



**NAVAL
POSTGRADUATE
SCHOOL**

MONTEREY, CALIFORNIA

THESIS

**IMPLEMENTATION OF THE FAST FOURIER
TRANSFORM ONBOARD CFTP-7 SPACE EXPERIMENT**

by

Alan Walker

June 2018

Thesis Advisor:

Herschel H. Loomis

Co-Advisor:

James H. Newman

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 June 2018	3. REPORT TYPE AND DATES COVERED Master's thesis	
4. TITLE AND SUBTITLE IMPLEMENTATION OF THE FAST FOURIER TRANSFORM ONBOARD CFTP-7 SPACE EXPERIMENT			5. FUNDING NUMBERS	
6. AUTHOR(S) Alan Walker				
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 A	
13. ABSTRACT (maximum 200 words) A satellite to be used as a testbed for experiments such as the Configurable Fault Tolerant Processor (CFTP) was designed at the Naval Postgraduate School. This processor consists of a Field Programmable Gate Array (FPGA), which may be reprogrammed by receiving a signal from a source external to the satellite. Experimentation of a high-speed pipelined and fault tolerant Fast Fourier Transform (FFT) was conducted for use within the CFTP. In this thesis, we detail the development and testing of a high-speed pipelined FFT in which fault tolerance can be applied at a later opportunity. Xilinx Vivado ISE® was utilized to synthesize behavioral Verilog to program an FPGA. Xilinx Vivado ISE's® simulation suite produced waveforms to demonstrate functionality. Launch of CFTP is planned for FY18 aboard NPSat-1.				
14. SUBJECT TERMS FPGA, NPSat-1, satellite, fault tolerance, FFT, DFT, reprogrammable computers, CFTP, MidStar-1, CFTP-1, CFTP-7, Parseval's theorem			15. NUMBER OF PAGES 117	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**IMPLEMENTATION OF THE FAST FOURIER TRANSFORM ONBOARD
CFTP-7 SPACE EXPERIMENT**

Alan A. Walker III
Lieutenant, United States Navy
BS, U.S. Naval Academy, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
June 2018**

Approved by: Herschel H. Loomis
Advisor

James H. Newman
Co-Advisor

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

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A satellite to be used as a testbed for experiments such as the Configurable Fault Tolerant Processor (CFTP) was designed at the Naval Postgraduate School. This processor consists of a Field Programmable Gate Array (FPGA), which may be reprogrammed by receiving a signal from a source external to the satellite. Experimentation of a high-speed pipelined and fault tolerant Fast Fourier Transform (FFT) was conducted for use within the CFTP. In this thesis, we detail the development and testing of a high-speed pipelined FFT in which fault tolerance can be applied at a later opportunity. Xilinx Vivado ISE® was utilized to synthesize behavioral Verilog to program an FPGA. Xilinx Vivado ISE's® simulation suite produced waveforms to demonstrate functionality. Launch of CFTP is planned for FY18 aboard NPSat-1.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	OBJECTIVES	1
B.	DEFINING THE PROBLEM.....	3
C.	ORGANIZATION	5
D.	ADDITIONAL DOCUMENTATION.....	5
II.	BACKGROUND AND PRIOR WORK.....	7
A.	CFTP HISTORY.....	7
1.	Midshipmen Space Technology Applications Research-1.....	7
2.	Naval Postgraduate School Satellite-1	8
B.	LITERATURE REVIEW	9
1.	Academic Advisors.....	9
2.	Research History	10
III.	DESIGN AND IMPLEMENTATION	13
A.	SYSTEM SPECIFICATIONS.....	13
1.	Two’s Complement	13
2.	Fixed-Point.....	14
3.	Complex Numbers	15
B.	FAST FOURIER TRANSFORM ALGORITHMS	16
1.	Bit Reversal.....	16
2.	The 8-Point DFT	17
C.	HIGH-SPEED PIPELINED FFT ARCHITECTURE	18
1.	Radix-2 Pipeline Butterfly Machine Architecture	19
2.	2-Stage Pipelined Complex Multiplier Architecture	20
IV.	TESTING AND EVALUATION.....	23
A.	TESTING PLAN.....	23
1.	System Test Plan	23
2.	Component Test Plan	24
3.	Component Timing	25
B.	TEST INPUTS.....	27
C.	FFT SYSTEM AND COMPONENT TESTING.....	28
1.	Bit Reversed Ping-Pong Buffer	29
2.	Radix-2 Pipeline Butterfly Machine Sub-Component Testing.....	33

V.	END TO END TESTING / INTEGRATION TESTING	51
A.	ALL TEST VECTOR ANALYSIS.....	51
B.	TEST VECTOR ANALYSIS.....	55
	1. Bit Reversed Ping Pong Buffer	55
	2. First-Stage BFM and Ping Pong Buffer	57
	3. Second-Stage BFM and Ping-Pong Buffer	58
	4. Third-Stage BFM and Ping-Pong Buffer.....	59
C.	TIMING ERROR WHILE INTEGRATING.....	61
VI.	CONCLUSION	63
A.	ACADEMIC VALUE	63
B.	THESIS SUMMARY	63
C.	RECOMMENDATIONS FOR FUTURE WORK.....	63
D.	CLOSING REMARKS.....	64
	APPENDIX A. FILE STRUCTURE	65
	APPENDIX B. SOURCE CODE	67
	APPENDIX C. HARDWARE CONSTRAINTS FILE.....	91
	LIST OF REFERENCES	95
	INITIAL DISTRIBUTION LIST	97

LIST OF FIGURES

Figure 1.	Implementation of Parseval’s Theorem with Two Duplicate FFTs. Source: [2].....	2
Figure 2.	External View of MidStar-1 Showing the Location of CFTP-1. Adapted from [6].....	8
Figure 3.	Expanded View of NPSat-1 That Shows Location of CFTP-7. Adapted from [7].....	9
Figure 4.	Demonstration of Two’s Complement Operation.....	14
Figure 5.	Fixed-Point Signed Binary Representation. Binary Point Place After Bit 0.....	14
Figure 6.	Flow Graph of Simplified Butterfly Machine Computation Requiring Only One Complex Multiplication. Source: [4].	16
Figure 7.	Bit Reversed Demonstration	17
Figure 8.	Flow Graph of 8-point DFT Using the Butterfly Machine Computation. Source: [4].....	17
Figure 9.	Rearrangement of Figure 1 to Allow Each Stage to Have a Constant Geometry Permitting Sequential Data Accessing and Storage. Source: [4].....	18
Figure 10.	Basic Pipelined FFT Structure. $N = 2^l$, Where N Is the FFT Word Size. Adapted from [15].....	18
Figure 11.	Detailed High Speed Pipelined FFT Structure. Adapted from [15].....	19
Figure 12.	Pipeline Radix-2 Butterfly Machine (BFM) Architecture, Level q . Adapted from [15].....	20
Figure 13.	Two-Stage Complex Multiplier	21
Figure 14.	System Level Test Plan for $N = 8$ FFT	24
Figure 15.	Component Level Test Plan for $N = 8$ BFM. Adapted from [15].	25
Figure 16.	Butterfly Machine Timing Diagram Part A. Adapted from [15].	26
Figure 17.	Butterfly Timing Machine Diagram Part B. Adapted from [15].	26

Figure 18.	Injected $X_q(t)$ Inputs in Binary	27
Figure 19.	Bit-Reversed Ping-Pong Buffer Variables and Arrays	29
Figure 20.	Bit-Reversed Ping-Pong Buffer Loading, Code Snippet	30
Figure 21.	Bit-Reversed Ping-Pong Buffer Output $X_q(t)$	31
Figure 22.	Variable <i>Indexbr</i> transposed from Counter[2:0].....	31
Figure 23.	Pong Buffer Loading in Bit Reversed Order, Ping Empty	32
Figure 24.	Startup Delay Between Bit-Reversed Ping-Pong Buffer Output and BFM	32
Figure 25.	Ping-Buffer Loading in Bit Reversed Order, Pong Full	33
Figure 26.	Butterfly Machine Variable Declaration Code Snippet	33
Figure 27.	“Top” and “Bottom” Multiplexer Code Snippet.....	34
Figure 28.	Delay into Both Multiplexer Code Snippet	34
Figure 29.	Internal Multiplexer Code Snippet.....	35
Figure 30.	Top Multiplexer Half Scale	35
Figure 31.	Top Multiplexer Three-Clock-Cycle Delay.....	36
Figure 32.	Top Multiplexer Output, Hexadecimal	36
Figure 33.	Top Multiplexer Output, Fixed-Point Binary	37
Figure 34.	Top Multiplexer Output, Binary	37
Figure 35.	Multiplexer Internal Code Snippet.....	37
Figure 36.	Bottom Multiplexer Half-scale Code Snippet.....	38
Figure 37.	Bottom Multiplexer Output with Multiplier as a Black Box	38
Figure 38.	Test Plan for Multiplier and Sub-Component testing	39
Figure 39.	Multiplier Variables Code Snippet	40
Figure 40.	Code Snippet for <i>twoComp</i> Module Call.....	40
Figure 41.	Code Snippet for <i>twoComp</i> Module Internals	41

Figure 42.	Sign Bit Extraction Code Snippet.....	41
Figure 43.	Module Call for Behavioral Multiplication on Four Unsigned Complex Numbers Code Snippet.....	42
Figure 44.	Unsigned Multiplication Module Code Snippet	42
Figure 45.	Behavioral Multiplication Code Snippet	43
Figure 46.	Output Truncation Code Snippet	43
Figure 47.	Exclusive-Or of Sign-Bit Code Snippet.....	44
Figure 48.	Code Snippet for <i>twoCompRedo</i> Module Call.....	44
Figure 49.	Module <i>twoCompRedo</i> Code Snippet.....	44
Figure 50.	Adder/Subtractor for Complex Multiplication Code Snippet	45
Figure 51.	Complex Multiplication Module Testing.....	45
Figure 52.	Conversion of Multiplier Inputs to Signed Magnitudes	46
Figure 53.	Multiplier Test Points. Fixed Point.....	47
Figure 54.	Adder / Subtractor Variables Code Snippet	47
Figure 55.	Adder / Subtractor Code Snippet	48
Figure 56.	Adder / Subtractor Adds on Even Clock-Cycles.....	48
Figure 57.	First-Stage Ping-Pong Buffer Variables Code Snippet.....	48
Figure 58.	First-Stage Ping Buffer Input Code Snippet	49
Figure 59.	First-Stage Ping Buffer Input Code Snippet (Continued).....	49
Figure 60.	First-Stage Ping-Pong Buffer Simulation	50
Figure 61.	Generic Constant Geometric FFT. Adapted from [4].....	52
Figure 62.	Test Vector-1 {1, 1, 1, 1, 1, 1, 1, 1}. Adapted from [4].....	53
Figure 63.	Test Vector-2 {1, 0, 0, 0, 0, 0, 0, 0}. Adapted from [4].....	53
Figure 64.	Test Vector-3 {0, 0, 0, 0, 0, 0, 0, 0}. Adapted from [4].....	54

Figure 65.	Test Vector-4 {0, 0, 0, 1, 0, 0, 0, 0} Displayed after Bit-Reversed to {0, 0, 0, 0, 0, 0, 1, 0}. Adapted from [4].....	55
Figure 66.	Bit-Reversed Ping-Pong Buffer Initializing.....	56
Figure 67.	Bit-Reversed Ping-Pong Buffer Output	56
Figure 68.	First-Stage BFM Output and First-Stage Ping-Pong Buffer Initialization	57
Figure 69.	First-Stage Ping-Pong Buffer Outputting and Second-Stage BFM Initializing	58
Figure 70.	Second-Stage BFM Outputting and Second-Stage Ping-Pong Buffer Initializing	58
Figure 71.	Second-Stage Ping-Pong Buffer Outputting and Third-Stage BFM Initializing	59
Figure 72.	Third-Stage BFM Outputting and Third-Stage Ping-Pong Initializing	60
Figure 73.	Third-Stage Ping-Pong Buffer With Final Scaled Result {1, 0, 0, 0, 0, 0, 0, 0}.....	60
Figure 74.	Highlighted Timing Variables within the BFM. Adapted from [15].....	61
Figure 75.	Parseval's Theorem Implementation Illustration. Source: [2].....	64

LIST OF TABLES

Table 1.	State Diagram for Parseval's Theorem Implemented with Two Redundant FFTs as Pictured in Figure 1.....	3
Table 2.	Comparison of DFT Calculation Size to FFT Calculation Size to Demonstrate Efficiency of Algorithm	4
Table 3.	18-Bit Fixed-Point Binary Number with a 16-bit Radix Point Examples.....	15
Table 4.	$X_q(t)$ Inputs Injected into FFT	28
Table 5.	Test Vector Input and Expected Output.....	51

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

BFM	Butterfly machine
CFTP	Configurable Fault Tolerant Processor
COTS	commercial off-the-shelf
DFT	discrete Fourier transform
DSP	digital signal processing
FFT	fast Fourier transform
FPGA	field-programmable gate array
HDL	Hardware Description Language
LTE	Long-Term Evolution
MidSTAR-1	Midshipman Space Technology Applications Research-1
MSB	most significant bit
MUX	multiplexer
NPS	Naval Postgraduate School
NPSAT-1	Naval Postgraduate School Satellite-1
OFDM	orthogonal frequency-division multiplexer
RF	radio frequency
RPR	reduced precision redundancy
SDR	software defined radio
SEU	single-event upset
TMR	triple-modular redundancy
VHDL	VHSIC Hardware Description Language
WiMAX	Worldwide Interoperability for Microwave Access

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I thank the dedicated faculty and staff of the Naval Postgraduate School. I give a special thank you to my primary thesis advisor, Professor Herschel Loomis, as he spent hours coaching me through code development and explaining the historical context. I am fortunate that I had the opportunity to take part in a project that has been developing for over 20 years, culminating Dr. Loomis's career with one final launch and a retirement.

Thank you to my parents, Alan and Claudette Walker, and to my resource sponsor. My time here has included some of my most enjoyable experiences while serving within the United States Navy. To all who may come after, continue to believe in yourself, set goals, get mentors, dream, have fun, and always enjoy the moment.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The Naval Postgraduate School (NPS) worked with government sponsors to design and build a small satellite that is scheduled to be launched into space in June 2018. Because of radiation effects, computer systems that are utilized in space must be able to detect and/or correct digital bit errors. The Configurable Fault-Tolerant Processor (CFTP) experiment is a field-programmable gate array (FPGA) implementation of a digital processor to test means for correcting radiation-induced faults in digital processors.

In 2005, Coudeyras [1] tested and demonstrated that radiation in the space environment can cause single-event upsets (SEU) that can have unknown, sometimes unrecoverable, effects on electrical systems. A fast Fourier transform (FFT) that can be used to implement Parseval's theorem in a way that corrects single bit errors [2] is designed and will be implemented on the CFTP as a space experiment on the Naval Postgraduate School Satellite One (NPSAT-1). This implementation will detect and correct SEUs caused by radiation in space.

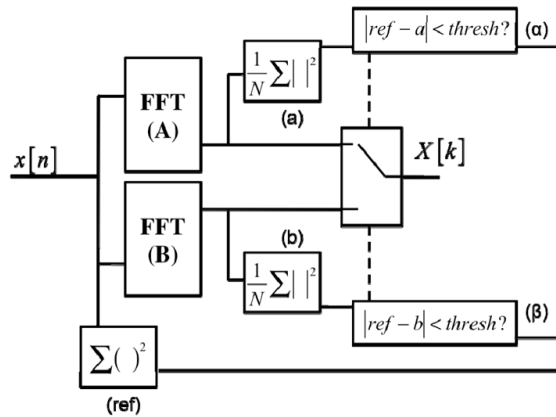
Specifically, in this thesis, we detail the development and testing of a high speed pipelined FFT. Caleb Humberd performed similar research during his time at NPS in 2011 [2]. His research differed in that he implemented a proprietary Xilinx Integrated Synthesis Environment® (ISE) designed FFT that was based on proprietary intellectual property [2]. The FFT code developed in this thesis is open source, easily modified, and documented, which has important pedagogical utility. Sample data was processed through the FFT code to verify functionality.

A. OBJECTIVES

In this thesis, we expand upon the engineering of the fault-tolerant FFT design, implementation, and execution. The FFT can be utilized to compress radio signals to reduce buffer memory and downlink bandwidth. Use of Parseval's theorem enables the FFT to perform error detection and correction as shown in Figure 1, increasing the reliability of the signal.

A sequence $x[n]$ flows into identical and parallel FFT A and FFT B, and the power is calculated in the frequency-domain. In addition, the sequence $x[n]$ has the power calculated from the time-domain. Parseval's theorem states that the FFT's frequency-domain power must equal the time-domain power. The output signal defaults to FFT A unless the FFT A power differs from the reference time-domain power by a threshold deviation. If the threshold deviation is detected, FFT B becomes the output signal. Detailed within Table 1 are the output options that may result based on the results of implementing Parseval's theorem. This will eventually be an experiment on the memory of the CFTP-1 at launch.

2NlogN Redundant FFT



$$\text{if } [|ref - a| < thresh] \text{ then } \{X[k] = FFT_A\} \text{ else } \{X[k] = FFT_B\}$$

Figure 1. Implementation of Parseval's Theorem with Two Duplicate FFTs. Source: [2].

Table 1. State Diagram for Parseval's Theorem Implemented with Two Redundant FFTs as Pictured in Figure 1.

ref	FFT (A)	FFT (B)	(a)	(b)	ref - a	ref - b	Output $X[k]$
Correct	Correct	Correct	Correct	Correct	Zero	Zero	FFT (A)
Correct	Correct	Incorrect	Correct	Incorrect	Zero	Not Zero	FFT (A)
Correct	Incorrect	Correct	Incorrect	Correct	Not Zero	Zero	FFT (B)
Correct	Incorrect	Incorrect	Incorrect	Incorrect	Not Zero	Not Zero	FFT (A)
Incorrect	Correct	Correct	Correct	Correct	Not Zero	Not Zero	FFT (A)
Incorrect	Correct	Incorrect	Correct	Incorrect	Not Zero	Not Zero	FFT (A)
Incorrect	Incorrect	Correct	Incorrect	Correct	Not Zero	Not Zero	FFT (B)
Incorrect	Incorrect	Incorrect	Incorrect	Incorrect	Not Zero	Not Zero	FFT (A)

In this thesis, a behavioral Verilog definition of a pipeline eight-point FFT was developed, simulated, and implemented in a Xilinx Kintex-7 FPGA. This FFT is the basic element for a test of the Parseval's theorem-protected SEU-tolerant FFT.

B. DEFINING THE PROBLEM

The discrete Fourier transform (DFT) is an integral component in digital signal processing (DSP). Understanding signals is important due to their prevalence everywhere.

Social communications between people, physical communications between people and machines, or machine to machine communications are done through signals. Signals present themselves in nature as continuous-time analog quantities. Once digitized and converted by sampling to discrete time they become sequences. A sequence is "a continuous or connected series such as a set of elements ordered so that they can be labeled with positive integers" [3]. Discrete-time samples are roughly comparable to digital signals and are treated the same in our case. The system produces outputs at the same rate at which the continuous signal is sampled, producing a real-time system. The Fourier transform of the sampled signal gives the frequency-domain representation of the time-domain sequence. The DFT is a representation of the Fourier transform that is bounded in time and is defined by the Fourier-transform-pair [4]

$$\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n e^{-j\left(\frac{2\pi}{N}\right)kn} = \sum_{n=0}^{N-1} x_n W_N^{kn} \\
x_n &= \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{j\left(\frac{2\pi}{N}\right)kn} = \sum_{k=0}^{N-1} X_k W_N^{-kn}
\end{aligned}
\tag{1}$$

where $W_N^{kn} = e^{-j(2\pi/N)kn}$ is known as a twiddle factor, X_k is the calculated time to frequency, x_n is the calculate frequency to time, n is the current sample, and k is a constant.

The DFT calculation requires N^2 calculations where N is the sample size. The DFT is one of the most important equations for DSP. It is useful in an orthogonal frequency-division multiplexer (OFDM) demodulators and modulators. Long-Term Evolution (LTE) signal processing, WIFI, and Worldwide Interoperability for Microwave Access (WiMax) signals are examples of signals that the DFT is utilized to analyze. Efficient algorithms have been developed to calculate the DFT. The fast Fourier transform is the signal processing algorithm that is used to reproduce to perform signal processing in this research. The FFT requires $N \log_2 N$ multiply-add operations [4]. To demonstrate the efficiency of the FFT over the DFT, Table 2 is provided.

Table 2. Comparison of DFT Calculation Size to FFT Calculation Size to Demonstrate Efficiency of Algorithm

	N	1000	10^6	10^9	10^{12}
DFT Calculation	N^2	10^6	10^{12}	10^{18}	10^{24}
FFT Calculation	$N \log_2 N$	10^4	20×10^6	30×10^9	40×10^{12}

The FFT is loaded into an FPGA as part of the Configurable Fault Tolerant Processor (CFTP) for operation within the space environment. Since electronics in space

must be able to produce the correct calculations in spite of the risk of SEUs, protection of the FFT is accomplished by implementing Parseval's theorem.

C. ORGANIZATION

Background on Configurable Fault Tolerant Processor research performed at NPS is covered in Chapter II. First, an introduction of Academic Advisors that have facilitated the research is given, following with a review of literature. In Chapter III, we discuss the implementation of the FFT. In addition, the design methodology of the FFT is also included in Chapter III. In Chapter IV, we discuss the test vectors that were utilized to confirm proper operation of the FFT. In Chapter V, we summarize and draw conclusions from the thesis research as well as provide recommendations for future work.

D. ADDITIONAL DOCUMENTATION

The Verilog source code for the FFT is included in appendixes.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND PRIOR WORK

The United States Navy (USN), United States Air Force (USAF), and Department of Defense (DoD) agencies have been designing, purchasing, manufacturing, launching, and operating space systems since 1957. These systems are designed and operated to support the warfighter and strategic decision makers. The acquisition life cycle time and cost for a constellation of proprietary and military specification satellites is greater than DoD leadership likes. In addition, space and the aerospace industry have been commercialized enough to allow the U.S. government to decrease risk and cost by purchasing entire systems or constructing custom systems utilizing commercial-off-the-shelf (COTS) components. The utilization of an FPGA to perform signal processing is becoming routine. The use of a reconfigurable processor, an antenna, a demodulator, an analog-to-digital converter, and a modulator can be utilized as one system named a software-defined radio (SDR). In short, a SDR is a reconfigurable signal processor.

The reprogrammable nature of an SDR makes it convenient for utilization in space. Satellite operators and engineers now have the ability to add or change capability by uploading software while a satellite is on orbit as long as the receiving antenna, analog-to-digital converter, FPGA, and downlink antenna are installed prior to launch. The control module also needs a connection to load the FPGA.

Space is a challenging environment for electronics to operate. The space environment can cause digital bits to flip, thus being read in error. Coudeyras confirmed this through research on FPGAs at Crooker Research Laboratory [1]. Prior research has been performed to allow an FPGA to detect errors and/or correct errors. Triple modular redundancy (TMR) and reduced precision redundancy (RPR) are two techniques that accomplish error correction. This is discussed in the literature review.

A. CFTP HISTORY

1. Midshipmen Space Technology Applications Research-1

The U.S. Naval Academy designed Midshipmen Space Technology Application Research-1 (MidSTAR) to incorporate a test bed for fault-tolerant techniques applied to

FPGAs developed by NPS. This consisted of two FPGAs, which were called the Configurable Fault-Tolerant Processor (CFTP). In September 2006, the United States Air Force launched CFTP into Low Earth Orbit (LEO) onboard the host Space Test Program (STP-1) satellite. CFTP detected seven single-event upsets, mainly while flying through the south-Atlantic Anomaly, during its 492 km, 46 degree inclination orbit [1]. The location of CFTP is displayed in Figure 2.

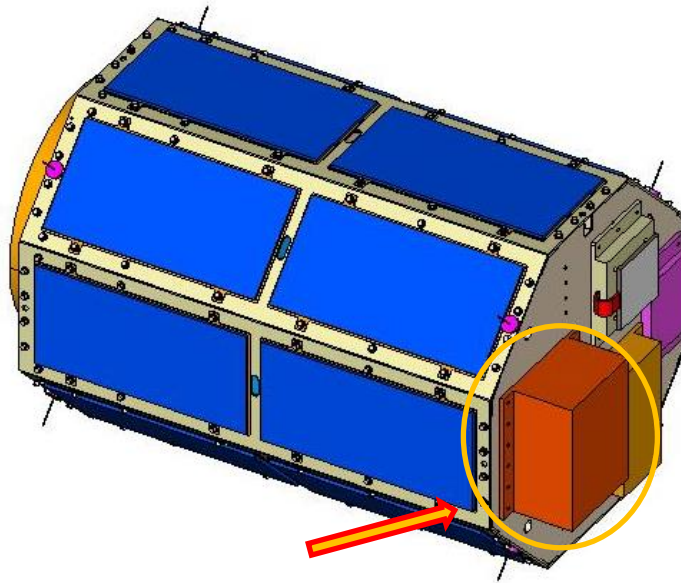


Figure 2. External View of MidStar-1 Showing the Location of CFTP-1.
Adapted from [6].

2. Naval Postgraduate School Satellite-1

NPS's effort on CFTP-7, which is currently in production, makes MidSTAR CFTP-1 obsolete. CFTP-7 is designed to be fault tolerant due to the use of TMR techniques. Its increased capability includes memory and processor improvements. The utilization of partial reconfigurations also increases capability. Four experiments will be preloaded into CFTP-7 at launch. This research project details one of the four experiments. Naval Postgraduate School Satellite-1 (NPSat-1) is scheduled to fly at 560 km at 35.4 degrees inclination and is scheduled to launch in FY18. An expanded view of the satellite is displayed in Figure 3.

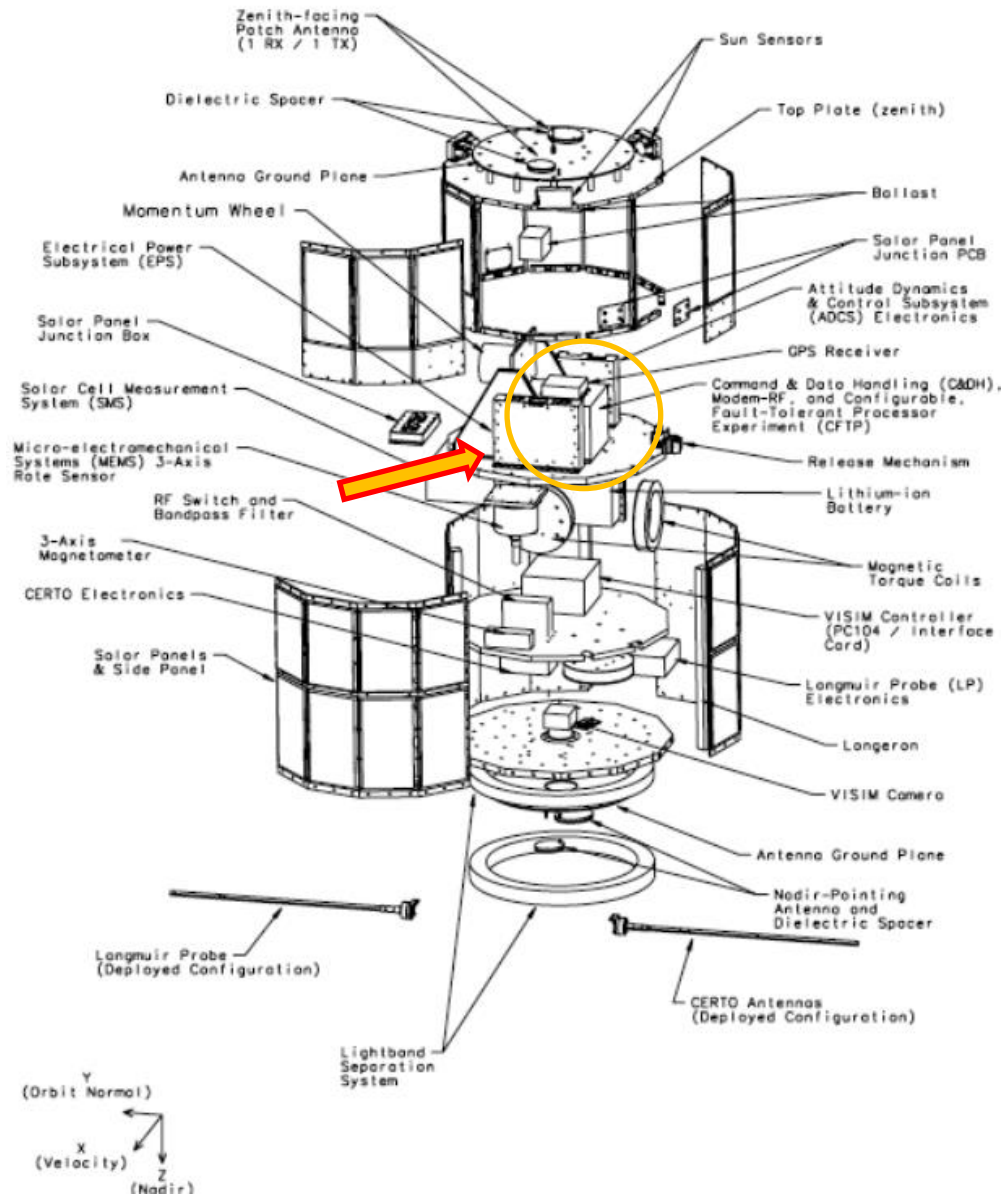


Figure 3. Expanded View of NPSat-1 That Shows Location of CFTP-7. Adapted from [7].

B. LITERATURE REVIEW

1. Academic Advisors

Dr. Alan A. Ross and Dr. Herschel H. Loomis have been the primary advisors for the CFTP project, completing a combined total of twenty theses and three dissertations. Dr. Ross (Lt. Col., USAF, retired) earned his PhD from the University of California, Davis,

in 1978 and recently retired as a professor in computer engineering. Dr. Loomis earned his PhD from Massachusetts Institute of Technology (MIT) in 1963. Dr. James H. Newman, chair, Space Systems Academic Group (SSAG), has participated as an advisor recently on CFTP projects. He earned his PhD from Rice University in 1984 and served as an astronaut from 1990–2008.

2. Research History

In June 2003, Dean Ebert researched the design trade offs for CFTP-1's initial concepts. In 2005, James Coudeyras completed his research in partnership with the Crocker Nuclear Laboratory in Davis, California in which he executes his radiation test plan utilizing their proton radiation beam [1]. He proved that space radiation has an impact on electronics, and these errors are known as SEUs. Also in December 2005, Peter Majewicz completed his research of a fault-tolerance technique called Triple Modular Redundancy. TMR instantiates three modules in parallel. They utilize a majority voter to correct errors in, at most, one component [8]. In December 2006, Gerald Caldwell completed his research on the design challenges present while utilizing two FPGAs [9]. In September 2008, David Dwiggins, Jr., redesigned the X1 control FPGA to be fault tolerant and added a microcontroller to manage internal components [10].

In December 2008, Margaret Sullivan completed her research implementing and analyzing a new method of fault tolerance called Reduced Precision Redundancy (RPR) [11]. She concluded that "RPR provided very good recovery from errors caused by SEU in spacecraft systems" [11]. To be clear, RPR protects a satellite's arithmetic module from SEU just like TMR. In addition, RPR has a lower power cost than TMR. RPR was developed by PhD student Josh Snodgrass in September 2006. He performed research on fault tolerance by means of reducing the precision of the redundant copies of a precise number used for error detection and correction. He named this method RPR. RPR applies only to arithmetic operations, and he proved this technique as viable using live proton radiation testing [12].

In December 2009, Jeremy Livingston completed his design to compress a wideband radio signal into a narrowband signal [13]. In December 2011, Caleb Humberd

completed his research on a FFT based compression algorithm and discovered a way to use Parseval's theorem to correct single-component errors in an FFT. In March 2016, Andrew Jackson completed his design of a TMR embedded MIPS processor architecture with a majority-output voter to combat single-event upsets for NPSat-1 on orbit [14].

The most important prior works directly related to this thesis are a thesis written by Michael Zimmer and a dissertation by Raymond Bernstein. Zimmer designed a radix-4 FFT that operated at 45 MHz, had a floating point multiplier and adder, and consisted of 20-bit words [15]. Bernstein utilized a FFT to design a vector-processing computer. He discovered that the structure of the memory system for a vector-based computer favored the butterfly machine (BFM) operation [3]. The BFM is a visible representation of the FFT computation and is described in detail in Chapter III.

The FFT described in this thesis is a product of 20 years of space and computer research. As we continue to operate complex electronics in space, the understanding of the impact of space radiation, how to mitigate those impacts in a cost effective manner, and to continue to improve every aspect of a computer system means research like this will continue to build on itself. In Chapter III, the FFT development and design choices are presented and matured.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DESIGN AND IMPLEMENTATION

Chapter III is organized into three major sections. Specifications are given in Section A for the end system. The FFT is described from a mathematical perspective in Section B. The architecture that must be implemented to realize the specified FFT is detailed in Section C. This algorithm was coded in the Verilog design language utilizing the Xilinx Vivado® design suite. It builds on known block diagrams and structures to realize the code.

A. SYSTEM SPECIFICATIONS

The FFT design consists of the 18-bit signed two's complement number system, a fixed point rational number representation with the binary point between the 16th and 17th bits. In addition, two 18-bit words representing the real and imagery part of a complex number system are utilized. Sample size N is equal to 8. Sub-sections are organized to elaborate on these details.

1. Two's Complement

Two's complement is a popular number system because it allows for the representation of negative numbers within a binary adder which may perform addition or subtraction. For an N -bit word, the range of values can be represented as -2^{N-1} to $2^{N-1}-1$. Commonly, the most significant bit (MSB) is utilized to determine if the number is positive or negative. A "0" in the MSB represents a positive number, and a "1" in the MSB represents a negative number. The value of an N -bit word when utilizing two's complement can be determined as [15]

$$V_{2's} = (-b_{N-1})(2^{N-1}) + \sum_{i=0}^{N-2} b_i 2^i \quad (2)$$

A simpler way to compute the two's complement of a number is to invert all bits in a binary number and then add 1. This converts a positive number to a negative number representation. An example of how this operation works is shown in Figure 4.

$$\begin{array}{r}
\underline{0101} \text{ (5)} \\
1010 \\
+ \quad \underline{1} \\
1011 \text{ (-5)}
\end{array}$$

Figure 4. Demonstration of Two's Complement Operation

2. Fixed-Point

When utilizing a fixed-point rational number representation, the binary point is established by the user to satisfy a design goal. Unlike the integer representation where, the binary point is to the right of the least-significant bit (LSB) [15], the binary point is located anywhere to the left of the LSB. Within this design, the binary point is after bit 0 as shown in Figure 5. This bounds values to be less than two and equal to or greater than negative two.

s	0.	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16
---	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----

Figure 5. Fixed-Point Signed Binary Representation.
Binary Point Place After Bit 0

The formula to determine a value when a fixed point two's complement representation is used is given by

$$V_{FixedPoint} = (-b_0)(2^{\bar{N}-1}) + \sum_{i=1}^{\bar{N}-2} b_i 2^{\bar{N}-1-i}, \quad b_i \in \{0,1\}, \quad (3)$$

where b_i represents the value of the i numbered bit, b_0 represents the value of the 0th bit, \bar{N} represents the total number of bits. Examples of equivalent radixes are shown in Table 3.

Table 3. 18-Bit Fixed-Point Binary Number with a 16-bit Radix Point Examples

Binary	Hex	Integer	Fixed-Point Binary	Actual	Exponential
01111111111111111111	1FFFF	131071	1.99998474121	<2	-
010000000000000000	10000	65536	1.00000000000	1	2^1
001000000000000000	8000	32768	0.50000000000	1/2	2^{-1}
000100000000000000	4000	16384	0.25000000000	1/4	2^{-2}
000010000000000000	2000	8192	0.12500000000	1/8	2^{-3}
000001000000000000	1000	4096	0.06250000000	1/16	2^{-4}
000000100000000000	0800	2048	0.03125000000	1/32	2^{-5}
000000010000000000	0400	1024	0.01562500000	1/64	2^{-6}
000000001000000000	0200	512	0.00781250000	1/128	2^{-7}
000000000100000000	0100	256	0.00390625000	1/256	2^{-8}
000000000010000000	0080	128	0.00195312000	1/512	2^{-9}
000000000001000000	0040	64	0.00097656000	1/1024	2^{-10}
000000000000100000	0020	32	0.00048828125	1/2048	2^{-11}
000000000000010000	0010	16	0.00024414062	1/4096	2^{-12}
000000000000001000	0008	8	0.00012207031	1/8192	2^{-13}
000000000000000100	0004	4	0.00006103515	1/16384	2^{-14}
000000000000000010	0002	2	0.00003051757	1/32768	2^{-15}
000000000000000001	0001	1	0.00001525878	1/65536	2^{-16}
000101010101010110	2AAB	10923	0.33334351	.33333	-
001011010011111110	5A7F	23167	.70700073	$\sqrt{2}/2$	-
00000000000010100	000A	10	.00030518	π	2^{-14} $+ 2^{-16}$

3. Complex Numbers

In implementing the arithmetic to compute the discrete Fourier transform, there is need to represent complex numbers. When transposing from the time domain to the frequency domain, W_N^{kn} introduces an imaginary component. A complex number is $a + jb$ where a is the real component of a complex number, jb is the imaginary component of a complex number, and $\sqrt{-1} = j$. In performing complex multiplication, two complex numbers can be expanded to produce

$$(a + jb)(c + jd) = (ac - bd) + (ad + bc)j \quad (4)$$

B. FAST FOURIER TRANSFORM ALGORITHMS

DSP progressed greatly when Cooley and Tukey discovered the FFT algorithm in 1965 [4]. The simplified BFM computation demonstrated in Figure 6 is a visual representation of the smallest element of a decomposed FFT calculation [4] and is in fact the two-point DFT. This element is used to construct the constant geometry decimation-in-time algorithm this research implements [4]. The twiddle factor, W_N^{kn} , seen in equation (1), is within this diagram. Simply, the twiddle factor is a predictable multiplier needed to correctly calculate the FFT that is factor of sample size, time, and BFM stage. The twiddle W_N^{kn} can be expanded to $e^{-j(2\pi/N)kn}$ [4]. In addition, odd samples are multiplied by the twiddle factor.

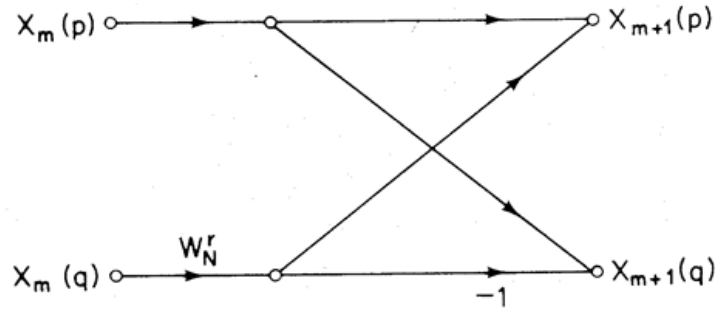


Figure 6. Flow Graph of Simplified Butterfly Machine Computation Requiring Only One Complex Multiplication. Source: [4].

1. Bit Reversal

The first step to utilizing the FFT within a DSP algorithm is to perform a bit reversal on the sample data. Bit reversal is a misnomer. Nothing is being done to the internal bits of the data; however, the data is being stored within alternating working buffers with the index transformed by reversing its bits as demonstrated in Figure 7.

$$\begin{array}{ll}
X_o[000] = \underline{x}[000] = x_m[p] & X_o[0] = x[0] \\
X_o[001] = \underline{x}[100] = x_m[q] & X_o[1] = x[4] \\
X_o[010] = \underline{x}[010] = x_m[p] & X_o[2] = x[2] \\
X_o[011] = \underline{x}[110] = x_m[q] & X_o[3] = x[6] \\
X_o[100] = \underline{x}[001] = x_m[p] & X_o[4] = x[1] \\
X_o[101] = \underline{x}[101] = x_m[q] & X_o[5] = x[5] \\
X_o[110] = \underline{x}[011] = x_m[p] & X_o[6] = x[3] \\
X_o[111] = \underline{x}[111] = x_m[q] & X_o[7] = x[7]
\end{array}$$

Figure 7. Bit Reversed Demonstration

2. The 8-Point DFT

Once the equation for the FFT is decomposed into odd and even sets, Figure 6 can be utilized to create Figure 8 to yield an algorithm for an 8-point DFT, which shows the bit-reversal applied to the input sequence. This algorithm is detailed in depth within *Discrete-Time Signal Processing* by Oppenheim and Schaffer [4]. To accommodate the needs of satellite-based DSP, a high speed pipelined FFT is desired. Constructing an FFT for pipelined operation is demonstrated in Figure 8. Data is loaded sequentially into the butterfly machine (BFM) computation alternating between $x_m(p)$ and $x_m(q)$.

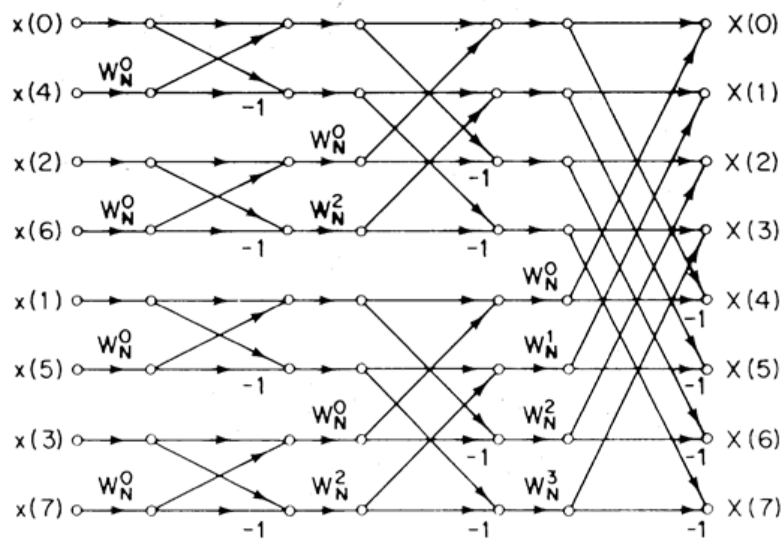


Figure 8. Flow Graph of 8-point DFT Using the Butterfly Machine Computation. Source: [4].

The algorithm being implemented is detailed in Figure 9. The constant geometry FFT is organized to take advantage of repeating patterns from stage to stage within the FFT structure. These patterns ease construction of a pipelined algorithm.

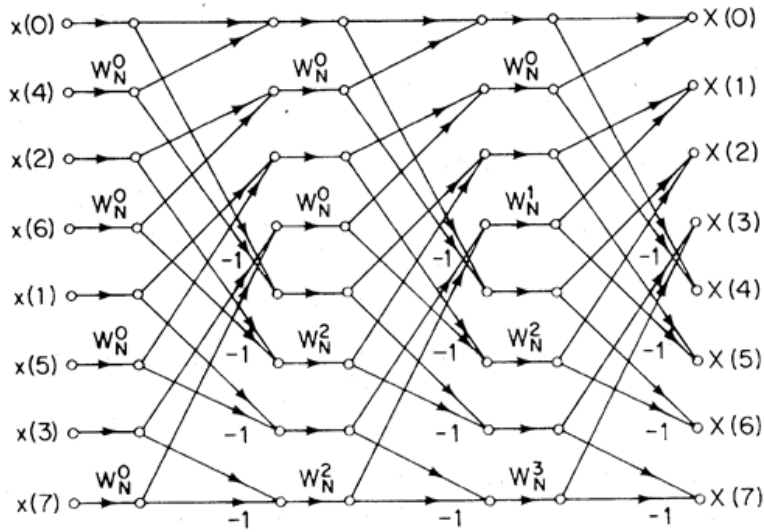


Figure 9. Rearrangement of Figure 1 to Allow Each Stage to Have a Constant Geometry Permitting Sequential Data Accessing and Storage. Source: [4]

C. HIGH-SPEED PIPELINED FFT ARCHITECTURE

A high speed pipelined FFT is demonstrated in Figure 10. Specifically, the image depicts a digit-reversed block, alternating memory buffers, and the BFM computations. There are three memory buffer/BFM pairs within a $N=8$ FFT.

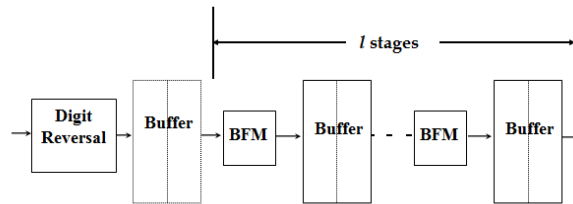


Figure 10. Basic Pipelined FFT Structure. $N = 2^l$, Where N Is the FFT Word Size. Adapted from [15].

Although similar to Figure 10, Figure 11 has added timing details to ensure that $x(i)$ is placed into the correct slot of the output BFM ping-pong buffer. While the first buffer is filled according to the bit reversed ordering discussed in this chapter, successive buffers are filled to allow for efficient pipelined operation. They are filled according to $k = 0, \frac{N}{2}, 1, \frac{N}{2} + 1, 2, \frac{N}{2} + 2, 3, \frac{N}{2} + 3, 4, \frac{N}{2} + 4, \dots, N - 1$ [15]. For $N = 8$, $k = 0, 4, 1, 5, 2, 6, 3, 7$. This order allows $x(i)$ to be fully utilized in both calculations in which it is involved before moving on to $x(i+1)$.

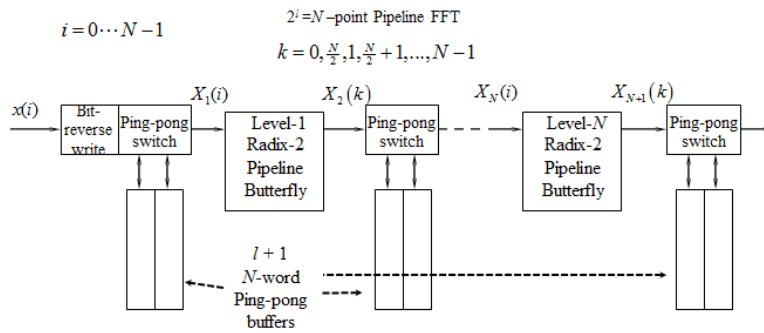


Figure 11. Detailed High Speed Pipelined FFT Structure. Adapted from [15].

1. Radix-2 Pipeline Butterfly Machine Architecture

A BFM that can be coded utilizing smaller blocks is displayed in Figure 12. It depicts two memory delays one clock-cycle long, two 2-input multiplexers, one memory delay $d_m + 1$ cycles long, a complex multiplier, and a complex adder / subtracter. Additionally, timing details and twiddle factors are calculated within the image. This entire circuit is made complex by utilizing separate registers for the real and imaginary portion of the complex number.

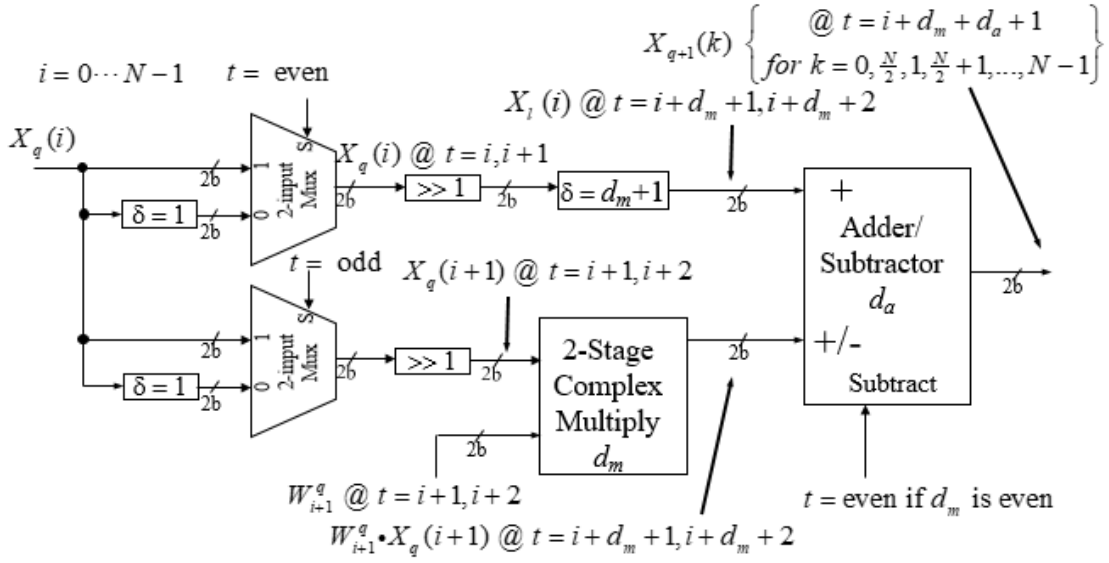


Figure 12. Pipeline Radix-2 Butterfly Machine (BFM) Architecture, Level q . Adapted from [15].

2. 2-Stage Pipelined Complex Multiplier Architecture

A one-stage pipeline real multiplier and a one-stage pipeline real adder is utilized to implement a two-stage pipeline complex multiplier. This is depicted in Figure 13. The multiplication must be performed on unsigned positive numbers. A module is inserted to perform a two's complement computation on negative numbers and strip off the sign bit, reducing numbers within the multiplier to 17 bits. After multiplication, there is an expansion to 34 bits, and a truncation is required to strip off the 17 least significant bits (LSBs). The truncated output has the sign added back and is two's complemented into a negative number if the exclusive-or of the signs of the multiplicand represent a negative product. An addition or subtraction is required to complete the complex product formation as a signed two's complement representation of the real and imaginary parts of the product.

Detailed Complex Multiplier

$$(a + jb)(c + jd) = (a*c - b*d) + j(a*d + b*c)$$

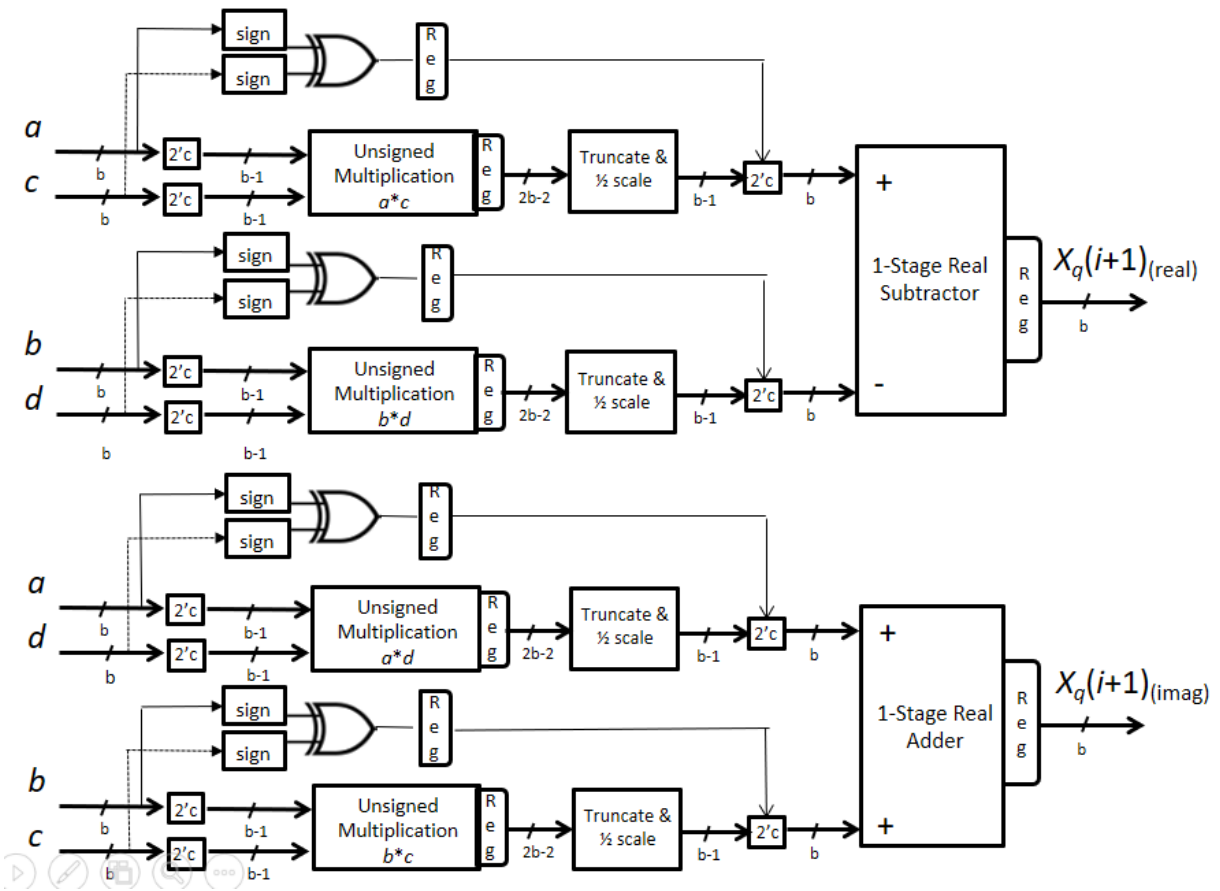


Figure 13. Two-Stage Complex Multiplier

Timing is important throughout the development of this design. Accessing a memory element, in this case a register, requires one clock cycle. This results in a one clock cycle delay in which the data is available for use. Testing must be done to ensure that multiplication and addition occur among the proper data elements. The delay of the multiplier d_m is equal to two clock cycles. The delay of the adder d_a is equal to one clock cycle. Timing is analyzed with simulation throughout the testing of the algorithm. This gives a visual depiction of where specific data elements align with the clock-cycle.

This design was instantiated in a Xilinx Artix-7 FPGA with Verilog hardware description language (HDL). The implementation and testing are discussed in Chapter IV.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. TESTING AND EVALUATION

Three major concepts are presented in Chapter IV. Planning and managing the testing of the FFT is in Section A. Here we find visual testing plans and major interfaces within the FFT that were tested. Sample inputs that were tested are listed in Section B. They showed up in hexadecimal, binary, and signed decimal throughout testing. Finally, the testing that was performed is demonstrated and displayed. The code that produced the test is described in detail, and simulations that were produced are analyzed. Verification of the code confirms that what was designed in Chapter III was produced and is usable for the intended space experiment purpose.

A. TESTING PLAN

This section is divided into three sub-sections. The test plan for the FFT is detailed in Sub-section 1. Here the FFT was considered as an entire system. Interfaces between code blocks, henceforth known as “modules,” were labeled to allow for correlation within simulations. The BFM module within the FFT is detailed in Sub-section 2. Being a smaller portion of the larger system FFT, it was labeled a component due to it consisting of multiple modules itself. This component performs multiplication on two signed binary numbers. The timing associated with this clocked systems is described and analyzed within Sub-section 3. Timing analysis ensures that data arrives for processing at the right time and location.

1. System Test Plan

A systematic approach to software testing was important due to the complex nature of the project. A system level test plan can be seen in Figure 14. Test points for simulation inputs and outputs are marked by numbers. Boxes represent the aforementioned components of the FFT. Test point S0 yields the initial vector that was sent to get digit reversed and stored. Test point S2 gives the output. Test point S3 is the outcome of the first stage BFM computation. Test point S4 is located at the end of stage 1.

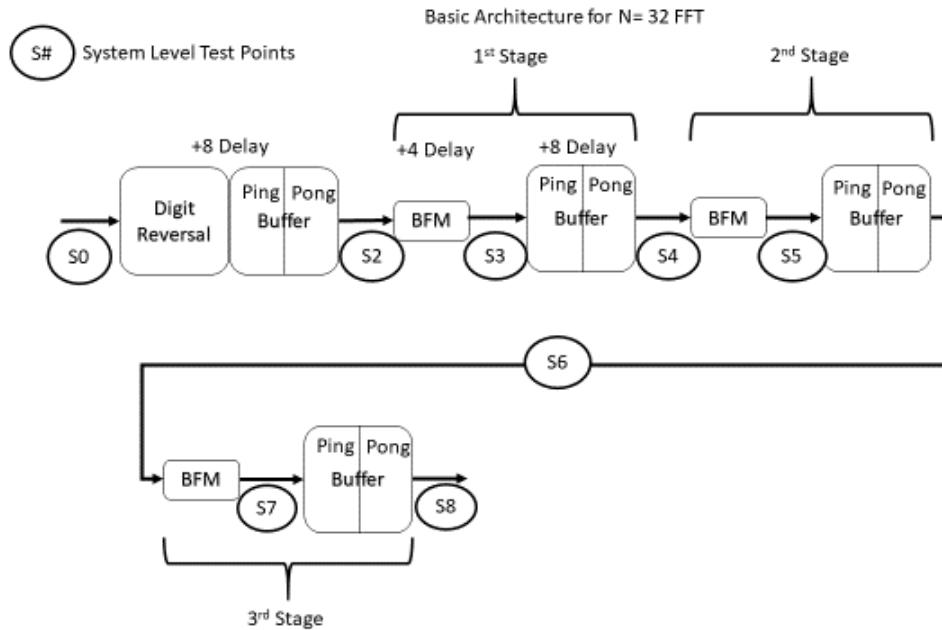


Figure 14. System Level Test Plan for $N = 8$ FFT

2. Component Test Plan

The component-level test plan for the BFM can be seen in Figure 15. The numbers on the BFM diagram represent input/output points that were examined to verify results and timing against expected values. Point C0 allows the observation of inputs into the BFM. From C1, we get the result of the data after a one clock-cycle delay. Point C2 allows us to observe the result of a 2-to-1 multiplexer (mux) that was selected on even time, and C3 allows us to observe the results of a 2-to-1 mux selected on odd time. Points C4 and C6 allow observation of the result of a shift right by one operation which results in a scale-by-half operation necessary to prevent the overflow in the BFM output adder / subtractor. This effectively divides the fixed-point binary number by two. Point C5 allows observation of the result of $d_m + 1$, where d_m is the delay of the multiplier and is equal to two. Point C7 allows observational of the result of the multiplication between Point C4 and the twiddle factor discussed earlier. Point C8 allows observation of the result of addition or subtraction of Points C5 and C7.

The operand was chosen by a selector set by the time. Subtraction occurred on even time if d_m was even. The results were sent to the first, second, and third ping pong buffer respective to the BFM stage as shown in Figure 14, S3, S5, and S7, respectively. Those ping pong buffers satisfy the constant-geometry reordering seen in Figure 9.

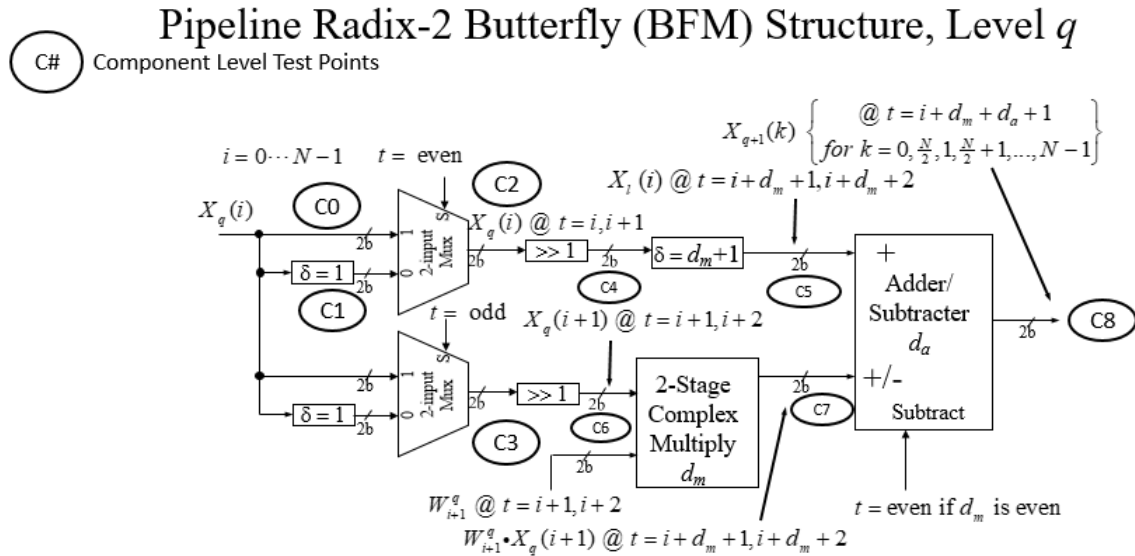


Figure 15. Component Level Test Plan for $N = 8$ BFM.
Adapted from [15].

3. Component Timing

The FPGA is connected to an internal clock running at 100 MHz. Variable CLK100MHZ was used to reflect this within the FFT code and waveforms. This physical clock was reduced to a 50 MHz clock and seen as clk_50 within the FFT code and waveforms. This 50 MHz clock drives multiple code snippets. *Registers* are triggered to load on the rising edge of the clock. This is where the clock transitions from a binary 0 to binary 1.

The processor operates in a pipelined fashion as demonstrated in Figure 16 and 17. This means that multiple processing elements occur in parallel. The first eight samples processed through the BFM are displayed in Figure 16 with a generic $X_q(t)$.

Butterfly Timing Diagram

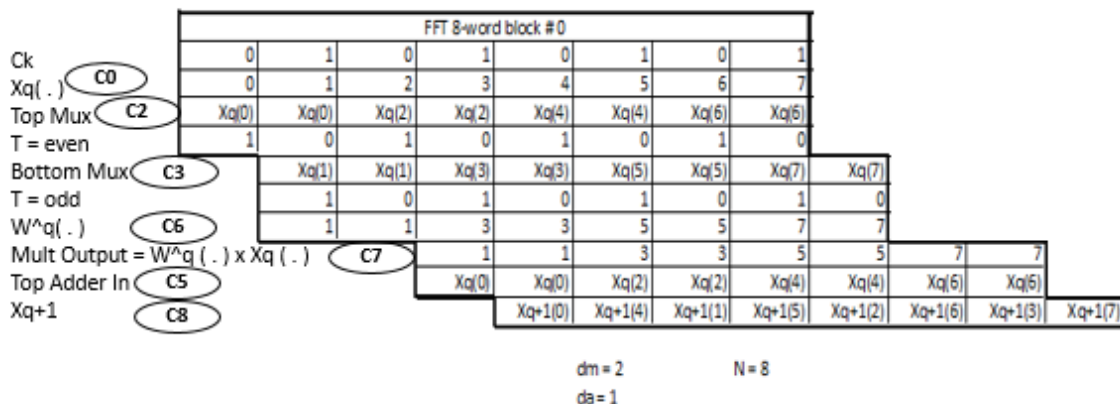


Figure 16. Butterfly Machine Timing Diagram Part A.
Adapted from [15].

The next eight samples that are processed through the BFM are displayed in Figure 17. Within $X_q(t)$, t is less than 8 and greater than or equal to 0. The $W^q(t)$ listed here was for test purposes only and is not a real value. Each successive N -word block was processed by the BFM and generated corresponding N -frequency components.

Butterfly Timing Diagram (continued)

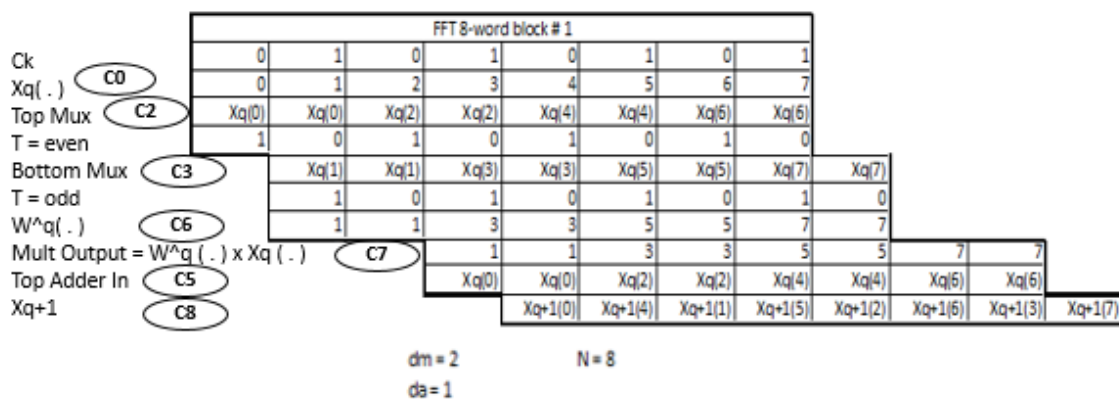


Figure 17. Butterfly Timing Machine Diagram Part B.
Adapted from [15].

Figures 16 and 17 were integral to the analysis of the FFT timing. Timing was synchronized by staggering the initialization of each BFM to accommodate the pipelined delays in the data input to the FFT.

B. TEST INPUTS

To verify outputs, there needs to be a known input for which expected outputs can be calculated. Starting with fixed point number 1.0, 2^{-16} was subtracted from each iteration to generate identifiable inputs. During some of the simulations, we see a hexadecimal representation. Figure 18 shows displays in binary within the simulation environment.



Figure 18. Injected $X_q(t)$ Inputs in Binary

Injected inputs have been listed in Table 4 in three possible formats to enhance readability. The stream continues without repeating; however, only the first sixteen have been listed.

Table 4. $X_q(t)$ Inputs Injected into FFT

$X_q(t)$	Binary	Fixed-Point Binary	Hexadecimal
$X_q(0)$	010000000000000000	1.0	10000
$X_q(1)$	001111111111111111	.99998474	0ffff
$X_q(2)$	001111111111111110	.99996948	0ffffe
$X_q(3)$	001111111111111101	.999954236	0ffffd
$X_q(4)$	001111111111111100	.9999389648	0ffffc
$X_q(5)$	001111111111111011	.9999237060	0ffffb
$X_q(6)$	001111111111111010	.999908447	0ffffa
$X_q(7)$	001111111111111001	.9998931	0fff9
$X_q(8)$	001111111111111000	.999877929	0fff8
$X_q(9)$	0011111111111110111	.9998626708	0fff7
$X_q(a)$	0011111111111110110	.999847412	0fff6
$X_q(b)$	0011111111111110101	.99983215	0fff5
$X_q(c)$	0011111111111110100	.999816894	0fff4
$X_q(d)$	0011111111111110011	.9998016357	0fff3
$X_q(e)$	0011111111111110010	.99978637	0fff2
$X_q(f)$	0011111111111110001	.99978484	0fff1

C. FFT SYSTEM AND COMPONENT TESTING

In this section, we detail each portion of the FFT code in snippets. The FFT is coded in behavioral Verilog. Vivado ISE’s® simulation tool was utilized to provide the simulation waveforms that demonstrated the functionality of the major components of Figures 11 and 12. Xilinx Vivado ISE® synthesized the behavioral Verilog into a hardware definition that detailed the interconnections of gates and registers. The Xilinx Vivado synthesizer produced a realization that was instantiated in an FPGA to perform successive 8-point FFTs in a pipelined fashion on 18-bit signal samples. Timing challenges as a result

of the pipelined nature of the design were present; however, timing diagrams were utilized within the testing process to ensure synchronization of data and verification of results.

1. Bit Reversed Ping-Pong Buffer

The Bit-Reversed Ping-Pong Buffer provides internal storage for the data to be delivered in bit-reversed order to the BFM and allows data to flow in “blocks” based on the FFT’s designed N . This Bit-Reversed Ping-Pong Buffer receives inputs in a sequential fashion, stores in a shuffled order, and outputs in a sequential order after $N = 8$ clock-cycles. A switch then occurs. The other half of the buffer receives inputs in a sequential fashion, while the previously filled buffer outputs in sequential order for $N = 8$ clock-cycles. This cycle repeats indefinitely.

The variables that are needed to code the Bit-Reversed Ping-Pong buffer are presented in Figure 19. $XqPing_Real$, $XqPing_Imag$, $XqPong_Real$, and $XqPong_Imag$ are two-dimensional arrays consisting of eight-eighteen bit words with the MSB in the left-most digit. Variables $transpose$ and $indexbr$ are both three bits wide, and the MSB is the left-most digit. Variables declared as *reg*, register, are not loaded with data until after a low-to-high clock-cycle. Variables declared as *wire* are placed immediately, but there is no storage mechanism for data.

```

reg [17:0] XqPing_Real [7:0];           // register for Ping real component
reg [17:0] XqPing_Imag [7:0];         // register for Ping imag component
reg [17:0] XqPong_Real [7:0];         // register for Pong real component
reg [17:0] XqPong_Imag [7:0];         // register for Pong imag component
wire [2:0] tranpose;                   // wire used to tranpose counter
reg [2:0] indexbr;                     // index utilized for bit reversal

```

Figure 19. Bit-Reversed Ping-Pong Buffer Variables and Arrays

To implement the bit reversal, an indexed value was created, shown as variable $indexbr$ in Figure 20. Variable $indexbr$ was used to transpose $\{transpose[2], transpose[1], transpose[0]\}$ into $\{transpose[0], transpose[1], transpose[2]\}$. $Indexbr$ was then used to address the arrays $XqPing_Real$, $XqPing_Imag$, $XqPong_Real$, and $XqPong_Imag$ to direct the input stream $X_q(t)$ into the correct memory element. Bit counter[3], which is the MSB

of a four-bit counter, switched between even and odd every eight clock-cycles and triggered the *if/else* statement to load the Bit-Reversed Ping-Buffer or the Bit-Reversed Pong-Buffer. Variables *ping_loading* or *pong_loading* are set to high to indicate the active storage buffer on the simulations. A code snippet of the Bit-Reversed Ping-Pong Buffer loading into memory is shown in Figure 20.

```

76 always@(posedge clk)
77     begin
78         // counter transposed due to bit reversal
79         indexbr <= {tranpose[0],tranpose[1],tranpose[2]};
80
81         //The Pong buffer is outputted when the Ping buffer is
82         //being filled up. Ping and Pong then alternate
83         //The MSB of counter is utilized as
84         //my switch bit. Buffers are complex numbers.
85
86         //Alternating loading Ping and Pong buffers
87         //based on MSB counter. Results in 16
88         //cycle period.
89         if (counter[3] == 1'b1) begin
90             XqPing_Real[indexbr] <= XqIn_PingPong_Real;
91             XqPing_Imag[indexbr] <= XqIn_PingPong_Imag;
92             ping_loading <= 1'b1;
93             pong_loading <= 1'b0;
94         end
95         else begin
96             XqPong_Real[indexbr] <= XqIn_PingPong_Real;
97             XqPong_Imag[indexbr] <= XqIn_PingPong_Imag;
98             pong_loading <= 1'b1;
99             ping_loading <= 1'b0;
100        end
101    end

```

Figure 20. Bit-Reversed Ping-Pong Buffer Loading, Code Snippet

The Bit-Reversed Ping-Pong buffer outputs the reordered $X_q(t)$ in variables *XqOut_PingPongw_Real* and *XqOut_PingPongw_Imag*. A conditional statement triggered on the MSB of the four-bit counter controls which buffer loads. Synthesized designs connect wires from the data of the Bit-Reversed Ping buffer or Bit-Reversed Pong buffer to the output variables depending on the result of the conditional statement. A low signal connects the Bit-Reversed Ping buffer, and a high signal connects the Bit-Reversed Pong buffer. The code snippet in Figure 21 displays the Bit-Reversed Ping-Pong buffer output process.

```

64 assign XqOut_PingPongw_Real = (counter[3]==1'b0) ? XqPing_Real[counter[2:0]] : XqPong_Real[counter[2:0]];
65 assign XqOut_PingPongw_Imag = (counter[3]==1'b0) ? XqPing_Imag[counter[2:0]] : XqPong_Imag[counter[2:0]];

```

Figure 21. Bit-Reversed Ping-Pong Buffer Output $X_q(t)$

The bit-reversed index, seen as *indexbr* within Figure 22, displays the bit reversal. Variable *indexbr* is a reversal of the transpose bits. The display is listed in binary; however, a conversion to decimal reveals the relationship discussed in Figure 7.

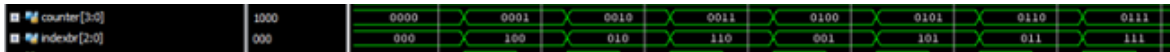


Figure 22. Variable *Indexbr* transposed from Counter[2:0]

Next, simulation waveforms were used to demonstrate the proper function of the code. Clock-cycles, buffer-memory, and data values can be observed producing the expected results in Figure 23. In this image Pong was loaded in the sequential order indicated by the binary high bit in the *pong_loading* register. In addition, Figure 23 contains the successful Bit-Reversed Pong-Buffer instantiation and displays the $X_q(t)$ {10000, 0ffff, 0fffe, 0fffd, 0fffc, 0fffb, 0fffa, 0fff9} being stored into the two-dimensional memory element as {10000, 0fffc, 0fffe, 0fffa, 0fff, 0fffb, 0fffd}. This is the expected bit reversed order. Eight clock-cycles after $X_q(t)$ was received as an input into the Bit-Reversed Ping-Pong Buffer, the input sequence outputs in the bit-reversed order.

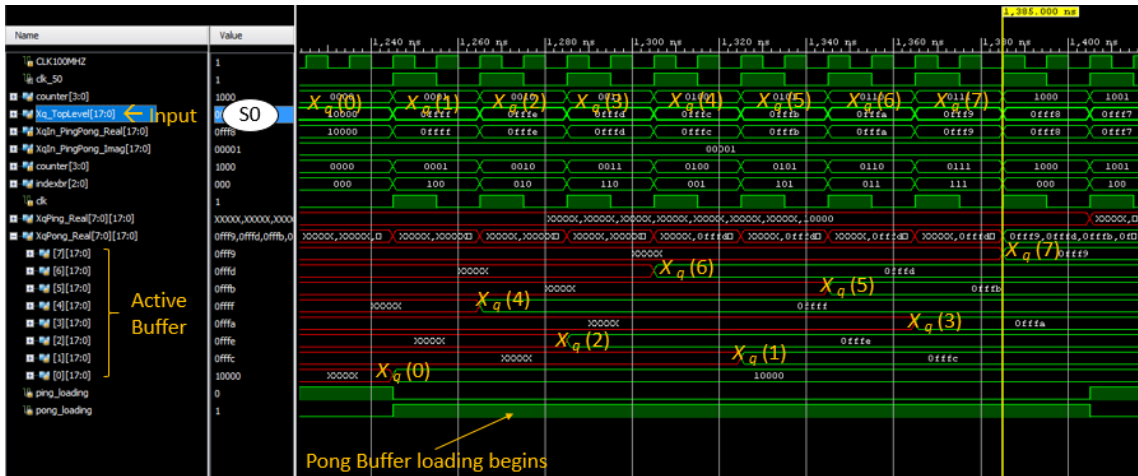


Figure 23. Pong Buffer Loading in Bit Reversed Order, Ping Empty

The Bit-Reversed Pong-Buffer load required the eight clock-cycles shown in Figure 24 to get the first $X_q(t)$ block of data to the BFM. This startup-delay only occurs once. It is required for every BFM stage.

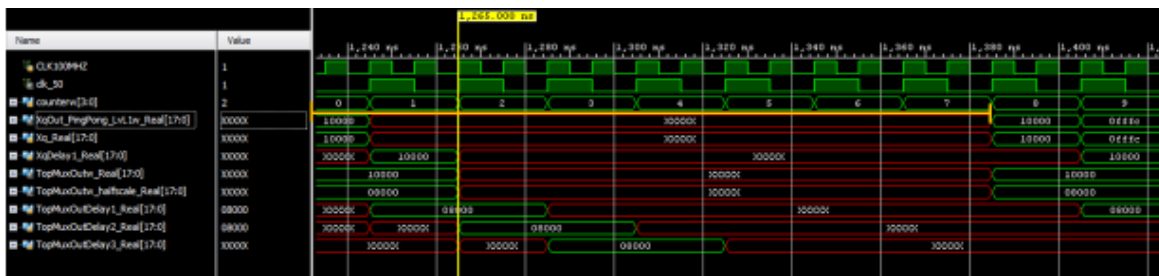


Figure 24. Startup Delay Between Bit-Reversed Ping-Pong Buffer Output and BFM

The Bit-Reversed Ping-Pong buffer switched loads as shown in Figure 25. The Bit-Reversed Pong buffer was filled as shown in Figure 23 and simultaneously outputted data in a sequential order, while Bit-Reversed Ping buffer was filled in bit reversed order. Input data, known as $X_q(t)$, holds the values of {0fff8, 0fff7, 0fff6, 0fff5, 0fff4, 0fff3, 0fff2, 0fff1} and is stored in the two-dimensional Bit-Reversed Ping buffer as {0fff8, 0fff4, 0fff6, 0fff2, 0fff7, 0fff3, 0fff5, 0fff1}. These values represent the second block of data outputted to the first BFM.

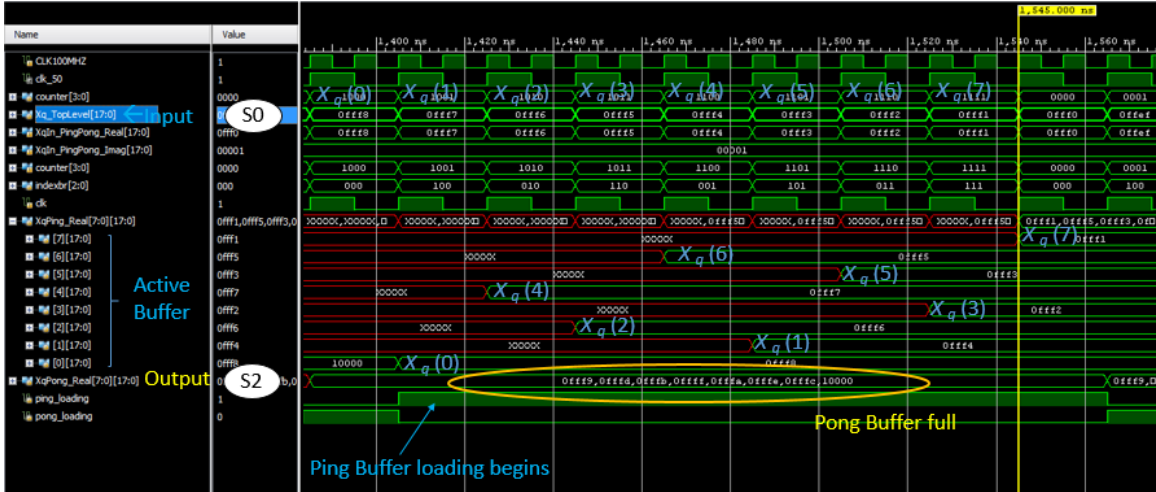


Figure 25. Ping-Buffer Loading in Bit Reversed Order, Pong Full

2. Radix-2 Pipeline Butterfly Machine Sub-Component Testing

The BFM is the primary processing element of the FFT. BFM's operate on blocks of data and are designed specifically for the N -block they are processing. This BFM is operating on $N = 8$ clock-cycles. Variables for the BFM were declared as *wires* with exception of the registers needed to introduce required delays as shown in Figure 26. These delays are needed for data synchronization within the BFM. All needed variables are 18 bits wide with the MSB to the left. Variables are paired to form the real and imaginary portions of a complex number.

```

34 (* dont_touch = "true" *)reg [17:0] TopMuxOutDelay1_Real, TopMuxOutDelay2_Real, TopMuxOutDelay3_Real;
35 (* dont_touch = "true" *)reg [17:0] TopMuxOutDelay1_Imag, TopMuxOutDelay2_Imag, TopMuxOutDelay3_Imag;
36 (* dont_touch = "true" *)wire [17:0] TopMuxOutw_Real, BottomMuxOutw_Real;
37 (* dont_touch = "true" *)wire [17:0] TopMuxOutw_Imag, BottomMuxOutw_Imag;
38 (* dont_touch = "true" *)wire [17:0] TopMuxOutw_halfscale_Real, BottomMuxOutw_halfscale_Real;
39 (* dont_touch = "true" *)wire [17:0] TopMuxOutw_halfscale_Imag, BottomMuxOutw_halfscale_Imag;
40 (* dont_touch = "true" *)reg [17:0] XqDelay1_Real;
41 (* dont_touch = "true" *)reg [17:0] XqDelay1_Imag;
42 (* dont_touch = "true" *)wire [17:0] MultOutw_Real;
43 (* dont_touch = "true" *)wire [17:0] MultOutw_Imag;
44 (* dont_touch = "true" *)wire [17:0] AdderOutw_Real;
45 (* dont_touch = "true" *)wire [17:0] AdderOutw_Imag;

```

Figure 26. Butterfly Machine Variable Declaration Code Snippet

Two multiplexers are needed. One multiplexer, visibly the “top” multiplexer within the architecture diagram (Figure 15), selects on $X_q(t)$ or $X_q(t-1)$ based on even time

while the “bottom” multiplexer selects $X_q(t)$ or $X_q(t-1)$ based on odd time. This results in data falling on even time slots passing through the “top” multiplexer and data falling on odd time slots passing through the “bottom” multiplexer. Within the code, a single module is instantiated for both “top and “bottom” multiplexer; however, the input for the *clk* are opposed between multiplexers. Specifically, when the “top” multiplexer receives a high *clk* signal, the “bottom” multiplexer receives a low *clk* signal. Both continue to cycle between low and high *clk* signals. The code snippet for both multiplexers are shown in Figure 27.

```

47 // ----- 2 to 1 Multiplexer Instantiated for top -----
48 two2oneMux two2oneMux_Top
49 (
50     .in1_real(Xq_Real),
51     .in1_imag(Xq_Imag),
52     .in2_real(XqDelay1_Real),
53     .in2_imag(XqDelay1_Imag),
54     .clk(~counterw[0]),
55     .out_real(TopMuxOutw_Real),
56     .out_imag(TopMuxOutw_Imag)
57 );
58
59 // ----- 2 to 1 Multiplexer Instantiated for bottom -----
60 two2oneMux two2oneMux_Bottom
61 (
62     .in1_real(Xq_Real),
63     .in1_imag(Xq_Imag),
64     .in2_real(XqDelay1_Real),
65     .in2_imag(XqDelay1_Imag),
66     .clk(counterw[0]),
67     .out_real(BottomMuxOutw_Real),
68     .out_imag(BottomMuxOutw_Imag)
69 );

```

Figure 27. “Top” and “Bottom” Multiplexer Code Snippet

The delay producing $X_q(t-1)$ into both multiplexers is coded by sending the signal into a register. By loading after a clock-cycle, we introduce a delay into the data. This is an expected, and required, delay and is shown coded in Figure 28.

```

111 XqDelay1_Real <= Xq_Real;
112 XqDelay1_Imag <= Xq_Imag;

```

Figure 28. Delay into Both Multiplexer Code Snippet

a. *Top Multiplexer, Half Scale, and 3-Cycle-Delay Sub-Component Test*

The internals of the multiplexer is a conditional statement selecting on even and odd time. If time is even, selection is on $X_q(t-1)$. If time is odd, selection is on $X_q(t)$. Variables are 18 bits with the MSB to the left. The code snippet for the internals of the multiplexer is shown in Figure 29.

```
22 module two2oneMux (  
23     input wire signed [17:0] in1_real, // a  
24     input wire signed [17:0] in1_imag, // jb  
25     input wire signed [17:0] in2_real, // c  
26     input wire signed [17:0] in2_imag, // jd  
27     input wire                clk,  
28     output wire signed [17:0] out_real, // c or a  
29     output wire signed [17:0] out_imag // jd or jb  
30 );  
31  
32     assign out_real = (clk==1'b0) ? in2_real : in1_real; //Real Component  
33     assign out_imag = (clk==1'b0) ? in2_imag : in1_imag; //Imag Component  
34 endmodule
```

Figure 29. Internal Multiplexer Code Snippet

An arithmetic shift right by one moves the binary point one bit to the left. When performed on the data, outputs of the “top” multiplexer” are reduced by a factor of-one-half. The code in Figure 30 was utilized for this shift operation.

```
98     assign TopMuxOutw_halfscale_Real = TopMuxOutw_Real >> 1;  
99     assign TopMuxOutw_halfscale_Imag = TopMuxOutw_Imag >> 1;
```

Figure 30. Top Multiplexer Half Scale

Half-scaled data from the “top” multiplexer was then delayed by three clock-cycles. Implementation of this was performed utilizing registers. Each register store operation introduces a one clock-cycle delay. The code can be seen in Figure 31.

```

105 TopMuxOutDelay1_Real <= TopMuxOutw_halfscale_Real;
106 TopMuxOutDelay1_Imag <= TopMuxOutw_halfscale_Imag;
107 TopMuxOutDelay2_Real <= TopMuxOutDelay1_Real;
108 TopMuxOutDelay2_Imag <= TopMuxOutDelay1_Imag;
109 TopMuxOutDelay3_Real <= TopMuxOutDelay2_Real;
110 TopMuxOutDelay3_Imag <= TopMuxOutDelay2_Imag;

```

Figure 31. Top Multiplexer Three-Clock-Cycle Delay

BFM test point C0 indicates the input of the BFM. The delay into the multiplexer can be read at test point C1. Test point C2 presents the output of the “top” multiplexer, C4 shows the data after the half-scale operation, and C5 displays the output to the top multiplexer after a three clock-cycle delay. Half-scaling data results in a loss of significance while truncating. The multiplexer and the delayed outputs are shown in Figures 32, 33, and 34. Functionally, they are the same waveform; however, the radix has been changed to support readability from different perspectives. Hexadecimal, fixed-point binary, and binary views are present. The delay being observed was needed to synchronize data with the bottom multiplexer due to the multiplier delay. Registers store data internal to the multiplier operation resulting in a two clock-cycle delay. Test point C5 demonstrates the delay of C2 after the half-scale operation seen in C4.

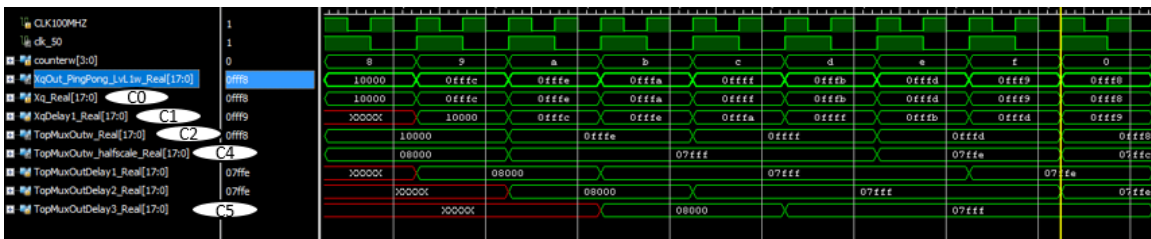


Figure 32. Top Multiplexer Output, Hexadecimal

A simulation view for fixed-point binary, 16-bit radix notation can be seen in Figure 33. This allows the decimal number to be read. Recall that values have been normalized to $-2 \leq X_q(t) < 2$.

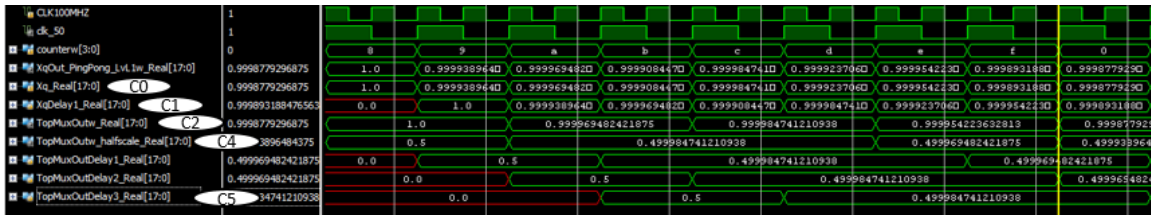


Figure 33. Top Multiplexer Output, Fixed-Point Binary

A binary view has also been provided and can be seen in Figure 34. Each view has utility based on what information is desired.

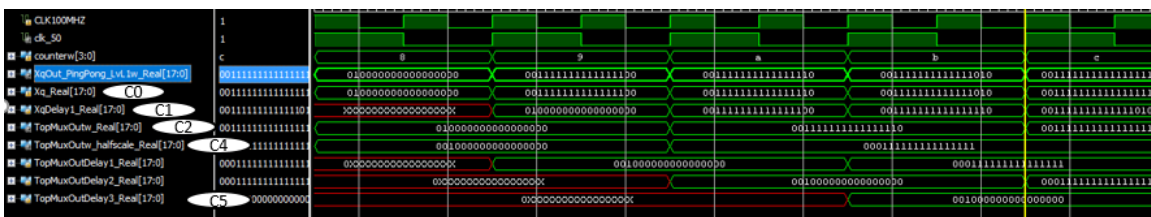


Figure 34. Top Multiplexer Output, Binary

b. Bottom Multiplexer Sub-Component Test

The “bottom” multiplexer utilizes the same module call to *two2oneMux*. The input change representing odd time, binds odd data points to this multiplexer as shown in Figure 35.

```

22 module two2oneMux (
23     input wire signed [17:0] in1_real, // a
24     input wire signed [17:0] in1_imag, // jb
25     input wire signed [17:0] in2_real, // c
26     input wire signed [17:0] in2_imag, // jd
27     input wire clk,
28     output wire signed [17:0] out_real, // c or a
29     output wire signed [17:0] out_imag // jd or jb
30 );
31
32 assign out_real = (clk==1'b0) ? in2_real : in1_real; //Real Component
33 assign out_imag = (clk==1'b0) ? in2_imag : in1_imag; //Imag Component
34 endmodule

```

Figure 35. Multiplexer Internal Code Snippet

Data in the “bottom” multiplexer is half scaled utilizing the same methodology as the “top” multiplexer. The output to the multiplexer is shifted right by one binary place utilizing the code in Figure 36.

```

100     assign BottomMuxOutw_halfscale_Real = BottomMuxOutw_Real >> 1;
101     assign BottomMuxOutw_halfscale_Imag = BottomMuxOutw_Imag >> 1;

```

Figure 36. Bottom Multiplexer Half-scale Code Snippet

The bottom multiplexer simulation is displayed in Figure 37 and outputs its $X_q(t)$ at odd ts . We can see this at test point C3. Test point C6 is the half-scale. Test point C7 is the output of the multiplier. The multiplier in this instance is viewed via a black box analysis. This means that only the inputs and outputs are analyzed.

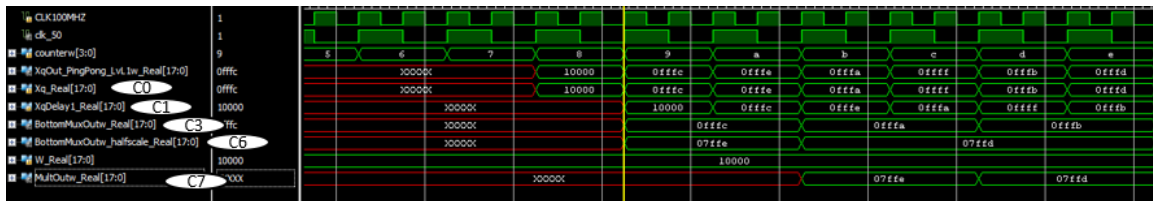


Figure 37. Bottom Multiplexer Output with Multiplier as a Black Box

c. Rate_OneHalf_complex_Multiply Sub-Component Test

In section, we provide insight into the internal workings of the multiplier. Performing complex multiplication is not a trivial operation, and this code consists of many module calls. Each module call added interfaces and processing elements that had to be tested.

As part of performing multiplication within hardware, data manipulation is required to get the data into a format that behavioral Verilog can process. To start, negative binary numbers are stored in signed two’s complement. The multiplication operation produces the anticipated result if operating on unsigned magnitude numbers; thus, signed two’s complement numbers are first converted to unsigned magnitude. Unsigned multiplication

is performed utilizing a behavioral operator. The sign is operated on separately and appended to the final resultant.

The final number must add the negative or positive sign that results from normal multiplication of signed numbers. The signs of the multiplicands are exclusive-or'd with each other to produce the correct sign. The new sign triggers a two's complement operation on the output of the truncated multiplication if the sign results in a negative number. The sign bit is then concatenated as the MSB to the truncated result. The multiplier test plan is provided in Figure 38 to guide a systematic way to test the multiplier and create common nomenclature for testable points.

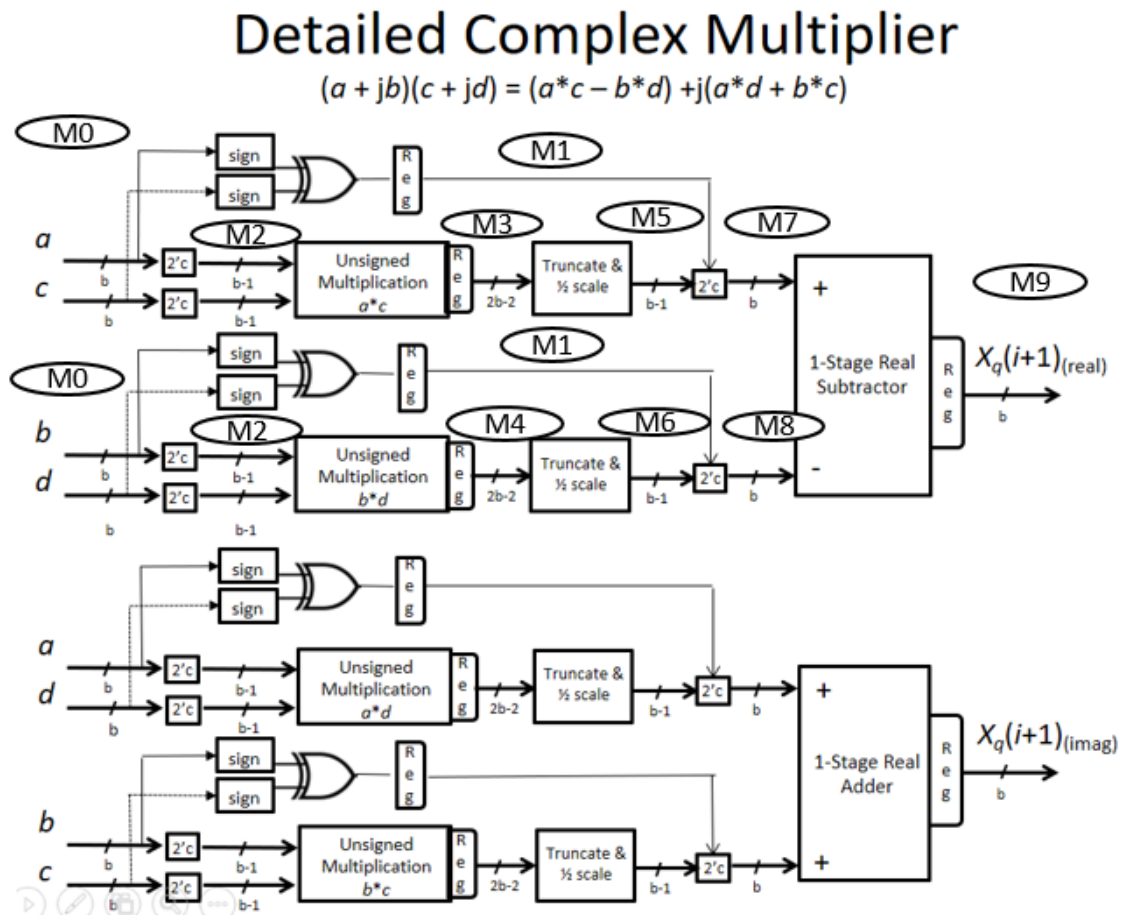


Figure 38. Test Plan for Multiplier and Sub-Component testing

The *wire* and register variables required for the multiplier coding are listed in Figure 39. Registers *sign_mult_left_Real*, *sign_mult_right_Real*, *sign_mult_left_Imag*, and *sign_mult_right_Imag* are utilized to store the result of complex multiplication on signed data. Four wires, one bit wide, indicate the sign of each portion of the complex number. Four wires, 18 bits wide, hold the concatenated 17-bit wide *unsigned_a*, *unsigned_jb*, *unsigned_c*, and *unsigned_jd* and their respective 1-bit signs.

```

11 (* dont_touch = "true" *) reg sign_mult_left_Real, sign_mult_right_Real, sign_mult_left_Imag, sign_mult_right_Imag;
12 (* dont_touch = "true" *) wire sign_a;
13 (* dont_touch = "true" *) wire sign_jb;
14 (* dont_touch = "true" *) wire sign_c;
15 (* dont_touch = "true" *) wire sign_jd;
16 (* dont_touch = "true" *) wire [16:0] unsigned_a; //
17 (* dont_touch = "true" *) wire [16:0] unsigned_jb; //
18 (* dont_touch = "true" *) wire [16:0] unsigned_c; //
19 (* dont_touch = "true" *) wire [16:0] unsigned_jd; //
20 (* dont_touch = "true" *) wire [16:0] trunc_left_Real; // (a^c)
21 (* dont_touch = "true" *) wire [16:0] trunc_right_Real; // (jb^jd)
22 (* dont_touch = "true" *) wire [16:0] trunc_left_Imag; // (a^jd)
23 (* dont_touch = "true" *) wire [16:0] trunc_right_Imag; // (jb^c)
24 (* dont_touch = "true" *) wire [17:0] signed_left_Real; // (a^c)
25 (* dont_touch = "true" *) wire [17:0] signed_right_Real; // (jb^jd)
26 (* dont_touch = "true" *) wire [17:0] signed_left_Imag; // (a^jd)
27 (* dont_touch = "true" *) wire [17:0] signed_right_Imag; // (jb^c)

```

Figure 39. Multiplier Variables Code Snippet

This code segment details four calls to *twoComp* module. Each portion of the complex number is individually passed into the module as an 18-bit binary number. A 17-bit unsigned magnitude number was returned as an output and connected to a wire as seen in Figure 40.

```

52 //2^c operator prior to multiplication. returns unsigned number. returns 17 bits
53 twoComp insta(.In(a), .Out(unsigned_a)); //extracts data portion a
54 twoComp instb(.In(jb), .Out(unsigned_jb)); //extracts data portion jb
55 twoComp instc(.In(c), .Out(unsigned_c)); //extracts data portion c
56 twoComp instd(.In(jd), .Out(unsigned_jd)); //extracts data portion jd

```

Figure 40. Code Snippet for *twoComp* Module Call

Internal to the *twoComp* module is a conditional statement that selects an output based on the value of the MSB. If the MSB is a digital high, the input is inverted and incremented by one bit; while a digital low returns a copy of the input. This can be seen

coded in Figure 41. The MSB was truncated off so an unsigned magnitude number could be returned to the parent module.

```
23 module twoComp(  
24   input wire [17:0] In,  
25   output wire [16:0] Out);  
26  
27   assign Out = In[17]?~In+1:In;
```

Figure 41. Code Snippet for *twoComp* Module Internals

The MSB that was truncated within the *twoComp* module was also stored for use later within the multiplier module. The coding for the multiplier module store operation is displayed in Figure 42.

```
58 // saves sign bit. returns 1 bit  
59 assign sign_a = a[17]; //extracts sign of a  
60 assign sign_jb = jb[17]; //extracts sign of jb  
61 assign sign_c = c[17]; //extracts sign of c  
62 assign sign_jd = jd[17]; //extracts sign of jd
```

Figure 42. Sign Bit Extraction Code Snippet

Unsigned numbers in the one's complement form can successfully pass through a behavioral multiplier and produce expected results. A module was called to perform behavioral multiplication on the 17-bit binary numbers seen in Figure 43. Four 17-bit binary numbers are returned.

```

34     mult18x18 unsigned_multiply
35     (
36     .clk(clk),
37     .a(unsigned_a),
38     .jb(unsigned_jb),
39     .c(unsigned_c),
40     .jd(unsigned_jd),
41     .Out_left_Real(trunc_left_Real),
42     .Out_right_Real(trunc_right_Real),
43     .Out_left_Imag(trunc_left_Imag),
44     .Out_right_Imag(trunc_right_Imag));

```

Figure 43. Module Call for Behavioral Multiplication on Four Unsigned Complex Numbers Code Snippet

The unsigned multiplier module creates registers to store the output of the multiplication. Registers *mult_left_Real*, *mult_right_Real*, *mult_left_Imag*, and *mult_right_Imag* are declared as 35-bit registers. Storage must be available to accommodate the largest possible multiple. The coding of these variables is displayed in Figure 44.

```

34     //Complex Multiplier
35     //           Real           Imag
36     //(a+jb)(c+jd)= (a*c - b*d) + j(b*c + a*d)
37     (* dont_touch = "true" *)reg [34:0] mult_left_Real;
38     (* dont_touch = "true" *)reg [34:0] mult_right_Real;
39     (* dont_touch = "true" *)reg [34:0] mult_left_Imag;
40     (* dont_touch = "true" *)reg [34:0] mult_right_Imag;

```

Figure 44. Unsigned Multiplication Module Code Snippet

Complex multiplication can be seen coded in Figure 45. This code snippet is within the unsigned multiplier module. Multiplications are performed at the positive edge of every clock-cycle, and the results are stored within one of the four 35-bit registers. When two 17-bit binary numbers multiply, they produce one 34-bit binary number.

```

50 always@(posedge clk)
51 begin
52     //Calculate real portion
53     mult_left_Real <= a * c;
54     mult_right_Real <= jb * jd;
55
56     //Calculate imag portion
57     mult_left_Imag <= a * jd;
58     mult_right_Imag <= jb * c;
59 end

```

Figure 45. Behavioral Multiplication Code Snippet

The parent module functions with 17-bit binary numbers. The 17 leading bits contain the relevant data. The trailing 17 bits add precision to the binary number; however, truncation is required to return product to the original 17-bit binary number format. The truncation code snippet is displayed in Figure 46.

```

42 assign Out_left_Real[16:0] = mult_left_Real[32:16]; // (a*c)
43 assign Out_right_Real[16:0] = mult_right_Real[32:16]; // (jb*jd)
44 assign Out_left_Imag[16:0] = mult_left_Imag[32:16]; // (a*jd)
45 assign Out_right_Imag[16:0] = mult_right_Imag[32:16]; // (jb*c)

```

Figure 46. Output Truncation Code Snippet

The exclusive-or operation is performed on the sign bits within the multiplier parent module on the rising edge of the clock-cycle to compute the new sign. This can be seen coded in Figure 47. If the sign of the product is negative, the result is a concatenation of the computed sign and the two's complement of the magnitude of the product. If the sign of the product is positive, the result is a concatenation of the computed sign and the unsigned magnitude of the product.

```

72 // Statements below will be executed on every clock rising edge
73
74 always@(posedge clk)
75 begin
76     sign_mult_left_Real  <= sign_a ^ sign_c;
77     sign_mult_right_Real <= sign_jb ^ sign_jd;
78     sign_mult_left_Imag  <= sign_a ^ sign_jd;
79     sign_mult_right_Imag <= sign_jb ^ sign_c;
80 end

```

Figure 47. Exclusive-Or of Sign-Bit Code Snippet

Module *twoCompRedo* receives the new sign bit and the truncated output of the multiplier and produces the two's complement 18-bit numbers. The code snippet for the module call is displayed in Figure 48.

```

46 //2's operator following multiplication. Concatenates the sign bit with a 2's number. returns 18 bits
47 twoCompRedo twoC_leftReal(.signbit(sign_mult_left_Real),.number(trunc_left_Real),.sign_number(signed_left_Real));
48 twoCompRedo twoC_rightReal(.signbit(sign_mult_right_Real),.number(trunc_right_Real),.sign_number(signed_right_Real));
49 twoCompRedo twoC_leftImag(.signbit(sign_mult_left_Imag),.number(trunc_left_Imag),.sign_number(signed_left_Imag));
50 twoCompRedo twoC_rightImag(.signbit(sign_mult_right_Imag),.number(trunc_right_Imag),.sign_number(signed_right_Imag));

```

Figure 48. Code Snippet for *twoCompRedo* Module Call

Peering inside of *twoCompRedo*, we see that module *twoCompRedo* consists of a conditional statement selected with the sign bit. If the sign bit is a binary one, every bit in the binary number is inverted and then one bit is added. That binary number is then concatenated with the sign-bit before being sent to a *wire* for output. If the sign bit is zero, the original binary number is outputted after being concatenated with the sign bit. The *twoCompRedo* module code snippet is displayed in Figure 49.

```

23 module twoCompRedo(
24     input wire signbit,
25     input wire [16:0] number,
26     output wire [17:0] sign_number
27 );
28
29 assign sign_number = signbit?{signbit,~number+1}:{signbit,number};
30 endmodule

```

Figure 49. Module *twoCompRedo* Code Snippet

The final stage consists of adding or subtracting the signed multiplier output. The real component of the complex number is subtracted to get the final real number. The imaginary component of the complex number is added to get the final imaginary number. Both numbers are sent to a wire for output to the BFM module and the code is displayed in Figure 50.

```

64     assign Xqout_real = signed_left_Real - signed_right_Real;
65     assign Xqout_imag = signed_left_Imag + signed_right_Imag;

```

Figure 50. Adder/Subtractor for Complex Multiplication Code Snippet

Two sets of complex numbers were tested through simulation. The input and output of $(1+.5j)(0.25+0.125j)$ and $(-1-0.5j)(1+j)$ is shown in Figure 51 as a simulation output.

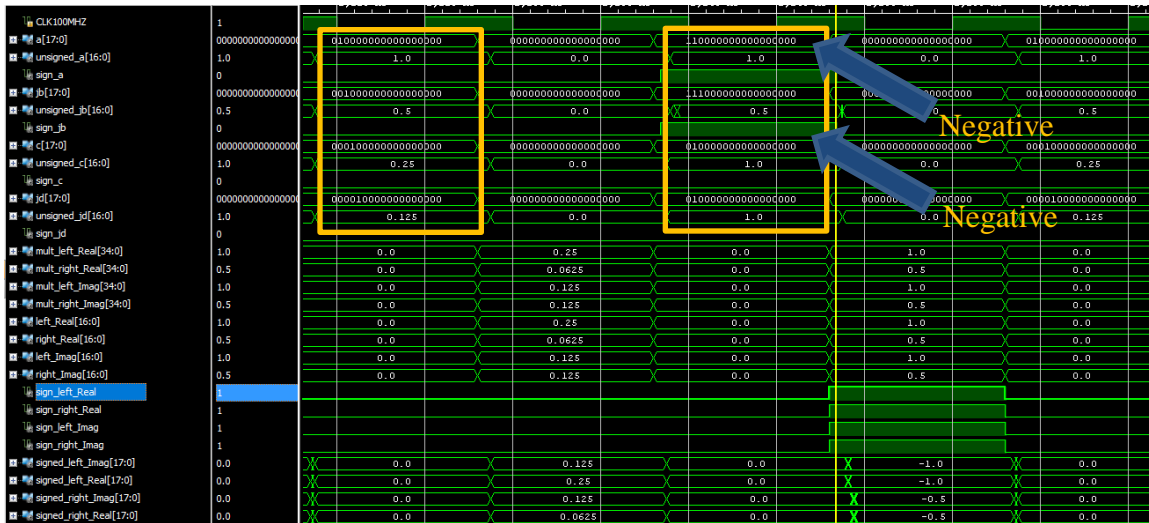


Figure 51. Complex Multiplication Module Testing

As mentioned previously, behavioral multiplication functions correctly on unsigned numbers. The seventeenth bit of data is the sign bit and must be stripped off. The circles within Figure 52 highlights the signal for the sign bit. Also, all negative numbers

must be two's complemented to return to their magnitude in preparation for multiplication. Inputs can be viewed against outputs in Figure 52. Point M2 demonstrates the magnitude.

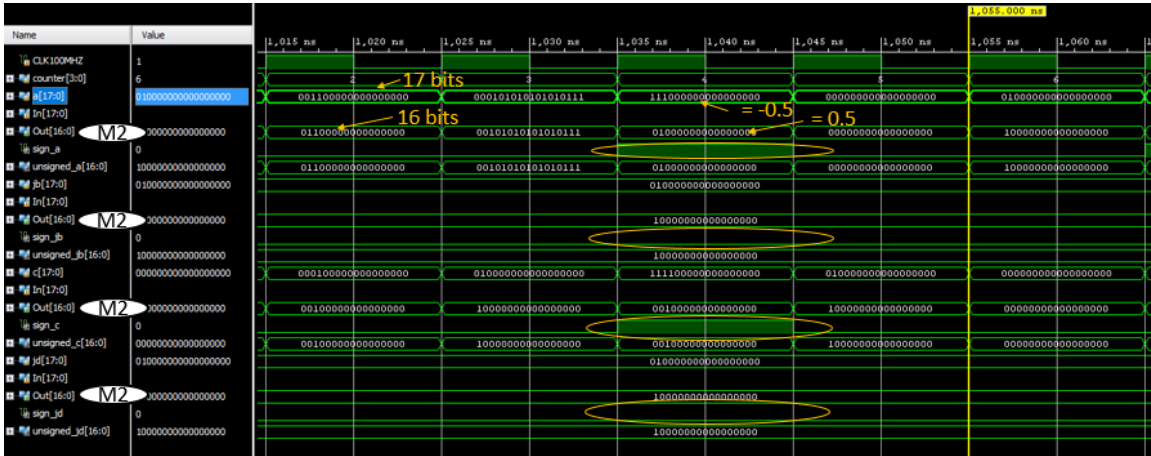


Figure 52. Conversion of Multiplier Inputs to Signed Magnitudes

Once all negative numbers are two's complemented, seen as test point M2, a behavioral multiplier multiplies the magnitudes. The multiplier produces a 34-bit unsigned number that is displayed as test points M3 and M4. The data we need are in the upper 17-bits of the data. The data was truncated to extract what was needed and the rest discarded. Test points M5 and M6 show the truncated data.

The original signs of the data were exclusive-or'd and can be seen at test point M1. The new sign was appended to the truncated multiplication output. If the sign indicates a negative number, the multiplication output is inverted and one bit is added. This is reflected as test points M7 and M8. Prior to being sent as an output to the adder / subtracter, test point M9 can be used to view the resultant value. The simulation labeled with applicable test points is displayed in Figure 53.

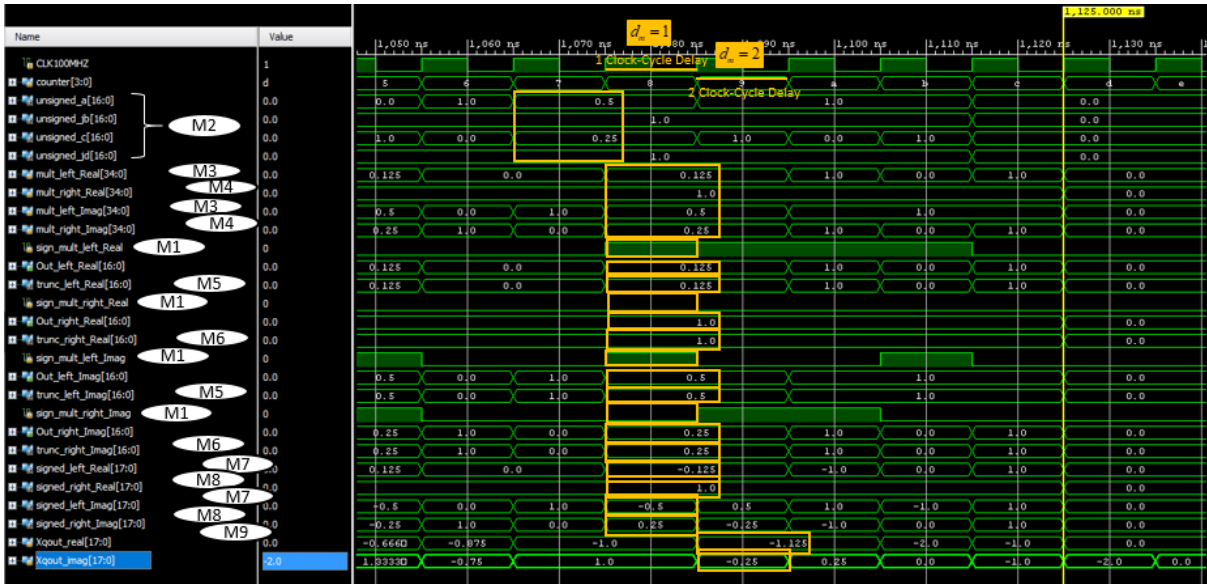


Figure 53. Multiplier Test Points. Fixed Point

d. Adder / Subtractor Component Sub-Component Test

The adder / subtractor combines the common results of the multiplication. The variables needed consist of two registers, *out_real* and *out_imag*, both 18 bits wide, and can be seen coded in Figure 54.

```

33      (* dont_touch = "true" *) reg [17:0] out_real;
34      (* dont_touch = "true" *) reg [17:0] out_imag;
35
36      //Complex Addition/Subtraction
37      //          Real    Imag
38      //(a+jb)+(c+jd) = a + c , j(b+d)
39      //(a+jb)-(c+jd) = a - c , j(b-d)
40      assign Xqout_real = out_real; //Real Component
41      assign Xqout_imag = out_imag; //Imag Component

```

Figure 54. Adder / Subtractor Variables Code Snippet

The adder / subtractor is the last component of the BFM. On the positive edge of every clock-cycle, a selection was made to either add or subtract. This selection is based on the value of *counter*. A digital high results in addition, while a digital low results in subtraction as seen in Figure 55.

```

43 always@(posedge clk)
44     begin
45         out_real <= (counterw==1'b1) ? TopIn_real + BottomIn_real : TopIn_real - BottomIn_real; //Real Component
46         out_imag <= (counterw==1'b1) ? TopIn_imag + BottomIn_imag : TopIn_imag - BottomIn_imag; //Imag Component
47     end

```

Figure 55. Adder / Subtractor Code Snippet

The simulation for the adder / subtractor is displayed in Figure 56. The top multiplexer input, seen as M7, was delayed by $d_m + 1$. The inputs were added on even clock cycles and subtracted on odd clock cycles, with the results being viewed at test point M9.

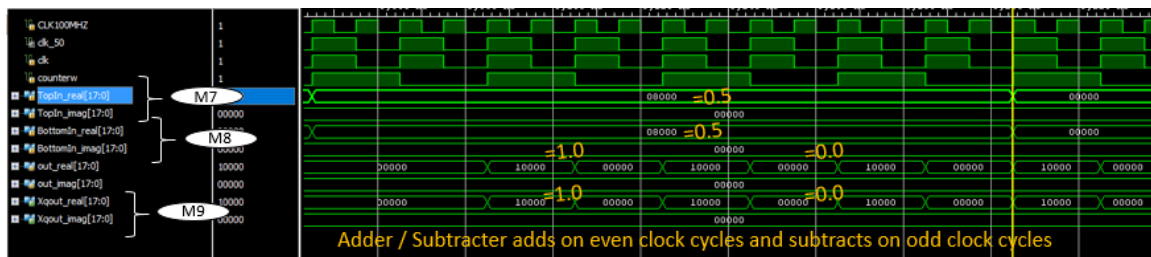


Figure 56. Adder / Subtractor Adds on Even Clock-Cycles

e. 2nd Stage Ping Pong Buffer Sub-Component Test

Four two-dimensional registers are utilized to store the real and imaginary components of the First-Stage Ping-Pong buffers. Variables *XqPing_Real*, *XqPing_Imag*, *XqPong_Real*, and *XqPong_Imag* are declared as 18-bit wide words. In addition, a 4-bit register *buffer_counter* was utilized to count. These can be seen in Figure 57.

```

58 (* dont_touch = "true" *) reg [17:0] XqPing_Real [7:0]; // register for Ping real component
59 (* dont_touch = "true" *) reg [17:0] XqPing_Imag [7:0]; // register for Ping imag component
60 (* dont_touch = "true" *) reg [17:0] XqPong_Real [7:0]; // register for Pong real component
61 (* dont_touch = "true" *) reg [17:0] XqPong_Imag [7:0]; // register for Pong imag component
62 (* dont_touch = "true" *) reg [3:0] buffer_counter;

```

Figure 57. First-Stage Ping-Pong Buffer Variables Code Snippet

The 4-bit *buffer_counter* was utilized to produce sixteen timeslots. Eight timeslots are needed for the First-Stage Ping buffer and eight timeslots are needed for the First-Stage

Pong buffer to fill according to $k = 0, \frac{N}{2}, 1, \frac{N}{2} + 1, 2, \frac{N}{2} + 2, 3, \frac{N}{2} + 3, 4, \frac{N}{2} + 4, \dots, N - 1$. Since $N = 8$, the fill order is $k = 0, 4, 1, 5, 2, 6, 3, 7$ and can be seen in Figures 58 and 59.

```

114      3'b0100: //Xq(4)
115      begin
116          XqPing_Real[0] <= XqIn_PingPong_Real;
117          XqPing_Imag[0] <= XqIn_PingPong_Imag;
118          buffer_counter = 4'd0;
119      end
120      3'b0101: //Xq(5)
121      begin
122          XqPing_Real[4] <= XqIn_PingPong_Real;
123          XqPing_Imag[4] <= XqIn_PingPong_Imag;
124          buffer_counter = 4'd1;
125      end
126      3'b0110: //Xq(6)
127      begin
128          XqPing_Real[1] <= XqIn_PingPong_Real;
129          XqPing_Imag[1] <= XqIn_PingPong_Imag;
130          buffer_counter = 4'd2;
131      end
132      3'b0111: //Xq(7)
133      begin
134          XqPing_Real[5] <= XqIn_PingPong_Real;
135          XqPing_Imag[5] <= XqIn_PingPong_Imag;
136          buffer_counter = 4'd3;
137      end
138      4'b1000: //Xq(8)
139      begin
140          XqPing_Real[2] <= XqIn_PingPong_Real;
141          XqPing_Imag[2] <= XqIn_PingPong_Imag;
142          buffer_counter = 4'd4;
143      end

```

Figure 58. First-Stage Ping Buffer Input Code Snippet

```

144      4'b1001: //Xq(9)
145      begin
146          XqPing_Real[6] <= XqIn_PingPong_Real;
147          XqPing_Imag[6] <= XqIn_PingPong_Imag;
148          buffer_counter = 4'd5;
149      end
150      4'b1010: //Xq(10)
151      begin
152          XqPing_Real[3] <= XqIn_PingPong_Real;
153          XqPing_Imag[3] <= XqIn_PingPong_Imag;
154          buffer_counter = 4'd6;
155      end
156      4'b1011: //Xq(11)
157      begin
158          XqPing_Real[7] <= XqIn_PingPong_Real;
159          XqPing_Imag[7] <= XqIn_PingPong_Imag;
160          buffer_counter = 4'd7;
161      end

```

Figure 59. First-Stage Ping Buffer Input Code Snippet (Continued)

Component level testing of the First-Stage Ping-Pong buffer demonstrates that inputs flow into the appropriate location at the desired time. The loading process of both the First-Stage Ping buffer and the First-Stage Pong buffer is annotated in Figure 60. The Pong buffer is filled prior to filling Ping buffer, which then alternates every eight clock-cycles. All subsequent stages of Ping-Pong buffer are identical and consistent with the $k = 0, \frac{N}{2}, 1, \frac{N}{2} + 1, 2, \frac{N}{2} + 2, 3, \frac{N}{2} + 3, 4, \frac{N}{2} + 4, \dots, N - 1$ loading scheme seen in Figure 59.

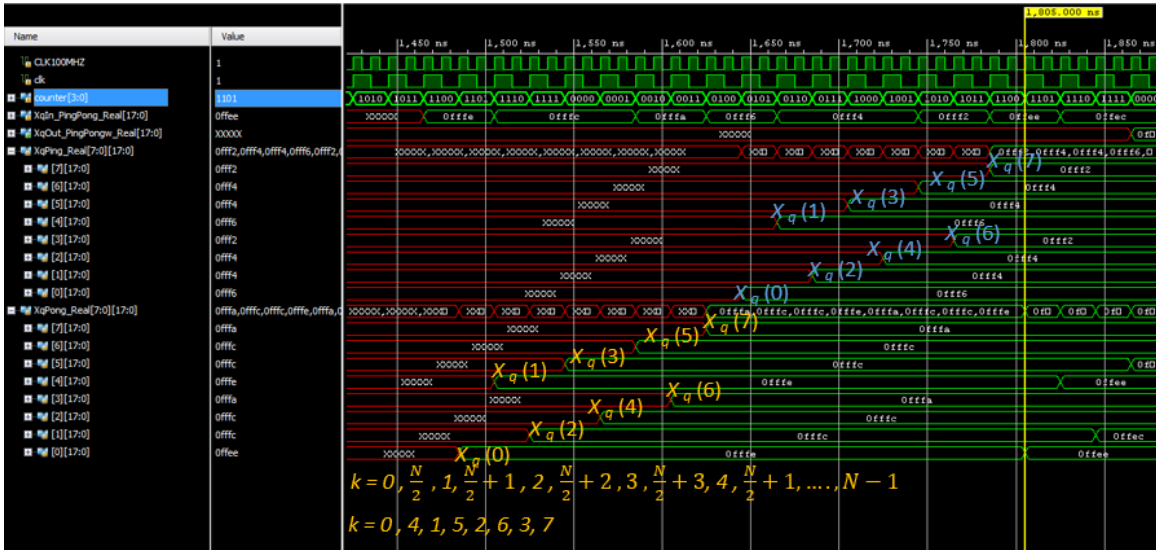


Figure 60. First-Stage Ping-Pong Buffer Simulation

The code of the FFT and simulations to verify operation at a component and sub-component level was described in this chapter. The integration of the FFT as a whole system is discussed in Chapter V.

V. END TO END TESTING / INTEGRATION TESTING

The operability of the Fast Fourier transform as a system is confirmed in this chapter. The four test vectors seen in Table 5 are utilized. How real $X_q(t)$ inputs into a FFT produce both real and imaginary outputs is shown in Table 5. Separate columns exist for real and imaginary portions of the complex number.

Table 5. Test Vector Input and Expected Output

Real Input	Real Output	Imaginary Output	Scaled Real Output	Scaled Imaginary Output
{1, 1, 1, 1, 1, 1, 1, 1}	{8, 0, 0, 0, 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, 0, 0, 0}
{1, 0, 0, 0, 0, 0, 0, 0}	{1, 1, 1, 1, 1, 1, 1, 1}	{0, 0, 0, 0, 0, 0, 0, 0}	$\{\frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}\}$	{0, 0, 0, 0, 0, 0, 0, 0}
{0, 0, 0, 0, 0, 0, 0, 0}	{0, 0, 0, 0, 0, 0, 0, 0}	{0, 0, 0, 0, 0, 0, 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}	{1, -0.707, 0, 0.707, -1, 0.707, 0, -0.707}	{0, -0.707, 1, -0.707, 0, 0.707, -1, 0.707}	$\{\frac{1}{8}, -\frac{0.707}{8}, 0, \frac{0.707}{8}, -\frac{1}{8}, \frac{0.707}{8}, 0, \frac{0.707}{8}\}$	$\{0, -\frac{0.707}{8}, \frac{1}{8}, -\frac{0.707}{8}, 0, \frac{0.707}{8}, -\frac{1}{8}, \frac{0.707}{8}\}$

Three major sections are present in this chapter. A higher level perspective that shows an analysis utilizing the constant geometry FFT architecture is provided in Section A. The detailed analysis of the first test vector in Table 5 is detailed in Section B. The flow of data is demonstrated through annotated simulations. The results of integration and a hypothesis as to why are captured in Section C.

A. ALL TEST VECTOR ANALYSIS

The generic FFT architecture utilized to implement the coding is being reinserted as Figure 61. Numbers in sequence multiply with each other, and numbers that are in parallel add where paths converge. Unlabeled arrowheads imply a multiplication by one. In addition, scaling is not displayed within the butterfly diagram.

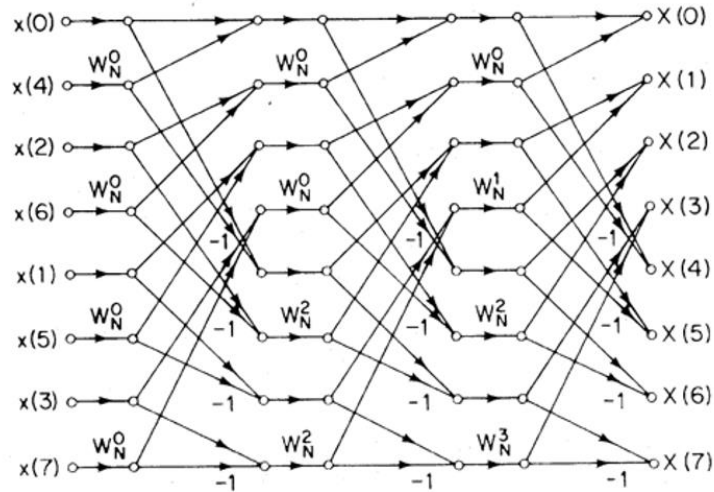


Figure 61. Generic Constant Geometric FFT. Adapted from [4].

Twiddle factors have been inserted in the form of their complex number equivalent. Starting with $\{1, 1, 1, 1, 1, 1, 1, 1\}$, after first stage processing $\{2, 2, 2, 2, 0, 0, 0, 0\}$ resulted. This was a result of the 1s on the top half of the FFT adding with the 1s on the bottom half of the FFT and alternating with the 1s of the top half of the FFT subtracting with the 1s of the bottom half of the FFT. Continuing with this trend, we find $\{4, 4, 0, 0, 0, 0, 0, 0\}$ outputted after the second stage. The third stage produced $\{8, 0, 0, 0, 0, 0, 0, 0\}$ after processing, which is scaled result of $\{1, 0, 0, 0, 0, 0, 0, 0\}$.

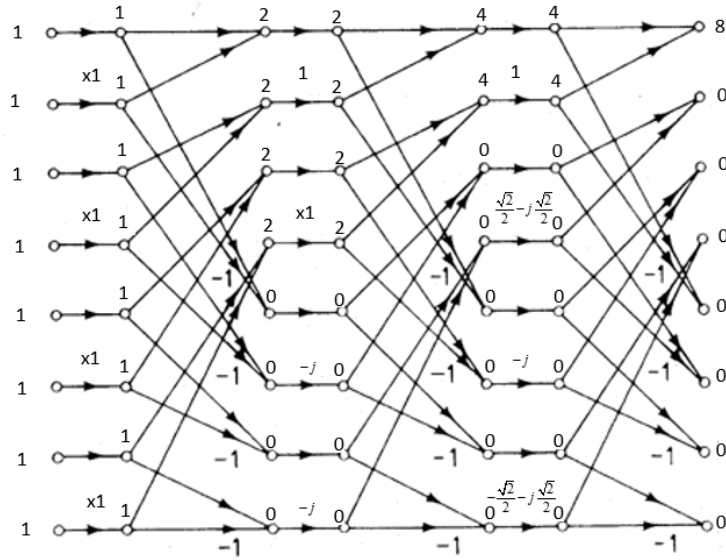


Figure 62. Test Vector-1 $\{1, 1, 1, 1, 1, 1, 1, 1\}$. Adapted from [4].

Test Vector-2 inputs $\{1, 0, 0, 0, 0, 0, 0, 0\}$. After the first stage processing, $\{1, 0, 0, 0, 1, 0, 0, 0\}$ resulted. Second stage processing produced $\{1, 0, 1, 0, 1, 0, 1, 0\}$. Third stage processing results in $\{1, 1, 1, 1, 1, 1, 1, 1\}$ and is scaled as $\{\frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}\}$. Test Vector-2 can be seen in Figure 63.

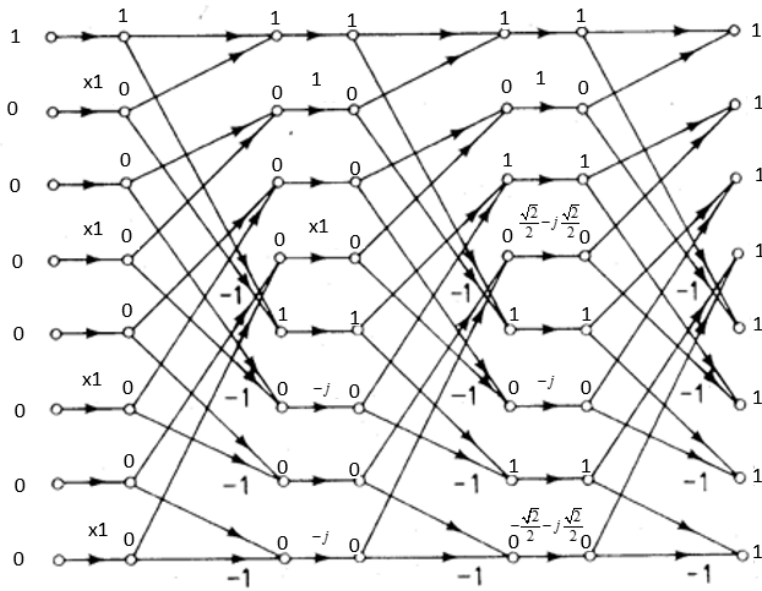


Figure 63. Test Vector-2 $\{1, 0, 0, 0, 0, 0, 0, 0\}$. Adapted from [4].

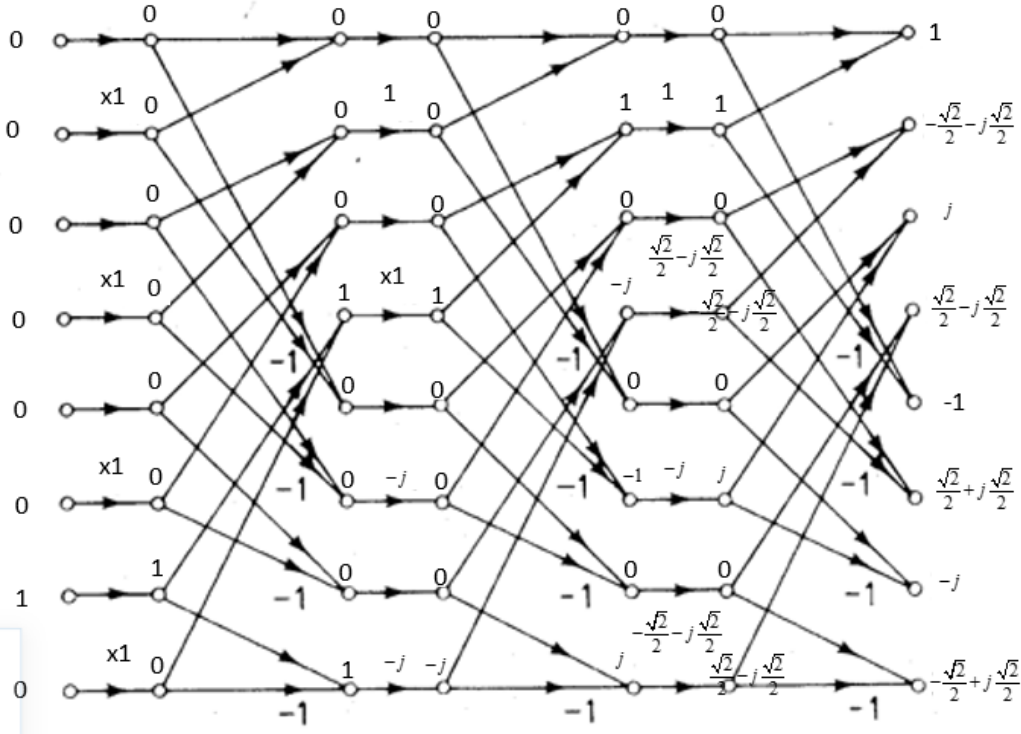


Figure 65. Test Vector-4 {0, 0, 0, 1, 0, 0, 0, 0} Displayed after Bit-Reversed to {0, 0, 0, 0, 0, 0, 1, 0}. Adapted from [4].

B. TEST VECTOR ANALYSIS

A detailed analysis of annotation simulations on Test Vector-1 is given in this section. This is done by looking at the simulation of each component of the FFT and following how the test data is processed throughout. Each BFM's require a counter initialization to zero. Each stage has been synchronized on an individual counter to allow for proper execution of the FFT at a system level.

1. Bit Reversed Ping Pong Buffer

Test Vector-1 was received as an input into the FFT algorithm. The test vector was driven from a test bench. In a real-life application, the signal is received from an analog-to-digital converter after receipt by an antenna, and the first half of the Bit-Reversed Ping-Pong buffer loads the data. The second half of the Bit-Reversed Ping Pong buffer will not yet have valid data as demonstrated in Figure 66.

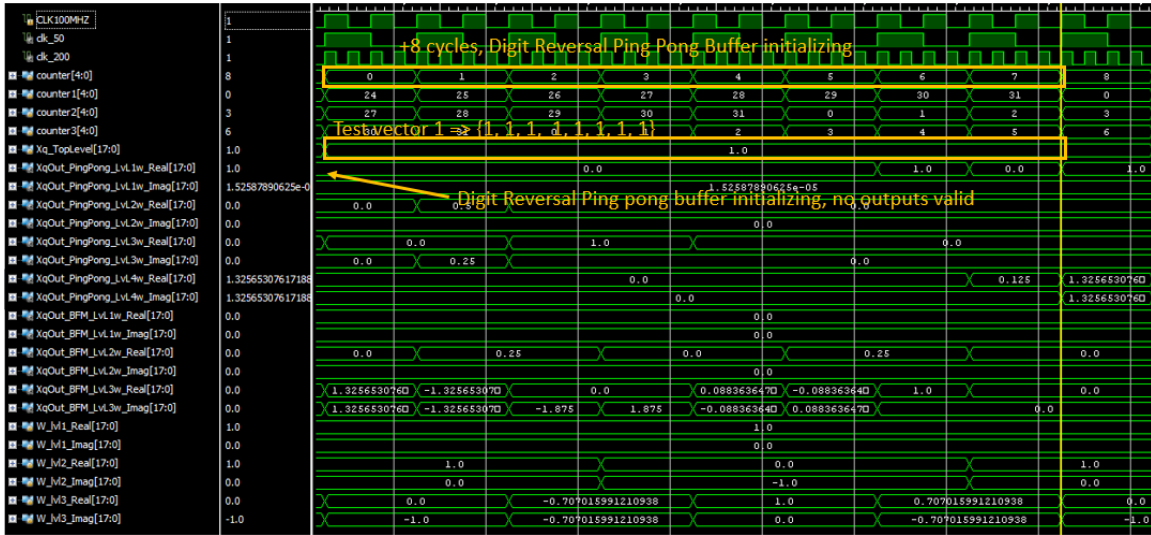


Figure 66. Bit-Reversed Ping-Pong Buffer Initializing

The Bit-Reversed Ping-Pong buffer outputted $\{1, 1, 1, 1, 1, 1, 1, 1\}$ as seen in Figure 67. Since this is a pipelined algorithm, the first stage BFM receives data while simultaneously processing data. Four clock-cycles are required for the BFM to completely process the data. The startup delay is equivalent to initialization of the BFM.

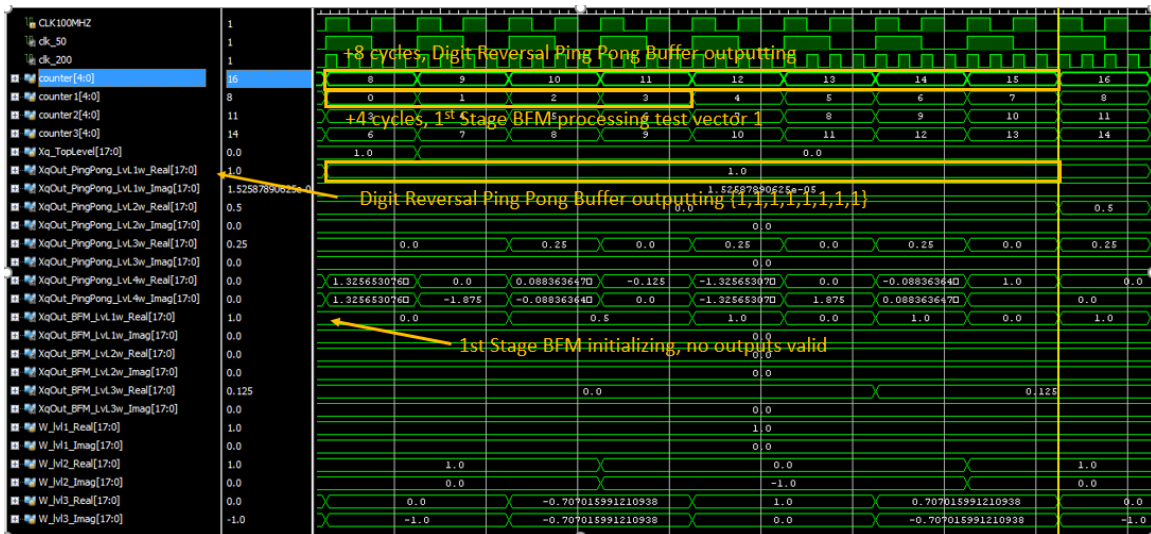


Figure 67. Bit-Reversed Ping-Pong Buffer Output

2. First-Stage BFM and Ping Pong Buffer

After a four clock-cycle initialization, the First-Stage BFM outputs {1, 1, 1, 1, 0, 0, 0, 0} as can be seen in Figure 68. These outputs were sent into the First-Stage Ping-Pong buffer. No valid outputs were present from the First-Stage Ping-Pong buffer until after initialization.

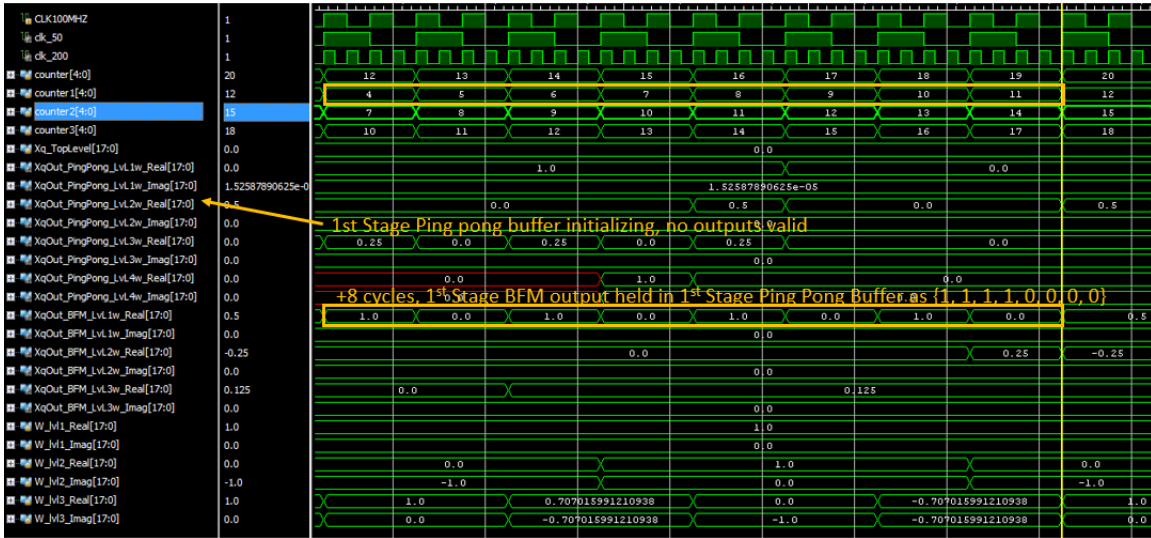


Figure 68. First-Stage BFM Output and First-Stage Ping-Pong Buffer Initialization

Outputs can be seen leaving the First-Stage Ping-Pong buffer in Figure 69. Also shown is the Second-Stage BFM initialization. Variable *counter2* drove the Second-Stage BFM; although, seen to be on count 16, the BFM interprets this as $16 \bmod(8)$, which is interpreted as zero.

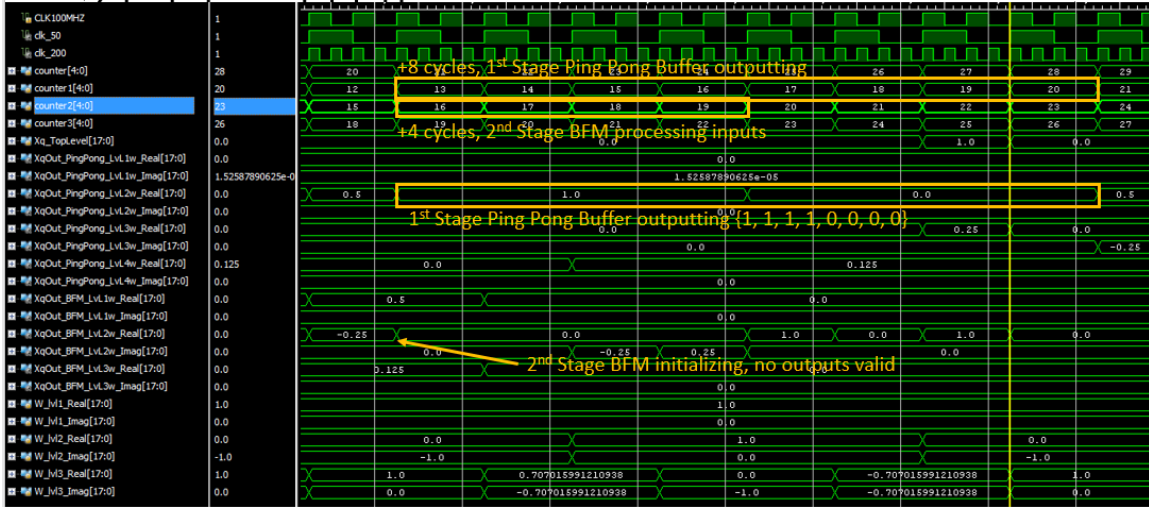


Figure 69. First-Stage Ping-Pong Buffer Outputting and Second-Stage BFM Initializing

3. Second-Stage BFM and Ping-Pong Buffer

Variable *counter2* was drove the Second-Stage BFM and Ping-Pong buffer. Starting at $20 \bmod 8$, which equates to four, the Second-Stage BFM outputted $\{1, 1, 0, 0, 0, 0, 0, 0\}$. Logically, values are released and rearranged as explained in Chapter III, Paragraph C to maintain a constant geometry FFT. See Figure 70 for a demonstration.

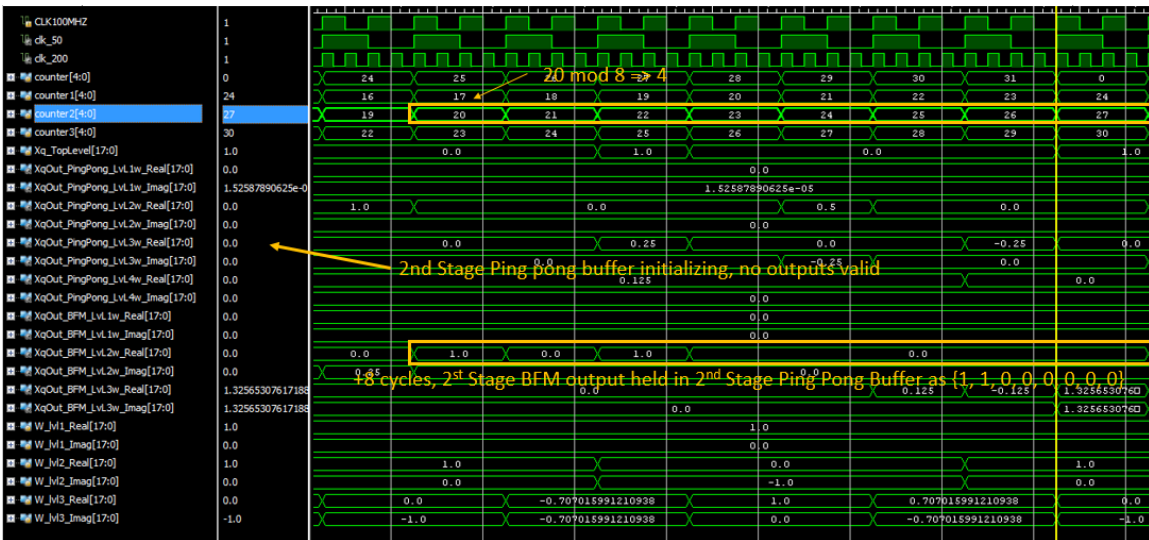


Figure 70. Second-Stage BFM Outputting and Second-Stage Ping-Pong Buffer Initializing

The Second-Stage Ping-Pong buffer outputted {1, 1, 0, 0, 0, 0, 0, 0}, and the Third-Stage BFM took four clock-cycles to initialize and process the values as displayed in Figure 71. Variable *counter3* was initialized to zero to synchronize with the Third-Stage BFM operation.

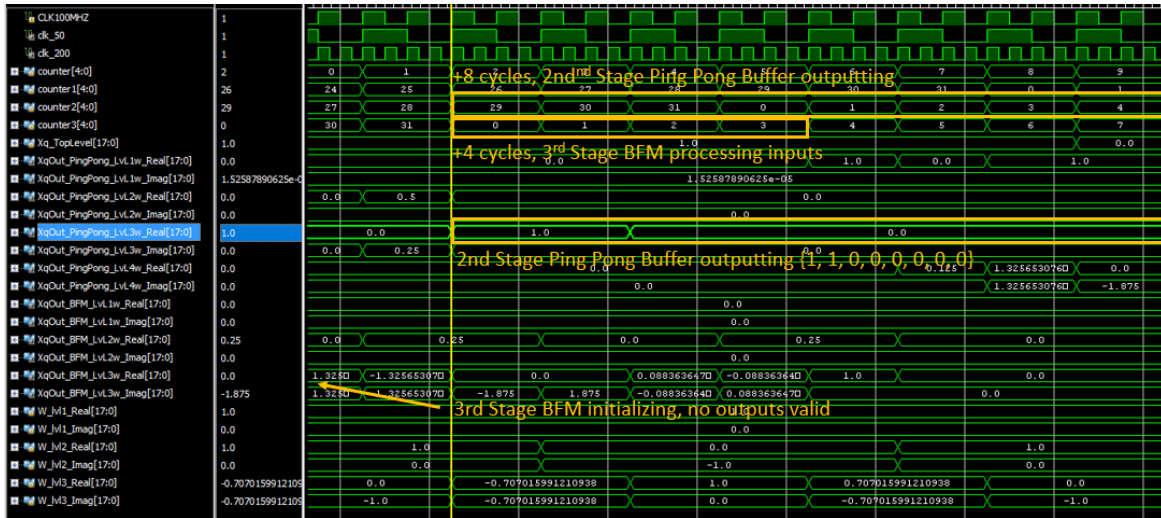


Figure 71. Second-Stage Ping-Pong Buffer Outputting and Third-Stage BFM Initializing

4. Third-Stage BFM and Ping-Pong Buffer

The Third-Stage BFM outputted {1, 0, 0, 0, 0, 0, 0, 0}. This is the expected result for Test Vector-1 and is demonstrated in Figure 72. Also note that this came four clock-cycles after the data was received into the BFM and twelve clock-cycles after this block of data was loaded into the Third-Stage Ping-Pong Buffer. Since the Forth-Stage Ping-Pong Buffer is loading its initial data, it does not output valid yet data in Figure 72

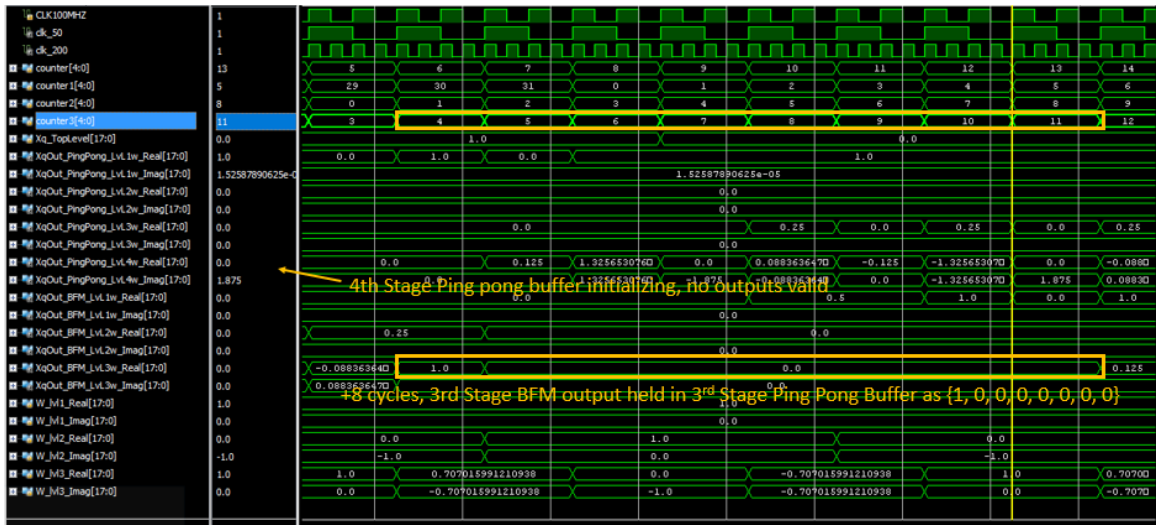


Figure 72. Third-Stage BFM Outputting and Third-Stage Ping-Pong Initializing

A test vector of $\{1, 1, 1, 1, 1, 1, 1, 1\}$ should have a scaled result of $\{1, 0, 0, 0, 0, 0, 0, 0\}$. Shown in Figure 73 the correct result is demonstrated for Test Vector-1 processed through the designed FFT.

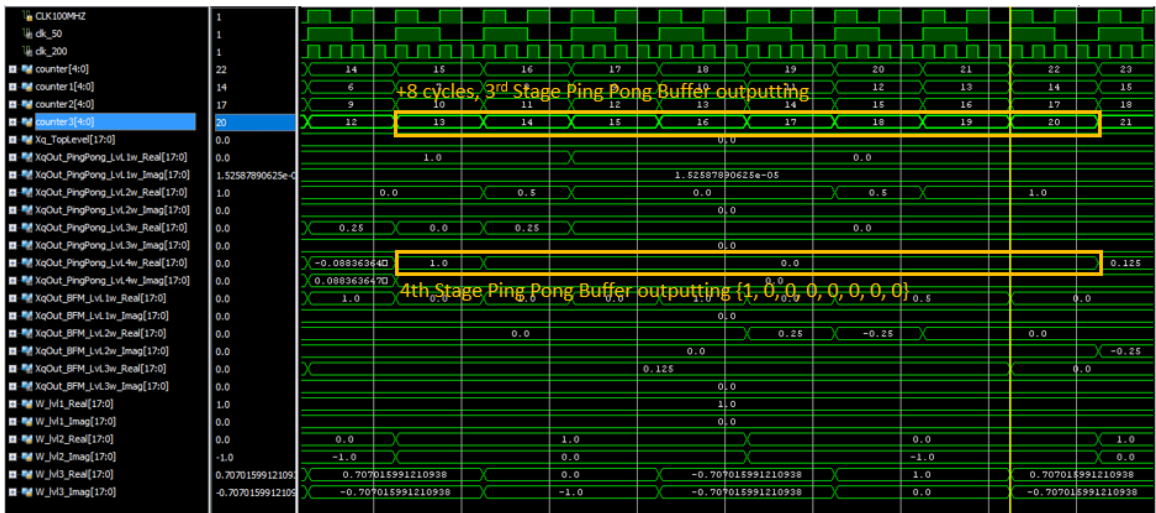


Figure 73. Third-Stage Ping-Pong Buffer With Final Scaled Result $\{1, 0, 0, 0, 0, 0, 0, 0\}$

The FFT was demonstrated successfully with Test Vector-1 in Chapter V. Timing was key to successful synchronization. Multiple counters were utilized to synchronize the stages of the FFT.

C. TIMING ERROR WHILE INTEGRATING

An analysis of Test Vector 3 proved unsuccessful. While individually analyzing the processing elements of the FFT proved successful, synchronizing the timing between the three levels of the BFM failed. Highlighted in Figure 74 are a few locations where incorrect timing can skew processing of data. This improperly processed data then replicates through the successive stages. With the error being localized to within a timing element, initialization of the BFM and setting the selectors on the multiplexer or adder / subtractor can be as simple as a one-bit inversion; however, isolating the exact bit has proven to be a challenge. A complete overhaul of the timing within the FFT is the recommended corrective action. This has the greatest chance of success while also producing a code that is usable by others.

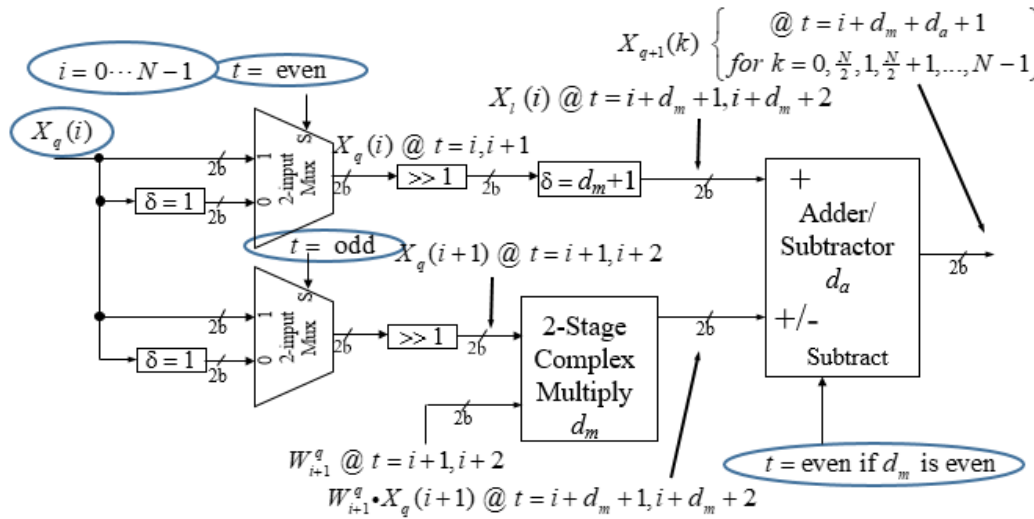


Figure 74. Highlighted Timing Variables within the BFM. Adapted from [15].

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

With 20 years of research into satellite development, in this thesis research we are closer to creating a rapidly deployable and reconfigurable signal processor that is space capable. Signal processing may be the basis for a mission element or an enabling component of a larger objective. It also has utility in every day communications.

A. ACADEMIC VALUE

An understanding of many academic areas was required for this research. A foundation in digital signal processing required a solid foundation in the Fast Fourier Transform theorems, principles, and mathematics. In addition, to conceptualize the final product, an understanding of the electronics needed to digitize an analog signal is required. With space being the intended application of the end product, knowledge of the space environment and its impact on electronics is also fundamental. A working knowledge of producing software code and an additional knowledge specifically in Verilog HDL was acquired throughout this research.

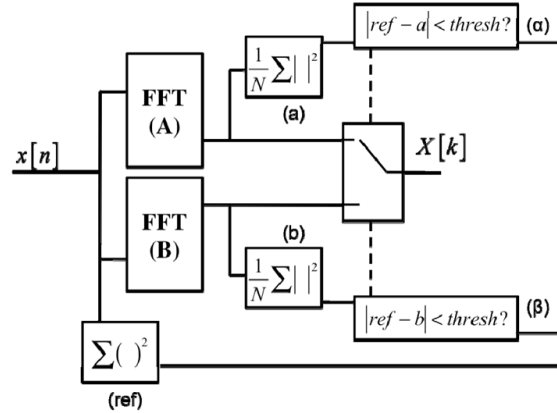
B. THESIS SUMMARY

The objective of this thesis was to give access to an open source signal processing algorithm that could eventually be implemented in an error detecting / error correcting algorithm for use within the space environment. Success was achieved in programming major portions of the highspeed pipelined FFT. The major modules needed to instantiate the FFT have been designed and tested. The FFT fails to integrate correctly. Timing and synchronization of data is the suspected cause. Once timing and synchronization is corrected, implementing Parseval's theorem will make this code space capable.

C. RECOMMENDATIONS FOR FUTURE WORK

Follow on work should consist of synchronizing the timing within the FFT and then utilization of the FFT to implement Parseval's theorem. Parseval's theorem makes this code space capable and is required before loading on to the FPGA for launch. Parseval's theorem is illustrated once more in Figure 75.

2NlogN Redundant FFT



$$\text{if } [|ref - a| < thresh] \text{ then } \{ X[k] = FFT_A \} \text{ else } \{ X[k] = FFT_B \}$$

Figure 75. Parseval's Theorem Implementation Illustration. Source: [2].

In addition, capability can be added to the FFT by increasing the sample size from $N = 8$ to $N = 32$. This will allow a more granular level of processing. This FFT will also need to be loaded onto NPSat-1's FPGA with operability testing to follow.

D. CLOSING REMARKS

In clocked systems, timing and synchronization of data is mandatory. Processing elements must have the right data at the right time to produce the correct results. When debugging software, have a plan and work in a systematic process. This research project was built on 20 years of work. A literature review was helpful and there are many documents to educate oneself.

APPENDIX A. FILE STRUCTURE

top_level (top_level.v)
 clk_mod – clk_module (clk_module.xci)
 clk_module (clk_module.v)
 inst – clk_module_clk_wiz (clk_module_clk_wiz.v)
PingPong_SwitchLevel1 – pingpong (pingpong.v)
BFMlevelOne – Radix2_BFM (Radix2_BFM.v)
 two2oneMux_Top – two2oneMux (two2oneMux.v)
 two2oneMux_Bottom – two2oneMux (two2oneMux.v)
 ComplexMultiply2Stage – multiply_complex (multiply_complex.v)
 Unsigned_multiply – mult18x18 (mult18x18.v)
 twoC_leftReal – twoCompRedo (twoCompRedo.v)
 twoC_rightReal – twoCompRedo (twoCompRedo.v)
 twoC_leftImag – twoCompRedo (twoCompRedo.v)
 twoC_rightImag – twoCompRedo (twoCompRedo.v)
 insta – twoComp (twoComp.v)
 instb – twoComp (twoComp.v)
 instc – twoComp (twoComp.v)
 instd – twoComp (twoComp.v)
 AdderandSubtractor1Stage – addsub18x18 (addsub18x18.v)
PingPong_SwitchLevel2 – pingpongK (pingpongK.v)
BFMlevelTwo – Radix2_BFM (Radix2_BFM.v)
 ...
 ... as shown above
 ...
PingPong_SwitchLevel3 – pingpongK (pingpongK.v)
BFMlevelThree – Radix2_BFM (Radix2_BFM.v)
 ...
 ... as shown above
 ...
PingPong_SwitchLevel4 – pingpongK (pingpongK.v)

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. SOURCE CODE

*****TOP_LEVEL.V*****

```
1 module top_level(  
2     input wire CLK100MHZ  
3 );  
4  
5 (* dont_touch = "true" *)wire clk_50, clk_200, clk_400;  
6 (* dont_touch = "true" *)reg [4:0] counter, counter1, counter2, counter3;  
7 (* dont_touch = "true" *)reg [17:0] driver;  
8 (* dont_touch = "true" *)reg INIT, reset;  
9 (* dont_touch = "true" *)reg [17:0] Xq_TopLevel;  
10 (* dont_touch = "true" *)wire [17:0] XqOut_PingPong_LvL1w_Real, XqOut_PingPong_LvL1w_Imag;  
11 (* dont_touch = "true" *)wire [17:0] XqOut_PingPong_LvL2w_Real, XqOut_PingPong_LvL2w_Imag;  
12 (* dont_touch = "true" *)wire [17:0] XqOut_PingPong_LvL3w_Real, XqOut_PingPong_LvL3w_Imag;  
13 (* dont_touch = "true" *)wire [17:0] XqOut_PingPong_LvL4w_Real, XqOut_PingPong_LvL4w_Imag;  
14 (* dont_touch = "true" *)wire [17:0] XqOut_BFM_LvL1w_Real, XqOut_BFM_LvL1w_Imag;  
15 (* dont_touch = "true" *)wire [17:0] XqOut_BFM_LvL2w_Real, XqOut_BFM_LvL2w_Imag;  
16 (* dont_touch = "true" *)wire [17:0] XqOut_BFM_LvL3w_Real, XqOut_BFM_LvL3w_Imag;  
17 (* dont_touch = "true" *)reg [17:0] W_lv11_Real, W_lv11_Imag;  
18 (* dont_touch = "true" *)reg [17:0] W_lv12_Real, W_lv12_Imag;  
19 (* dont_touch = "true" *)reg [17:0] W_lv13_Real, W_lv13_Imag;  
20  
21 //-----  
22 // The following must be inserted into your Verilog file for this  
23 // core to be instantiated. Change the instance name and port connections  
24 // (in parentheses) to your own signal names.  
25  
26 //----- Clock INSTANTIATION -----  
27 clk_module clk_mod  
28 (  
29     // Clock in ports  
30     .clk_in1(CLK100MHZ), // input clk  
31     // Clock out ports  
32     .clk_50(clk_50), // output clk_50  
33     .clk_200(clk_200), // output clk_200  
34     .clk_400(clk_400), // output clk_400  
35     // Status and control signals  
36     .reset(reset),  
37     .locked(locked)  
38 );  
39 // ----- End Clock INSTANTIATION -----  
40  
41 // ----- Begin Ping-pong switch Level 1 -----  
42 pingpong PingPong_SwitchLevel1  
43 (  
44     .XqIn_PingPong_Real(Xq_TopLevel),  
45     .XqIn_PingPong_Imag(1),  
46     .clk(clk_50),  
47     .counter(counter),  
48     .XqOut_PingPong_Real(XqOut_PingPong_LvL1w_Real),  
49     .XqOut_PingPong_Imag(XqOut_PingPong_LvL1w_Imag)  
50 );  
51 // ----- End Ping-pong switch Level 1 -----  
52  
53 // ----- Begin Radix2_BFM Level Q = 1 -----  
54 Radix2_BFM BFMlevelOne  
55 (  
56     .Xq_Real(XqOut_PingPong_LvL1w_Real),  
57     .Xq_Imag(XqOut_PingPong_LvL1w_Imag),  
58     .clk(clk_50),  
59     .W_Real(W_lv11_Real),  
60     .W_Imag(W_lv11_Imag),  
61     .counterw(counter1),  
62     .Radix2_BFMOut_Real(XqOut_BFM_LvL1w_Real),  
63     .Radix2_BFMOut_Imag(XqOut_BFM_LvL1w_Imag)  
64 );  
65 // ----- End Radix2_BFM Level Q = 1 -----  
66  
67 // ----- Begin Ping-pongK switch Level 2 -----  
68 pingpongK PingPong_SwitchLevel2  
69 (  
70     .XqIn_PingPong_Real(XqOut_BFM_LvL1w_Real), // input Xq real component  
71     .XqIn_PingPong_Imag(XqOut_BFM_LvL1w_Imag), // input Xq imag component  
72     .clk(clk_50), // input clock  
73     .counter(counter1), // input counter  
74     .XqOut_PingPong_Real(XqOut_PingPong_LvL2w_Real), // output Xq real component on a wire  
75     .XqOut_PingPong_Imag(XqOut_PingPong_LvL2w_Imag) // output Xq imag component on a wire  
76 );  
77 // ----- End Ping-pongK switch Level 2 -----  
78  
79 // ----- Begin Radix2_BFM Level Q = 2 -----  
80 Radix2_BFM BFMlevelTwo  
81 (  
82     .Xq_Real(XqOut_PingPong_LvL2w_Real),  
83     .Xq_Imag(XqOut_PingPong_LvL2w_Imag),  
84     .clk(clk_50),  
85     .W_Real(W_lv12_Real),  
86     .W_Imag(W_lv12_Imag),  
87     .counterw(counter2),  
88     .Radix2_BFMOut_Real(XqOut_BFM_LvL2w_Real),  
89     .Radix2_BFMOut_Imag(XqOut_BFM_LvL2w_Imag)  
90 );
```

```

93 // ----- Begin Ping-pong switch Level 3 -----
94 pingpongK PingPong_SwitchLevel3
95 (
96   .XqIn_PingPong_Real(XqOut_BFM_LvL3w_Real), // input Xq real component
97   .XqIn_PingPong_Imag(XqOut_BFM_LvL3w_Imag), // input Xq imag component
98   .clk(clk_50), // input clock
99   .counter(counter2), // input counter
100   .XqOut_PingPongw_Real(XqOut_PingPong_LvL3w_Real), // output Xq real component on a wire
101   .XqOut_PingPongw_Imag(XqOut_PingPong_LvL3w_Imag) // output Xq imag component on a wire
102 );
103 // ----- End Ping-pong switch Level 3 -----
104 // ----- Begin Radix2_BFM Level Q = 3 -----
105 Radix2_BFM_BFMLevelThree
106 (
107   .Xq_Real(XqOut_PingPong_LvL3w_Real),
108   .Xq_Imag(XqOut_PingPong_LvL3w_Imag),
109   .clk(clk_50),
110   .W_Real(W_LvL3_Real),
111   .W_Imag(W_LvL3_Imag),
112   .counterw(counter3),
113   .Radix2_BFMOut_Real(XqOut_BFM_LvL3w_Real),
114   .Radix2_BFMOut_Imag(XqOut_BFM_LvL3w_Imag)
115 );
116 // ----- Begin Radix2_BFM Level Q = 3 -----
117 // ----- Begin Ping-pong switch Level 4 -----
118 pingpongK PingPong_SwitchLevel4
119 (
120   .XqIn_PingPong_Real(XqOut_BFM_LvL3w_Real), // input Xq real component
121   .XqIn_PingPong_Imag(XqOut_BFM_LvL3w_Imag), // input Xq imag component
122   .clk(clk_50), // input clock
123   .counter(counter3), // input counter
124   .XqOut_PingPongw_Real(XqOut_PingPong_LvL4w_Real), // output Xq real component on a wire
125   .XqOut_PingPongw_Imag(XqOut_PingPong_LvL4w_Imag) // output Xq imag component on a wire
126 );
127 // ----- End Ping-pong switch Level 4 -----
128 // ----- End Ping-pong switch Level 4 -----
129 // Initial block (Only executed once when simulation start)
130 initial begin
131   counter = -1;
132   counter1 = -9;
133   counter2 = -6;
134   counter3 = -3;
135   INIT = 0;
136   reset = 0;
137   Xq_TopLevel = 18'b010000000000000000;
138   W_LvL1_Real = 1;
139   W_LvL1_Imag = 1;
140   W_LvL2_Real = 1;
141   W_LvL2_Imag = 1;
142   W_LvL3_Real = 1;
143   W_LvL3_Imag = 1;
144   W_LvL4_Real = 1;
145   W_LvL4_Imag = 1;
146 end
147 // Statements below will be executed on every clock rising edge
148 always@(posedge clk_50)
149 begin
150   case (counter[4:0])
151     5'b00000: //Xq(0)
152     begin
153       Xq_TopLevel <= 18'b010000000000000000; // 1
154     end
155     5'b00001: //Xq(1)
156     begin
157       Xq_TopLevel <= 18'b010000000000000000; // 1
158     end
159     5'b00010: //Xq(2)
160     begin
161       Xq_TopLevel <= 18'b010000000000000000; // 1
162     end
163     5'b00011: //Xq(3)
164     begin
165       Xq_TopLevel <= 18'b010000000000000000; // 1
166     end
167     5'b00100: //Xq(4)
168     begin
169       Xq_TopLevel <= 18'b010000000000000000; // 1
170     end
171     5'b00101: //Xq(5)
172     begin
173       Xq_TopLevel <= 18'b010000000000000000; // 1
174     end
175     5'b00110: //Xq(6)
176     begin
177       Xq_TopLevel <= 18'b010000000000000000; // 1
178     end
179   end
180

```

```

181 5'b00111: //Xq(7)
182     begin
183         Xq_TopLevel <= 18'b010000000000000000; // 1
184     end
185 5'b01000: //Xq(8)
186     begin
187         Xq_TopLevel <= 18'b000000000000000000; // 0
188     end
189 5'b01001: //Xq(9)
190     begin
191         Xq_TopLevel <= 18'b000000000000000000; // 0
192     end
193 5'b01010: //Xq(10)
194     begin
195         Xq_TopLevel <= 18'b000000000000000000; // 0
196     end
197 5'b01011: //Xq(11)
198     begin
199         Xq_TopLevel <= 18'b000000000000000000; // 0
200     end
201 5'b01100: //Xq(12)
202     begin
203         Xq_TopLevel <= 18'b000000000000000000; // 0
204     end
205 5'b01101: //Xq(13)
206     begin
207         Xq_TopLevel <= 18'b000000000000000000; // 0
208     end
209 5'b01111: //Xq(15)
210     begin
211         Xq_TopLevel <= 18'b000000000000000000; // 0
212     end
213 5'b10000: //Xq(16)
214     begin
215         Xq_TopLevel <= 18'b000000000000000000; // 0
216     end
217 5'b10001: //Xq(17)
218     begin
219         Xq_TopLevel <= 18'b000000000000000000; // 0
220     end
221 5'b10010: //Xq(17)
222     begin
223         Xq_TopLevel <= 18'b000000000000000000; // 0
224     end
225 5'b10011: //Xq(18)
226     begin
227         Xq_TopLevel <= 18'b000000000000000000; // 0
228     end
229 5'b10011: //Xq(19)
230     begin
231         Xq_TopLevel <= 18'b000000000000000000; // 0
232     end
233 5'b10100: //Xq(20)
234     begin
235         Xq_TopLevel <= 18'b000000000000000000; // 0
236     end
237 5'b10101: //Xq(21)
238     begin
239         Xq_TopLevel <= 18'b000000000000000000; // 0
240     end
241 5'b10110: //Xq(22)
242     begin
243         Xq_TopLevel <= 18'b000000000000000000; // 0
244     end
245 5'b10111: //Xq(23)
246     begin
247         Xq_TopLevel <= 18'b000000000000000000; // 0
248     end
249 5'b11000: //Xq(24)
250     begin
251         Xq_TopLevel <= 18'b000000000000000000; // 0
252     end
253 5'b11001: //Xq(25)
254     begin
255         Xq_TopLevel <= 18'b000000000000000000; // 0
256     end
257 5'b11010: //Xq(26)
258     begin
259         Xq_TopLevel <= 18'b010000000000000000; // 1
260     end
261 5'b11011: //Xq(27)
262     begin
263         Xq_TopLevel <= 18'b000000000000000000; // 0
264     end
265 5'b11100: //Xq(28)
266     begin
267         Xq_TopLevel <= 18'b000000000000000000; // 0
268     end
269 5'b11101: //Xq(29)
270     begin

```

```

271 |         Xq_TopLevel <= 18'b00000000000000000000 // 0
272 |     end
273 |     5'b11110: //Xq(30)
274 |     begin
275 |         Xq_TopLevel <= 18'b00000000000000000000 // 0
276 |     end
277 |     5'b11111: //Xq(31)
278 |     begin
279 |         Xq_TopLevel <= 18'b01000000000000000000 // 1
280 |     end
281 | endcase
282 |
283 | // Level 1 Twiddle
284 | case (counter1[2:0])
285 | 3'b000: //Xq(0)
286 |     begin
287 |         W_lv11_Real <= 18'b01000000000000000000 // 1
288 |         W_lv11_Imag <= 18'b00000000000000000000 // -j0
289 |     end
290 | 3'b001: //Xq(1)
291 |     begin
292 |         W_lv11_Real <= 18'b01000000000000000000 // 1
293 |         W_lv11_Imag <= 18'b00000000000000000000 // -j0
294 |     end
295 | 3'b010: //Xq(2)
296 |     begin
297 |         W_lv11_Real <= 18'b01000000000000000000 // 1
298 |         W_lv11_Imag <= 18'b00000000000000000000 // -j0
299 |     end
300 | 3'b011: //Xq(3)
301 |     begin
302 |         W_lv11_Real <= 18'b01000000000000000000 // 1
303 |         W_lv11_Imag <= 18'b00000000000000000000 // -j0
304 |     end
305 | 3'b100: //Xq(4)
306 |     begin
307 |         W_lv11_Real <= 18'b01000000000000000000 // 1
308 |         W_lv11_Imag <= 18'b00000000000000000000 // -j0
309 |     end
310 | 3'b101: //Xq(5)
311 |     begin
312 |         W_lv11_Real <= 18'b01000000000000000000 // 1
313 |         W_lv11_Imag <= 18'b00000000000000000000 // -j0
314 |     end
315 | 3'b110: //Xq(6)
316 |     begin
317 |         W_lv11_Real <= 18'b01000000000000000000 // 1
318 |         W_lv11_Imag <= 18'b00000000000000000000 // -j0
319 |     end
320 | 3'b111: //Xq(7)
321 |     begin
322 |         W_lv11_Real <= 18'b01000000000000000000 // 1
323 |         W_lv11_Imag <= 18'b00000000000000000000 // -j0
324 |     end
325 | endcase
326 |
327 | // Level 2 Twiddle
328 | case (counter2[3:0])
329 | 4'b0000:
330 |     begin
331 |         W_lv12_Real <= 18'b00000000000000000000 // 0-----
332 |         W_lv12_Imag <= 18'b10000000000000000000 // -j1-----
333 |     end
334 | 4'b0001:
335 |     begin
336 |         W_lv12_Real <= 18'b01000000000000000000 // 1-----
337 |         W_lv12_Imag <= 18'b00000000000000000000 // -j0-----
338 |     end
339 | 4'b0010:
340 |     begin
341 |         W_lv12_Real <= 18'b01000000000000000000 // 1-----
342 |         W_lv12_Imag <= 18'b00000000000000000000 // -j0-----
343 |     end
344 | 4'b0011:
345 |     begin
346 |         W_lv12_Real <= 18'b01000000000000000000 // 1-----
347 |         W_lv12_Imag <= 18'b00000000000000000000 // -j0-----
348 |     end
349 | 4'b0100:
350 |     begin
351 |         W_lv12_Real <= 18'b01000000000000000000 // 1-----
352 |         W_lv12_Imag <= 18'b00000000000000000000 // -j0-----
353 |     end
354 | 4'b0101:
355 |     begin
356 |         W_lv12_Real <= 18'b00000000000000000000 // 0-----
357 |         W_lv12_Imag <= 18'b10000000000000000000 // -j1-----
358 |     end
359 | 4'b0110:
360 |     begin

```

```

361 W_lv12_Real <= 18'b0000000000000000; // 0-----
362 W_lv12_Imag <= 18'b1000000000000000; // -j1-----
363 end
364 4'b0111:
365 begin
366 W_lv12_Real <= 18'b0000000000000000; // 0-----
367 W_lv12_Imag <= 18'b1000000000000000; // -j1-----
368 end
369 4'b1000:
370 begin
371 W_lv12_Real <= 18'b0000000000000000; // 0-----
372 W_lv12_Imag <= 18'b1000000000000000; // -j1-----
373 end
374 4'b1001:
375 begin
376 W_lv12_Real <= 18'b0100000000000000; // 1-----
377 W_lv12_Imag <= 18'b0000000000000000; // -j0-----
378 end
379 4'b1010:
380 begin
381 W_lv12_Real <= 18'b0100000000000000; // 1-----
382 W_lv12_Imag <= 18'b0000000000000000; // -j0-----
383 end
384 4'b1011:
385 begin
386 W_lv12_Real <= 18'b0100000000000000; // 1-----
387 W_lv12_Real <= 18'b0100000000000000; // 1-----
388 W_lv12_Imag <= 18'b0000000000000000; // -j0-----
389 end
390 4'b1101:
391 begin
392 W_lv12_Real <= 18'b0000000000000000; // 0-----
393 W_lv12_Imag <= 18'b1000000000000000; // -j1-----
394 end
395 4'b1110:
396 begin
397 W_lv12_Real <= 18'b0000000000000000; // 0-----
398 W_lv12_Imag <= 18'b1000000000000000; // -j1-----
399 end
400 4'b1111:
401 begin
402 W_lv12_Real <= 18'b0000000000000000; // 0-----
403 W_lv12_Imag <= 18'b1000000000000000; // -j1-----
404 end
405 4'b1111:
406 begin
407 W_lv12_Real <= 18'b0000000000000000; // 0-----
408 W_lv12_Imag <= 18'b1000000000000000; // -j1-----
409 end
410 endcase
411 // Level 3 Twiddle -----
412 case (counter3[3:0])
413 4'b0000:
414 begin
415 W_lv13_Real <= 18'b1010010110000000; // -sqrt(2)/2
416 W_lv13_Imag <= 18'b1010010110000000; // -sqrt(2)/2
417 end
418 4'b0001:
419 begin
420 W_lv13_Real <= 18'b0100000000000000; // 1
421 W_lv13_Imag <= 18'b0000000000000000; // -j0
422 end
423 4'b0010:
424 begin
425 W_lv13_Real <= 18'b0100000000000000; // 1
426 W_lv13_Imag <= 18'b0000000000000000; // -j0
427 end
428 4'b0011:
429 begin
430 W_lv13_Real <= 18'b0010110100111111; // sqrt(2)/2
431 W_lv13_Imag <= 18'b1101001011000000; // -sqrt(2)/2
432 end
433 4'b0100:
434 begin
435 W_lv13_Real <= 18'b0010110100111111; // sqrt(2)/2
436 W_lv13_Imag <= 18'b1101001011000000; // -sqrt(2)/2
437 end
438 4'b0101:
439 begin
440 W_lv13_Real <= 18'b0000000000000000; // 0
441 W_lv13_Imag <= 18'b1000000000000000; // -j1
442 end
443 4'b0110:
444 begin
445 W_lv13_Real <= 18'b0000000000000000; // 0
446 W_lv13_Imag <= 18'b1000000000000000; // -j1
447 end
448 4'b0111:
449 begin
450 W_lv13_Real <= 18'b1101001011000000; // -sqrt(2)/2

```

```

451     W_lv13_Imag <= 18'b110100101100000001; // -sqrt(2)/2
452     end
453 4'b1000:
454     begin
455     W_lv13_Real <= 18'b110100101100000001; // -sqrt(2)/2
456     W_lv13_Imag <= 18'b110100101100000001; // -sqrt(2)/2
457     end
458 4'b1001:
459     begin
460     W_lv13_Real <= 18'b100000000000000000; // 1
461     W_lv13_Imag <= 18'b000000000000000000; // -j0
462     end
463 4'b1010:
464     begin
465     W_lv13_Real <= 18'b010000000000000000; // 1
466     W_lv13_Imag <= 18'b000000000000000000; // -j0
467     end
468 4'b1011:
469     begin
470     W_lv13_Real <= 18'b001011010011111111; // sqrt(2)/2
471     W_lv13_Imag <= 18'b110100101100000001; // -jsqrt(2)/2
472     end
473 4'b1100:
474     begin
475     W_lv13_Real <= 18'b001011010011111111; // sqrt(2)/2
476     W_lv13_Imag <= 18'b110100101100000001; // -jsqrt(2)/2
477     W_lv13_Imag <= 18'b110000000000000000; // -j1
478     end
479 4'b1101:
480     begin
481     W_lv13_Real <= 18'b000000000000000000; // 0
482     W_lv13_Imag <= 18'b110000000000000000; // -j1
483     end
484 4'b1110:
485     begin
486     W_lv13_Real <= 18'b110100101100000001; // -sqrt(2)/2
487     W_lv13_Imag <= 18'b110100101100000001; // -jsqrt(2)/2
488     end
489 4'b1111:
490     begin
491     W_lv13_Real <= 18'b110100101100000001; // -sqrt(2)/2
492     W_lv13_Imag <= 18'b110100101100000001; // -jsqrt(2)/2
493     end
494     endcase
495 counter <= counter + 1;
496 counter1 <= counter1 + 1;
497 counter2 <= counter2 + 1;
498 counter3 <= counter3 + 1;
499 end
500 endmodule

```

*****CLK_MODULE.V*****
 *****© Copyright 2008 – 2013 Xilinx, Inc. All rights reserved
 [16]*****

```

1 // file: clk_module.v
2 //
3 // (c) Copyright 2008 - 2013 Xilinx, Inc. All rights reserved.
4 //
5 // This file contains confidential and proprietary information
6 // of Xilinx, Inc. and is protected under U.S. and
7 // international copyright and other intellectual property
8 // laws.
9 //
10 //
11 // DISCLAIMER
12 // This disclaimer is not a license and does not grant any
13 // rights to the materials distributed herewith. Except as
14 // otherwise provided in a valid license issued to you by
15 // Xilinx, and to the maximum extent permitted by applicable
16 // law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
17 // WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL WARRANTIES
18 // AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
19 // BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
20 // INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
21 // (2) Xilinx shall not be liable (whether in contract or tort,
22 // including negligence, or under any other theory of
23 // liability) for any loss or damage of any kind or nature
24 // related to, arising under or in connection with these
25 // materials, including for any direct, or any indirect,
26 // special, incidental, or consequential loss or damage
27 // (including loss of data, profits, goodwill, or any type of
28 // loss or damage suffered as a result of any action brought
29 // by a third party) even if such damage or loss was
30 // reasonably foreseeable or Xilinx had been advised of the
31 // possibility of the same.
32 //
33 // CRITICAL APPLICATIONS
34 // Xilinx products are not designed or intended to be fail-
35 // safe, or for use in any application requiring fail-safe
36 // performance, such as life-support or safety devices or
37 // systems, Class III medical devices, nuclear facilities,
38 // applications related to the deployment of airbags, or any
39 // other applications that could lead to death, personal
40 // injury, or severe property or environmental damage
41 // (individually and collectively, "Critical
42 // Applications"). Customer assumes the sole risk and
43 // liability of any use of Xilinx products in Critical
44 // Applications, subject only to applicable laws and
45 // regulations governing limitations on product liability.
46 //
47 // THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
48 // PART OF THIS FILE AT ALL TIMES.
49 //
50 //-----
51 // User entered comments
52 //-----
53 // None
54 //
55 //-----
56 // Output Clock Freq (MHz) Phase Duty Cycle Fk-to-Fk Phase
57 // Clock Freq (MHz) (degrees) (%) Jitter (ps) Error (ps)
58 //-----
59 // _clk_50 50.000 0.000 50.0 167.017 114.212
60 // _clk_200 200.000 0.000 50.0 126.455 114.212
61 // _clk_400 400.000 0.000 50.0 111.164 114.212
62 //-----
63 //
64 // Input Clock Freq (MHz) Input Jitter (UI)
65 //-----
66 // _primary 100.000 0.010
67 //
68 timescale lps/lps
69
70 (* CORE_GENERATION_INFO = "clk_module,clk_wiz_v5_3_1,(component_name=clk_module,use_phase_alignment=true,use_min_o_jitter=false,use_max_i_jitter=false,use_dyn_phase_shift=false,use_
71
72 module clk_module
73 (
74 // Clock in ports
75 input clk_in1,
76 // Clock out ports
77 output clk_50,
78 output clk_200,
79 output clk_400,
80 // Status and control signals
81 input reset,
82 output locked
83 );
84
85 clk_module_clk_wiz inst
86 (
87 // Clock in ports
88 .clk_in1(clk_in1),
89 // Clock out ports
90 .clk_50(clk_50),
91 .clk_200(clk_200),
92 .clk_400(clk_400),
93 // Status and control signals
94 .reset(reset),
95 .locked(locked)
96 );
97
98 endmodule

```

*****CLK_MODULE_CLK_WIZ.V*****
 *****© Copyright 2008 – 2013 Xilinx, Inc. All rights reserved
 [16]*****

```

1 // file: clk_module.v
2 //
3 //
4 // (c) Copyright 2008 - 2013 Xilinx, Inc. All rights reserved.
5 //
6 // This file contains confidential and proprietary information
7 // of Xilinx, Inc. and is protected under U.S. and
8 // international copyright and other intellectual property
9 // laws.
10 //
11 // DISCLAIMER
12 // This disclaimer is not a license and does not grant any
13 // rights to the materials distributed herewith. Except as
14 // otherwise provided in a valid license issued to you by
15 // Xilinx, and to the maximum extent permitted by applicable
16 // law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
17 // WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL WARRANTIES
18 // AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
19 // BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
20 // INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
21 // (2) Xilinx shall not be liable (whether in contract or tort,
22 // including negligence, or under any other theory of
23 // liability) for any loss or damage of any kind or nature
24 // related to, arising under or in connection with these
25 // materials, including for any direct, or any indirect,
26 // special, incidental, or consequential loss or damage
27 // (including loss of data, profits, goodwill, or any type of
28 // loss or damage suffered as a result of any action brought
29 // by a third party) even if such damage or loss was
30 // reasonably foreseeable or Xilinx had been advised of the
31 // possibility of the same.
32 //
33 // CRITICAL APPLICATIONS
34 // Xilinx products are not designed or intended to be fail-
35 // safe, or for use in any application requiring fail-safe
36 // performance, such as life-support or safety devices or
37 // systems, Class III medical devices, nuclear facilities,
38 // applications related to the deployment of airbags, or any
39 // other applications that could lead to death, personal
40 // injury, or severe property or environmental damage
41 // (individually and collectively, "Critical
42 // Applications"). Customer assumes the sole risk and
43 // liability of any use of Xilinx products in Critical
44 // Applications, subject only to applicable laws and
45 // regulations governing limitations on product liability.
46 //
47 // THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
48 // PART OF THIS FILE AT ALL TIMES.
49 //
50 //-----
51 // User entered comments
52 //-----
53 // None
54 //
55 //-----
56 // Output Output Phase Duty Cycle Pk-to-Pk Phase
57 // Clock Freq (MHz) (degrees) (%) Jitter (ps) Error (ps)
58 //-----
59 // _clk_50 50.000 0.000 50.0 167.017 114.212
60 // _clk_200 200.000 0.000 50.0 126.455 114.212
61 // _clk_400 400.000 0.000 50.0 111.164 114.212
62 //-----
63 //-----
64 // Input Clock Freq (MHz) Input Jitter (UI)
65 //-----
66 // _primary 100.000 0.010
67 //
68 `timescale 1ps/1ps
69
70 module clk_module_clk_wiz
71
72 // Clock in ports
73 input clk_in1,
74 // Clock out ports
75 output clk_50,
76 output clk_200,
77 output clk_400,
78 // Status and control signals
79 input reset,
80 output locked
81 );
82 // Input buffering
83 //-----
84 wire clk_in1_clk_module;
85 wire clk_in2_clk_module;
86 IBUF clkini_ibufg
87 (.O (clk_in1_clk_module),
88 .I (clk_in1));
89
90

```

```

91 // Clocking PRIMITIVE
92 //-----
93
94 // Instantiation of the MCMC PRIMITIVE
95 // * Unused inputs are tied off
96 // * Unused outputs are labeled unused
97
98 wire clk_50_clk_module;
99 wire clk_200_clk_module;
100 wire clk_400_clk_module;
101 wire clk_out4_clk_module;
102 wire clk_out5_clk_module;
103 wire clk_out6_clk_module;
104 wire clk_out7_clk_module;
105
106 wire [15:0] do_unused;
107 wire drdy_unused;
108 wire psdone_unused;
109 wire locked_int;
110 wire clkfbout_clk_module;
111 wire clkfbout_buf_clk_module;
112 wire clkfbouth_unused;
113 wire clkout0b_unused;
114 wire clkout1b_unused;
115 wire clkout2b_unused;
116 wire clkout3_unused;
117 wire clkfbstopped_unused;
118 wire clkinstopped_unused;
119 wire reset_high;
120
121
122
123
124
125 MCMC2_ADV
126 #(.BANDWIDTH ("OPTIMIZED"),
127 .CLKOUT4_CASCADE ("FALSE"),
128 .COMPENSATION ("ZHOLD"),
129 .STARTUP_WAIT ("FALSE"),
130 .DIVCLK_DIVIDE (1),
131 .CLKFBOUT_MULT_F (8.000),
132 .CLKFBOUT_PHASE (0.000),
133 .CLKFBOUT_USE_FINE_PS ("FALSE"),
134 .CLKOUT0_DIVIDE_F (16.000),
135 .CLKOUT0_PHASE (0.000),
136 .CLKOUT0_DUTY_CYCLE (0.500),
137 .CLKOUT0_USE_FINE_PS ("FALSE"),
138 .CLKOUT1_DIVIDE (4),
139 .CLKOUT1_PHASE (0.000),
140 .CLKOUT1_DUTY_CYCLE (0.500),
141 .CLKOUT1_USE_FINE_PS ("FALSE"),
142 .CLKOUT2_DIVIDE (2),
143 .CLKOUT2_PHASE (0.000),
144 .CLKOUT2_DUTY_CYCLE (0.500),
145 .CLKOUT2_USE_FINE_PS ("FALSE"),
146 .CLKINI_PERIOD (10.0))
147 mcm_adv_inst
148 // Output clocks
149 (
150 .CLKFBOUT (clkfbout_clk_module),
151 .CLKFBOUTB (clkfbouth_unused),
152 .CLKOUT0 (clk_50_clk_module),
153 .CLKOUT0B (clkout0b_unused),
154 .CLKOUT1 (clk_200_clk_module),
155 .CLKOUT1B (clkout1b_unused),
156 .CLKOUT2 (clk_400_clk_module),
157 .CLKOUT2B (clkout2b_unused),
158 .CLKOUT3 (clkout3_unused),
159 .CLKOUT3B (clkout3b_unused),
160 .CLKOUT4 (clkout4_unused),
161 .CLKOUT5 (clkout5_unused),
162 .CLKOUT6 (clkout6_unused),
163 // Input clock control
164 .CLKFBIN (clkfbout_buf_clk_module),
165 .CLKINI (clk_in1_clk_module),
166 .CLKINE (1'b0),
167 // Tied to always select the primary input clock
168 .CLKINSEL (1'b1),
169 // Ports for dynamic reconfiguration
170 .DADDR (7'h0),
171 .DCLK (1'b1),
172 .DEN (1'b0),
173 .DI (16'h0),
174 .DO (do_unused),
175 .DRDY (drdy_unused),
176 .DWE (1'b0),
177 // Ports for dynamic phase shift
178 .PSCLK (1'b0),
179 .PSEN (1'b0),
180 .PSINDEC (1'b0),

```

```
181 .PSDONE          (psdone_unused),
182 // Other control and status signals
183 .LOCKED          (locked_int),
184 .CLKINSTOPPED   (clkinstopped_unused),
185 .CLKFBSTOPPED   (clkfbstopped_unused),
186 .PWRMWN         (1'b0),
187 .RST            (reset_high));
188
189 assign reset_high = reset;
190
191 assign locked = locked_int;
192 // Clock Monitor clock assigning
193 //-----
194 // Output buffering
195 //-----
196
197 BUFG clkf_buf
198 (.O (clkfbout_buf_clk_module),
199  .I (clkfbout_clk_module));
200
201
202
203 BUFG clkout1_buf
204 (.O (clk_50),
205  .I (clk_50_clk_module));
206
207
208
209
210
211
212 BUFG clkout3_buf
213 (.O (clk_400),
214  .I (clk_400_clk_module));
215
216
217
218 endmodule
```

*****PINGPONG.V*****

```

1 timescale ns / ps
2 ///////////////////////////////////////////////////////////////////
3 // Company:
4 // Engineer:
5 //
6 // Create Date: 04/19/2017 04:41:54 PM
7 // Design Name:
8 // Module Name: pingpong
9 // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21
22 ///////////////////////////////////////////////////////////////////
23 //Ping//Pong// //One Register is being utilized for real component
24 // 0 // 0 // //One Register is being utilized for imag component
25 /////////////////////////////////////////////////////////////////// //i.e. Ping has a real and imaginary register
26 // 1 // 1 // //Pong has a real and imaginary register
27 /////////////////////////////////////////////////////////////////// //
28 // 2 // 2 // //Bit Reversal explained
29 /////////////////////////////////////////////////////////////////// //X[000] = x[000] or X[0] => x[0]
30 // 3 // 3 // //X[001] = x[100] or X[1] => x[4]
31 /////////////////////////////////////////////////////////////////// //X[010] = x[010] or X[2] => x[2]
32 // 4 // 4 // //X[011] = x[110] or X[3] => x[6]
33 /////////////////////////////////////////////////////////////////// //X[100] = x[001] or X[4] => x[1]
34 // 5 // 5 // //X[101] = x[101] or X[5] => x[5]
35 /////////////////////////////////////////////////////////////////// //X[110] = x[011] or X[6] => x[3]
36 // 6 // 6 // //X[111] = x[111] or X[7] => x[7]
37 /////////////////////////////////////////////////////////////////// //
38 // 7 // 7 // //
39 /////////////////////////////////////////////////////////////////// //
40
41 // complex input = (x + jz)
42 // complex output = (x + jz)
43
44 module pingpong(
45     input wire [17:0] XqIn_PingPong_Real, // input Xq real component
46     input wire [17:0] XqIn_PingPong_Imag, // input Xq imag component
47     input wire clk, // input clock
48     input wire [3:0] counter, // input counter
49     output wire [17:0] XqOut_PingPongw_Real, // output Xq real component on a wire
50     output wire [17:0] XqOut_PingPongw_Imag // output Xq imag component on a wire
51 );
52
53     reg [17:0] XqPing_Real [7:0]; // register for Ping real component
54     reg [17:0] XqPing_Imag [7:0]; // register for Ping imag component
55     reg [17:0] XqPong_Real [7:0]; // register for Pong real component
56     reg [17:0] XqPong_Imag [7:0]; // register for Pong imag component
57     wire [2:0] tranpose; // register used to tranpose counter
58     reg [2:0] indexbr; // index utilized for bit reversal
59
60     wire pingpongstatus;
61     reg ping_loading;
62     reg pong_loading;
63
64     assign XqOut_PingPongw_Real = (counter[3]==1'b0) ? XqPing_Real[counter[2:0]] : XqPong_Real[counter[2:0]]; //XqOut_PingPongw_Real; // register for output Xq real component
65     assign XqOut_PingPongw_Imag = (counter[3]==1'b0) ? XqPing_Imag[counter[2:0]] : XqPong_Imag[counter[2:0]]; //XqOut_PingPongw_Imag; // register for output Xq imag component
66     assign pingpongstatus = counter[3];
67     assign tranpose = counter + 1;
68     //always statement that occurs at every rising
69     //edge of the clock
70
71     // Initial block (Only executed once when simulation start)
72     initial begin
73         indexbr = 0;
74     end
75
76     always@(posedge clk)
77     begin
78         // counter transposed due to bit reversal
79         indexbr <= {tranpose[0],tranpose[1],tranpose[2]};
80
81         //The Pong buffer is outputted when the Ping buffer is
82         //being filled up. Ping and Pong then alternate
83         //The MSB of counter is utilized as
84         //my switch bit. Buffers are complex numbers.
85
86         //Alternating loading Ping and Pong buffers
87         //Based on MSB counter. Results in 16
88         //cycle period.
89         if (counter[3] == 1'b1) begin
90             XqPing_Real[indexbr] <= XqIn_PingPong_Real;

```

```
91     XqPing_Imag[indexbr] <= XqIn_PingPong_Imag;
92     ping_loading <= 1'b1;
93     pong_loading <= 1'b0;
94     end
95     else begin
96     XqPong_Real[indexbr] <= XqIn_PingPong_Real;
97     XqPong_Imag[indexbr] <= XqIn_PingPong_Imag;
98     ping_loading <= 1'b1;
99     pong_loading <= 1'b0;
100    end
101  end
102 endmodule
```

*****RADIX2_BFM.V*****

```

1  timescale 1ns / 1ps
2  ////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 04/08/2017 10:24:56 AM
7  // Design Name:
8  // Module Name: Radix2_BFM
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ////////////////////////////////////////////////////////////////////
21
22
23 module Radix2_BFM(
24     input wire [17:0] Xq_Real,
25     input wire [17:0] Xq_Imag,
26     input wire clk,
27     input wire [17:0] W_Real,
28     input wire [17:0] W_Imag,
29     input wire [2:0] counterw,
30     output wire [17:0] Radix2_BFMOut_Real,
31     output wire [17:0] Radix2_BFMOut_Imag
32 );
33
34 (* dont_touch = "true" *)reg [17:0] TopMuxOutDelay1_Real, TopMuxOutDelay2_Real, TopMuxOutDelay3_Real;
35 (* dont_touch = "true" *)reg [17:0] TopMuxOutDelay1_Imag, TopMuxOutDelay2_Imag, TopMuxOutDelay3_Imag;
36 (* dont_touch = "true" *)wire [17:0] TopMuxOutw_Real, BottomMuxOutw_Real;
37 (* dont_touch = "true" *)wire [17:0] TopMuxOutw_Imag, BottomMuxOutw_Imag;
38 (* dont_touch = "true" *)wire [17:0] TopMuxOutw_halfscale_Real, BottomMuxOutw_halfscale_Real;
39 (* dont_touch = "true" *)wire [17:0] TopMuxOutw_halfscale_Imag, BottomMuxOutw_halfscale_Imag;
40 (* dont_touch = "true" *)reg [17:0] XqDelay1_Real;
41 (* dont_touch = "true" *)reg [17:0] XqDelay1_Imag;
42 (* dont_touch = "true" *)wire [17:0] MultOutw_Real;
43 (* dont_touch = "true" *)wire [17:0] MultOutw_Imag;
44 (* dont_touch = "true" *)wire [17:0] AdderOutw_Real;
45 (* dont_touch = "true" *)wire [17:0] AdderOutw_Imag;
46
47 // ----- 2 to 1 Multiplexer Instantiated for top -----
48 twoToOneMux twoToOneMux_Top
49 (
50     .in1_real(Xq_Real),
51     .in1_imag(Xq_Imag),
52     .in2_real(XqDelay1_Real),
53     .in2_imag(XqDelay1_Imag),
54     .clk(~counterw[0]),
55     .out_real(TopMuxOutw_Real),
56     .out_imag(TopMuxOutw_Imag)
57 );
58
59 // ----- 2 to 1 Multiplexer Instantiated for bottom -----
60 twoToOneMux twoToOneMux_Bottom
61 (
62     .in1_real(Xq_Real),
63     .in1_imag(Xq_Imag),
64     .in2_real(XqDelay1_Real),
65     .in2_imag(XqDelay1_Imag),
66     .clk(counterw[0]),
67     .out_real(BottomMuxOutw_Real),
68     .out_imag(BottomMuxOutw_Imag)
69 );
70
71 // ----- 18 bit by 18 bit Multiplier Instantiated -----
72 multiply_complex ComplexMultiply2Stage
73 (
74     .clk(counterw[0]),
75     .a(BottomMuxOutw_halfscale_Real),
76     .jb(BottomMuxOutw_halfscale_Imag),
77     .c(W_Real),
78     .jd(W_Imag),
79     .Xqout_real(MultOutw_Real),
80     .Xqout_imag(MultOutw_Imag)
81 );
82
83 // ----- 18 bit by 18 bit Adder Instantiated -----
84 addsub18x18 AdderandSubtractor1Stage
85 (
86     .TopIn_real(TopMuxOutDelay3_Real),
87     .TopIn_imag(TopMuxOutDelay3_Imag),
88     .BottomIn_real(MultOutw_Real),
89     .BottomIn_imag(MultOutw_Imag),
90     .clk(clk),

```

```

91     .counterw(counterw[0]),
92     .Xqout_real(AdderOutw_Real),
93     .Xqout_imag(AdderOutw_Imag)
94 );
95
96 assign Radix2_BFMOut_Real = AdderOutw_Real;
97 assign Radix2_BFMOut_Imag = AdderOutw_Imag;
98 assign TopMuxOutw_halfscale_Real = TopMuxOutw_Real >> 1;
99 assign TopMuxOutw_halfscale_Imag = TopMuxOutw_Imag >> 1;
100 assign BottomMuxOutw_halfscale_Real = BottomMuxOutw_Real >> 1;
101 assign BottomMuxOutw_halfscale_Imag = BottomMuxOutw_Imag >> 1;
102
103 always@(posedge clk)
104 begin
105     TopMuxOutDelay1_Real <= TopMuxOutw_halfscale_Real;
106     TopMuxOutDelay1_Imag <= TopMuxOutw_halfscale_Imag;
107     TopMuxOutDelay2_Real <= TopMuxOutDelay1_Real;
108     TopMuxOutDelay2_Imag <= TopMuxOutDelay1_Imag;
109     TopMuxOutDelay3_Real <= TopMuxOutDelay2_Real;
110     TopMuxOutDelay3_Imag <= TopMuxOutDelay2_Imag;
111     XqDelay1_Real <= Xq_Real;
112     XqDelay1_Imag <= Xq_Imag;
113 end
114
115 endmodule

```

*****TWO2ONEMUX.V*****

```
1 timescale 1ns / 1ps
2 ///////////////////////////////////////////////////////////////////
3 // Company:
4 // Engineer:
5 //
6 // Create Date: 04/05/2017 03:43:36 PM
7 // Design Name: Complex Multiplexer
8 // Module Name: two2oneMux
9 // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21
22 module two2oneMux (
23     input wire signed [17:0] in1_real, // a
24     input wire signed [17:0] in1_imag, // jb
25     input wire signed [17:0] in2_real, // c
26     input wire signed [17:0] in2_imag, // jd
27     input wire clk,
28     output wire signed [17:0] out_real, // c or a
29     output wire signed [17:0] out_imag // jd or jb
30 );
31
32     assign out_real = (clk=='b0') ? in2_real : in1_real; //Real Component
33     assign out_imag = (clk=='b0') ? in2_imag : in1_imag; //Imag Component
34 endmodule
```

*****MULTIPLY_COMPLEX.V*****

```

1 module multiply_complex(
2   input wire clk,
3   input wire [17:0] a,
4   input wire [17:0] jb,
5   input wire [17:0] c,
6   input wire [17:0] jd,
7   output wire [17:0] Xqout_real,
8   output wire [17:0] Xqout_imag
9 );
10
11 (* dont_touch = "true" *) reg sign_mult_left_Real, sign_mult_right_Real, sign_mult_left_Imag, sign_mult_right_Imag;
12 (* dont_touch = "true" *) wire sign_a;
13 (* dont_touch = "true" *) wire sign_b;
14 (* dont_touch = "true" *) wire sign_c;
15 (* dont_touch = "true" *) wire sign_d;
16 (* dont_touch = "true" *) wire [16:0] unsigned_a; //
17 (* dont_touch = "true" *) wire [16:0] unsigned_b; //
18 (* dont_touch = "true" *) wire [16:0] unsigned_c; //
19 (* dont_touch = "true" *) wire [16:0] unsigned_d; //
20 (* dont_touch = "true" *) wire [16:0] trunc_left_Real; // (a*c)
21 (* dont_touch = "true" *) wire [16:0] trunc_right_Real; // (b*d)
22 (* dont_touch = "true" *) wire [16:0] trunc_left_Imag; // (a*d)
23 (* dont_touch = "true" *) wire [16:0] trunc_right_Imag; // (b*c)
24 (* dont_touch = "true" *) wire [17:0] signed_left_Real; // (a*c)
25 (* dont_touch = "true" *) wire [17:0] signed_right_Real; // (b*d)
26 (* dont_touch = "true" *) wire [17:0] signed_left_Imag; // (a*d)
27 (* dont_touch = "true" *) wire [17:0] signed_right_Imag; // (b*c)
28 //-----
29
30 //performs unsigned fixed point multiplication. truncates output reducing precision.
31 //returns 17 bits. Inputs are a clock, 17 bit unsigned fixed point numbers labeled as
32 //unsigned_a, unsigned_b, unsigned_c, and unsigned_d. Outputs are 17 bit unsigned fixed point
33 //numbers labeled as left_Real
34 mult1x18 unsigned_multiply
35 (
36   .clk(clk),
37   .a(unsigned_a),
38   .b(unsigned_b),
39   .c(unsigned_c),
40   .d(unsigned_d),
41   .Out_left_Real(trunc_left_Real),
42   .Out_right_Real(trunc_right_Real),
43   .Out_left_Imag(trunc_left_Imag),
44   .Out_right_Imag(trunc_right_Imag));
45
46 //2'c operator following multiplication. Concatenates the sign bit with a 2'c number. returns 18 bits
47 twoCompRedo twoC_leftReal(.signbit(sign_mult_left_Real),.number(trunc_left_Real),.sign_number(signed_left_Real));
48 twoCompRedo twoC_rightReal(.signbit(sign_mult_right_Real),.number(trunc_right_Real),.sign_number(signed_right_Real));
49 twoCompRedo twoC_leftImag(.signbit(sign_mult_left_Imag),.number(trunc_left_Imag),.sign_number(signed_left_Imag));
50 twoCompRedo twoC_rightImag(.signbit(sign_mult_right_Imag),.number(trunc_right_Imag),.sign_number(signed_right_Imag));
51
52 //2'c operator prior to multiplication. returns unsigned number. returns 17 bits
53 twoComp insta(.In(a),.Out(unsigned_a)); //extracts data portion a
54 twoComp instb(.In(jb),.Out(unsigned_b)); //extracts data portion b
55 twoComp instc(.In(c),.Out(unsigned_c)); //extracts data portion c
56 twoComp instd(.In(jd),.Out(unsigned_d)); //extracts data portion d
57
58 // saves sign bit. returns 1 bit
59 assign sign_a = a[17]; //extracts sign of a
60 assign sign_b = jb[17]; //extracts sign of b
61 assign sign_c = c[17]; //extracts sign of c
62 assign sign_d = jd[17]; //extracts sign of d
63
64 assign Xqout_real = signed_left_Real - signed_right_Real;
65 assign Xqout_imag = signed_left_Imag + signed_right_Imag;
66
67 // Initial block (Only executed once when simulation start)
68 initial
69   begin
70     end
71
72 // Statements below will be executed on every clock rising edge
73
74 always@(posedge clk)
75   begin
76     sign_mult_left_Real <= sign_a ^ sign_c;
77     sign_mult_right_Real <= sign_b ^ sign_d;
78     sign_mult_left_Imag <= sign_a ^ sign_d;
79     sign_mult_right_Imag <= sign_b ^ sign_c;
80   end
81 endmodule

```

*****MULT18X18.V*****

```

1  timescale 1ns / 1ps
2  //////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 03/25/2017 11:43:54 AM
7  // Design Name: Complex Multiplication
8  // Module Name: mult18x18
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 6.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////
21
22
23 module mult18x18(
24     input wire clk,
25     input wire [16:0] a,    //a
26     input wire [16:0] jb,  //jb
27     input wire [16:0] c,   //c
28     input wire [16:0] jd,  //jd
29     output wire [16:0] Out_left_Real,  //(a*c)
30     output wire [16:0] Out_right_Real, //(b*d)
31     output wire [16:0] Out_left_Imag,  //(a*jd)
32     output wire [16:0] Out_right_Imag; //(c*jb)
33
34     //Complex Multiplier
35     //      Real      Imag
36     // (a+jb)(c+jd) = (a*c - b*d) + j(b*d + a*d)
37     (* dont_touch = "true" *)reg [34:0] mult_left_Real;
38     (* dont_touch = "true" *)reg [34:0] mult_right_Real;
39     (* dont_touch = "true" *)reg [34:0] mult_left_Imag;
40     (* dont_touch = "true" *)reg [34:0] mult_right_Imag;
41
42     assign Out_left_Real[16:0] = mult_left_Real[32:16]; // (a*c)
43     assign Out_right_Real[16:0] = mult_right_Real[32:16]; // (jb*jd)
44     assign Out_left_Imag[16:0] = mult_left_Imag[32:16]; // (a*jd)
45     assign Out_right_Imag[16:0] = mult_right_Imag[32:16]; // (jb*c)
46
47     initial begin
48         end
49
50     always@(posedge clk)
51     begin
52         //Calculate real portion
53         mult_left_Real <= a * c;
54         mult_right_Real <= jb * jd;
55
56         //Calculate imag portion
57         mult_left_Imag <= a * jd;
58         mult_right_Imag <= jb * c;
59     end
60
61 endmodule

```

*****TWOCOMPRED0.V*****

```
1 `timescale 1ns / 1ps
2 ////////////////////////////////////////////////////////////////////
3 // Company:
4 // Engineer:
5 //
6 // Create Date: 05/23/2017 08:08:50 AM
7 // Design Name:
8 // Module Name: twoCompRedo
9 // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ////////////////////////////////////////////////////////////////////
21
22
23 module twoCompRedo(
24     input wire signbit,
25     input wire [16:0] number,
26     output wire [17:0] sign_number
27 );
28
29     assign sign_number = signbit?(signbit,-number+1):(signbit,number);
30 endmodule
```

*****TWOCOMP.V*****

```
1 timescale 1ns / 1ps
2 ///////////////////////////////////////////////////////////////////
3 // Company:
4 // Engineer:
5 //
6 // Create Date: 05/19/2017 03:16:22 AM
7 // Design Name:
8 // Module Name: Two's Complement
9 // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21
22
23 module twoComp(
24     input wire [17:0] In,
25     output wire [16:0] Out);
26
27     assign Out = In[17]?~In+1:In;
28
29 endmodule
```

*****ADDSUB18X18.V*****

```

1  timescale 1ns / 1ps
2  //////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 04/08/2017 09:41:42 AM
7  // Design Name: Complex Addition / Subtraction
8  // Module Name: addsub18x18
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////
21
22
23 module addsub18x18(
24     input wire [17:0] TopIn_real,    // a
25     input wire [17:0] TopIn_imag,    // jb
26     input wire [17:0] BottomIn_real, // c
27     input wire [17:0] BottomIn_imag, // jd
28     input wire        clk,
29     input wire        counterw,
30     output wire [17:0] Xqout_real,    // a + b or a - b
31     output wire [17:0] Xqout_imag ); // j(b + d) or j(b - d)
32
33     (* dont_touch = "true" *) reg [17:0] out_real;
34     (* dont_touch = "true" *) reg [17:0] out_imag;
35
36     //Complex Addition/Subtraction
37     //      Real  Imag
38     //(a+jb)+(c+jd) = a + c , j(b+d)
39     //(a+jb)-(c+jd) = a - c , j(b-d)
40     assign Xqout_real = out_real; //Real Component
41     assign Xqout_imag = out_imag; //Imag Component
42
43     always@(posedge clk)
44     begin
45         out_real <= (counterw==1'b1) ? TopIn_real + BottomIn_real : TopIn_real - BottomIn_real; //Real Component
46         out_imag <= (counterw==1'b1) ? TopIn_imag + BottomIn_imag : TopIn_imag - BottomIn_imag; //Imag Component
47     end
48 endmodule

```

*****PINGPONGK.V*****

```

1 | timescale ns / ps
2 |
3 | ////////////////////////////////////////////////////////////////////
4 | // Company:
5 | // Engineer:
6 | //
7 | // Create Date: 04/19/2017 04:41:54 PM
8 | // Design Name:
9 | // Module Name: pingpong
10 | // Project Name:
11 | // Target Devices:
12 | // Tool Versions:
13 | // Description:
14 | //
15 | // Dependencies:
16 | //
17 | // Revision:
18 | // Revision 0.01 - File Created
19 | // Additional Comments:
20 | //
21 | ////////////////////////////////////////////////////////////////////
22 |
23 | // for N=8
24 | //k = 0, N/2, 1, N/2+1, 2, N/2+2, 3, N/2+3;
25 | //k = {0, 4, 1, 5, 2, 6, 3, 7}
26 |
27 | ////////////////////////////////////////////////////////////////////
28 | //Ping/Pong// //One Register is being utilized for real component
29 | // 0 // 0 // //One Register is being utilized for imag component
30 | //////////////////////////////////////////////////////////////////// //1.e. Ping has a real and imaginary register
31 | // 1 // 1 // //Pong has a real and imaginary register
32 | ////////////////////////////////////////////////////////////////////
33 | // 2 // 2 // //Ping-pong fill order explained
34 | //////////////////////////////////////////////////////////////////// //X[000] = x[000] or X[0] => x[0]
35 | // 3 // 3 // //X[001] = x[100] or X[1] => x[4]
36 | //////////////////////////////////////////////////////////////////// //X[010] = x[001] or X[2] => x[1]
37 | // 4 // 4 // //X[011] = x[101] or X[3] => x[5]
38 | //////////////////////////////////////////////////////////////////// //X[100] = x[010] or X[4] => x[2]
39 | // 5 // 5 // //X[101] = x[110] or X[5] => x[6]
40 | //////////////////////////////////////////////////////////////////// //X[110] = x[011] or X[6] => x[3]
41 | // 6 // 6 // //X[111] = x[111] or X[7] => x[7]
42 | ////////////////////////////////////////////////////////////////////
43 | // 7 // 7 //
44 | ////////////////////////////////////////////////////////////////////
45 |
46 | // complex input = (x + jz)
47 | // complex output = (x + jz)
48 |
49 | module pingpongK(
50 |     input wire [17:0] XqIn_PingPong_Real, // input Xq real component
51 |     input wire [17:0] XqIn_PingPong_Imag, // input Xq imag component
52 |     input wire clk, // input clock
53 |     input wire [3:0] counter, // input counter
54 |     output wire [17:0] XqOut_PingPongw_Real, // output Xq real component on a wire
55 |     output wire [17:0] XqOut_PingPongw_Imag // output Xq imag component on a wire
56 | );
57 |
58 | (* dont_touch = "true" *) reg [17:0] XqPing_Real [7:0]; // register for Ping real component
59 | (* dont_touch = "true" *) reg [17:0] XqPing_Imag [7:0]; // register for Ping imag component
60 | (* dont_touch = "true" *) reg [17:0] XqPong_Real [7:0]; // register for Pong real component
61 | (* dont_touch = "true" *) reg [17:0] XqPong_Imag [7:0]; // register for Pong imag component
62 | (* dont_touch = "true" *) reg [3:0] buffer_counter;
63 | // (* dont_touch = "true" *) wire [3:0] select;
64 |
65 |
66 | assign XqOut_PingPongw_Real = (buffer_counter[3]==1'b1) ? XqPing_Real[buffer_counter[2:0]] : XqPong_Real[buffer_counter[2:0]]; //XqOut_PingPongw_Real; // register for output Xq real
67 | assign XqOut_PingPongw_Imag = (buffer_counter[3]==1'b1) ? XqPing_Imag[buffer_counter[2:0]] : XqPong_Imag[buffer_counter[2:0]]; //XqOut_PingPongw_Imag; // register for output Xq imag
68 | //assign select = buffer_counter + 1;
69 | //always statement that occurs at every rising
70 | //edge of the clock
71 |
72 | initial
73 | begin
74 |
75 | end
76 |
77 | always@(posedge clk)
78 | begin
79 |     //The Pong buffer is outputted when the Ping buffer is
80 |     //being filled up. Ping and Pong then alternate
81 |     //The MSB of counter is utilized as
82 |     //my switch bit. Buffers are complex numbers.
83 |
84 |     // Input code to Ping and Pong buffers
85 |     // based on k = 0, N/2, 1, N/2+1, 2, N/2+2, 3, N/2+3;
86 |     // k = {0, 4, 1, 5, 2, 6, 3, 7};
87 |     // Results in 16 cycle period.
88 |
89 |     case(counter)
90 |         3'b000: //Xq(0)

```

```

91     begin
92     XqPong_Real[2] <= XqIn_PingPong_Real;
93     XqPong_Imag[2] <= XqIn_PingPong_Imag;
94     buffer_counter = 4'd13;
95     end
96 3'b0001: //Xq(1)
97     begin
98     XqPong_Real[6] <= XqIn_PingPong_Real;
99     XqPong_Imag[6] <= XqIn_PingPong_Imag;
100    buffer_counter = 4'd13;
101    end
102 3'b0010: //Xq(2)
103     begin
104     XqPong_Real[3] <= XqIn_PingPong_Real;
105     XqPong_Imag[3] <= XqIn_PingPong_Imag;
106     buffer_counter = 4'd14;
107     end
108 3'b0011: //Xq(3)
109     begin
110     XqPong_Real[7] <= XqIn_PingPong_Real;
111     XqPong_Imag[7] <= XqIn_PingPong_Imag;
112     buffer_counter = 4'd15;
113     end
114 3'b0100: //Xq(4)
115     begin
116     XqPing_Real[0] <= XqIn_PingPong_Real;
117     XqPing_Imag[0] <= XqIn_PingPong_Imag;
118     buffer_counter = 4'd0;
119     end
120 3'b0101: //Xq(5)
121     begin
122     XqPing_Real[4] <= XqIn_PingPong_Real;
123     XqPing_Imag[4] <= XqIn_PingPong_Imag;
124     buffer_counter = 4'd1;
125     end
126 3'b0110: //Xq(6)
127     begin
128     XqPing_Real[1] <= XqIn_PingPong_Real;
129     XqPing_Imag[1] <= XqIn_PingPong_Imag;
130     buffer_counter = 4'd2;
131     end
132 3'b0111: //Xq(7)
133     begin
134     XqPing_Real[5] <= XqIn_PingPong_Real;
135     XqPing_Imag[5] <= XqIn_PingPong_Imag;
136     buffer_counter = 4'd3;
137     end
138 4'b1000: //Xq(8)
139     begin
140     XqPing_Real[2] <= XqIn_PingPong_Real;
141     XqPing_Imag[2] <= XqIn_PingPong_Imag;
142     buffer_counter = 4'd4;
143     end
144 4'b1001: //Xq(9)
145     begin
146     XqPing_Real[6] <= XqIn_PingPong_Real;
147     XqPing_Imag[6] <= XqIn_PingPong_Imag;
148     buffer_counter = 4'd5;
149     end
150 4'b1010: //Xq(10)
151     begin
152     XqPing_Real[3] <= XqIn_PingPong_Real;
153     XqPing_Imag[3] <= XqIn_PingPong_Imag;
154     buffer_counter = 4'd6;
155     end
156 4'b1011: //Xq(11)
157     begin
158     XqPing_Real[7] <= XqIn_PingPong_Real;
159     XqPing_Imag[7] <= XqIn_PingPong_Imag;
160     buffer_counter = 4'd7;
161     end
162 4'b1100: //Xq(12)
163     begin
164     XqPong_Real[0] <= XqIn_PingPong_Real;
165     XqPong_Imag[0] <= XqIn_PingPong_Imag;
166     buffer_counter = 4'd8;
167     end
168 4'b1101: //Xq(13)
169     begin
170     XqPong_Real[4] <= XqIn_PingPong_Real;
171     XqPong_Imag[4] <= XqIn_PingPong_Imag;
172     buffer_counter = 4'd9;
173     end
174 4'b1110: //Xq(14)
175     begin
176     XqPong_Real[1] <= XqIn_PingPong_Real;
177     XqPong_Imag[1] <= XqIn_PingPong_Imag;
178     buffer_counter = 4'd10;
179     end
180 4'b1111: //Xq(15)

```

```
181 |         begin
182 |             XqPong_Real[5] <= XqIn_PingPong_Real;
183 |             XqPong_Imag[5] <= XqIn_PingPong_Imag;
184 |             buffer_counter = 4'd11;
185 |         end
186 |     endcase
187 | end
188 | endmodule
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. HARDWARE CONSTRAINTS FILE

*****NEXYS4DDR_MASTER.XDC*****

```
1 ## This file is a general .xdc for the Nexys4 DDR Rev. C
2 ## To use it in a project:
3 ## - uncomment the lines corresponding to used pins
4 ## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project
5
6 ## Clock signal
7 set_property -dict ( PACKAGE_PIN E3 IOSTANDARD LVCMOS33 ) [get_ports CLK100MHZ]
8 create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];
9
10
11 ##Switches
12
13 set_property -dict ( PACKAGE_PIN J15 IOSTANDARD LVCMOS33 ) [get_ports { SW[0] }]; #IO_L24N_T3_R50_15 Sch=sw[0]
14 set_property -dict ( PACKAGE_PIN L16 IOSTANDARD LVCMOS33 ) [get_ports { SW[1] }]; #IO_L3N_T0_D05_EMCLK_14 Sch=sw[1]
15 set_property -dict ( PACKAGE_PIN M13 IOSTANDARD LVCMOS33 ) [get_ports { SW[2] }]; #IO_L6N_T0_D09_VREF_14 Sch=sw[2]
16 set_property -dict ( PACKAGE_PIN R16 IOSTANDARD LVCMOS33 ) [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
17 set_property -dict ( PACKAGE_PIN R17 IOSTANDARD LVCMOS33 ) [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
18 set_property -dict ( PACKAGE_PIN T18 IOSTANDARD LVCMOS33 ) [get_ports { SW[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
19 set_property -dict ( PACKAGE_PIN U18 IOSTANDARD LVCMOS33 ) [get_ports { SW[6] }]; #IO_L17N_T2_A13_D09_14 Sch=sw[6]
20 set_property -dict ( PACKAGE_PIN R13 IOSTANDARD LVCMOS33 ) [get_ports { SW[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
21 set_property -dict ( PACKAGE_PIN T8 IOSTANDARD LVCMOS18 ) [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
22 set_property -dict ( PACKAGE_PIN U8 IOSTANDARD LVCMOS18 ) [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
23 set_property -dict ( PACKAGE_PIN R16 IOSTANDARD LVCMOS33 ) [get_ports { SW[10] }]; #IO_L16P_T2_D05_ROW_B_14 Sch=sw[10]
24 set_property -dict ( PACKAGE_PIN T13 IOSTANDARD LVCMOS33 ) [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
25 set_property -dict ( PACKAGE_PIN H6 IOSTANDARD LVCMOS33 ) [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
26 set_property -dict ( PACKAGE_PIN U12 IOSTANDARD LVCMOS33 ) [get_ports { SW[13] }]; #IO_L20P_T3_A09_D24_14 Sch=sw[13]
27 set_property -dict ( PACKAGE_PIN U11 IOSTANDARD LVCMOS33 ) [get_ports { SW[14] }]; #IO_L19N_T3_A09_D05_VREF_14 Sch=sw[14]
28 set_property -dict ( PACKAGE_PIN V10 IOSTANDARD LVCMOS33 ) [get_ports { SW[15] }]; #IO_L21F_T3_D05_14 Sch=sw[15]
29
30
31 ## LEDs
32
33 set_property -dict ( PACKAGE_PIN H17 IOSTANDARD LVCMOS33 ) [get_ports { LED[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
34 set_property -dict ( PACKAGE_PIN K15 IOSTANDARD LVCMOS33 ) [get_ports { LED[1] }]; #IO_L24P_T3_R51_15 Sch=led[1]
35 set_property -dict ( PACKAGE_PIN J13 IOSTANDARD LVCMOS33 ) [get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
36 set_property -dict ( PACKAGE_PIN N14 IOSTANDARD LVCMOS33 ) [get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
37 set_property -dict ( PACKAGE_PIN R18 IOSTANDARD LVCMOS33 ) [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
38 set_property -dict ( PACKAGE_PIN U17 IOSTANDARD LVCMOS33 ) [get_ports { LED[5] }]; #IO_L18N_T2_A11_D07_14 Sch=led[5]
39 set_property -dict ( PACKAGE_PIN U17 IOSTANDARD LVCMOS33 ) [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
40 set_property -dict ( PACKAGE_PIN U16 IOSTANDARD LVCMOS33 ) [get_ports { LED[7] }]; #IO_L18P_T2_A10_D28_14 Sch=led[7]
41 set_property -dict ( PACKAGE_PIN V16 IOSTANDARD LVCMOS33 ) [get_ports { LED[8] }]; #IO_L16N_T1_A15_D31_14 Sch=led[8]
42 set_property -dict ( PACKAGE_PIN T15 IOSTANDARD LVCMOS33 ) [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
43 set_property -dict ( PACKAGE_PIN U14 IOSTANDARD LVCMOS33 ) [get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
44 set_property -dict ( PACKAGE_PIN T16 IOSTANDARD LVCMOS33 ) [get_ports { LED[11] }]; #IO_L15N_T2_D05_DOUT_C00_B_14 Sch=led[11]
45 set_property -dict ( PACKAGE_PIN V15 IOSTANDARD LVCMOS33 ) [get_ports { LED[12] }]; #IO_L16P_T2_C01_B_14 Sch=led[12]
46 set_property -dict ( PACKAGE_PIN W14 IOSTANDARD LVCMOS33 ) [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
47 set_property -dict ( PACKAGE_PIN V12 IOSTANDARD LVCMOS33 ) [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
48 set_property -dict ( PACKAGE_PIN W11 IOSTANDARD LVCMOS33 ) [get_ports { LED[15] }]; #IO_L21N_T3_D05_A06_D22_14 Sch=led[15]
49
50 set_property -dict ( PACKAGE_PIN R12 IOSTANDARD LVCMOS33 ) [get_ports { LED16_B }]; #IO_L6P_T0_D06_14 Sch=led16_b
51 set_property -dict ( PACKAGE_PIN M16 IOSTANDARD LVCMOS33 ) [get_ports { LED16_G }]; #IO_L10P_T1_D14_14 Sch=led16_g
52 set_property -dict ( PACKAGE_PIN N15 IOSTANDARD LVCMOS33 ) [get_ports { LED16_R }]; #IO_L11P_T1_SRCC_14 Sch=led16_r
53 set_property -dict ( PACKAGE_PIN G14 IOSTANDARD LVCMOS33 ) [get_ports { LED17_B }]; #IO_L15N_T2_D05_ADV_B_15 Sch=led17_b
54 set_property -dict ( PACKAGE_PIN R11 IOSTANDARD LVCMOS33 ) [get_ports { LED17_G }]; #IO_0_14 Sch=led17_g
55 set_property -dict ( PACKAGE_PIN N16 IOSTANDARD LVCMOS33 ) [get_ports { LED17_R }]; #IO_L11N_T1_SRCC_14 Sch=led17_r
56
57
58 ##7 segment display
59
60 set_property -dict ( PACKAGE_PIN T10 IOSTANDARD LVCMOS33 ) [get_ports { CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
61 set_property -dict ( PACKAGE_PIN R10 IOSTANDARD LVCMOS33 ) [get_ports { CB }]; #IO_25_14 Sch=cb
62 set_property -dict ( PACKAGE_PIN K16 IOSTANDARD LVCMOS33 ) [get_ports { CC }]; #IO_25_15 Sch=cc
63 set_property -dict ( PACKAGE_PIN K13 IOSTANDARD LVCMOS33 ) [get_ports { CD }]; #IO_L17P_T2_A26_15 Sch=cd
64 set_property -dict ( PACKAGE_PIN P15 IOSTANDARD LVCMOS33 ) [get_ports { CE }]; #IO_L13P_T2_MRCC_14 Sch=ce
65 set_property -dict ( PACKAGE_PIN T11 IOSTANDARD LVCMOS33 ) [get_ports { CF }]; #IO_L18P_T3_A10_D26_14 Sch=cf
66 set_property -dict ( PACKAGE_PIN L18 IOSTANDARD LVCMOS33 ) [get_ports { CG }]; #IO_L4P_T0_D04_14 Sch=cg
67
68 set_property -dict ( PACKAGE_PIN H15 IOSTANDARD LVCMOS33 ) [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
69
70 set_property -dict ( PACKAGE_PIN J17 IOSTANDARD LVCMOS33 ) [get_ports { AN[0] }]; #IO_L23P_T3_F0E_B_15 Sch=an[0]
71 set_property -dict ( PACKAGE_PIN J18 IOSTANDARD LVCMOS33 ) [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
72 set_property -dict ( PACKAGE_PIN T9 IOSTANDARD LVCMOS33 ) [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
73 set_property -dict ( PACKAGE_PIN J14 IOSTANDARD LVCMOS33 ) [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
74 set_property -dict ( PACKAGE_PIN P14 IOSTANDARD LVCMOS33 ) [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
75 set_property -dict ( PACKAGE_PIN T14 IOSTANDARD LVCMOS33 ) [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
76 set_property -dict ( PACKAGE_PIN K2 IOSTANDARD LVCMOS33 ) [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
77 set_property -dict ( PACKAGE_PIN U13 IOSTANDARD LVCMOS33 ) [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
78
79
80 ##Buttons
81
82 set_property -dict ( PACKAGE_PIN C12 IOSTANDARD LVCMOS33 ) [get_ports { CPU_RESETN }]; #IO_L3P_T0_D05_AD1P_15 Sch=cpu_resetn
83
84 set_property -dict ( PACKAGE_PIN N17 IOSTANDARD LVCMOS33 ) [get_ports { BTNC }]; #IO_L9P_T1_D05_14 Sch=btnc
85 set_property -dict ( PACKAGE_PIN M18 IOSTANDARD LVCMOS33 ) [get_ports { BTNU }]; #IO_L4N_T0_D05_14 Sch=btnu
86 set_property -dict ( PACKAGE_PIN P17 IOSTANDARD LVCMOS33 ) [get_ports { BTNL }]; #IO_L12P_T1_MRCC_14 Sch=btlnl
87 set_property -dict ( PACKAGE_PIN M17 IOSTANDARD LVCMOS33 ) [get_ports { BTNR }]; #IO_L10N_T1_D15_14 Sch=btnr
88 set_property -dict ( PACKAGE_PIN P18 IOSTANDARD LVCMOS33 ) [get_ports { BTND }]; #IO_L8N_T1_D05_D13_14 Sch=btnd
89
90
```

```

91 ##Pmod Headers
92
93
94 ##Pmod Header JA
95
96 #set_property -dict ( PACKAGE_PIN C17 IOSTANDARD LVCMOS33 ) [get_ports ( JA[1] )]; #IO_L20N_T3_A19_15 Sch=wja[1]
97 #set_property -dict ( PACKAGE_PIN D18 IOSTANDARD LVCMOS33 ) [get_ports ( JA[2] )]; #IO_L21N_T3_D05_A18_15 Sch=wja[2]
98 #set_property -dict ( PACKAGE_PIN E18 IOSTANDARD LVCMOS33 ) [get_ports ( JA[3] )]; #IO_L21E_T3_D05_15 Sch=wja[3]
99 #set_property -dict ( PACKAGE_PIN G17 IOSTANDARD LVCMOS33 ) [get_ports ( JA[4] )]; #IO_L18N_T2_A12_15 Sch=wja[4]
100 #set_property -dict ( PACKAGE_PIN D17 IOSTANDARD LVCMOS33 ) [get_ports ( JA[7] )]; #IO_L16N_T2_A17_15 Sch=wja[7]
101 #set_property -dict ( PACKAGE_PIN E17 IOSTANDARD LVCMOS33 ) [get_ports ( JA[8] )]; #IO_L16P_T2_A22_15 Sch=wja[8]
102 #set_property -dict ( PACKAGE_PIN F18 IOSTANDARD LVCMOS33 ) [get_ports ( JA[9] )]; #IO_L22N_T3_A16_15 Sch=wja[9]
103 #set_property -dict ( PACKAGE_PIN G18 IOSTANDARD LVCMOS33 ) [get_ports ( JA[10] )]; #IO_L22P_T3_A17_15 Sch=wja[10]
104
105
106 ##Pmod Header JB
107
108 #set_property -dict ( PACKAGE_PIN D14 IOSTANDARD LVCMOS33 ) [get_ports ( JB[1] )]; #IO_L11P_TO_A0P_18 Sch=wjb[1]
109 #set_property -dict ( PACKAGE_PIN F16 IOSTANDARD LVCMOS33 ) [get_ports ( JB[2] )]; #IO_L14N_T2_SRCC_15 Sch=wjb[2]
110 #set_property -dict ( PACKAGE_PIN G16 IOSTANDARD LVCMOS33 ) [get_ports ( JB[3] )]; #IO_L13N_T2_MRCC_15 Sch=wjb[3]
111 #set_property -dict ( PACKAGE_PIN H14 IOSTANDARD LVCMOS33 ) [get_ports ( JB[4] )]; #IO_L15P_T2_D05_15 Sch=wjb[4]
112 #set_property -dict ( PACKAGE_PIN E16 IOSTANDARD LVCMOS33 ) [get_ports ( JB[7] )]; #IO_L11N_T1_SRCC_15 Sch=wjb[7]
113 #set_property -dict ( PACKAGE_PIN F13 IOSTANDARD LVCMOS33 ) [get_ports ( JB[8] )]; #IO_L15P_TO_A0P_15 Sch=wjb[8]
114 #set_property -dict ( PACKAGE_PIN G13 IOSTANDARD LVCMOS33 ) [get_ports ( JB[9] )]; #IO_G_15 Sch=wjb[9]
115 #set_property -dict ( PACKAGE_PIN H16 IOSTANDARD LVCMOS33 ) [get_ports ( JB[10] )]; #IO_L13P_T2_MRCC_15 Sch=wjb[10]
116
117 #set_property -dict ( PACKAGE_PIN F6 IOSTANDARD LVCMOS33 ) [get_ports ( JC[2] )]; #IO_L19N_T3_VREF_35 Sch=wjc[2]
118 #set_property -dict ( PACKAGE_PIN J2 IOSTANDARD LVCMOS33 ) [get_ports ( JC[3] )]; #IO_L22N_T3_35 Sch=wjc[3]
119 #set_property -dict ( PACKAGE_PIN G6 IOSTANDARD LVCMOS33 ) [get_ports ( JC[4] )]; #IO_L19P_T3_35 Sch=wjc[4]
120 #set_property -dict ( PACKAGE_PIN E7 IOSTANDARD LVCMOS33 ) [get_ports ( JC[7] )]; #IO_L6P_TO_35 Sch=wjc[7]
121 #set_property -dict ( PACKAGE_PIN J3 IOSTANDARD LVCMOS33 ) [get_ports ( JC[8] )]; #IO_L22P_T3_35 Sch=wjc[8]
122 #set_property -dict ( PACKAGE_PIN J4 IOSTANDARD LVCMOS33 ) [get_ports ( JC[9] )]; #IO_L21P_T3_D05_35 Sch=wjc[9]
123 #set_property -dict ( PACKAGE_PIN E6 IOSTANDARD LVCMOS33 ) [get_ports ( JC[10] )]; #IO_L5P_TO_AD13P_35 Sch=wjc[10]
124
125
126 ##Pmod Header JD
127
128
129
130 #set_property -dict ( PACKAGE_PIN H4 IOSTANDARD LVCMOS33 ) [get_ports ( JD[1] )]; #IO_L21N_T3_D05_35 Sch=wjd[1]
131 #set_property -dict ( PACKAGE_PIN H1 IOSTANDARD LVCMOS33 ) [get_ports ( JD[2] )]; #IO_L17P_T2_35 Sch=wjd[2]
132 #set_property -dict ( PACKAGE_PIN G1 IOSTANDARD LVCMOS33 ) [get_ports ( JD[3] )]; #IO_L17N_T2_35 Sch=wjd[3]
133 #set_property -dict ( PACKAGE_PIN G3 IOSTANDARD LVCMOS33 ) [get_ports ( JD[4] )]; #IO_L20N_T3_35 Sch=wjd[4]
134 #set_property -dict ( PACKAGE_PIN H2 IOSTANDARD LVCMOS33 ) [get_ports ( JD[7] )]; #IO_L15P_T2_D05_35 Sch=wjd[7]
135 #set_property -dict ( PACKAGE_PIN G4 IOSTANDARD LVCMOS33 ) [get_ports ( JD[8] )]; #IO_L20P_T3_35 Sch=wjd[8]
136 #set_property -dict ( PACKAGE_PIN G2 IOSTANDARD LVCMOS33 ) [get_ports ( JD[9] )]; #IO_L15N_T2_D05_35 Sch=wjd[9]
137 #set_property -dict ( PACKAGE_PIN F3 IOSTANDARD LVCMOS33 ) [get_ports ( JD[10] )]; #IO_L13N_T2_MRCC_35 Sch=wjd[10]
138
139
140
141
142 ##Pmod Header JXADC
143
144 #set_property -dict ( PACKAGE_PIN A14 IOSTANDARD LVDS ) [get_ports ( XA_N[1] )]; #IO_L8N_T1_D05_AD3N_15 Sch=xa_n[1]
145 #set_property -dict ( PACKAGE_PIN A13 IOSTANDARD LVDS ) [get_ports ( XA_P[1] )]; #IO_L9P_T1_D05_AD3P_15 Sch=xa_p[1]
146 #set_property -dict ( PACKAGE_PIN A16 IOSTANDARD LVDS ) [get_ports ( XA_N[2] )]; #IO_L8N_T1_AD10N_15 Sch=xa_n[2]
147 #set_property -dict ( PACKAGE_PIN A15 IOSTANDARD LVDS ) [get_ports ( XA_P[2] )]; #IO_L9P_T1_AD10P_15 Sch=xa_p[2]
148 #set_property -dict ( PACKAGE_PIN B17 IOSTANDARD LVDS ) [get_ports ( XA_N[3] )]; #IO_L7N_T1_AD2N_15 Sch=xa_n[3]
149 #set_property -dict ( PACKAGE_PIN B16 IOSTANDARD LVDS ) [get_ports ( XA_P[3] )]; #IO_L7P_T1_AD2P_15 Sch=xa_p[3]
150 #set_property -dict ( PACKAGE_PIN A18 IOSTANDARD LVDS ) [get_ports ( XA_N[4] )]; #IO_L10N_T1_AD11N_15 Sch=xa_n[4]
151 #set_property -dict ( PACKAGE_PIN B18 IOSTANDARD LVDS ) [get_ports ( XA_P[4] )]; #IO_L10P_T1_AD11P_15 Sch=xa_p[4]
152
153
154 ##VGA Connector
155
156 #set_property -dict ( PACKAGE_PIN A3 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_R[0] )]; #IO_L8N_T1_AD14N_35 Sch=wvga_r[0]
157 #set_property -dict ( PACKAGE_PIN B4 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_R[1] )]; #IO_L7N_T1_AD6N_35 Sch=wvga_r[1]
158 #set_property -dict ( PACKAGE_PIN C5 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_R[2] )]; #IO_L1N_TO_AD4N_35 Sch=wvga_r[2]
159 #set_property -dict ( PACKAGE_PIN A4 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_R[3] )]; #IO_L9P_T1_AD14P_35 Sch=wvga_r[3]
160
161 #set_property -dict ( PACKAGE_PIN C6 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_G[0] )]; #IO_L1P_TO_AD4P_35 Sch=wvga_g[0]
162 #set_property -dict ( PACKAGE_PIN A5 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_G[1] )]; #IO_L3N_TO_D05_AD8N_35 Sch=wvga_g[1]
163 #set_property -dict ( PACKAGE_PIN B6 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_G[2] )]; #IO_L2N_TO_AD12N_35 Sch=wvga_g[2]
164 #set_property -dict ( PACKAGE_PIN A6 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_G[3] )]; #IO_L3P_TO_D05_AD8P_35 Sch=wvga_g[3]
165
166 #set_property -dict ( PACKAGE_PIN B7 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_B[0] )]; #IO_L2P_TO_AD12P_35 Sch=wvga_b[0]
167 #set_property -dict ( PACKAGE_PIN C7 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_B[1] )]; #IO_L4N_TO_35 Sch=wvga_b[1]
168 #set_property -dict ( PACKAGE_PIN D7 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_B[2] )]; #IO_L6N_TO_VREF_35 Sch=wvga_b[2]
169 #set_property -dict ( PACKAGE_PIN D8 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_B[3] )]; #IO_L4P_TO_35 Sch=wvga_b[3]
170
171 #set_property -dict ( PACKAGE_PIN B11 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_HS )]; #IO_L4P_TO_15 Sch=wvga_hs
172 #set_property -dict ( PACKAGE_PIN B12 IOSTANDARD LVCMOS33 ) [get_ports ( VGA_VS )]; #IO_L3N_TO_D05_AD1N_15 Sch=wvga_vs
173
174
175 ##Micro SD Connector
176
177 #set_property -dict ( PACKAGE_PIN E2 IOSTANDARD LVCMOS33 ) [get_ports ( SD_RESET )]; #IO_L14P_T2_SRCC_35 Sch=sd_reset
178 #set_property -dict ( PACKAGE_PIN A1 IOSTANDARD LVCMOS33 ) [get_ports ( SD_CD )]; #IO_L8N_T1_D05_AD3N_15 Sch=sd_cd
179 #set_property -dict ( PACKAGE_PIN B1 IOSTANDARD LVCMOS33 ) [get_ports ( SD_SCK )]; #IO_L9P_T1_D05_AD7P_35 Sch=sd_sck
180 #set_property -dict ( PACKAGE_PIN C1 IOSTANDARD LVCMOS33 ) [get_ports ( SD_CMD )]; #IO_L16N_T2_35 Sch=sd_cmd

```

```

181 #set_property -dict ( PACKAGE_PIN C2 IOSTANDARD LVCMOS33 ) [get_ports ( SD_DAT[0] )]; #IO_L16P_T2_35 Sch=sd_dat[0]
182 #set_property -dict ( PACKAGE_PIN E1 IOSTANDARD LVCMOS33 ) [get_ports ( SD_DAT[1] )]; #IO_L18N_T2_35 Sch=sd_dat[1]
183 #set_property -dict ( PACKAGE_PIN F1 IOSTANDARD LVCMOS33 ) [get_ports ( SD_DAT[2] )]; #IO_L18P_T2_35 Sch=sd_dat[2]
184 #set_property -dict ( PACKAGE_PIN D2 IOSTANDARD LVCMOS33 ) [get_ports ( SD_DAT[3] )]; #IO_L14N_T2_SRCC_35 Sch=sd_dat[3]
185
186
187 ##Accelerometer
188
189 #set_property -dict ( PACKAGE_PIN E15 IOSTANDARD LVCMOS33 ) [get_ports ( ACL_MISO )]; #IO_L11P_T1_SRCC_15 Sch=acl_miso
190 #set_property -dict ( PACKAGE_PIN F14 IOSTANDARD LVCMOS33 ) [get_ports ( ACL_MOSI )]; #IO_L5N_TO_AD0W_15 Sch=acl_mosi
191 #set_property -dict ( PACKAGE_PIN F15 IOSTANDARD LVCMOS33 ) [get_ports ( ACL_SCLK )]; #IO_L14P_T2_SRCC_15 Sch=acl_sclk
192 #set_property -dict ( PACKAGE_PIN D15 IOSTANDARD LVCMOS33 ) [get_ports ( ACL_CSN )]; #IO_L12P_T1_MRCC_15 Sch=acl_csn
193 #set_property -dict ( PACKAGE_PIN B13 IOSTANDARD LVCMOS33 ) [get_ports ( ACL_INT[1] )]; #IO_L2P_TO_AD0P_15 Sch=acl_int[1]
194 #set_property -dict ( PACKAGE_PIN C16 IOSTANDARD LVCMOS33 ) [get_ports ( ACL_INT[2] )]; #IO_L20P_T3_A20_15 Sch=acl_int[2]
195
196
197 ##Temperature Sensor
198
199 #set_property -dict ( PACKAGE_PIN C14 IOSTANDARD LVCMOS33 ) [get_ports ( TMP_SCL )]; #IO_L1N_TO_AD0N_15 Sch=tmp_scl
200 #set_property -dict ( PACKAGE_PIN C15 IOSTANDARD LVCMOS33 ) [get_ports ( TMP_SDA )]; #IO_L12N_T1_MRCC_15 Sch=tmp_sda
201 #set_property -dict ( PACKAGE_PIN D13 IOSTANDARD LVCMOS33 ) [get_ports ( TMP_INT )]; #IO_L6N_TO_VREF_15 Sch=tmp_int
202 #set_property -dict ( PACKAGE_PIN B14 IOSTANDARD LVCMOS33 ) [get_ports ( TMP_CT )]; #IO_L2N_TO_AD0N_15 Sch=tmp_ct
203
204 ##Omnidirectional Microphone
205
206 #set_property -dict ( PACKAGE_PIN J5 IOSTANDARD LVCMOS33 ) [get_ports ( M_CLK )]; #IO_25_35 Sch=m_clk
207 ##PWM Audio Amplifier
208
209 #set_property -dict ( PACKAGE_PIN A11 IOSTANDARD LVCMOS33 ) [get_ports ( AUD_PWM )]; #IO_L4N_TO_15 Sch=aud_pwm
210 #set_property -dict ( PACKAGE_PIN D12 IOSTANDARD LVCMOS33 ) [get_ports ( AUD_SD )]; #IO_L6P_TO_15 Sch=aud_sd
211
212
213 ##USB-RS232 Interface
214
215
216
217 #set_property -dict ( PACKAGE_PIN C4 IOSTANDARD LVCMOS33 ) [get_ports ( UART_TXD_IN )]; #IO_L7P_T1_AD0P_35 Sch=uart_txd_in
218 #set_property -dict ( PACKAGE_PIN D4 IOSTANDARD LVCMOS33 ) [get_ports ( UART_RXD_OUT )]; #IO_L11N_T1_SRCC_35 Sch=uart_rxd_out
219 #set_property -dict ( PACKAGE_PIN D3 IOSTANDARD LVCMOS33 ) [get_ports ( UART_CTS )]; #IO_L12N_T1_MRCC_35 Sch=uart_cts
220 #set_property -dict ( PACKAGE_PIN E5 IOSTANDARD LVCMOS33 ) [get_ports ( UART_RTS )]; #IO_L5N_TO_AD13N_35 Sch=uart_rts
221
222
223
224 ##USB HID (PS/2)
225
226 #set_property -dict ( PACKAGE_PIN F4 IOSTANDARD LVCMOS33 ) [get_ports ( PS2_CLK )]; #IO_L13P_T2_MRCC_35 Sch=ps2_clk
227 #set_property -dict ( PACKAGE_PIN B2 IOSTANDARD LVCMOS33 ) [get_ports ( PS2_DATA )]; #IO_L10N_T1_AD15N_35 Sch=ps2_data
228
229
230 ##SMSC Ethernet PHY
231
232 #set_property -dict ( PACKAGE_PIN C9 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_MDC )]; #IO_L11P_T1_SRCC_16 Sch=eth_mdc
233 #set_property -dict ( PACKAGE_PIN A9 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_MDIO )]; #IO_L14N_T2_SRCC_16 Sch=eth_mdio
234 #set_property -dict ( PACKAGE_PIN B3 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_RSTN )]; #IO_L10P_T1_AD15P_35 Sch=eth_rstn
235 #set_property -dict ( PACKAGE_PIN D9 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_CRSDV )]; #IO_L6N_TO_VREF_16 Sch=eth_crsv
236 #set_property -dict ( PACKAGE_PIN C10 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_RXERR )]; #IO_L13N_T2_MRCC_16 Sch=eth_rxerr
237 #set_property -dict ( PACKAGE_PIN C11 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_RXD[0] )]; #IO_L13P_T2_MRCC_16 Sch=eth_rxd[0]
238 #set_property -dict ( PACKAGE_PIN D10 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_RXD[1] )]; #IO_L19N_T3_VREF_16 Sch=eth_rxd[1]
239 #set_property -dict ( PACKAGE_PIN B9 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_TXEN )]; #IO_L11N_T1_SRCC_16 Sch=eth_txen
240 #set_property -dict ( PACKAGE_PIN A10 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_TXD[0] )]; #IO_L14P_T2_SRCC_16 Sch=eth_txd[0]
241 #set_property -dict ( PACKAGE_PIN A8 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_TXD[1] )]; #IO_L12N_T1_MRCC_16 Sch=eth_txd[1]
242 #set_property -dict ( PACKAGE_PIN D5 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_REFCLK )]; #IO_L11P_T1_SRCC_35 Sch=eth_refclk
243 #set_property -dict ( PACKAGE_PIN B8 IOSTANDARD LVCMOS33 ) [get_ports ( ETH_INTN )]; #IO_L12P_T1_MRCC_16 Sch=eth_intn
244
245
246 ##Quad SPI Flash
247
248 #set_property -dict ( PACKAGE_PIN K17 IOSTANDARD LVCMOS33 ) [get_ports ( QSPI_DQ[0] )]; #IO_L1P_TO_D00_MOSI_14 Sch=qspi_dq[0]
249 #set_property -dict ( PACKAGE_PIN K18 IOSTANDARD LVCMOS33 ) [get_ports ( QSPI_DQ[1] )]; #IO_L1N_TO_D01_DIN_14 Sch=qspi_dq[1]
250 #set_property -dict ( PACKAGE_PIN L14 IOSTANDARD LVCMOS33 ) [get_ports ( QSPI_DQ[2] )]; #IO_L2P_TO_D02_14 Sch=qspi_dq[2]
251 #set_property -dict ( PACKAGE_PIN M14 IOSTANDARD LVCMOS33 ) [get_ports ( QSPI_DQ[3] )]; #IO_L1N_TO_D03_14 Sch=qspi_dq[3]
252 #set_property -dict ( PACKAGE_PIN L13 IOSTANDARD LVCMOS33 ) [get_ports ( QSPI_CSN )]; #IO_L6P_TO_FCS_B_14 Sch=qspi_csn

```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [1] J. C. Coudeyras, "Radiation testing of the Configurable Fault Tolerant Processor (CFTP) for space-based applications," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2005.
- [2] C. J. Humberd, "A compression algorithm for field programmable gate arrays in the space environment," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2011.
- [3] R. F. Bernstein, Jr. "A pipelined vector processing and memory architecture for cyclostationary processing," Ph.D. dissertation, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 1995.
- [4] A. V. Oppenheim and R. W. Schaffer, *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, USA, 1989.
- [5] D. A. Ebert, "Design and development of a Configurable Fault-Tolerant Processor (CFTP) for space applications," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2003.
- [6] MidSTAR. (n.d.). [Online]. Available FTP: \\it154512\CFTP Directory: DesignDocs\Midstar File: midstar_external_view
- [7] D Sakoda. (n.d.). NPSAT-1 Spacecraft architecture and technology demonstration satellite [Online]. Available FTP: \\it154512\CFTP Directory: NPSat1\Sponsor091806 File: NPSAT1_Brief
- [8] P. J. Majewicz, "Implementation of a Configurable Fault Tolerant Processor (CFTP) using Internal Triple Modular Redundancy (TMR)," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2005.
- [9] G. W. Caldwell, "Implementation of Configurable Fault Tolerant Processor (CFTP) experiments," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2006.
- [10] D. E. Dwiggins, "Fault tolerant microcontroller for the configurable Fault Tolerant Processor," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2008.
- [11] M. A. Sullivan, "Reduced precision redundancy applied to arithmetic operations in field programmable gate arrays for satellite control and sensor systems," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2008.

- [12] J. D. Snodgrass, "Low-power fault tolerance for spacecraft FPGA-based numerical computing," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2009.
- [13] J. V. Livingston, "A field programmable gate array based software defined radio design for the space environment," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2009.
- [14] A. S. Jackson, "Implementation of the configurable fault tolerant system experiment on NPSAT-1," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, March 2016.
- [15] M. L. Zimmer, "A VLSI design of a radix-4 floating point FFT butterfly," M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 1991.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California