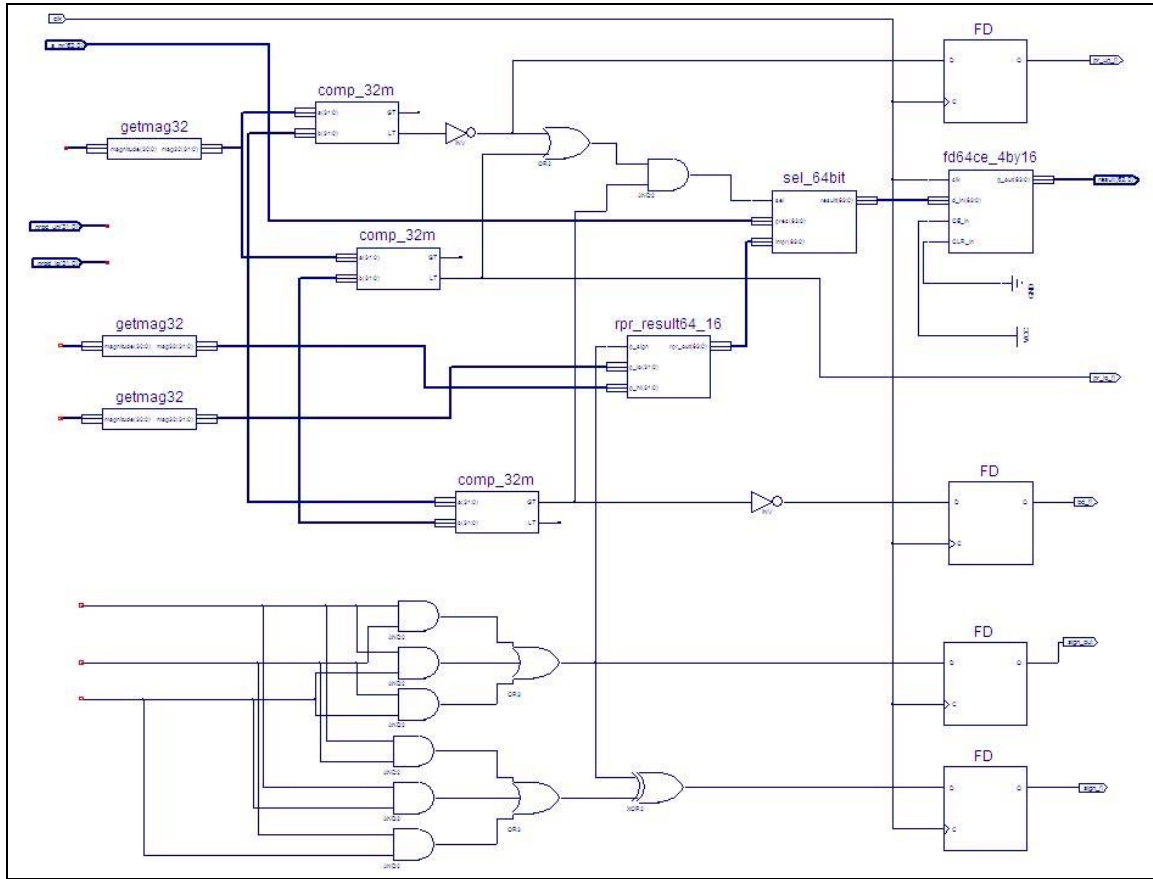


Figure 71. RPR 16/64 Multiplication - Voter Module with 2r-bit Comparators.

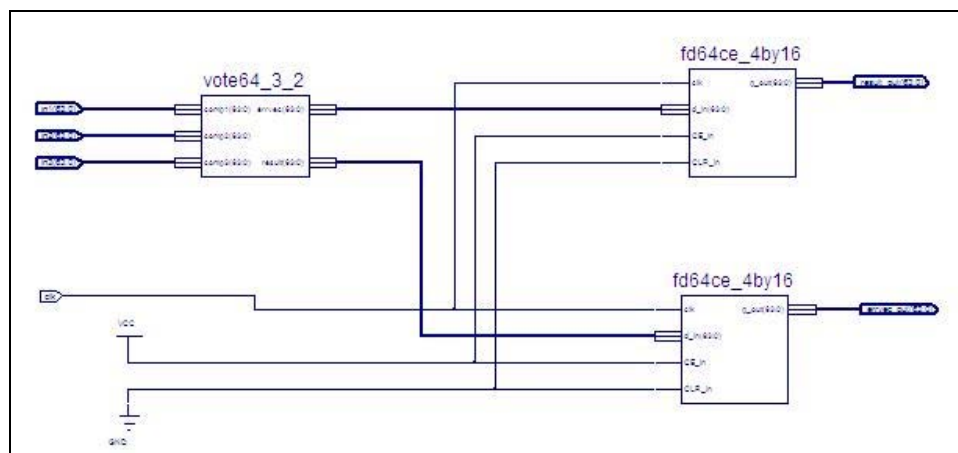


Figure 72. TMR 64-bit Multiplication - Voter Module (Bitwise Majority).

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. COMPOUND OPERATION BOUND TESTING

B.1 TESTING UPPER/LOWER BOUNDS ON MATRIX MULTIPLICATION

```
%% MANUAL MATRIX MULT WITH BOUND CALCS AND COMPARISON
clear all; close all; clc; format compact; format short

sizem = 5;                                % what size (square) matrix?
scale = 100;                              % scale for determining error in bounds
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
mat1 = 2*rand(sizem)-1                    % generate initial values
mat2 = 2*rand(sizem)-1

m1l = floor(scale*mat1)/scale              % generate lower bounds
m2l = floor(scale*mat2)/scale

m1u = ceil(scale*mat1);                   % generate upper bounds
m2u = ceil(scale*mat2);

for i=1:sizem                             % make UPPER bound < LOWER bound
    for j = 1:sizem
        if (m1u(i,j) == mat1(i,j))        % alter for integers
            m1u(i,j) = m1u(i,j) + 1;
        end
        if (m2u(i,j) == mat2(i,j))
            m2u(i,j) = m2u(i,j) + 1;
        end
    end
end
m1u = m1u/scale
m2u = m2u/scale

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
prod = mat1*mat2;                        % compute precise product

pl = zeros(sizem);    %mat1l*mat2l;    % INITIALIZE
pu = zeros(sizem);    %mat1u*mat2u;

for i=1:sizem          %fill rows of bound product matrices
    for j=1:sizem      %fill cols of bound product matrices

        for k=1:sizem % each term in the inner product
            if (mat1(i,k)>0 && mat2(k,j)>0) % prod>0          THIS IS EQ 12
                putemp = m1u(i,k)*m2u(k,j);
                pltemp = m1l(i,k)*m2l(k,j);
                pu(i,j) = pu(i,j) + putemp;
                pl(i,j) = pl(i,j) + pltemp;
            else if (mat1(i,k)<0 && mat2(k,j)>0) %prod<0
                putemp = m1u(i,k)*m2l(k,j);
                pltemp = m1l(i,k)*m2u(k,j);
                pu(i,j) = pu(i,j) + putemp;
                pl(i,j) = pl(i,j) + pltemp;
            else if (mat1(i,k)>0 && mat2(k,j)<0) %prod<0
                putemp = m1l(i,k)*m2u(k,j);
                pltemp = m1u(i,k)*m2l(k,j);
                pu(i,j) = pu(i,j) + putemp;
                pl(i,j) = pl(i,j) + pltemp;
            else if (mat1(i,k)<0 && mat2(k,j)<0) %prod>0
                putemp = m1l(i,k)*m2l(k,j);
                pltemp = m1u(i,k)*m2u(k,j);
                pu(i,j) = pu(i,j) + putemp;
                pl(i,j) = pl(i,j) + pltemp;
            end
        end
    end
end
```

```

        end
    end
end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
checku = gt(prod,pu) %conduct bound checks (should all be 0)
checkl = gt(pl,prod)
checkb = gt(pl,pu)
% PRINT ERROR MESSAGES, if applicable
for i=1:size
    %fill rows of bound product matrices
    for j=1:size
        %fill cols of bound product matrices
        fprintf('\n (%d,%d): ',i,j);
        if ((checku(i,j) || checkl(i,j))&& checkb(i,j))
            fprintf(' Bound error detected. Use PRECISE result. ');
        else if checku(i,j) || checkl(i,j)
            fprintf(' Error detected. Use REDUCED-PRECISION result. ');
        else fprintf(' No error detected. Use PRECISE result. ');
        end
    end
end
end
end
end

```

B.2 CODE TESTING UPPER/LOWER BOUNDS ON FFT OPERATION

```

%% MANUAL FFT WITH BOUNDS
clear all; close all; clc; format compact; format short;
% NOTE: must COMPUTE REAL and IMAG PARTS SEPARATELY (for bounds!)
% RESERVE 'i' AS IMAGINARY SQRT(-1).
scale = 100;
N=16; % number of input points
n=N; % number of output points (keep same as input for simplicity)
w_inc = exp(-2*pi*sqrt(-1)/N) % define twiddle factor (increment) wn
% Generate wn lookup table - FULL PRECISION
for k=1:n
    for j=1:N
        w(k,j) = w_inc^((j-1)*(k-1));
    end
end
wr = real(w);
wi = imag(w);
%disp(w)
% NOTE: w IS DIAGONAL -- INEFFICIENT TO STORE WHOLE MATRIX... but don't
% worry about that here!!

% Generate w bound matrices (not sure if this is necessary...)
wrl = floor(scale*wr)/scale % generate lower bounds
wil = floor(scale*wi)/scale

wru = ceil(scale*wr); % generate upper bounds
wiu = ceil(scale*wi);
for k=1:N
    for j = 1:N
        if (wru(k,j) == wr(k,j)) % alter for integers
            wru(k,j) = wru(k,j) + 1;
        end
        if (wiu(k,j) == wi(k,j))
            wiu(k,j) = wiu(k,j) + 1;
        end
    end
end
wru = wru/scale
wiu = wiu/scale

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generate input vector and placeholder output vector
xr= 2*rand(N,1) - 1; % generate input points (vector)

```

```

XXr = zeros(n,1);    % initialize output (points) vector, reals
XXi = zeros(n,1);    % initialize output (points) vector, imags

% Generate input bound vectors (bounds on input values)
xrl = floor(scale*xr)/scale;    % generate real input lower bounds
xru = ceil(scale*xr);    % generate real input upper bounds
for k=1:N
    if (xru(k) == xr(k))    % increment upper bounds of int reals
        xru(k) = xru(k) + 1;
    end
end
xru = xru/scale;

xi = zeros(N,1);    % placeholders for imag parts
% xil = xi;    % (xi NOT NEEDED when samples x are all REAL)
% xiu = xi;

xr_in = [xrl xr xru]    % check that xl < x < xu (SIGNED)
% xi_in = [xil xi xiu]    % PLACEHOLDER VECTORS ONLY for imag

XXrl = zeros(n,1);    % initialize output lower bounds (real)
XXil = zeros(n,1);    % initialize output lower bounds (imag)
XXru = zeros(n,1);    % initialize output upper bounds (real)
XXiu = zeros(n,1);    % initialize output upper bounds (imag)

% Perform transform on exact values xr + xi*i (although all xi = 0 here)
% (Equation from MATLAB Help on function "fft(x)")
for k=1:n
    for j=1:N
        XXr(k) = XXr(k) + (xr(j)*wr(k,j) + xi(j)*wi(k,j)); % REAL parts
        XXi(k) = XXi(k) + (xr(j)*wi(k,j) + xi(j)*wi(k,j)); % IMAG parts
    end
end

XXX = fft(xr);    % ACTUAL FFT for check/compare...
compare_ReIm_Actual = [XXr XXi XXX]
% (columns 1 and 2 should equal real/imag parts of col 3)

% NOW APPLY TO BOUNDS

% Create logical sign tests on x and w
% NOTE: would also need x_imag IF the input set included complex samples.
xrnn = logical(sign(xr)+1) % element is TRUE if corresp # is NONNEGATIVE
wrnn = logical(sign(wr)+1)
winn = logical(sign(wi)+1)

% Write expressions for: XXrl XXru XXil XXiu
% NOTE: if the input included complex samples, would need to use:
%   XXru(k) = XXru(k) + (xr(j)*wr(k,j) + xi(j)*wi(k,j)); % REAL parts
%   XXiu(k) = XXiu(k) + (xr(j)*wi(k,j) + xi(j)*wi(k,j)); % IMAG parts
%   XXrl(k) = XXrl(k) + (xr(j)*wr(k,j) + xi(j)*wi(k,j)); % REAL parts
%   XXil(k) = XXil(k) + (xr(j)*wi(k,j) + xi(j)*wi(k,j)); % IMAG parts
% BUT as it stands, xi(j) = 0 for all j so we don't need the final term!!
for k=1:n
    for j=1:N
        if (xrnn(j) && wrnn(k,j) && winn(k,j))    % OPERANDS ARE DIFF BOUNDS
            XXru(k) = XXru(k) + (xru(j)*wru(k,j));    % BASED ON SIGNS OF INPUT
            XXiu(k) = XXiu(k) + (xru(j)*wiu(k,j));    % (Eq 12 again)
            XXrl(k) = XXrl(k) + (xrl(j)*wrl(k,j));
            XXil(k) = XXil(k) + (xrl(j)*wil(k,j));
        else if (xrnn(j) && wrnn(k,j) && ~winn(k,j))
            XXru(k) = XXru(k) + (xru(j)*wru(k,j));
            XXiu(k) = XXiu(k) + (xrl(j)*wiu(k,j));
            XXrl(k) = XXrl(k) + (xrl(j)*wrl(k,j));
            XXil(k) = XXil(k) + (xru(j)*wil(k,j));
        else if (xrnn(j) && ~wrnn(k,j) && winn(k,j))
            XXru(k) = XXru(k) + (xrl(j)*wru(k,j));
            XXiu(k) = XXiu(k) + (xru(j)*wiu(k,j));
            XXrl(k) = XXrl(k) + (xru(j)*wrl(k,j));
        end
    end
end

```

```

XXil(k) = XXil(k) + (xrl(j)*wil(k,j));
    else if (xrnn(j) && ~wrnn(k,j) && ~winn(k,j))
XXru(k) = XXru(k) + (xrl(j)*wru(k,j));
XXiu(k) = XXiu(k) + (xrl(j)*wui(k,j));
XXrl(k) = XXrl(k) + (xru(j)*wrl(k,j));
XXil(k) = XXil(k) + (xru(j)*wil(k,j));
    else if (~xrnn(j) && wrnn(k,j) && winn(k,j))
XXru(k) = XXru(k) + (xru(j)*wrl(k,j));
XXiu(k) = XXiu(k) + (xru(j)*wil(k,j));
XXrl(k) = XXrl(k) + (xrl(j)*wru(k,j));
XXil(k) = XXil(k) + (xrl(j)*wui(k,j));
    else if (~xrnn(j) && wrnn(k,j) && ~winn(k,j))
XXru(k) = XXru(k) + (xru(j)*wrl(k,j));
XXiu(k) = XXiu(k) + (xrl(j)*wil(k,j));
XXrl(k) = XXrl(k) + (xrl(j)*wru(k,j));
XXil(k) = XXil(k) + (xru(j)*wui(k,j));
    else if (~xrnn(j) && ~wrnn(k,j) && winn(k,j))
XXru(k) = XXru(k) + (xrl(j)*wrl(k,j));
XXiu(k) = XXiu(k) + (xru(j)*wil(k,j));
XXrl(k) = XXrl(k) + (xru(j)*wru(k,j));
XXil(k) = XXil(k) + (xrl(j)*wui(k,j));
    else if (~xrnn(j) && ~wrnn(k,j) && ~winn(k,j))
XXru(k) = XXru(k) + (xrl(j)*wrl(k,j));
XXiu(k) = XXiu(k) + (xrl(j)*wil(k,j));
XXrl(k) = XXrl(k) + (xru(j)*wru(k,j));
XXil(k) = XXil(k) + (xru(j)*wui(k,j));
    end
    end
    end
    end
    end
    end
    end
end

% compare_bounds_Real = [XXrl XXr XXru]; % should be col1<col2<col3
% compare_bounds_Imag = [XXil XXi XXiu]; % (if you want to see the values)

% Conduct checks of precise against bounds and bounds against each other
checkRu = gt(XXr,XXru);
checkRl = gt(XXrl,XXr);
checkRb = gt(XXrl,XXru);
ErrorsReal_UpLoBd = [checkRu checkRl checkRb]
checkIu = gt(XXi,XXiu);
checkIl = gt(XXil,XXi);
checkIb = gt(XXil,XXiu);
ErrorsImag_UpLoBd = [checkIu checkIl checkIb]

% end of FFT execution with bound testing

```

APPENDIX C. SYSTEM SIMULATION CODE

C.1 ADCS ERROR INJECTION MODEL SCRIPT FOR TESTING

```
%% Experiment with BRMS PD Control Model - Error Injection
clear angle_hist ctrl_NOerr ctrl_witherr snr;
close all; clc; format compact; format long;

% Set simulation parameters
time = 50;
simstep = 0.025; %fixed-step size (called in Config Params - Solver)

% Set error injection parameters
errtime = 5;
errtype = 2; % 0 for NONE, 1 for spike (transient), 2 for step (config/persist)
snr = [ 27 inf inf ]; % 3 dB for no fault-tol; 27 for r=8, 51 for r=16
pin = .025;
% Run simulation
[t,x] = sim('main_err1',time);

% Plot Data (don't forget to check figure TITLES for trials!!)
close;clc
% Plot both attitude and control in one plot - SUBPLOTS
figure(13);
subplot(2,1,1),plot(t,angle_hist(:,1),'-b',t,angle_hist(:,2),'--k',...
    t,angle_hist(:,3),'-.k');
    grid on
    h = findobj(gca,'type','line');
    set(h,'linewidth',2);
    xlabel('Time (s)','fontsize',16);
    ylabel('Angular Position (\phi,\theta,\psi) (\circ)','fontsize',16);
    title('RPR (\itrm = 8) Persistent Error in T_x - Angular Position',...
        'fontsize',24);
% title('Unbounded Persistent Error in T_x - Angular Position (500 s)',...
% 'fontsize',24);
    lh1 = legend('Roll (\phi)','Pitch (\theta)','Yaw (\psi)');
    %ylim([0 1.3]);
subplot(2,1,2),plot(t,ctrl_NOerr(:,1),'-b',t,ctrl_NOerr(:,2),'--k',...
    t,ctrl_NOerr(:,3),'-.k');
    grid on
    h = findobj(gca,'type','line');
    set(h,'linewidth',2);
    xlabel('Time (s)','fontsize',16);
    ylabel('Control Command (T_x,T_y,T_z) (N-m)','fontsize',16);
    title('RPR (\itrm = 8) Persistent Error in T_x - Commanded Control',...
        'fontsize',24);
% title('Unbounded Persistent Error in T_x - Commanded Control (500 s)',...
% 'fontsize',24);
    lh2 = legend('T_x','T_y','T_z');
    %ylim([0 1.3]);

% Plot both attitude and control in one plot - OVERLAID AXES
figure(14);
title('Unbounded Persistent Error in T_x - Angular Position and Commanded
Control','fontsize',24);
%title('RPR (\itrm = 8) Persistent Error in T_x - Angular Position and Commanded
Control','fontsize',24);
hl1 = line(t,angle_hist(:,1),'Color','b','linestyle','--');
hl2 = line(t,angle_hist(:,2),'Color','b','linestyle','--');
hl3 = line(t,angle_hist(:,3),'Color','b','linestyle','-.');
grid on
ax1 = gca;
set(ax1,'YColor','b');
```

```

xlabel('Time (s)','fontsize',16);
ylabel('Angular Position (\phi,\theta,\psi) (\circ)','fontsize',16);
l1 = legend('Roll (\phi)','Pitch (\theta)','Yaw (\psi)','location','n');
ax2 = axes('Position',get(ax1,'Position'),...
    'YAxisLocation','right',...
    'Color','none',...
    'XColor','k','YColor','k');
%ylim([-0.06 .08]);
ylabel('Control Command (T_x,T_y,T_z) (N-m)','fontsize',16);
hl4 = line(t,ctrl_NOerr(:,1),'Color','k','linestyle','--','Parent',ax2);
hl5 = line(t,ctrl_NOerr(:,2),'Color','k','linestyle','-','Parent',ax2);
hl6 = line(t,ctrl_NOerr(:,3),'Color','k','linestyle','-','Parent',ax2);
l2 = legend('T_x','T_y','T_z','location','s');
set(l2,'Color','white');

ha1 = findobj(gcf,'type','axes');
ha2 = findobj(ha1,'type','line');
set(ha2,'linewidth',2);

% end of ADCS performance evaluation code

```

C.2 FFT ERROR INJECTION MODEL SCRIPT FOR TESTING

```

%% FFT: FLOATING-POINT WITH AWGN INJECTION
clear all; close all; clc; format compact; format long

% Set FFT parameters
N = 16; % for now
sets = 1000; % sim time, also num FFTs (batches?) calculated
% Set error injection parameters
errsig = 7;
%errtrial = 1;
%errtype = 0; % 0 for spike (data/trans), 1 for step (config/persist)
errvec = zeros(N,1); % initialize error selection vector
errvec(errsig) = 1; % specify which signal has error

% Set noise parameters
nseed = floor(1000*rand(N,1));
snr = 0; % 0 for no fault-tolerance, **TBD** for RPR fault-tol
vccin = 2.5; %volts
curin = 10; % mA
pin = vccin * 0.001*curin
% NOTE: Vccin nominally 2.5 V @ < 100 mA for Xilinx 1
% --> set power in "signal in" to P = I*V for noise!

% Run simulation
sim('newFFT_16pt_snr1',sets);
% note: EACH TRIAL is a COLUMN of data in output arrays.

% Check (relative) error in input and output
in_err = abs((fft_in_witherr - fft_in_NOerr)./ fft_in_NOerr)';
out_err = abs((fft_out_witherr - fft_out_NOerr)./fft_out_NOerr)';

in_erravg = mean(in_err,2)
out_erravg = mean(out_err,2)

% end of FFT error simulation

```

APPENDIX D. BRMSS CODE FOR DYNAMICS MODEL AND CONTROLLER

D.1 EFFECT OF APPENDAGES ON SYSTEM MOI (J)

```

clear all; close all; format compact; clc;

% Use experimental rigid-body MOI from paper (J. J. Kim, NPS, 2008)
Jb = [130.34 3.01 10.52; 3.02 174.64 -0.40; 10.52 -0.40 181.23];
%mrvec = [0.00196; 0.00481; 0.19695]; %center of gravity vector, m*r

% Calculate moment of inertia of each appendage
m = 22.6796; %kg, mass of each cyl. "end" (2 per bar/appendage, 50lb)
%NOTE: each cyl end mass is modeled as a POINT MASS @ R
r = 0.15; %m, radius of end cylinder (CHECK THIS??)
R = 0.727; %m, radius from center of bar to CENTER of end (point) mass
%NOTE: applies to all 3 appendages (any orient'n)
%Im1 = m*R^2; %kg-m^2, MOI of one end mass (PARALLEL AXIS THM)
%Im2 = Im1; %kg-m^2, MOI of other end mass
M = 11.3398; %kg, mass of bar (1 per appendage, 25lb)
a = 0.1; %m, width of bar (in plane of rotation, any orient'n)
b = 1.27; %m, length of bar (connecting two end masses, < 2R)
c = 0.02; %m, EST thickness of bar (thin dim)

Ma = M + 2*m; %kg, total mass of one appendage (any 1,2,3)

% Principle moments of inertia of appendages in their local body axes
Ia1xx = (1/12)*M*(a^2+b^2)+2*m*R^2;
Ia1yy = (1/12)*M*(a^2+c^2)+2*0.5*m*r^2;
Ia1zz = (1/12)*M*(b^2+c^2)+2*m*R^2;

Ja1 = diag([Ia1xx Ia1yy Ia1zz]); Ja1pr = diag(Ja1);
Ja2 = Ja1; Ja2pr = Ja1pr;
Ja3 = diag([Ia1yy Ia1zz Ia1xx]); Ja3pr = diag(Ja3);

% NOTE: Now calculate torsional SPRING CONSTANT for appendages
% fn = sqrt(kspring/MOIapp) for rotational bodies
fn = 0.1; %Hz, tuned natural frequency of each appendage (from J.J. Kim)
k = fn^2*Ia1xx; %N-m/rad, torsional spring constant for each app.
% Spring constant is the same for all; x-axis for 1/2 & z-axis for 3.

% Coordinate transformations using rotation matrices from body to apps
AL = -pi/4; %rad, angle between body X and appendage 1
BE = -3*pi/4; %rad, angle between body X and appendage 2
%(NOTE: appendage 3 is ALIGNED with body Y axis, so no angle is needed)
g1 = 0; %rad, angle position of appendage 1 w.r.t. spring1 X axis
g2 = 0; %rad, angle position of appendage 2 w.r.t. spring2 X axis
g3 = 0; %rad, angle position of appendage 3 w.r.t. spring3 X axis
% Rotation matrices for Appendage 1
bCs1 = [ cos(AL) sin(AL) 0;
        -sin(AL) cos(AL) 0;
        0 0 1 ];
s1Ca1 = [1 0 0;
         0 cos(g1) sin(g1);
         0 -sin(g1) cos(g1)];
bCa1 = bCs1*s1Ca1;
% Rotation matrices for Appendage 2
bCs2 = [ cos(BE) sin(BE) 0;
        -sin(BE) cos(BE) 0;
        0 0 1 ];
s2Ca2 = [1 0 0;
         0 cos(g2) sin(g2);
         0 -sin(g2) cos(g2)];

```

```

bCa2 = bCs2*s2Ca2;
% Rotation matrices for Appendage 3
bCa3 = [ cos(g3) sin(g3) 0;
        -sin(g3) cos(g3) 0;
         0      0      1 ];
% (ONLY ONE ROTATION NEEDED for appendage 3)

% Expressing each appendage's MOI in CENTRAL BODY FRAME
bJ_a1 = bCa1*Ja1*bCa1'; %coord txfmation (Likins eq 8.35a, p.427)
bJ_a2 = bCa2*Ja2*bCa2'; %coord txfmation (Likins eq 8.35a, p.427)
bJ_a3 = bCa3*Ja3*bCa3'; %coord txfmation (Likins eq 8.35a, p.427)

% Vectors from central body origin 0 to each appendage (spring)
Rb = 0.762; %m, radius of central body
Ra1 = 0.28575; %m, extension of appendage 1 arm beyond rigid body radius
Ra2 = Ra1; %m, extension of appendage 2 arm beyond rigid body radius
Ra3 = 0.36195; %m, extension of appendage 3 arm beyond rigid body radius
S1 = Rb + Ra1;
S2 = Rb + Ra2;
S3 = Rb + Ra3;
sa1 = [S1*cos(AL) S1*sin(AL) 0]';
sa2 = [S2*cos(BE) S2*sin(BE) 0]';
sa3 = [ 0 S3 0]';

% Contribution of each appendage to MOI of total BRMSS system
% (build skew-symmetric matrices)
st1 = [ 0 -sa1(3) sa1(2);
        sa1(3) 0 -sa1(1);
        -sa1(2) sa1(1) 0 ];
st2 = [ 0 -sa2(3) sa2(2);
        sa2(3) 0 -sa2(1);
        -sa2(2) sa2(1) 0 ];
st3 = [ 0 -sa3(3) sa3(2);
        sa3(3) 0 -sa3(1);
        -sa3(2) sa3(1) 0 ];

bJba1 = bJ_a1 - Ma*st1*st1; %kg-m^2 (all three bJba(n))
bJba2 = bJ_a2 - Ma*st2*st2; % YES it's '-!!' -->EX 8.4.9 p.432-3, ref
bJba3 = bJ_a3 - Ma*st3*st3; % fig 8.5 p 424 & eq 8.28!!
% (TOTAL MOI is larger than bJ_a(n), *despite* '-!. :) )
% ** REMEMBER: if you do NOT assume gamma = 0, these are all time-varying!

% TOTAL BRMSS MOI:
Jtot = Jb + bJba1 + bJba2 + bJba3; %time-invariant *IF* GAMMA(n) = 0.
Jpr = diag(Jtot);
Jd = diag(Jpr); % drop cross-terms to get just PRINCIPLE AXES on DIAG.
%JdInv = inv(Jd); % inverse of J... for pre-mult for EOM?
Jgs = Ia1xx*eye(3);
Jeom = [ Jd zeros(3);
         zeros(3) Jgs ];
Jsys = [ Jeom zeros(6);
         zeros(6) eye(6) ];
Jinv = inv(Jsys); % inversion of MOI matrix for EOM calcs

```

D.2 STATE-SPACE SYSTEM DESCRIPTION/CONTROL USING (J)

% NOTE: See ch V for EOM and steps to simplify them in order to get A,B!!

```

% State-Space Model of Linear Dynamics System WITH NOISE:
% xdot(t) = A(t)*x(t) + B(t)u(t) + v(t)
% y(t) = C(t)*x(t) + w(t)
% where x is the state vector, y are the observable states, u is the
% control, v is the model uncertainty (noise), w is the measurement
% uncertainty (noise), A is the system/plant, B the control distribution,
% and C is the selection matrix that pulls observable states from x.
n=12; q=3; p=3; % dim of x, y, u

```

```

%      qx      qy      qz      g1      g2      g3 ]
Acoeffs = -k*[ 3      0      0      -cos(AL)  -cos(BE)  0 ;
               0      3      0      sin(AL)   sin(BE)  0 ;
               0      0      3      0          0       -1 ;
               -cos(AL) sin(AL) 0      1          0       0 ;
               -cos(BE) sin(BE) 0      0          1       0 ;
               0      0      -1      0          0       1 ];
% '-k' because they are going on the other side of the equation now.
zeta = 0.0001;
wn = 2*pi*fn;
Adamp= -2*zeta*wn*[ 0 0 0 0 0 0 ;
                   0 0 0 0 0 0 ;
                   0 0 0 0 0 0 ;
                   0 0 0 1 0 0 ;
                   0 0 0 0 1 0 ;
                   0 0 0 0 0 1 ];

Atemp = [ Adamp Acoeffs; eye(n/2) zeros(n/2) ];
A = Jinv*Atemp;
Btemp = [ eye(p); zeros(n-p,p) ];
B = Jinv*Btemp;
C = [eye(q) zeros(q,n-q)];
D = zeros(p);

% Asub = [ -3*k/Jpr(1) 0 0      k*cos(AL)/Jpr(1)  k*cos(BE)/Jpr(1) 0 ;
%           0 -3*k/Jpr(2) 0      -k*sin(AL)/Jpr(2) -k*sin(BE)/Jpr(2) 0 ;
%           0 0 -3*k/Jpr(3)      0          0      k/Jpr(3) ;
%           k*cos(AL)/Ja1pr(1) -k*sin(AL)/Ja1pr(1) 0      -k/Ja1pr(1) 0 0 ;
%           k*cos(BE)/Ja2pr(1) -k*sin(BE)/Ja2pr(1) 0      0      -k/Ja2pr(1) 0 ;
%           0          0          0      k/Ja3pr(3) 0 0 -k/Ja3pr(3) ];
% zeta = 0.0001;
% wn = 2*pi*fn;
% Adam = 2*zeta*wn*[zeros(3) zeros(3);
%                  zeros(3) eye(3) ];
% %from Ch.V, Eq 32 (# for now)
% Bsub = [1/Jpr(1) 0 0; 0 1/Jpr(2) 0; 0 0 1/Jpr(3) ];
% A = [Adam Asub; eye(n/2) zeros(n/2)];
% B = [Bsub; zeros(n-p,p)];
% C = [eye(q) zeros(q,n-q)];
% D = zeros(p);
% Ts = [];

%brmss = ss(A,B,C,D,Ts); % ESTABLISHES THE STATE-SPACE SYSTEM 'brmss'
% NOTE: 'Ts' dictates DISCRETE TIME MODEL, step unspecified. CHECK THIS!!
brmss = ss(A,B,C,D) % CONTINUOUS TIME state-space system 'brmss'
% Now refer to brmss.a, b, c, d in the plant/system dynamics subsystem of model

% Check controllability of the system
% (if rank of controllability matrix = rank of system, it IS CONTROLLABLE)
wantzero = rank(A)-rank(ctrb(brmss))

% Generate noise and weighting matrices
QXU = eye(15); % PLACEHOLDER
QWV = eye(15); % PLACEHOLDER

LQGreg = lqg(brmss,QXU,QWV)
% Now refer to LQGreg.a, b, c, d in the controller in the Simulink model

% end of file as of November 2008

```

D.3 SYSTEM AND CONTROLLER A,B,C,D MATRICES (OUTPUT)

```

a =
      x1      x2      x3      x4      x5      x6
x7      x8

```

x1	0	0	0	0	0	0	-
0.004451	0						
x2	0	0	0	0	0	0	0
0	-0.003383						
x3	0	0	0	0	0	0	0
0	0						
x4	0	0	0	0	-4.927e-006	0	0
0.007071	0.007071						
x5	0	0	0	0	-4.927e-006	0	-
0.007071	0.007071						
x6	0	0	0	0	0	0	-4.927e-006
0	0						
x7	1	0	0	0	0	0	0
0	0						
x8	0	1	0	0	0	0	0
0	0						
x9	0	0	1	0	0	0	0
0	0						
x10	0	0	0	1	0	0	0
0	0						
x11	0	0	0	0	1	0	0
0	0						
x12	0	0	0	0	0	0	1
0	0						

	x9	x10	x11	x12
x1	0	0.001049	-0.001049	0
x2	0	0.0007975	0.0007975	0
x3	-0.002805	0	0	0.0009351
x4	0	-0.01	0	0
x5	0	0	-0.01	0
x6	0.01	0	0	-0.01
x7	0	0	0	0
x8	0	0	0	0
x9	0	0	0	0
x10	0	0	0	0
x11	0	0	0	0
x12	0	0	0	0

b =

	u1	u2	u3
x1	0.005817	0	0
x2	0	0.004422	0
x3	0	0	0.003666
x4	0	0	0
x5	0	0	0
x6	0	0	0
x7	0	0	0
x8	0	0	0
x9	0	0	0
x10	0	0	0
x11	0	0	0
x12	0	0	0

c =

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12
y1	1	0	0	0	0	0	0	0	0	0	0	0
y2	0	1	0	0	0	0	0	0	0	0	0	0
y3	0	0	1	0	0	0	0	0	0	0	0	0

d =

	u1	u2	u3
y1	0	0	0
y2	0	0	0
y3	0	0	0

Continuous-time model.

wantzero =

0

```

a =
      x1_e      x2_e      x3_e      x4_e      x5_e      x6_e
x7_e
  x1_e      -1.155   -7.076e-017   -3.34e-017   -0.01741    0.01741   -3.253e-017
-0.01425
  x2_e      -6.557e-017      -1.136   -3.241e-016   -0.007316   -0.007316   -6.444e-018
2.331e-018
  x3_e      2.838e-018   -2.807e-016      -1.123   -6.857e-017   -9.205e-017      -0.003563
-1.27e-019
  x4_e      -0.01345    0.005075   -8.261e-015   -4.927e-006      0      0
0.007071
  x5_e      0.01345    0.005075   -7.713e-015      0   -4.927e-006      0
-0.007071
  x6_e      1.758e-015    9.306e-015    0.02145      0      0   -4.927e-006
0
  x7_e      1.414    4.559e-015    2.977e-015      0      0      0
0
  x8_e      3.491e-016      1.414   -1.328e-014      0      0      0
0
  x9_e      6.673e-015   -2.76e-014      1.414      0      0      0
0
  x10_e      -6.173      -6.173   -2.93e-014      1      0      0
0
  x11_e      6.173      -6.173   -6.688e-014      0      1      0
0
  x12_e      1.107e-014   -4.351e-014      -8.727      0      0      1
0

```

```

      x8_e      x9_e      x10_e      x11_e      x12_e
x1_e      7.251e-018   -6.581e-018    0.003893   -0.003893    2.55e-018
x2_e      -0.01111   -1.84e-017    0.003158    0.003158    3.309e-018
x3_e      -1.815e-017   -0.009296    8.454e-018    7.009e-018    0.003784
x4_e      0.007071      0      -0.01      0      0
x5_e      0.007071      0      0      -0.01      0
x6_e      0      0.01      0      0      -0.01
x7_e      0      0      0      0      0
x8_e      0      0      0      0      0
x9_e      0      0      0      0      0
x10_e      0      0      0      0      0
x11_e      0      0      0      0      0
x12_e      0      0      0      0      0

```

```

b =
      y1      y2      y3
x1_e      1.015    5.848e-017   -2.672e-017
x2_e      5.848e-017    1.011    1.851e-016
x3_e      -2.672e-017    1.851e-016    1.009
x4_e      0.01345   -0.005075    8.261e-015
x5_e      -0.01345   -0.005075    7.713e-015
x6_e      -1.758e-015   -9.306e-015    -0.02145
x7_e      -0.4142   -4.559e-015   -2.977e-015
x8_e      -3.491e-016   -0.4142    1.328e-014
x9_e      -6.673e-015    2.76e-014   -0.4142
x10_e      6.173      6.173    2.93e-014
x11_e      -6.173      6.173    6.688e-014
x12_e      -1.107e-014    4.351e-014    8.727

```

```

c =
      x1_e      x2_e      x3_e      x4_e      x5_e      x6_e
x7_e      x8_e
  u1      -24.09   -2.112e-015   -1.034e-014   -2.993    2.993   -5.593e-015
-1.685    1.247e-015
  u2      -1.605e-015      -28.14   -3.143e-014   -1.655   -1.655   -1.457e-015
5.273e-016      -1.748
  u3      -6.515e-015   -2.606e-014      -31.09   -1.87e-014   -2.511e-014      -0.9718   -
3.465e-017   -4.951e-015

      x9_e      x10_e      x11_e      x12_e

```

u1	-1.131e-015	0.4888	-0.4888	4.384e-016
u2	-4.162e-015	0.5338	0.5338	7.485e-016
u3	-1.77	2.306e-015	1.912e-015	0.7771

d =

	y1	y2	y3
u1	0	0	0
u2	0	0	0
u3	0	0	0

Input groups:

Name	Channels
Measurement	1,2,3

Output groups:

Name	Channels
Controls	1,2,3

Continuous-time model.

LIST OF REFERENCES

- [1] "Digital Hardware," class notes for EC4530, Department of Electrical and Computer Engineering, Naval Postgraduate School, Summer 2008.
- [2] "Space Radiation Effects," class notes for SS3035, Space Systems Academic Group, Naval Postgraduate School, Spring 2007.
- [3] J. Snodgrass, "Low-Power Fault Tolerance For Spacecraft FPGA-Based Numerical Computing," Ph.D. dissertation, Naval Postgraduate School, Monterey, CA, 2006.
- [4] A. C. Tribbel, D. J. Gorney, J.B. Blake, H.C. Koons, M. Schulz, A.L. Vampola, R. L. Walterschied, J. R. Wertz, "The Space Environment," in *Space Mission Analysis and Design*, 3rd Edition, J. R. Wertz and W. J. Larson, Ed. El Segundo: Microcosm Press, 1999, pp. 203-221.
- [5] P. Nordin and M. K. Kong, "Hardness and Survivability Requirements," in *Space Mission Analysis and Design*, 3rd Edition, J. R. Wertz and W. J. Larson, Ed. El Segundo: Microcosm Press, 1999, pp. 221-238.
- [6] R. W. Hamming, "Error Detecting and Correcting Codes," in *Bell System Technical Journal*, 1950, pp. 147-160.
- [7] S. B. Wicker, *Error Control Coding for Digital Communication and Storage*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
- [8] "Triple Modular Redundancy," class notes for EC4810, Department of Electrical and Computer Engineering, Naval Postgraduate School, Summer 2007.
- [9] J. Coudeyras, "Radiation Testing of the Configurable Fault Tolerant Processor (CFTP) For Space-based Applications," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2005.
- [10] D. Ebert, "Design and Development of a Configurable Fault-Tolerant Processor (CFTP) for Space Applications," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2003.
- [11] P. Majewicz, "Implementation of a Configurable Fault-Tolerant Processor (CFTP) Using Internal Triple Modular Redundancy (TMR)," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2005.

- [12] G. Caldwell, "Implementation of Configurable Fault Tolerant Processor (CFTP) Experiments," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2006.
- [13] B. N. Agrawal, Design of Geosynchronous Spacecraft. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- [14] J. S. Eterno, "Attitude Determination and Control," in Space Mission Analysis and Design, 3rd Edition, J. R. Wertz and W. J. Larson, Ed. El Segundo: Microcosm Press, 1999, pp. 354-380.
- [15] J. P.B. Vreeburg, "Spacecraft Manuevers and Sloss Control," IEEE Control Systems Magazine, pp. 12-16, June 2005.
- [16] T. E. Suttles and R. E. Beverly, "Model for solar torque effects on DSCS II," Journal of Astronautical Sciences, vol. 24, pp. 165-184, 1976.
- [17] C. B. Spence, Jr., "Environmental Torques," in Spacecraft Attitude Determination and Control, J. R. Wertz, Ed. Springer, 1978, pp. 566-583.
- [18] I. M. Ross, Control and Optimization: An Introduction to Principles and Applications, Electronic Edition. Monterey, CA: Naval Postgraduate School, 2005.
- [19] "Design of PID Control Law for Rotation About a Single Axis for a Rigid Body Spacecraft," class notes for AE3818, Department of Mechanical and Astronautical Engineering, Naval Postgraduate School, Fall 2007.
- [20] J. H. Reed, Software Radio: A Modern Approach to Radio Engineering. Upper Saddle River, NJ: Prentice-Hall, 2002.
- [21] R. Cristi, Modern Digital Signal Processing. Pacific Grove: Brooks/Cole, 2004.
- [22] A. V. Oppenheim , R. W. Schaffer, *Digital Signal Processing*, Prentice-Hall, 1975.
- [23] D. A. Wright, "Field-Programmable Gate Array-Based Software Defined Radio," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2008.
- [24] Xilinx, Inc., "Fast Fourier Transform v4.1," Xilinx® LogiCore Product Specification DS260, April 2, 2007.
- [25] Xilinx, Inc., "Virtex™ 2.5 V Field Programmable Gate Arrays Architectural Description," Xilinx Product Specification DS003-2 (v2.8.1), December 9, 2002.

- [26] J. H. Wilkinson, Rounding Errors in Algebraic Processes. Englewood Cliffs, New Jersey: Prentice-Hall, © 1963 British Crown.
- [27] R. K. Richards, Arithmetic Operations in Digital Computers. Princeton, New Jersey: D. Van Nostrand Company, Inc., 1955.
- [28] B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs. New York: Oxford University Press, 2000.
- [29] “Pipeline Division,” class notes for EC4830, Department of Electrical and Computer Engineering, Naval Postgraduate School, Spring 2008.
- [30] Xilinx, Inc., “Virtex™ 2.5 V Field Programmable Gate Arrays Electrical Characteristics,” Xilinx Product Specification DS003-3 (v3.2), September 10, 2002.
- [31] J. J. Kim (private communication), 2008.
- [32] J. J. Kim and B.N. Agrawal, “Automatic Mass Balancing of Air-Bearing Based Three-Axis Rotational Spacecraft Simulator,” Naval Postgraduate School, Monterey, CA, October 2008.
- [33] P. W. Likins, Elements of Engineering Mechanics. New York: Mc Graw-Hill, 1973.
- [34] J. L. Junkins, Y. Kim, Introduction to Dynamics and Control of Flexible Structures. Washington, D.C.: American Institute of Aeronautics and Astronautics, Inc., 1993.
- [35] R. F. Stengel, Optimal Control and Estimation. Mineola, New York: Dover, 1994.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Chairman, MAE Department, Millsaps
Naval Postgraduate School
Monterey, California
4. Chairman, ECE Department, Knorr
Naval Postgraduate School
Monterey, California
5. Professor Brij Agrawal
Naval Postgraduate School
Monterey, California
6. Professor Herschel H. Loomis, Jr.
Naval Postgraduate School
Monterey, California
7. Professor Alan A. Ross, Jr.
Naval Postgraduate School
Monterey, California
8. Ms. Donna Miller
Naval Postgraduate School
Monterey, California
9. AFIT/CIP
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio