2018-06

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

Walker, Alan A. III

Monterey, CA; Naval Postgraduate School


# 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

THIS PAGE INTENTIONALLY LEFT BLANK

| REPORT DOCUMENTATION PAGE | | | *Form Approved OMB No. 0704-0188* |
|---|---|---|---|



| **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

# LIST OF FIGURES

# LIST OF TABLES

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.

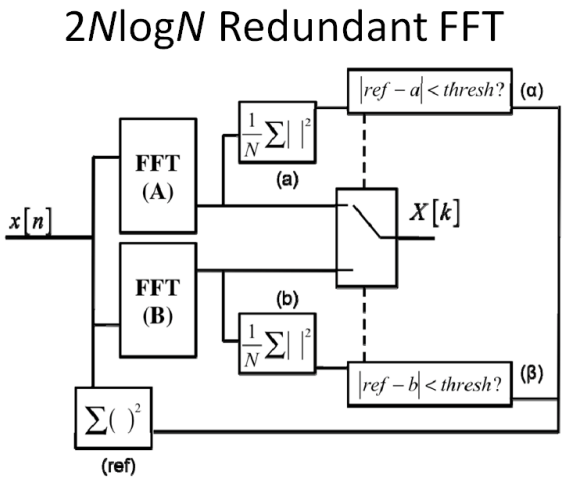
In 2005, Coudeyras [1] tested and demonstrated that radiation in the space environment can cause single-event upsets (SEU) that can have unknown, sometimes unrecoverable, effects on electrical systems. A fast Fourier transform (FFT) that can be used to implement Parseval's theorem in a way that corrects 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 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.




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

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]

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\left(\frac{2\pi}{N}\right)kn} = \sum_{n=0}^{N-1} x_n W_N^{kn}$$
$$x_n = \frac{1}{N}\sum_{n=0}^{N-1} X_k e^{j\left(\frac{2\pi}{N}\right)kn} = \sum_{n=0}^{N-1} X_k W_N^{-kn} \quad , \qquad (1)$$

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 | $N\log_2 N$ | $10^4$ | $20 \times 10^6$ | $30 \times 10^9$ | $40 \times 10^{12}$ |

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 convertor, 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.

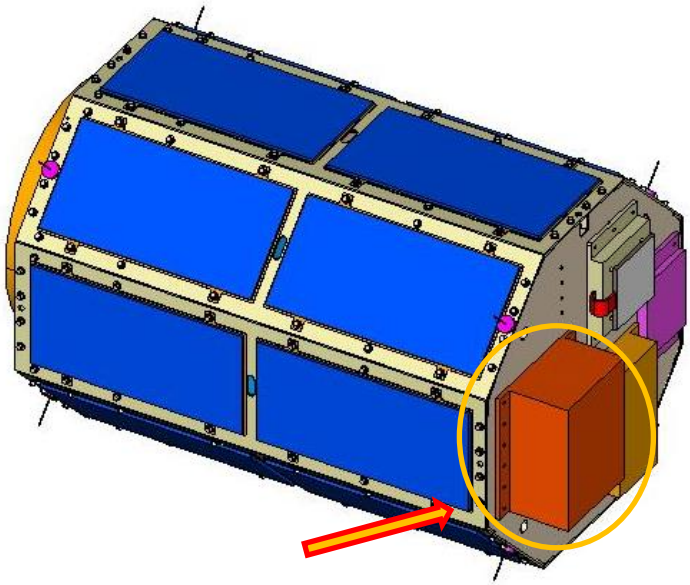


Figure 2. External View of MidStar-1 Showing the Location of CFTP-1. Adapted from [6].

## 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.

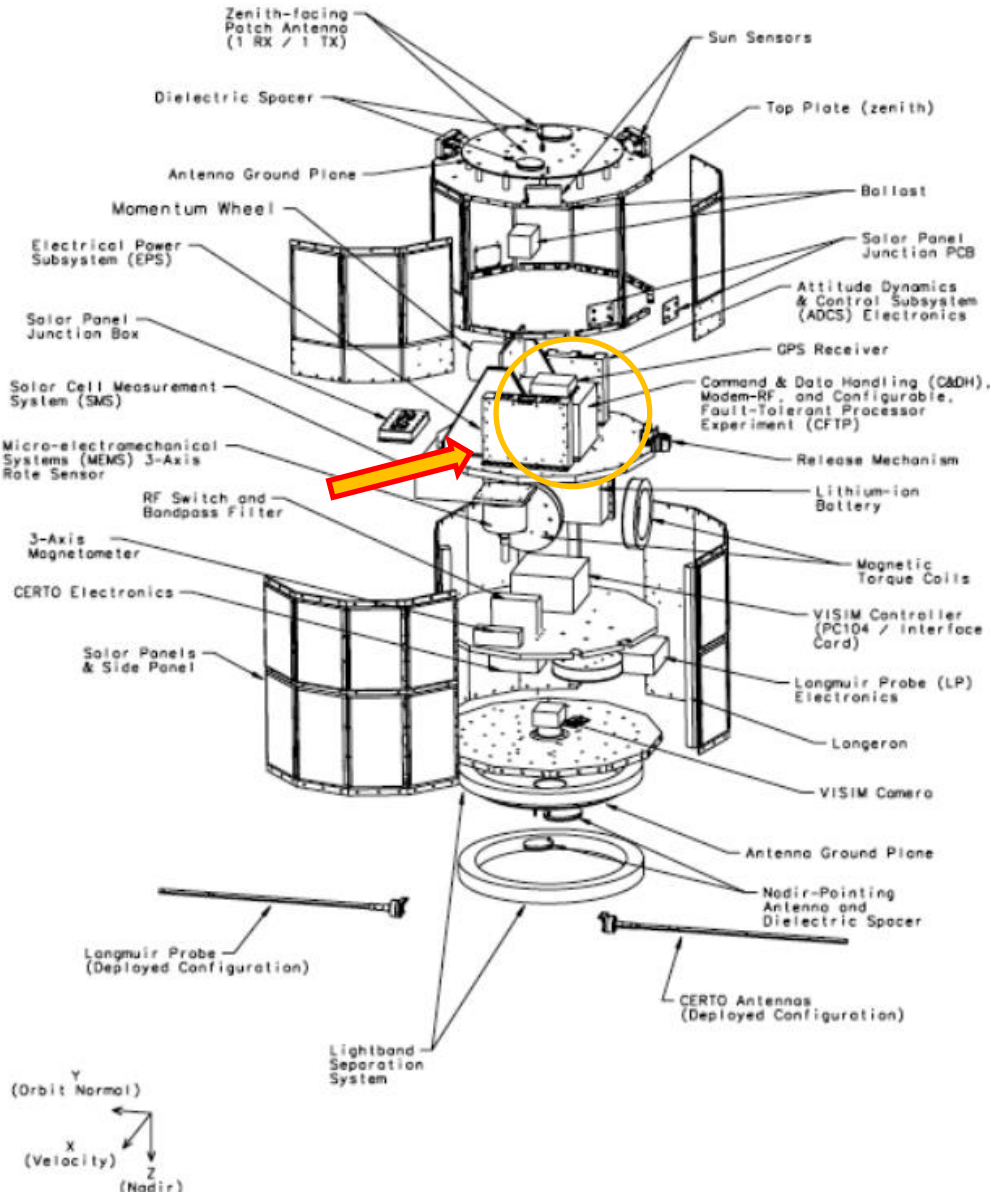



Figure 3. Expanded View of NPSat-1 That Shows Location of CFTP-7. Adapted from [7].

## 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]

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

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.

0101 ( 5)
1010
+ 1
1011 (-5)

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.

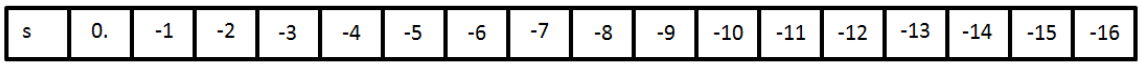



Figure 5. Fixed-Point Signed Binary Representation.
Binary Point Place After Bit 0

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

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

where $b_i$ represents the value of the $i$ numbered bit, $b_0$ represents the value of the $0^{th}$ bit, $\overline{N}$ 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 |
|---|---|---|---|---|---|
| 011111111111111111 | 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 | $\sqrt{2}/2$ | - |
| 000000000000010100 | 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

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

## 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.

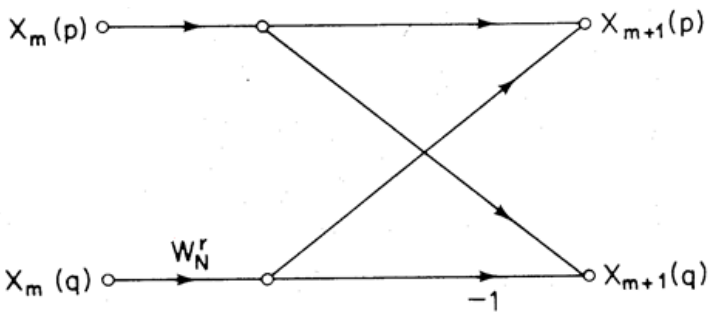



Figure 6. Flow Graph of Simplified Butterfly Machine Computation Requiring Only One Complex Multiplication. Source: [4].

### 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.

$$X_0[000] = x[000] = x_m[p] \qquad X_0[0] = x[0]$$
$$X_0[001] = x[100] = x_m[q] \qquad X_0[1] = x[4]$$
$$X_0[010] = x[010] = x_m[p] \qquad X_0[2] = x[2]$$
$$X_0[011] = x[110] = x_m[q] \qquad X_0[3] = x[6]$$
$$X_0[100] = x[001] = x_m[p] \qquad X_0[4] = x[1]$$
$$X_0[101] = x[101] = x_m[q] \qquad X_0[5] = x[5]$$
$$X_0[110] = x[011] = x_m[p] \qquad X_0[6] = x[3]$$
$$X_0[111] = x[111] = x_m[q] \qquad X_0[7] = x[7]$$

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 Schafer [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 $x_m(q)$.

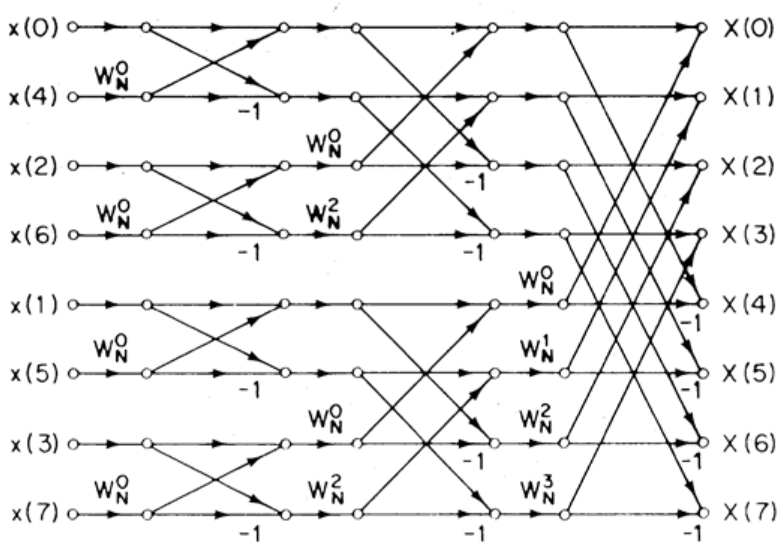



Figure 8. Flow Graph of 8-point DFT Using the Butterfly Machine Computation. Source: [4].

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.

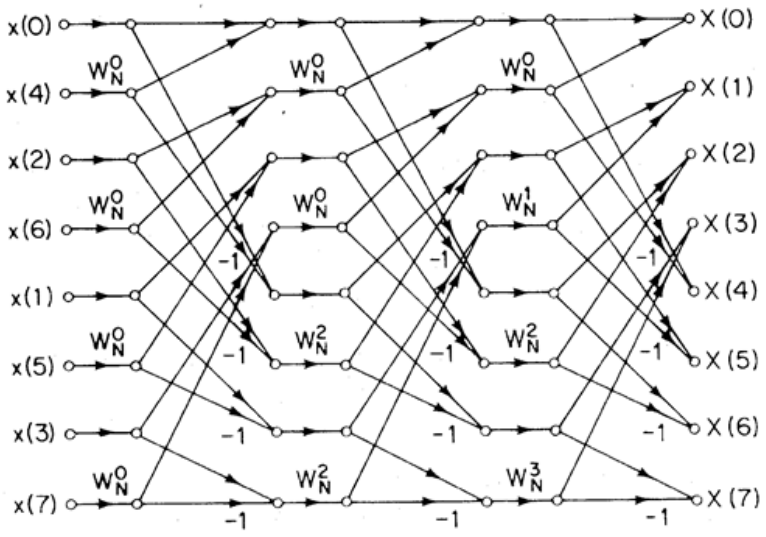



Figure 9. Rearrangement of Figure 1 to Allow Each Stage to Have a Constant Geometry Permitting Sequential Data Accessing and Storage. Source: [4]

## 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.

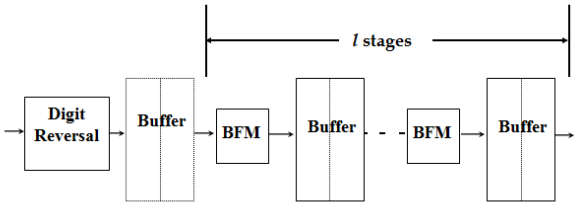



Figure 10. Basic Pipelined FFT Structure. $N = 2^l$, Where $N$ Is the FFT Word Size. Adapted from [15].

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 $k = 0, \frac{N}{2}, 1, \frac{N}{2}+1, 2, \frac{N}{2}+2, 3, \frac{N}{2}+3, 4, \frac{N}{2}+1, \ldots, N-1$ [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)*.

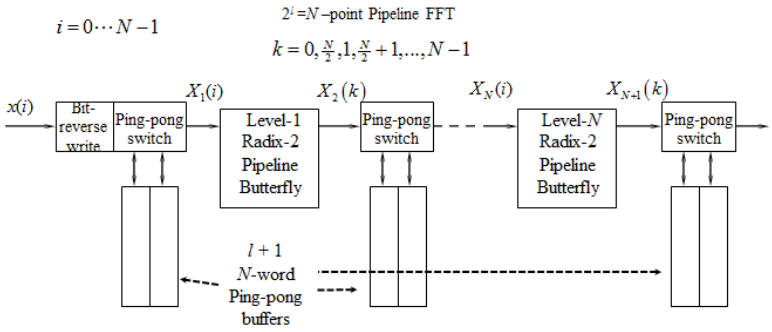



Figure 11. Detailed High Speed Pipelined FFT Structure. Adapted from [15].

### 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.

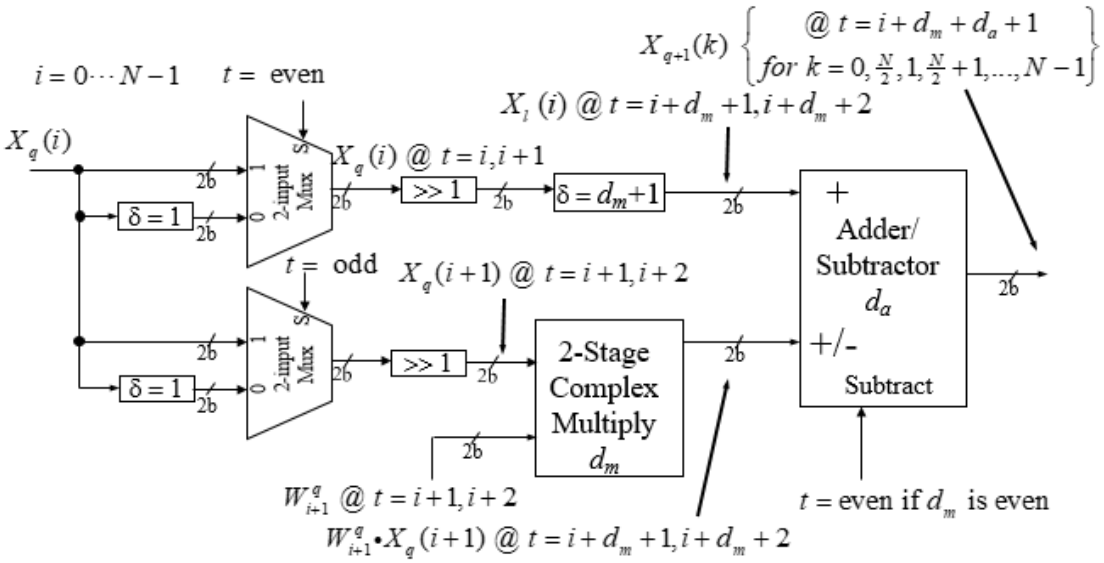



Figure 12. Pipeline Radix-2 Butterfly Machine (BFM) Architecture, Level *q*. Adapted from [15].

## 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.

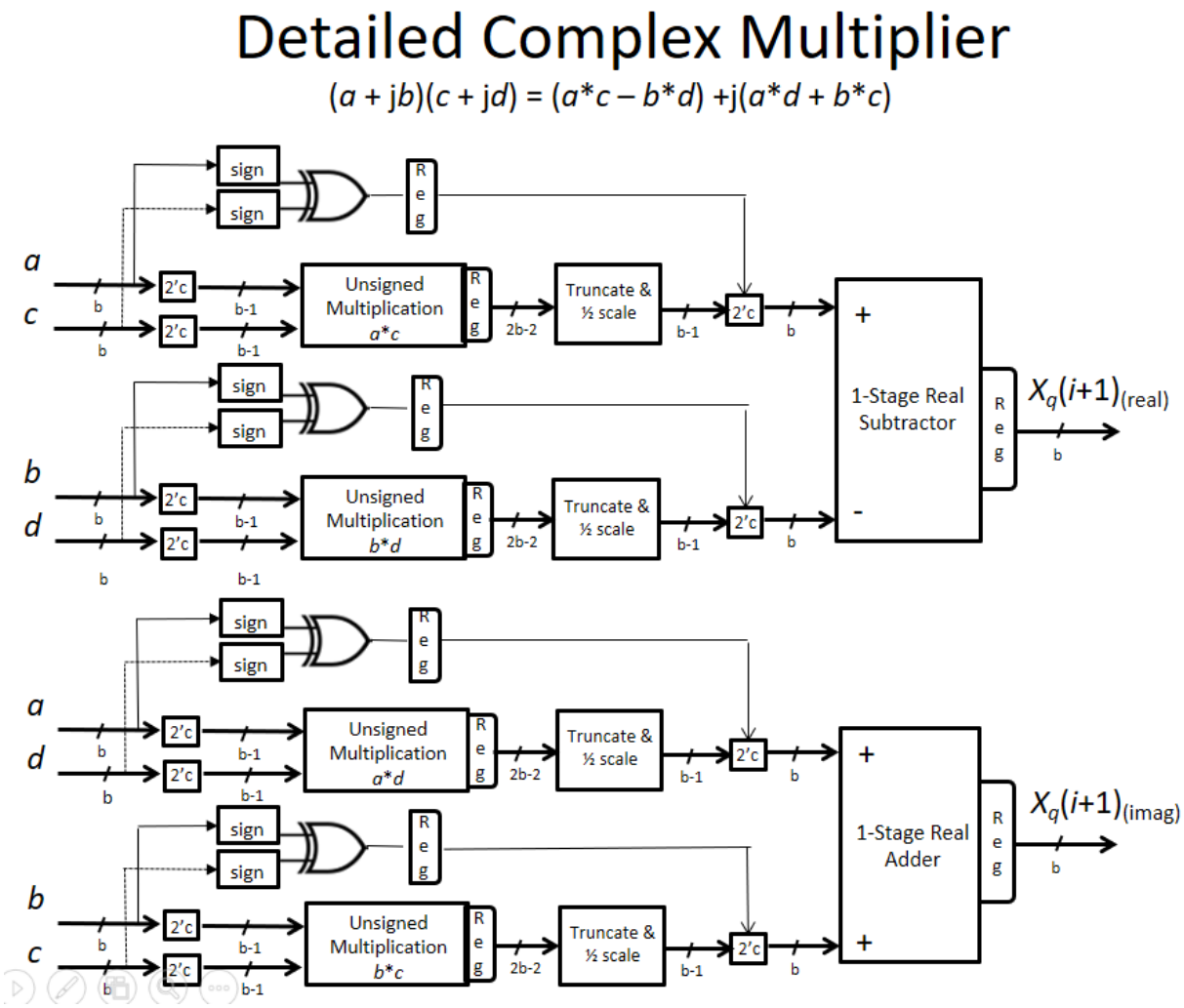



Figure 13. Two-Stage Complex Multiplier

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.

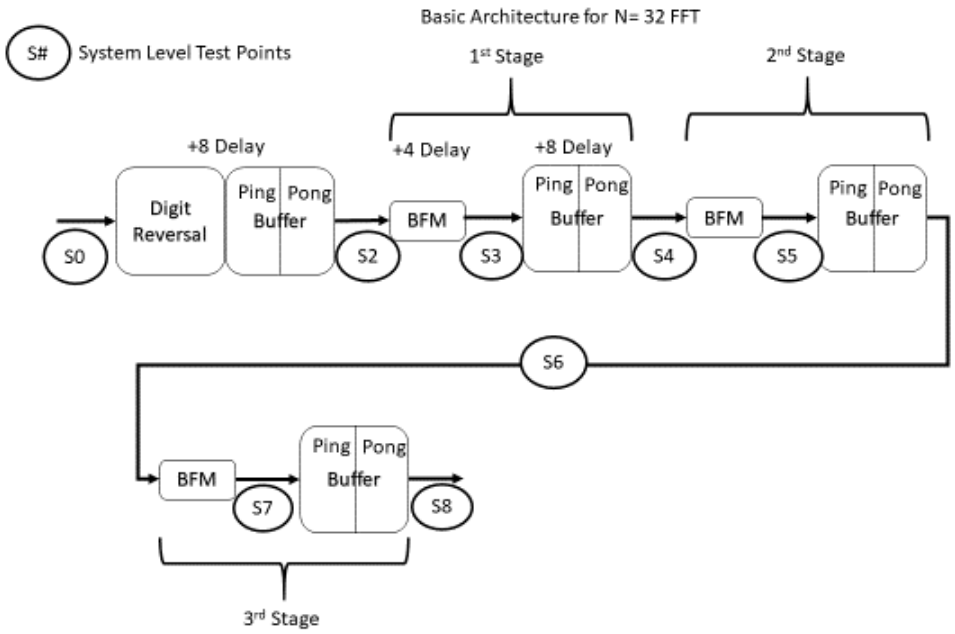



Figure 14. System Level Test Plan for $N = 8$ FFT

## 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 / subtracter. 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.

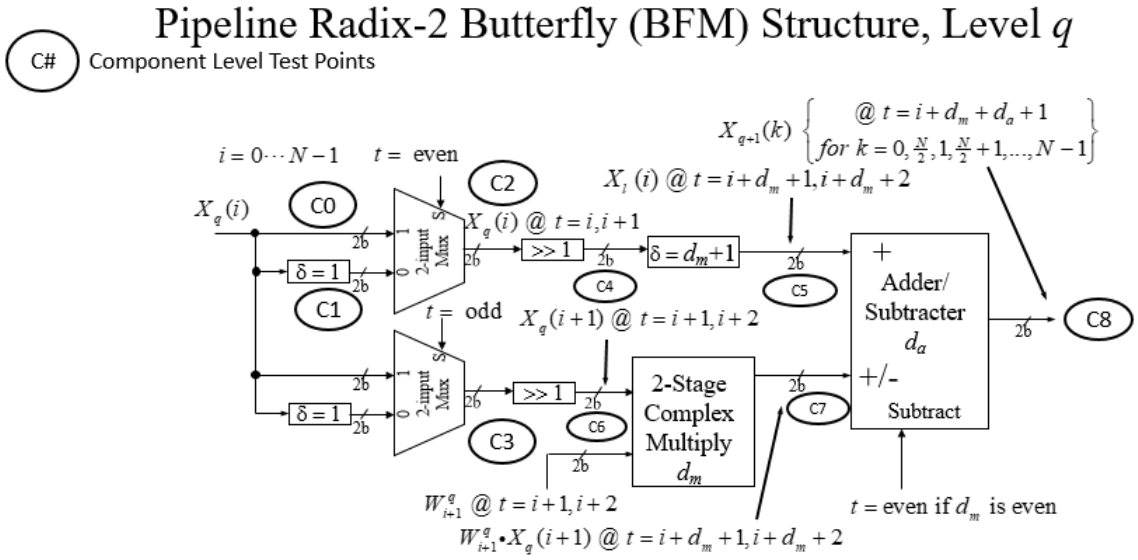



Figure 15. Component Level Test Plan for $N = 8$ BFM.
Adapted from [15].

### 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 $X_q(t)$.

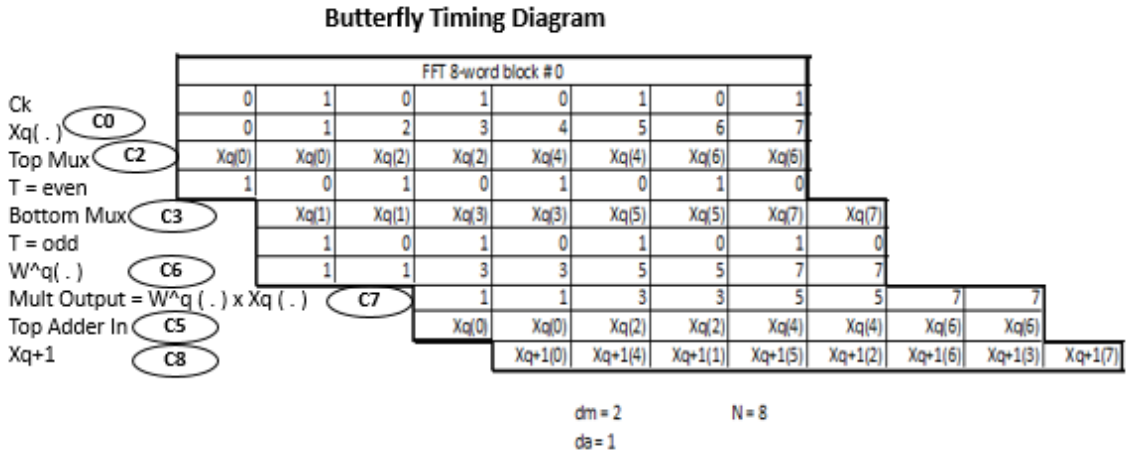



Figure 16. Butterfly Machine Timing Diagram Part A.
Adapted from [15].

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.

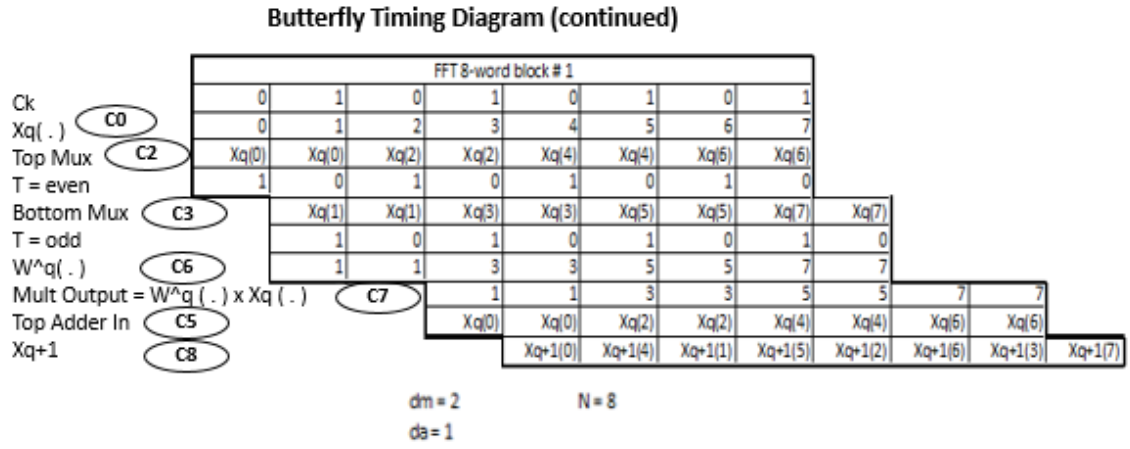



Figure 17. Butterfly Timing Machine Diagram Part B.
Adapted from [15].

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 | 0fffe |
| $X_q(3)$ | 001111111111111101 | .9999542236 | 0fffd |
| $X_q(4)$ | 001111111111111100 | .9999389648 | 0fffc |
| $X_q(5)$ | 001111111111111011 | .9999237060 | 0fffb |
| $X_q(6)$ | 001111111111111010 | .999908447 | 0fffa |
| $X_q(7)$ | 001111111111111001 | .9998931 | 0fff9 |
| $X_q(8)$ | 001111111111111000 | .999877929 | 0fff8 |
| $X_q(9)$ | 001111111111110111 | .9998626708 | 0fff7 |
| $X_q(a)$ | 001111111111110110 | .999847412 | 0fff6 |
| $X_q(b)$ | 001111111111110101 | .99983215 | 0fff5 |
| $X_q(c)$ | 001111111111110100 | .999816894 | 0fff4 |
| $X_q(d)$ | 001111111111110011 | .9998016357 | 0fff3 |
| $X_q(e)$ | 001111111111110010 | .99978637 | 0fff2 |
| $X_q(f)$ | 001111111111110001 | .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 t*ranspose* 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 outputed 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, 0ffff, 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.

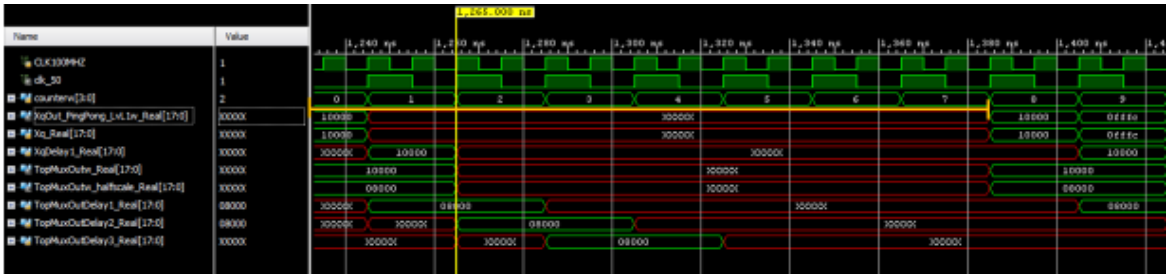


Figure 24. Startup Delay Between Bit-Reversed Ping-Pong Buffer Output and BFM

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.

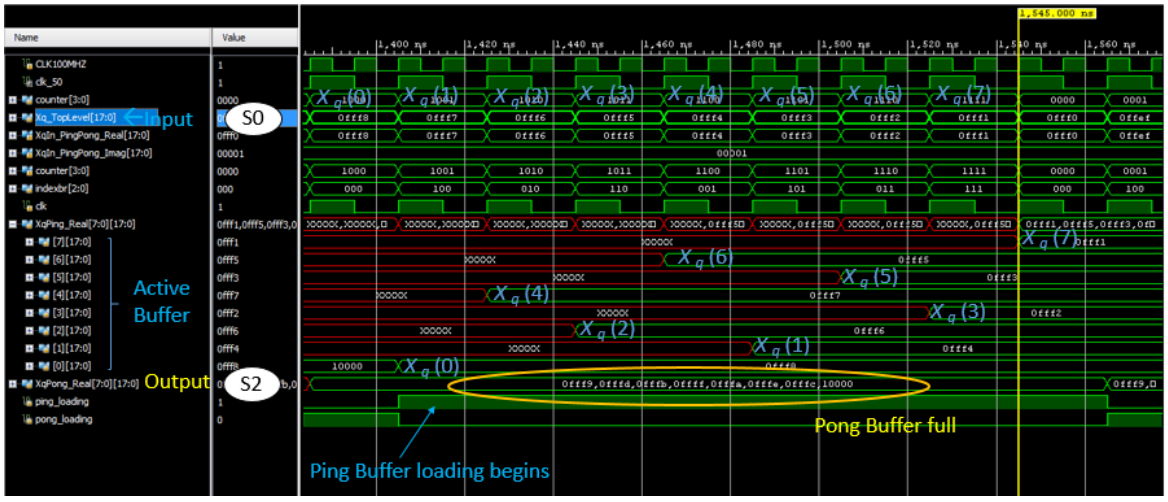



Figure 25. Ping-Buffer Loading in Bit Reversed Order, Pong Full

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

The BFM is the primary processing element of the FFT. BFMs 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.

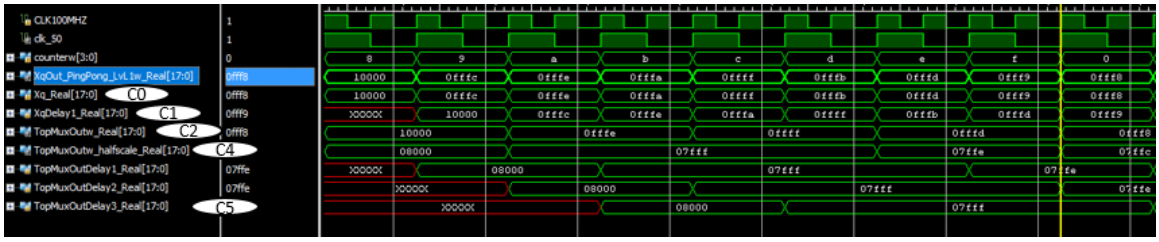



Figure 32. Top Multiplexer Output, Hexadecimal

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 $-2 \le X_q(t) < 2$.

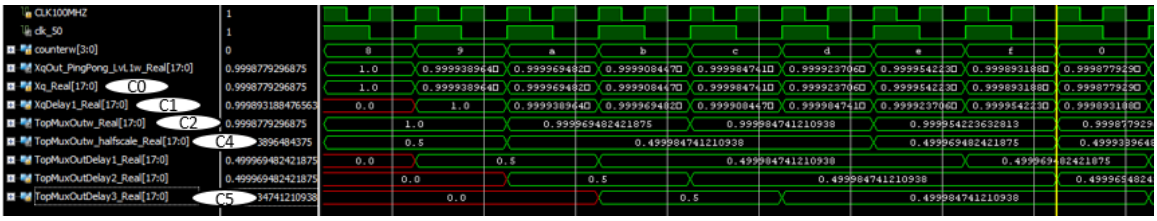



Figure 33. Top Multiplexer Output, Fixed-Point Binary

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

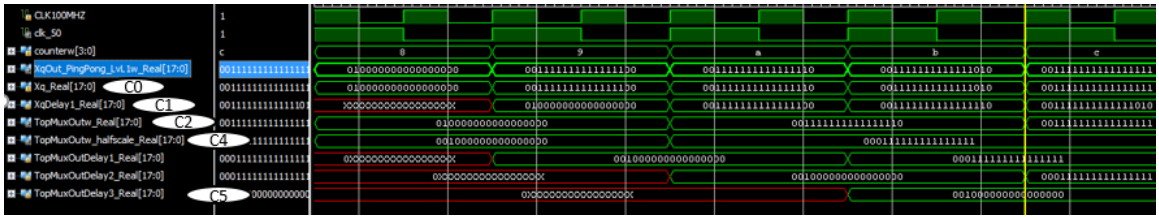



Figure 34. Top Multiplexer Output, Binary

### *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 *t*s. 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.

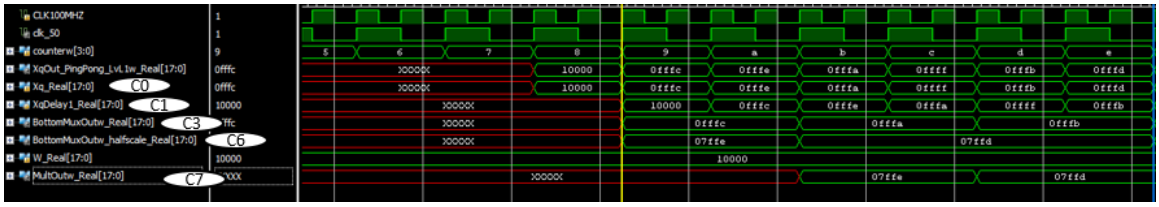



Figure 37. Bottom Multiplexer Output with Multiplier as a Black Box

### *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 resuls 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.

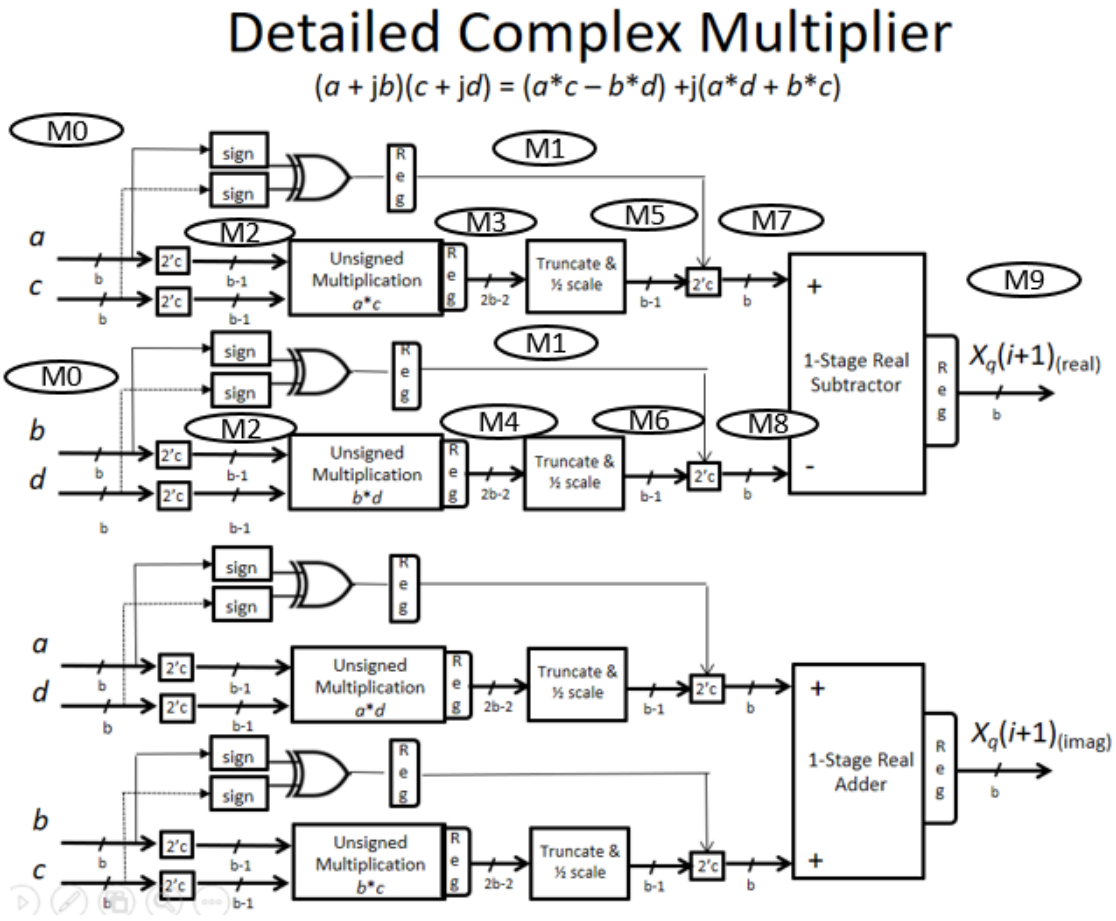



Figure 38. Test Plan for Multiplier and Sub-Component testing

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'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));
```

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/Subtracter 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.

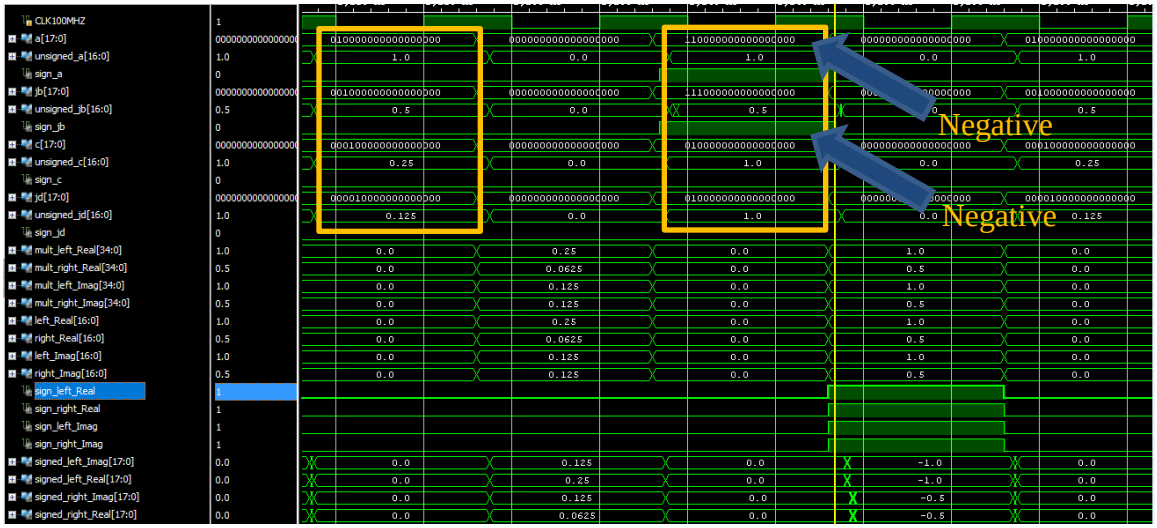



Figure 51. Complex Multiplication Module Testing

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.

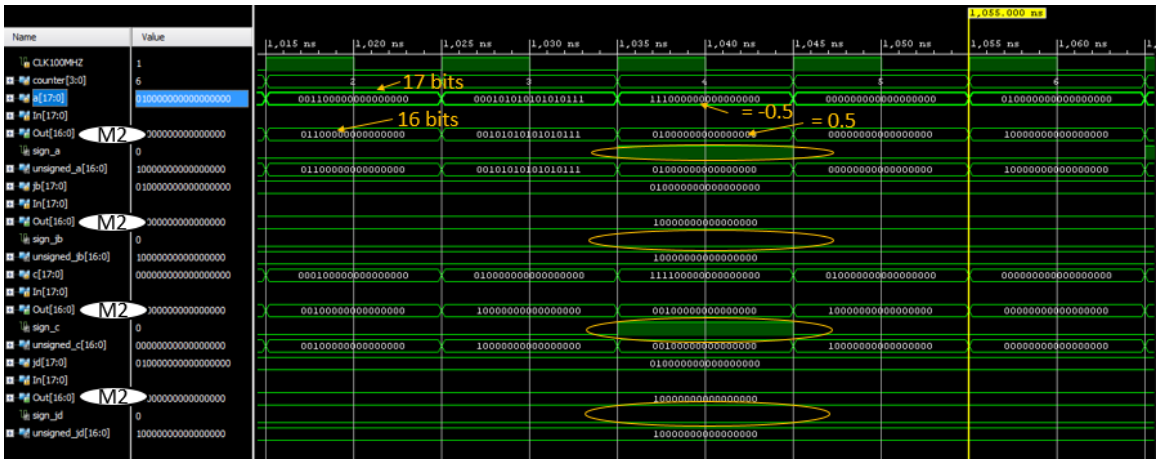



Figure 52. Conversion of Multiplier Inputs to Signed Magnitudes

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.

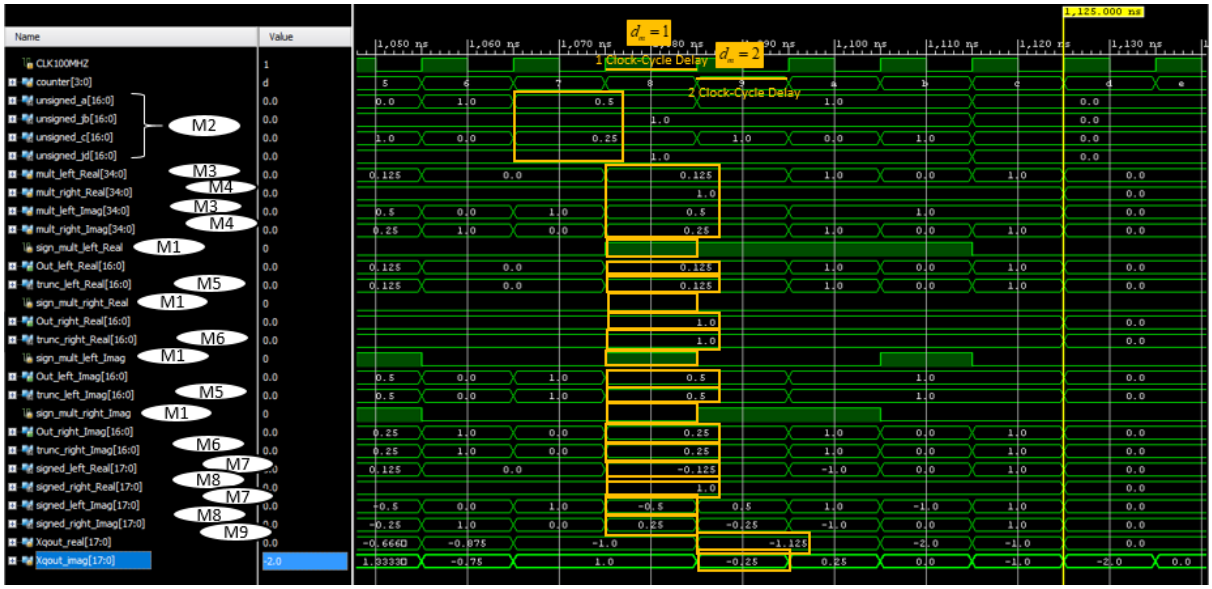



Figure 53. Multiplier Test Points. Fixed Point

### *d. Adder / Subtracter Component Sub-Component Test*

The adder / subtracter 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 / Subtracter Variables Code Snippet

The adder / subtracter 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.

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

Figure 55. Adder / Subtracter Code Snippet

The simulation for the adder / subtracter 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.

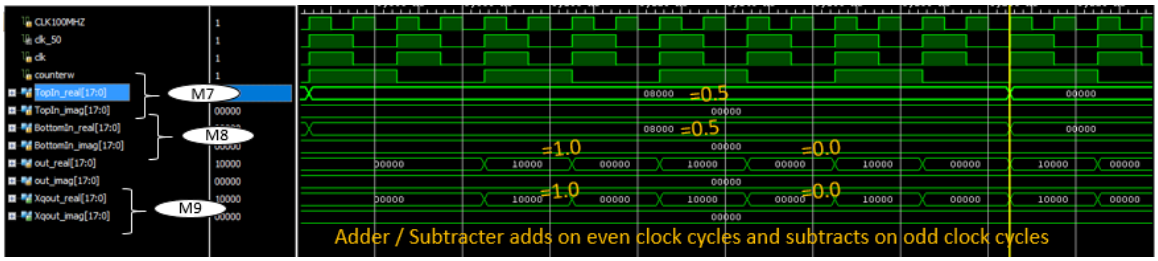



Figure 56. Adder / Subtracter Adds on Even Clock-Cycles

### *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*, *XqPing_Real*, and *XqPing_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.

```
    (* dont_touch = "true" *) reg  [17:0] XqPing_Real [7:0];          // register for Ping real component
    (* dont_touch = "true" *) reg  [17:0] XqPing_Imag [7:0];          // register for Ping imag component
    (* dont_touch = "true" *) reg  [17:0] XqPong_Real [7:0];          // register for Pong real component
    (* dont_touch = "true" *) reg  [17:0] XqPong_Imag [7:0];          // register for Pong imag component
    (* 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 $k = 0, \frac{N}{2}, 1, \frac{N}{2}+1, 2, \frac{N}{2}+2, 3, \frac{N}{2}+3, 4, \frac{N}{2}+1, \ldots, N-1$. 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 $k = 0, \frac{N}{2}, 1, \frac{N}{2}+1, 2, \frac{N}{2}+2, 3, \frac{N}{2}+3, 4, \frac{N}{2}+1, \ldots, N-1$ loading scheme seen in Figure 59.

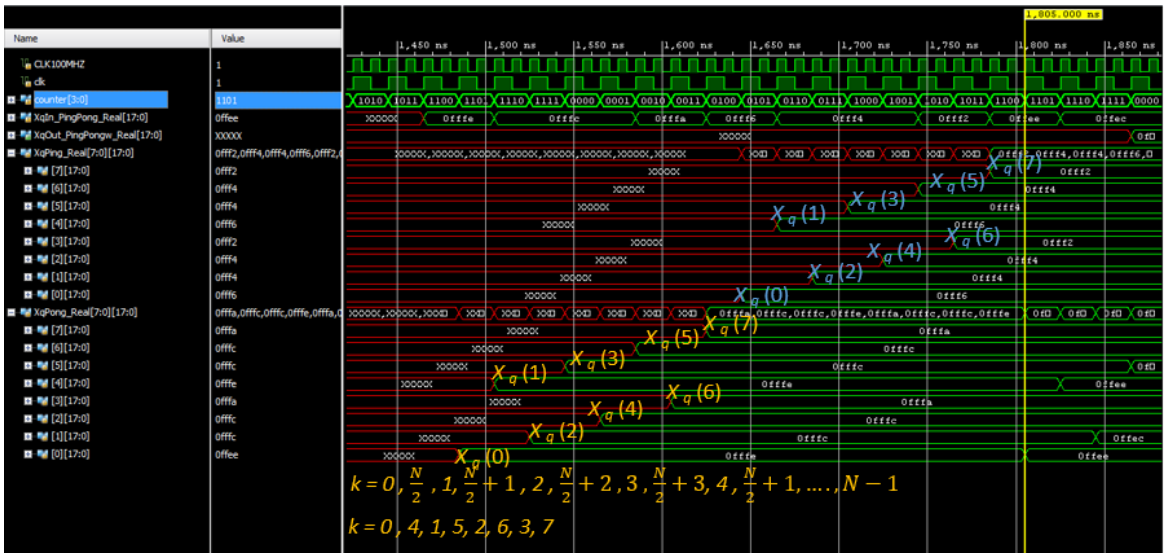



Figure 60. First-Stage Ping-Pong Buffer Simulation

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, -.707} | {0, -0.707, 1, -0.707, 0, 0.707, -1, 0.707} | $\{\frac{1}{8}, -\frac{0.707}{8}, 0, \frac{0.707}{8}, -\frac{1}{8}, \frac{.707}{8}, 0, \frac{.707}{8}\}$ | $\{0, -\frac{0.707}{8}, \frac{1}{8}, -\frac{0.707}{8}, 0, \frac{0.707}{8}, -\frac{1}{8}, \frac{0.707}{8}\}$ |

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.

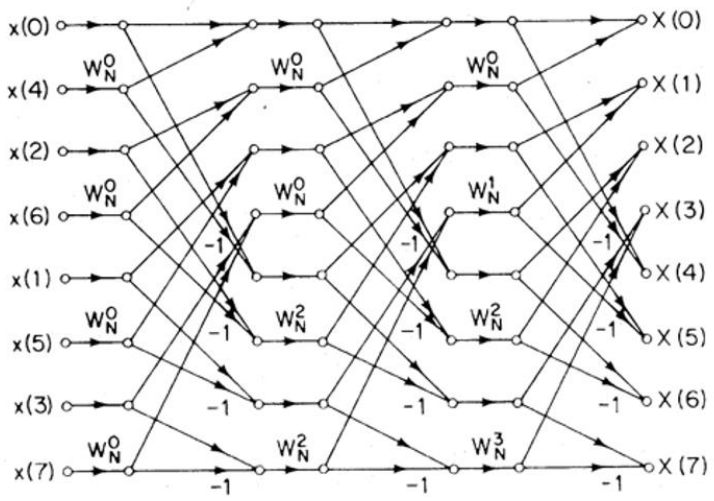



Figure 61. Generic Constant Geometric FFT. Adapted from [4].

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}.

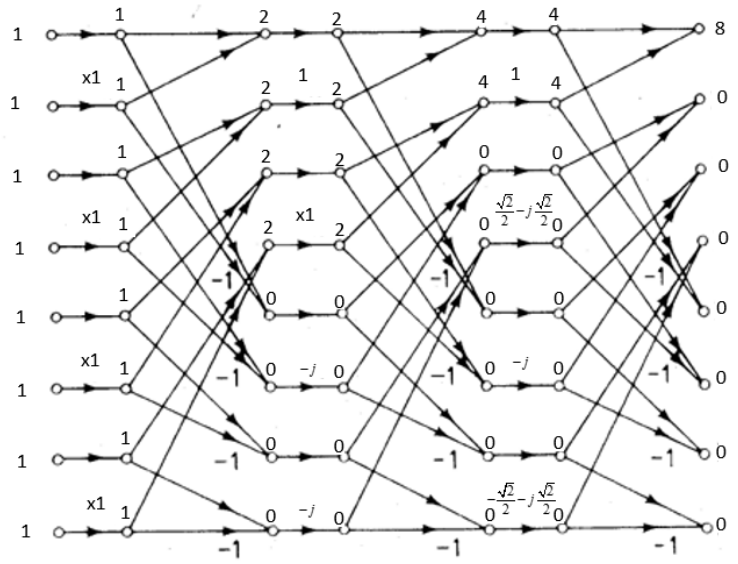


Figure 62. Test Vector-1 {1, 1, 1, 1, 1, 1, 1, 1}. Adapted from [4].

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.

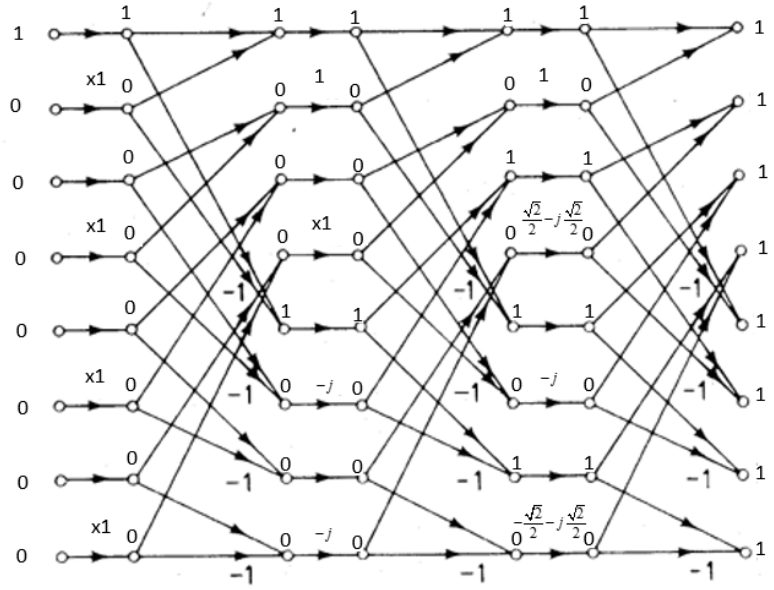


Figure 63. Test Vector-2 {1, 0, 0, 0, 0, 0, 0, 0}. Adapted from [4].

Test Vector-3 is displayed in Figure 64. Inputs of {0, 0, 0, 0, 0, 0, 0, 0} result in {0, 0, 0, 0, 0, 0, 0, 0} being processing through all three stages with a final output of {0, 0, 0, 0, 0, 0, 0, 0} and a scaled result of {0, 0, 0, 0, 0, 0, 0, 0}.

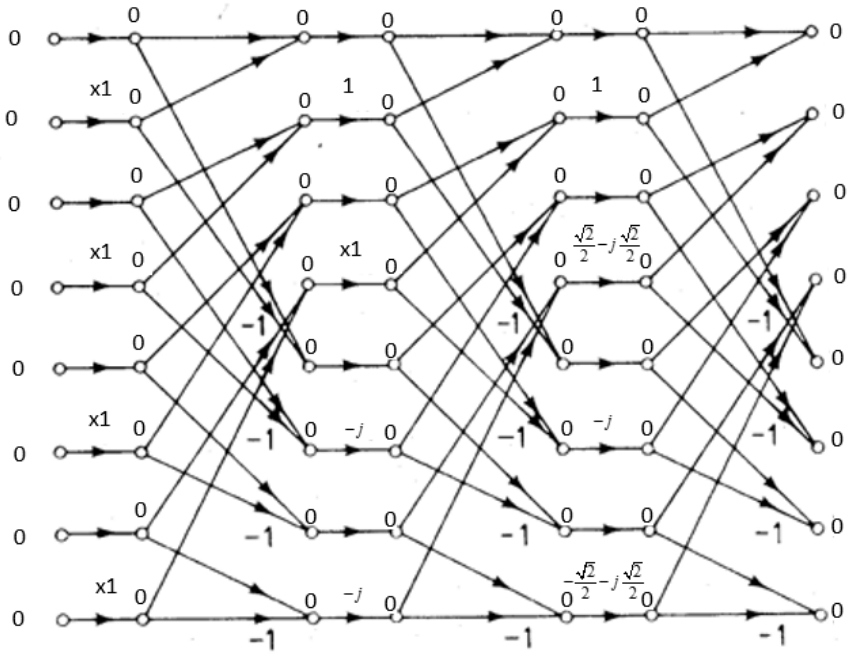



Figure 64. Test Vector-3 {0, 0, 0, 0, 0, 0, 0, 0}. Adapted from [4].

Test Vector-4 began with {0, 0, 0, 1, 0, 0, 0, 0}. First stage processing resulted in {0, 0, 0, 1, 0, *0*, 0, 1,}. Second stage processing produced {0, 1, *0*, *-j*, 0, -1, *0*, *j*}. Third stage processing resulted in {1, $-\frac{\sqrt{2}}{2}-j\frac{\sqrt{2}}{2}$ , *j*, $\frac{\sqrt{2}}{2}-j\frac{\sqrt{2}}{2}$, -1, $\frac{\sqrt{2}}{2}+j\frac{\sqrt{2}}{2}$, *-j*, $-\frac{\sqrt{2}}{2}+j\frac{\sqrt{2}}{2}$ }. This is the only test vector which yields a non-zero imaginary component. Test Vector-4 can be seen in Figure 65.

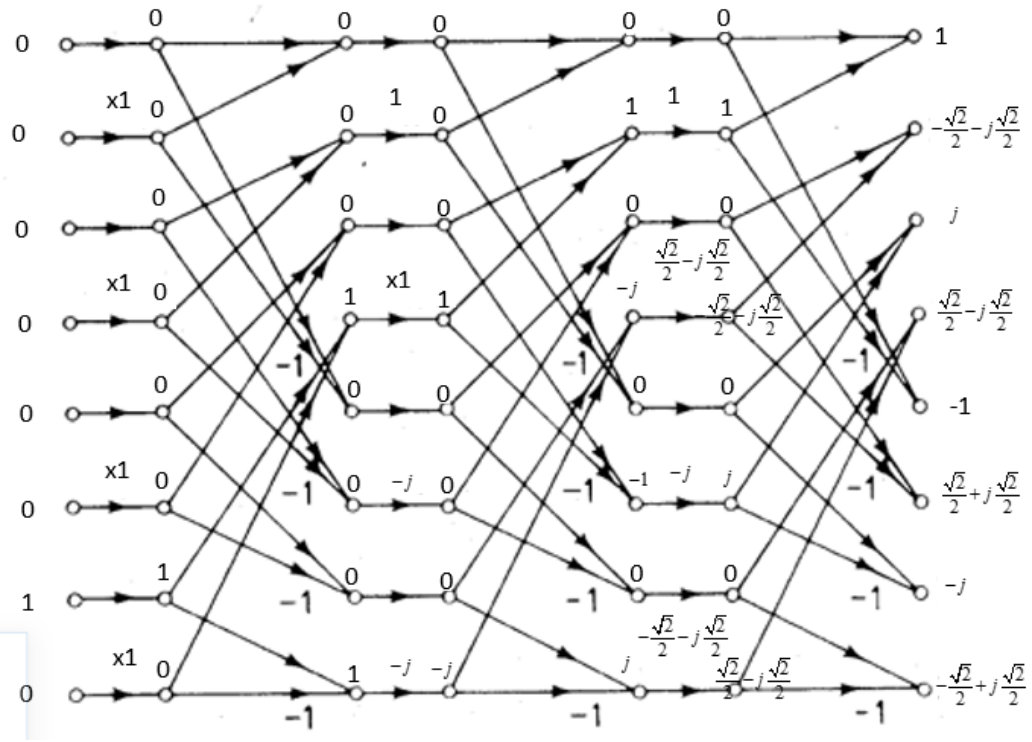


Figure 65. Test Vector-4 {0, 0, 0, 1, 0, 0, 0, 0} Displayed after Bit-Reversed to {0, 0, 0, 0, 0, 0, 1, 0}. Adapted from [4].

## 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 BFMs 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.

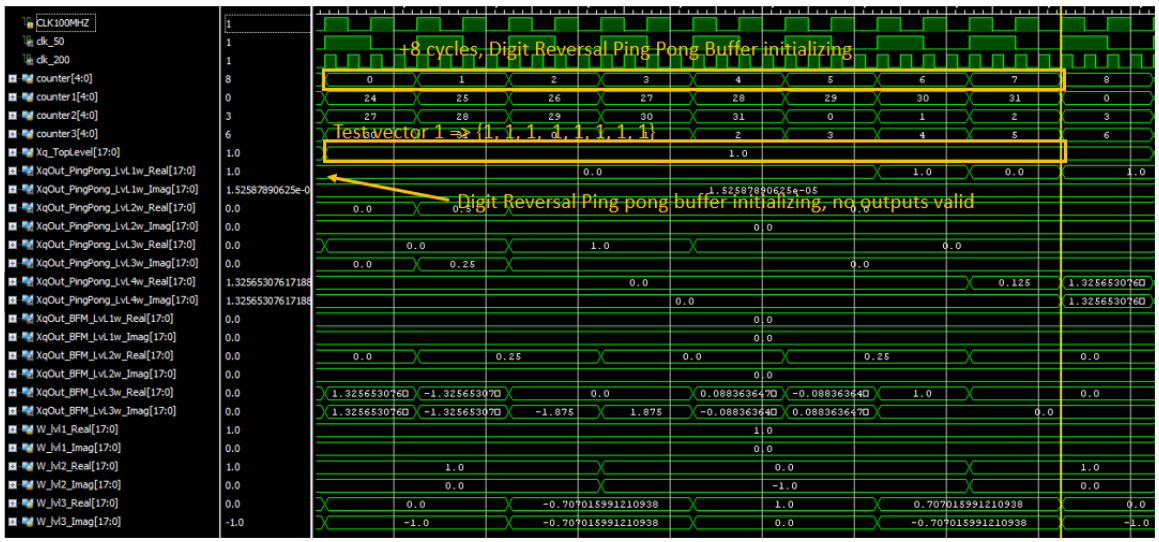



Figure 66. Bit-Reversed Ping-Pong Buffer Initializing

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.

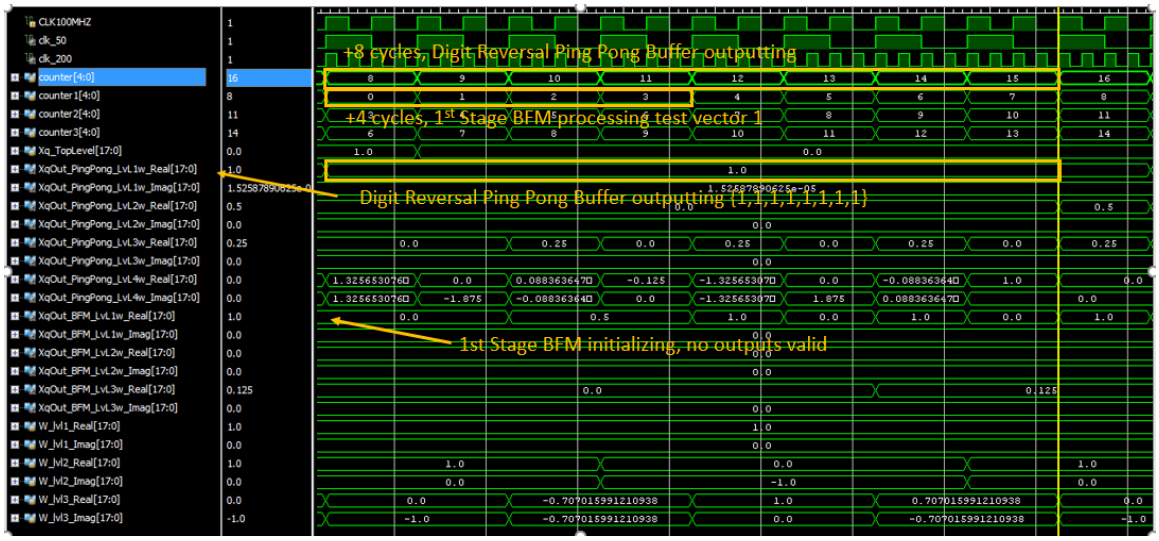



Figure 67. Bit-Reversed Ping-Pong Buffer Output

### 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.

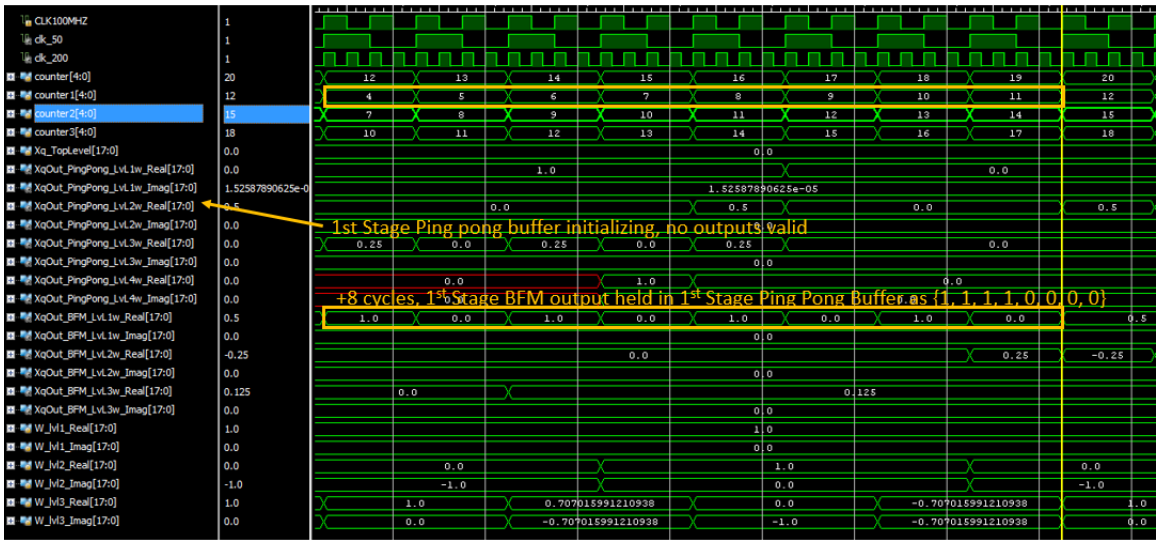



Figure 68. First-Stage BFM Output and First-Stage Ping-Pong Buffer Initialization

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.

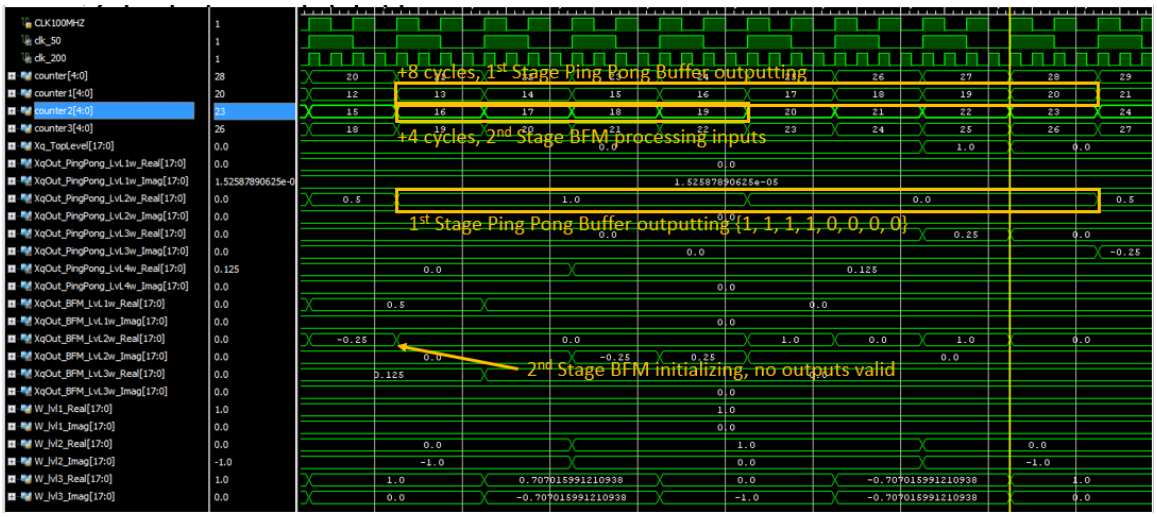



Figure 69. First-Stage Ping-Pong Buffer Outputting and Second-Stage BFM Initializing

### 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.

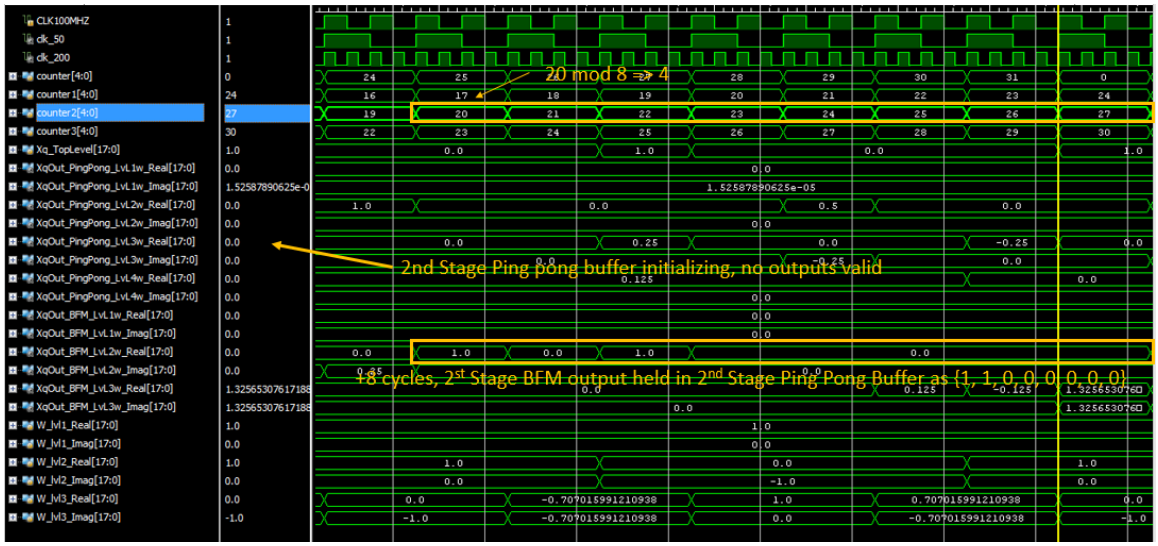



Figure 70. Second-Stage BFM Outputting and Second-Stage Ping-Pong Buffer Initializing

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.

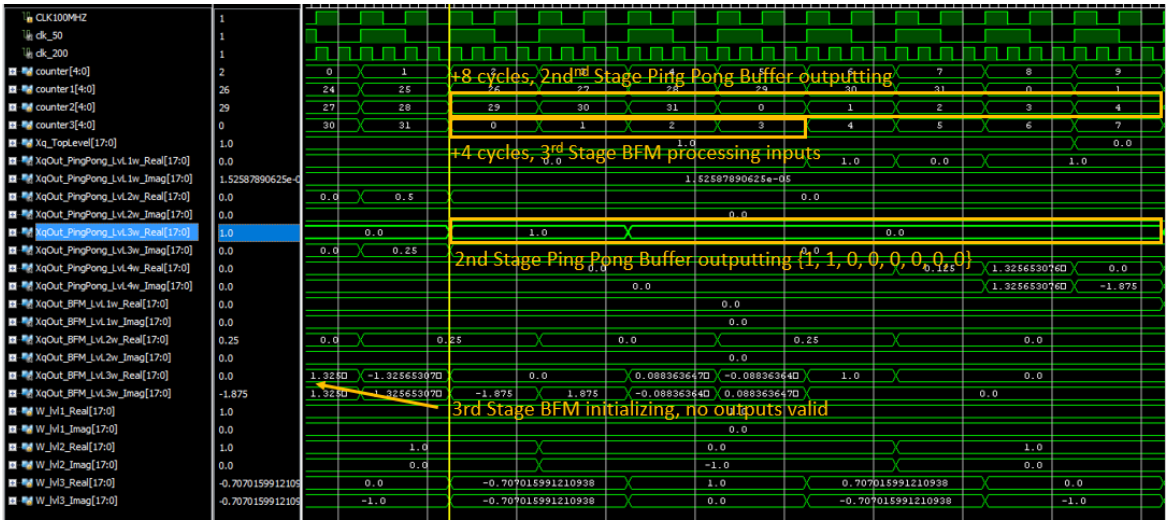



Figure 71. Second-Stage Ping-Pong Buffer Outputting and Third-Stage BFM Initializing

### 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

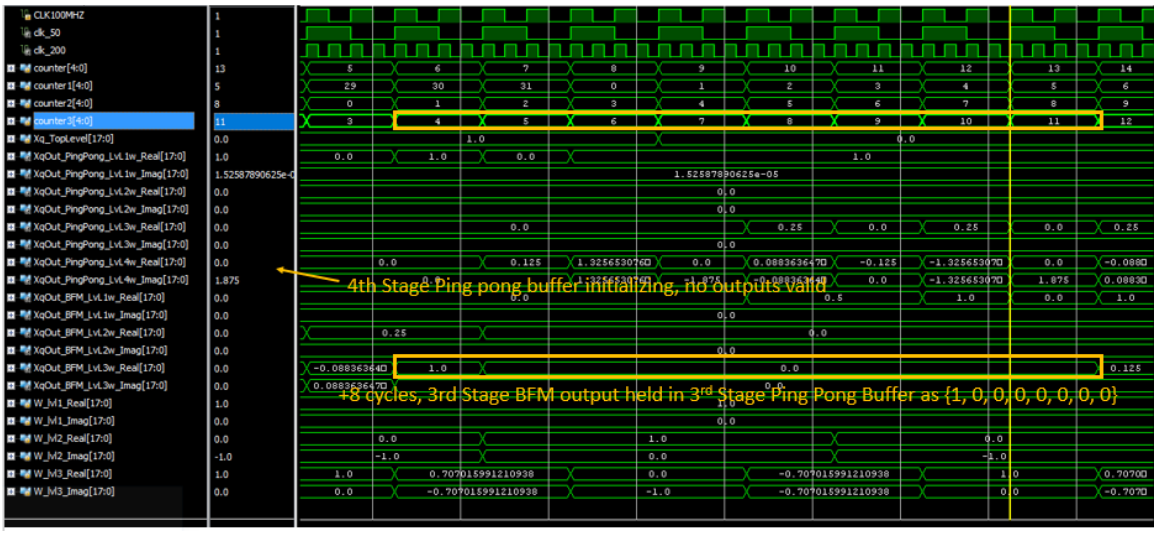



Figure 72. Third-Stage BFM Outputting and Third-Stage Ping-Pong Initializing

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.

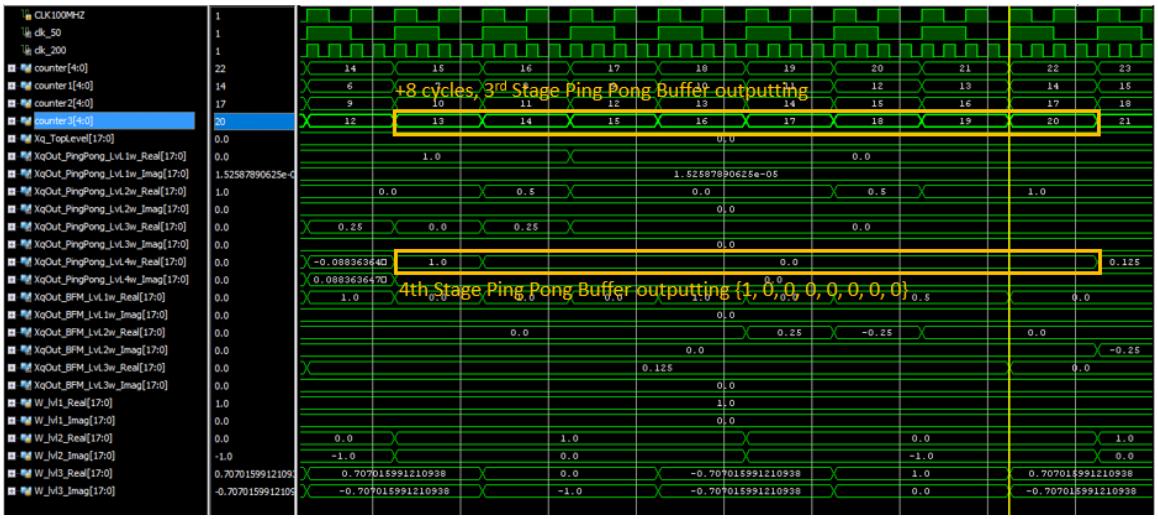



Figure 73. Third-Stage Ping-Pong Buffer With Final Scaled Result {1, 0, 0, 0, 0 0, 0, 0}

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 / subtracter 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.

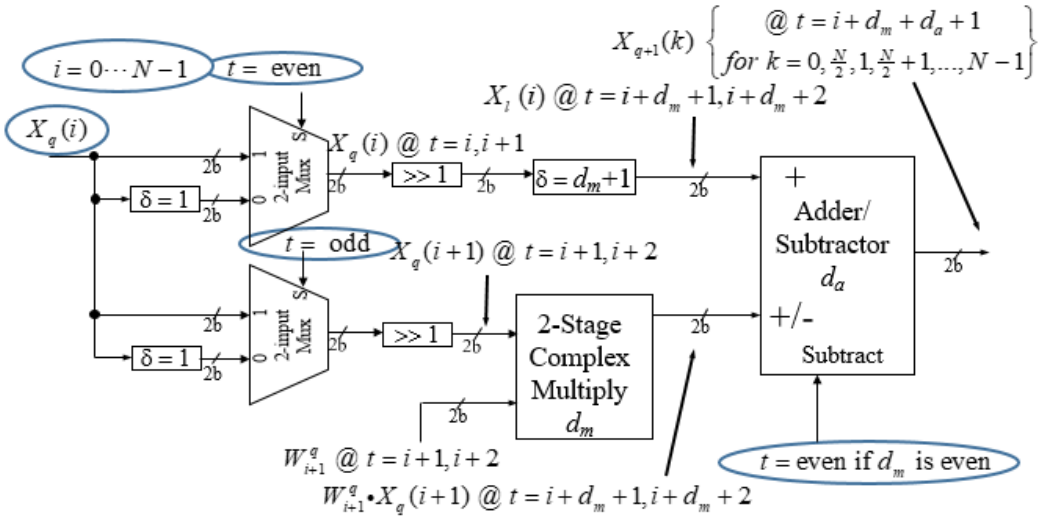



Figure 74. Highlighted Timing Variables within the BFM. Adapted from [15].

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.

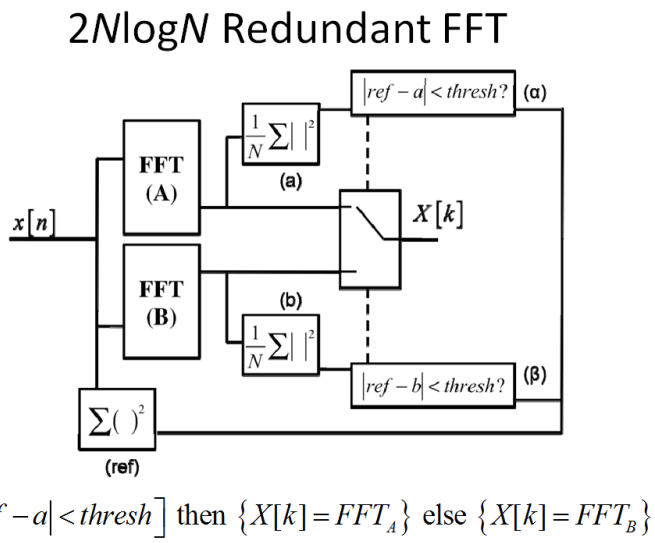



Figure 75. Parseval's Theorem Implementation Illustration. Source: [2].

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**********

```
module top_level(
    input wire CLK100MHZ
);

(* dont_touch = "true" *)wire clk_50, clk_200, clk_400;
(* dont_touch = "true" *)reg  [4:0] counter, counter1, counter2, counter3;
(* dont_touch = "true" *)reg  [17:0] driver;
(* dont_touch = "true" *)reg  INIT, reset;
(* dont_touch = "true" *)reg  [17:0] Xq_TopLevel;
(* dont_touch = "true" *)wire [17:0] XqOut_PingPong_LvL1w_Real, XqOut_PingPong_LvL1w_Imag;
(* dont_touch = "true" *)wire [17:0] XqOut_PingPong_LvL2w_Real, XqOut_PingPong_LvL2w_Imag;
(* dont_touch = "true" *)wire [17:0] XqOut_PingPong_LvL3w_Real, XqOut_PingPong_LvL3w_Imag;
(* dont_touch = "true" *)wire [17:0] XqOut_PingPong_LvL4w_Real, XqOut_PingPong_LvL4w_Imag;
(* dont_touch = "true" *)wire [17:0] XqOut_BFM_LvL1w_Real, XqOut_BFM_LvL1w_Imag;
(* dont_touch = "true" *)wire [17:0] XqOut_BFM_LvL2w_Real, XqOut_BFM_LvL2w_Imag;
(* dont_touch = "true" *)wire [17:0] XqOut_BFM_LvL3w_Real, XqOut_BFM_LvL3w_Imag;
(* dont_touch = "true" *)reg  [17:0] W_lvl1_Real, W_lvl1_Imag;
(* dont_touch = "true" *)reg  [17:0] W_lvl2_Real, W_lvl2_Imag;
(* dont_touch = "true" *)reg  [17:0] W_lvl3_Real, W_lvl3_Imag;

//-----------------------------------------------------------------------
    // The following must be inserted into your Verilog file for this
    // core to be instantiated. Change the instance name and port connections
    // (in parentheses) to your own signal names.

    //----------- Clock INSTANTIATION ----------
      clk_module clk_mod
       (
       // Clock in ports
        .clk_in1(CLK100MHZ),      // input clk
        // Clock out ports
        .clk_50(clk_50),     // output clk_50
        .clk_200(clk_200),   // output clk_200
        .clk_400(clk_400),   // output clk_400
        // Status and control signals
        .reset(reset),
        .locked(locked)
       );
    // ------ End Clock INSTANTIATION ---------

    // ------ Begin Ping-pong switch Level 1 ------
        pingpong PingPong_SwitchLevel1
            (
            .XqIn_PingPong_Real(Xq_TopLevel),
            .XqIn_PingPong_Imag(1),
            .clk(clk_50),
            .counter(counter),
            .XqOut_PingPongw_Real(XqOut_PingPong_LvL1w_Real),
            .XqOut_PingPongw_Imag(XqOut_PingPong_LvL1w_Imag)
            );
    // ------ End Ping-pong switch Level 1 ------

    // ------ Begin Radix2_BFM Level Q = 1 ------
        Radix2_BFM BFMlevelOne
            (
            .Xq_Real(XqOut_PingPong_LvL1w_Real),
            .Xq_Imag(XqOut_PingPong_LvL1w_Imag),
            .clk(clk_50),
            .W_Real(W_lvl1_Real),
            .W_Imag(W_lvl1_Imag),
            .counterw(counter1),
            .Radix2_BFMOut_Real(XqOut_BFM_LvL1w_Real),
            .Radix2_BFMOut_Imag(XqOut_BFM_LvL1w_Imag)
            );
    // ------   End Radix2_BFM Level Q = 1 ------

    // ------ Begin Ping-pongK switch Level 2 ------
        pingpongK PingPong_SwitchLevel2
            (
            .XqIn_PingPong_Real(XqOut_BFM_LvL1w_Real),        // input Xq real component
            .XqIn_PingPong_Imag(XqOut_BFM_LvL1w_Imag),        // input Xq imag component
            .clk(clk_50),                                    // input clock
            .counter(counter1),                               // input counter
            .XqOut_PingPongw_Real(XqOut_PingPong_LvL2w_Real), // output Xq real component on a wire
            .XqOut_PingPongw_Imag(XqOut_PingPong_LvL2w_Imag)  // output Xq imag component on a wire
            );
    // ------ End Ping-pongK switch Level 2 ------

    // ------ Begin Radix2_BFM Level Q = 2 ------
        Radix2_BFM BFMlevelTwo
            (
            .Xq_Real(XqOut_PingPong_LvL2w_Real),
            .Xq_Imag(XqOut_PingPong_LvL2w_Imag),
            .clk(clk_50),
            .W_Real(W_lvl2_Real),
            .W_Imag(W_lvl2_Imag),
            .counterw(counter2),
            .Radix2_BFMOut_Real(XqOut_BFM_LvL2w_Real),
            .Radix2_BFMOut_Imag(XqOut_BFM_LvL2w_Imag)
            );
```

```
    // ------ Begin Ping-pong switch Level 3 ------
        pingpongK PingPong_SwitchLevel3
            (
            .XqIn_PingPong_Real(XqOut_BFM_LvL2w_Real),        // input Xq real component
            .XqIn_PingPong_Imag(XqOut_BFM_LvL2w_Imag),        // input Xq imag component
            .clk(clk_50),                                    // input clock
            .counter(counter2),                              // input counter
            .XqOut_PingPongw_Real(XqOut_PingPong_LvL3w_Real), // output Xq real component on a wire
            .XqOut_PingPongw_Imag(XqOut_PingPong_LvL3w_Imag)  // output Xq imag component on a wire
            );
    // ------ End Ping-pong switch Level 3 ------

    // ------ Begin Radix2_BFM Level Q = 3 ------
        Radix2_BFM BFMlevelThree
            (
            .Xq_Real(XqOut_PingPong_LvL3w_Real),
            .Xq_Imag(XqOut_PingPong_LvL3w_Imag),
            .clk(clk_50),
            .W_Real(W_lvl3_Real),
            .W_Imag(W_lvl3_Imag),
            .counterw(counter3),
            .Radix2_BFMOut_Real(XqOut_BFM_LvL3w_Real),
            .Radix2_BFMOut_Imag(XqOut_BFM_LvL3w_Imag)
            );
    // ------ Begin Radix2_BFM Level Q = 3 ------

    // ------ Begin Ping-pong switch Level 4 ------
        pingpongK PingPong_SwitchLevel4
            (
            .XqIn_PingPong_Real(XqOut_BFM_LvL3w_Real),        // input Xq real component
            .XqIn_PingPong_Imag(XqOut_BFM_LvL3w_Imag),        // input Xq imag component
            .clk(clk_50),                                    // input clock
            .counter(counter3),                              // input counter
            .XqOut_PingPongw_Real(XqOut_PingPong_LvL4w_Real), // output Xq real component on a wire
            .XqOut_PingPongw_Imag(XqOut_PingPong_LvL4w_Imag)  // output Xq imag component on a wire
            );
    // ------ End Ping-pong switch Level 4 ------

// Initial block (Only executed once when simulation start)
    initial begin
        counter = -1;
        counter1 = -9;
        counter2 = -6;
        counter3 = -3;
        INIT = 0;
        reset = 0;
        Xq_TopLevel = 18'b010000000000000000;
        W_lvl1_Real = 1;
        W_lvl1_Imag = 1;
        W_lvl2_Real = 1;
        W_lvl2_Imag = 1;
        W_lvl3_Real = 1;
        W_lvl3_Imag = 1;
    end

// Statements below will be executed on every clock rising edge

always@(posedge clk_50)
    begin
    case (counter[4:0])
          5'b00000:  //Xq(0)
                begin
                Xq_TopLevel <= 18'b010000000000000000; // 1
                end
          5'b00001:  //Xq(1)
                begin
                Xq_TopLevel <= 18'b010000000000000000; // 1
                end
          5'b00010:  //Xq(2)
                begin
                Xq_TopLevel <= 18'b010000000000000000; // 1
                end
          5'b00011:  //Xq(3)
                begin
                Xq_TopLevel <= 18'b010000000000000000; // 1
                end
          5'b00100:  //Xq(4)
                begin
                Xq_TopLevel <= 18'b010000000000000000; // 1
                end
          5'b00101:  //Xq(5)
                begin
                Xq_TopLevel <= 18'b010000000000000000; // 1
                end
          5'b00110:  //Xq(6)
                begin
                Xq_TopLevel <= 18'b010000000000000000;  // 1
                end
```

```
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
211             Xq_TopLevel <= 18'b000000000000000000; // 0
212             end
213         5'b01111:  //Xq(15)
214             begin
215             Xq_TopLevel <= 18'b000000000000000000; // 0
216             end
217         5'b10000:  //Xq(16)
218             begin
219             Xq_TopLevel <= 18'b000000000000000000; // 0
220             end
221         5'b10001:  //Xq(17)
222             begin
223             Xq_TopLevel <= 18'b000000000000000000; // 0
224             end
225         5'b10010:  //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
```

```
                Xq_TopLevel <= 18'b000000000000000000; // 0
                end
            5'b11110:  //Xq(30)
                begin
                Xq_TopLevel <= 18'b000000000000000000; // 0
                end
            5'b11111:  //Xq(31)
                begin
                Xq_TopLevel <= 18'b010000000000000000; // 1
                end
        endcase

// Level 1 Twiddle
    case (counter1[2:0])
    3'b000:  //Xq(0)
            begin
            W_lvl1_Real <= 18'b010000000000000000; // 1
            W_lvl1_Imag <= 18'b000000000000000000; // -j0
            end
    3'b001:  //Xq(1)
            begin
            W_lvl1_Real <= 18'b010000000000000000; // 1
            W_lvl1_Imag <= 18'b000000000000000000; // -j0
            end
    3'b010:  //Xq(2)
            begin
            begin
            W_lvl1_Real <= 18'b010000000000000000; // 1
            W_lvl1_Imag <= 18'b000000000000000000; // -j0
            end
    3'b100:  //Xq(4)
            begin
            W_lvl1_Real <= 18'b010000000000000000;  // 1
            W_lvl1_Imag <= 18'b000000000000000000;  // -j0
            end
    3'b101:  //Xq(5)
            begin
            W_lvl1_Real <= 18'b010000000000000000;  // 1
            W_lvl1_Imag <= 18'b000000000000000000;  // -j0
            end
    3'b110:  //Xq(6)
            begin
            W_lvl1_Real <= 18'b010000000000000000;  // 1
            W_lvl1_Imag <= 18'b000000000000000000;  // -j0
            end
    3'b111:  //Xq(7)
            begin
            W_lvl1_Real <= 18'b010000000000000000;  // 1
            W_lvl1_Imag <= 18'b000000000000000000;  // -j0
            end
    endcase

// Level 2 Twiddle
    case (counter2[3:0])
    4'b0000:
        begin
        W_lvl2_Real <= 18'b000000000000000000; // 0--------
        W_lvl2_Imag <= 18'b110000000000000000; // -j1---------
        end
    4'b0001:
        begin
        W_lvl2_Real <= 18'b010000000000000000; // 1----------
        W_lvl2_Imag <= 18'b000000000000000000; // -j0-----------
        end
    4'b0010:
        begin
        W_lvl2_Real <= 18'b010000000000000000; // 1------------
        W_lvl2_Imag <= 18'b000000000000000000; // -j0-----------
        end
    4'b0011:
        begin
        W_lvl2_Real <= 18'b010000000000000000; // 1------------
        W_lvl2_Imag <= 18'b000000000000000000; // -j0--------------
        end
    4'b0100:
        begin
        W_lvl2_Real <= 18'b010000000000000000;  // 1-------------
        W_lvl2_Imag <= 18'b000000000000000000;  // -j0-----------
        end
    4'b0101:
        begin
        W_lvl2_Real <= 18'b000000000000000000;  // 0--------------
        W_lvl2_Imag <= 18'b110000000000000000;  // -j1------------
        end
    4'b0110:
        begin
```

```
            W_lvl2_Real <= 18'b000000000000000000;  // 0-----------------
            W_lvl2_Imag <= 18'b110000000000000000;  // -j1--------------
            end
        4'b0111:
            begin
            W_lvl2_Real <= 18'b000000000000000000;  // 0--------
            W_lvl2_Imag <= 18'b110000000000000000;  // -j1----------
            end
        4'b1000:
            begin
            W_lvl2_Real <= 18'b000000000000000000; // 0-------
            W_lvl2_Imag <= 18'b110000000000000000; // -j1-------
            end
        4'b1001:
            begin
            W_lvl2_Real <= 18'b010000000000000000; // 1------
            W_lvl2_Imag <= 18'b000000000000000000; // -j0-------
            end
        4'b1010:
            begin
            W_lvl2_Real <= 18'b010000000000000000; // 1--------
            W_lvl2_Imag <= 18'b000000000000000000; // -j0-------
            end
        4'b1011:
            begin
            W_lvl2_Real <= 18'b010000000000000000; // 1----------
            W_lvl2_Real <= 18'b010000000000000000; // 1----------
            W_lvl2_Imag <= 18'b000000000000000000; // -j0------
            end
        4'b1101:
            begin
            W_lvl2_Real <= 18'b000000000000000000; // 0-------
            W_lvl2_Imag <= 18'b110000000000000000; // -j1----------
            end
        4'b1110:
            begin
            W_lvl2_Real <= 18'b000000000000000000; // 0------
            W_lvl2_Imag <= 18'b110000000000000000; // -j1------
            end
        4'b1111:
            begin
            W_lvl2_Real <= 18'b000000000000000000; // 0------
            W_lvl2_Imag <= 18'b110000000000000000; // -j1-----
            end
        endcase

// Level 3 Twiddle    ----------------------------------------------
        case (counter3[3:0])
        4'b0000:
            begin
            W_lvl3_Real <= 18'b110100101100000001; // -sqrt(2)/2
            W_lvl3_Imag <= 18'b110100101100000001; // -jsqrt(2)/2
            end
        4'b0001:
            begin
            W_lvl3_Real <= 18'b010000000000000000; // 1
            W_lvl3_Imag <= 18'b000000000000000000; // -j0
            end
        4'b0010:
            begin
            W_lvl3_Real <= 18'b010000000000000000; // 1
            W_lvl3_Imag <= 18'b000000000000000000; // -j0
            end
        4'b0011:
            begin
            W_lvl3_Real <= 18'b001011010011111111; // sqrt(2)/2
            W_lvl3_Imag <= 18'b110100101100000001; // -jsqrt(2)/2
            end
        4'b0100:
            begin
            W_lvl3_Real <= 18'b001011010011111111;  // sqrt(2)/2
            W_lvl3_Imag <= 18'b110100101100000001;  //-jsqrt(2)/2
            end
        4'b0101:
            begin
            W_lvl3_Real <= 18'b000000000000000000;  // 0
            W_lvl3_Imag <= 18'b110000000000000000;  // -j1
            end
        4'b0110:
            begin
            W_lvl3_Real <= 18'b000000000000000000;  // 0
            W_lvl3_Imag <= 18'b110000000000000000;  // -j1
            end
        4'b0111:
            begin
            W_lvl3_Real <= 18'b110100101100000001;  // -sqrt(2)/2
```

```
451             W_lvl3_Imag <= 18'b110100101100000001;  // -sqrt(2)/2
452             end
453         4'b1000:
454             begin
455             W_lvl3_Real <= 18'b110100101100000001; // -sqrt(2)/2
456             W_lvl3_Imag <= 18'b110100101100000001; // -sqrt(2)/2
457             end
458         4'b1001:
459             begin
460             W_lvl3_Real <= 18'b010000000000000000; // 1
461             W_lvl3_Imag <= 18'b000000000000000000; // -j0
462             end
463         4'b1010:
464             begin
465             W_lvl3_Real <= 18'b010000000000000000; // 1
466             W_lvl3_Imag <= 18'b000000000000000000; // -j0
467             end
468         4'b1011:
469             begin
470             W_lvl3_Real <= 18'b001011010011111111; //  sqrt(2)/2
471             W_lvl3_Imag <= 18'b110100101100000001; // -jsqrt(2)/2
472             end
473         4'b1100:
474             begin
475             W_lvl3_Real <= 18'b001011010011111111; // sqrt(2)/2
476             W_lvl3_Imag <= 18'b110100101100000001; // -jsqrt(2)/2
481             W_lvl3_Imag <= 18'b110000000000000000; // -j1
482             end
483         4'b1110:
484             begin
485             W_lvl3_Real <= 18'b000000000000000000; // 0
486             W_lvl3_Imag <= 18'b110000000000000000; // -j1
487             end
488         4'b1111:
489             begin
490             W_lvl3_Real <= 18'b110100101100000001; // -sqrt(2)/2
491             W_lvl3_Imag <= 18'b110100101100000001; // -jsqrt(2)/2
492             end
493         endcase
494
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
2  // file: clk_module.v
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 (* 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
2  // file: clk_module.v
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 clkin1_ibufg
87    (.O (clk_in1_clk_module),
88     .I (clk_in1));
89
90
```

```
  // Clocking PRIMITIVE
  //------------------------------------

  // Instantiation of the MMCM PRIMITIVE
  //    * Unused inputs are tied off
  //    * Unused outputs are labeled unused

  wire        clk_50_clk_module;
  wire        clk_200_clk_module;
  wire        clk_400_clk_module;
  wire        clk_out4_clk_module;
  wire        clk_out5_clk_module;
  wire        clk_out6_clk_module;
  wire        clk_out7_clk_module;

  wire [15:0] do_unused;
  wire        drdy_unused;
  wire        psdone_unused;
  wire        locked_int;
  wire        clkfbout_clk_module;
  wire        clkfbout_buf_clk_module;
  wire        clkfboutb_unused;
    wire clkout0b_unused;
   wire clkout1b_unused;
   wire clkout2b_unused;
   wire clkout3_unused;
  wire        clkfbstopped_unused;
  wire        clkinstopped_unused;
  wire        reset_high;

  MMCME2_ADV
  #(.BANDWIDTH            ("OPTIMIZED"),
    .CLKOUT4_CASCADE      ("FALSE"),
    .COMPENSATION         ("ZHOLD"),
    .STARTUP_WAIT         ("FALSE"),
    .DIVCLK_DIVIDE        (1),
    .CLKFBOUT_MULT_F      (8.000),
    .CLKFBOUT_PHASE       (0.000),
    .CLKFBOUT_USE_FINE_PS ("FALSE"),
    .CLKOUT0_DIVIDE_F     (16.000),
    .CLKOUT0_PHASE        (0.000),
    .CLKOUT0_DUTY_CYCLE   (0.500),
    .CLKOUT0_USE_FINE_PS  ("FALSE"),
    .CLKOUT1_DIVIDE       (4),
    .CLKOUT1_PHASE        (0.000),
    .CLKOUT1_DUTY_CYCLE   (0.500),
    .CLKOUT1_USE_FINE_PS  ("FALSE"),
    .CLKOUT2_DIVIDE       (2),
    .CLKOUT2_PHASE        (0.000),
    .CLKOUT2_DUTY_CYCLE   (0.500),
    .CLKOUT2_USE_FINE_PS  ("FALSE"),
    .CLKIN1_PERIOD        (10.0))
  mmcm_adv_inst
    // Output clocks
   (
    .CLKFBOUT             (clkfbout_clk_module),
    .CLKFBOUTB            (clkfboutb_unused),
    .CLKOUT0              (clk_50_clk_module),
    .CLKOUT0B             (clkout0b_unused),
    .CLKOUT1              (clk_200_clk_module),
    .CLKOUT1B             (clkout1b_unused),
    .CLKOUT2              (clk_400_clk_module),
    .CLKOUT2B             (clkout2b_unused),
    .CLKOUT3              (clkout3_unused),
    .CLKOUT3B             (clkout3b_unused),
    .CLKOUT4              (clkout4_unused),
    .CLKOUT5              (clkout5_unused),
    .CLKOUT6              (clkout6_unused),
     // Input clock control
    .CLKFBIN              (clkfbout_buf_clk_module),
    .CLKIN1               (clk_in1_clk_module),
    .CLKIN2               (1'b0),
     // Tied to always select the primary input clock
    .CLKINSEL             (1'b1),
    // Ports for dynamic reconfiguration
    .DADDR                (7'h0),
    .DCLK                 (1'b0),
    .DEN                  (1'b0),
    .DI                   (16'h0),
    .DO                   (do_unused),
    .DRDY                 (drdy_unused),
    .DWE                  (1'b0),
    // Ports for dynamic phase shift
    .PSCLK                (1'b0),
    .PSEN                 (1'b0),
    .PSINCDEC             (1'b0),
```

```
    .PSDONE             (psdone_unused),
    // Other control and status signals
    .LOCKED             (locked_int),
    .CLKINSTOPPED       (clkinstopped_unused),
    .CLKFBSTOPPED       (clkfbstopped_unused),
    .PWRDWN             (1'b0),
    .RST                (reset_high));

  assign reset_high = reset;

  assign locked = locked_int;
// Clock Monitor clock assigning
//--------------------------------------
 // Output buffering
  //-----------------------------------


  BUFG clkf_buf
   (.O (clkfbout_buf_clk_module),
    .I (clkfbout_clk_module));



  BUFG clkout1_buf
   (.O   (clk_50),
    .I   (clk_50_clk_module));


  BUFG clkout3_buf
   (.O   (clk_400),
    .I   (clk_400_clk_module));



endmodule
```

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

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/19/2017 04:41:54 PM
// Design Name:
// Module Name: pingpong
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

//////////////
//Ping//Pong//   //One Register is being utilized for real component
// 0  // 0  //   //One Register is being utilized for imag component
//////////////   //i.e. Ping has a real and imaginery register
// 1  // 1  //   //Pong has a real and imaginery register
//////////////   //
// 2  // 2  //   //Bit Reversal explained
//////////////   //X[000] = x[000]  or X[0] => x[0]
// 3  // 3  //   //X[001] = x[100]  or X[1] => x[4]
//////////////   //X[010] = x[010]  or X[2] => x[2]
// 4  // 4  //   //X[011] = x[110]  or X[3] => x[6]
//////////////   //X[100] = x[001]  or X[4] => x[1]
// 5  // 5  //   //X[101] = x[101]  or X[5] => x[5]
//////////////   //X[110] = x[011]  or X[6] => x[3]
// 6  // 6  //   //X[111] = x[111]  or X[7] => x[7]
//////////////   //
// 7  // 7  //   //
//////////////   //

// complex input = (x + jz)
// complex output = (x + jz)

module pingpong(
    input wire [17:0] XqIn_PingPong_Real,    // input Xq real component
    input wire [17:0] XqIn_PingPong_Imag,    // input Xq imag component
    input wire clk,                         // input clock
    input wire [3:0] counter,                // input counter
    output wire [17:0] XqOut_PingPongw_Real, // output Xq real component on a wire
    output wire [17:0] XqOut_PingPongw_Imag  // output Xq imag component on a wire
    );

    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;                      // register used to tranpose counter
    reg [2:0] indexbr;                       // index utilized for bit reversal

    wire pingpongstatus;
    reg ping_loading;
    reg pong_loading;

assign XqOut_PingPongw_Real = (counter[3]==1'b0) ? XqPing_Real[counter[2:0]] : XqPong_Real[counter[2:0]];          //XqOut_PingPong_Real;  // register for output Xq real component
assign XqOut_PingPongw_Imag = (counter[3]==1'b0) ? XqPing_Imag[counter[2:0]] : XqPong_Imag[counter[2:0]];          //XqOut_PingPong_Imag;  // register for output Xq imag component
assign pingpongstatus = counter[3];
assign tranpose = counter + 1;
//always statement that occurs at every rising
//edge of the clock

// Initial block (Only executed once when simulation start)
    initial begin
       indexbr = 0;
    end

always@(posedge clk)
    begin
        // counter transposed due to bit reversal
        indexbr <= {tranpose[0],tranpose[1],tranpose[2]};

        //The Pong buffer is outputed when the Ping buffer is
        //being filled up.  Ping and Pong then alternate
        //The MSB of counter is utilized as
        //my switch bit.  Buffers are complex numbers.

        //Alternating loading Ping and Pong buffers
        //based on MSB counter.  Results in 16
        //cycle period.
        if (counter[3] == 1'b1) begin
            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
102 endmodule
```

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

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/08/2017 10:24:56 AM
// Design Name:
// Module Name: Radix2_BFM
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Radix2_BFM(
    input  wire [17:0] Xq_Real,
    input  wire [17:0] Xq_Imag,
    input  wire clk,
    input  wire [17:0] W_Real,
    input  wire [17:0] W_Imag,
    input  wire [2:0]  counterw,
    output wire [17:0] Radix2_BFMOut_Real,
    output wire [17:0] Radix2_BFMOut_Imag
    );

    (* dont_touch = "true" *)reg  [17:0] TopMuxOutDelay1_Real, TopMuxOutDelay2_Real, TopMuxOutDelay3_Real;
    (* dont_touch = "true" *)reg  [17:0] TopMuxOutDelay1_Imag, TopMuxOutDelay2_Imag, TopMuxOutDelay3_Imag;
    (* dont_touch = "true" *)wire [17:0] TopMuxOutw_Real, BottomMuxOutw_Real;
    (* dont_touch = "true" *)wire [17:0] TopMuxOutw_Imag, BottomMuxOutw_Imag;
    (* dont_touch = "true" *)wire [17:0] TopMuxOutw_halfscale_Real, BottomMuxOutw_halfscale_Real;
    (* dont_touch = "true" *)wire [17:0] TopMuxOutw_halfscale_Imag, BottomMuxOutw_halfscale_Imag;
    (* dont_touch = "true" *)reg  [17:0] XqDelay1_Real;
    (* dont_touch = "true" *)reg  [17:0] XqDelay1_Imag;
    (* dont_touch = "true" *)wire [17:0] MultOutw_Real;
    (* dont_touch = "true" *)wire [17:0] MultOutw_Imag;
    (* dont_touch = "true" *)wire [17:0] AdderOutw_Real;
    (* dont_touch = "true" *)wire [17:0] AdderOutw_Imag;

    // ------ 2 to 1 Multiplexer Instantiated for top ------
    two2oneMux two2oneMux_Top
    (
        .in1_real(Xq_Real),
        .in1_imag(Xq_Imag),
        .in2_real(XqDelay1_Real),
        .in2_imag(XqDelay1_Imag),
        .clk(~counterw[0]),
        .out_real(TopMuxOutw_Real),
        .out_imag(TopMuxOutw_Imag)
    );

    // ------ 2 to 1 Multiplexer Instantiated for bottom ------
    two2oneMux two2oneMux_Bottom
    (
        .in1_real(Xq_Real),
        .in1_imag(Xq_Imag),
        .in2_real(XqDelay1_Real),
        .in2_imag(XqDelay1_Imag),
        .clk(counterw[0]),
        .out_real(BottomMuxOutw_Real),
        .out_imag(BottomMuxOutw_Imag)
    );

    // ------ 18 bit by 18 bit Multiplier Instantiated -------
    multiply_complex ComplexMultiply2Stage
    (
        .clk(counterw[0]),
        .a(BottomMuxOutw_halfscale_Real),
        .jb(BottomMuxOutw_halfscale_Imag),
        .c(W_Real),
        .jd(W_Imag),
        .Xqout_real(MultOutw_Real),
        .Xqout_imag(MultOutw_Imag)
    );

    // ------ 18 bit by 18 bit Adder Instantiated ------
    addsub18x18 AdderandSubtractor1Stage
    (
        .TopIn_real(TopMuxOutDelay3_Real),
        .TopIn_imag(TopMuxOutDelay3_Imag),
        .BottomIn_real(MultOutw_Real),
        .BottomIn_imag(MultOutw_Imag),
        .clk(clk),
```

```
        .counterw(counterw[0]),
        .Xqout_real(AdderOutw_Real),
        .Xqout_imag(AdderOutw_Imag)
    );

    assign Radix2_BFMOut_Real = AdderOutw_Real;
    assign Radix2_BFMOut_Imag = AdderOutw_Imag;
    assign TopMuxOutw_halfscale_Real = TopMuxOutw_Real >> 1;
    assign TopMuxOutw_halfscale_Imag = TopMuxOutw_Imag >> 1;
    assign BottomMuxOutw_halfscale_Real = BottomMuxOutw_Real >> 1;
    assign BottomMuxOutw_halfscale_Imag = BottomMuxOutw_Imag >> 1;

    always@(posedge clk)
        begin
        TopMuxOutDelay1_Real <= TopMuxOutw_halfscale_Real;
        TopMuxOutDelay1_Imag <= TopMuxOutw_halfscale_Imag;
        TopMuxOutDelay2_Real <= TopMuxOutDelay1_Real;
        TopMuxOutDelay2_Imag <= TopMuxOutDelay1_Imag;
        TopMuxOutDelay3_Real <= TopMuxOutDelay2_Real;
        TopMuxOutDelay3_Imag <= TopMuxOutDelay2_Imag;
        XqDelay1_Real <= Xq_Real;
        XqDelay1_Imag <= Xq_Imag;
        end

endmodule
```

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

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/05/2017 03:43:36 PM
// Design Name: Complex Mutiplexer
// Module Name: two2oneMux
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module two2oneMux (
    input  wire signed [17:0] in1_real,  // a
    input  wire signed [17:0] in1_imag,  // jb
    input  wire signed [17:0] in2_real,  // c
    input  wire signed [17:0] in2_imag,  // jd
    input  wire               clk,
    output wire signed [17:0] out_real, // c   or   a
    output wire signed [17:0] out_imag  // jd  or  jb
    );

    assign out_real = (clk==1'b0) ? in2_real : in1_real;  //Real Component
    assign out_imag = (clk==1'b0) ? in2_imag : in1_imag;  //Imag Component
endmodule
```

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

```
module multiply_complex(
    input wire clk,
    input wire [17:0] a,
    input wire [17:0] jb,
    input wire [17:0] c,
    input wire [17:0] jd,
    output wire [17:0] Xqout_real,
    output wire [17:0] Xqout_imag
);

    (* dont_touch = "true" *) reg sign_mult_left_Real, sign_mult_right_Real, sign_mult_left_Imag, sign_mult_right_Imag;
    (* dont_touch = "true" *) wire sign_a;
    (* dont_touch = "true" *) wire sign_jb;
    (* dont_touch = "true" *) wire sign_c;
    (* dont_touch = "true" *) wire sign_jd;
    (* dont_touch = "true" *) wire [16:0] unsigned_a; //
    (* dont_touch = "true" *) wire [16:0] unsigned_jb; //
    (* dont_touch = "true" *) wire [16:0] unsigned_c; //
    (* dont_touch = "true" *) wire [16:0] unsigned_jd; //
    (* dont_touch = "true" *) wire [16:0] trunc_left_Real;    //(a*c)
    (* dont_touch = "true" *) wire [16:0] trunc_right_Real;   //(jb*jd)
    (* dont_touch = "true" *) wire [16:0] trunc_left_Imag;    //(a*jd)
    (* dont_touch = "true" *) wire [16:0] trunc_right_Imag;   //(jb*c)
    (* dont_touch = "true" *) wire [17:0] signed_left_Real;    //(a*c)
    (* dont_touch = "true" *) wire [17:0] signed_right_Real;   //(jb*jd)
    (* dont_touch = "true" *) wire [17:0] signed_left_Imag;    //(a*jd)
    (* dont_touch = "true" *) wire [17:0] signed_right_Imag;   //(jb*c)
//-------------------------------------------------------------------------------

    //performs unsigned fixed point multiplication.  truncates output reducing precision.
    //returns 17 bits.  Inputs are a clock, 17 bit unsigned fixed point numbers labeled as
    //unsigned_a, unsigned_jb, unsigned_c, and unsigned_jd.  Outputs are 17 bit unsigned fixed point
    //numbers labeled as left_Real
    mult18x18 unsigned_multiply
    (
    .clk(clk),
    .a(unsigned_a),
    .jb(unsigned_jb),
    .c(unsigned_c),
    .jd(unsigned_jd),
    .Out_left_Real(trunc_left_Real),
    .Out_right_Real(trunc_right_Real),
    .Out_left_Imag(trunc_left_Imag),
    .Out_right_Imag(trunc_right_Imag));

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

    //2'c operatof prior to multiplication.  returns unsigned number. returns 17 bits
    twoComp insta(.In(a),.Out(unsigned_a)); //extracts data portion a
    twoComp instb(.In(jb),.Out(unsigned_jb)); //extracts data portion jb
    twoComp instc(.In(c),.Out(unsigned_c)); //extracts data portion c
    twoComp instd(.In(jd),.Out(unsigned_jd)); //extracts data portion jd

    // saves sign bit.  returns 1 bit
    assign sign_a = a[17]; //extracts sign of a
    assign sign_jb = jb[17]; //extracts sign of jb
    assign sign_c = c[17]; //extracts sign of c
    assign sign_jd = jd[17]; //extracts sign of jd

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

// Initial block (Only executed once when simulation start)
    initial
        begin
        end

// Statements below will be executed on every clock rising edge

always@(posedge clk)
    begin
    sign_mult_left_Real  <= sign_a ^ sign_c;
    sign_mult_right_Real <= sign_jb ^ sign_jd;
    sign_mult_left_Imag  <= sign_a ^ sign_jd;
    sign_mult_right_Imag <= sign_jb ^ sign_c;
    end
endmodule
```

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

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/25/2017 11:43:54 AM
// Design Name: Complex Multiplication
// Module Name: mult18x18
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module mult18x18(
    input wire clk,
    input wire [16:0] a,     //a
    input wire [16:0] jb,     //jb
    input wire [16:0] c,     //c
    input wire [16:0] jd,     //jd
    output wire [16:0] Out_left_Real,    //(a*c)
    output wire [16:0] Out_right_Real,   //(b*d)
    output wire [16:0] Out_left_Imag,    //(a*jd)
    output wire [16:0] Out_right_Imag);  //(c*jb)

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

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

initial begin
end

    always@(posedge clk)
        begin
        //Calculate real portion
        mult_left_Real <= a * c;
        mult_right_Real <= jb * jd;

        //Calculate imag portion
        mult_left_Imag <= a * jd;
        mult_right_Imag <= jb * c;
        end

endmodule
```

**********TWOCOMPREDO.V**********

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 05/23/2017 08:08:50 AM
// Design Name:
// Module Name: twoCompRedo
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module twoCompRedo(
input wire signbit,
input wire [16:0] number,
output wire [17:0] sign_number
);

assign sign_number = signbit?{signbit,~number+1}:{signbit,number};
endmodule
```

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

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 05/19/2017 03:16:22 AM
// Design Name:
// Module Name: Two's Complement
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module twoComp(
input wire [17:0] In,
output wire [16:0] Out);

assign Out = In[17]?~In+1:In;

endmodule
```

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

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/08/2017 09:41:42 AM
// Design Name: Complex Addition / Subtraction
// Module Name: addsub18x18
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module addsub18x18(
    input  wire [17:0] TopIn_real,    // a
    input  wire [17:0] TopIn_imag,    // jb
    input  wire [17:0] BottomIn_real,    // c
    input  wire [17:0] BottomIn_imag,    // jd
    input  wire        clk,
    input  wire        counterw,
    output wire [17:0] Xqout_real,   //   a + b   or    a - b
    output wire [17:0] Xqout_imag ); // j(b + d)  or   j(b - d)

    (* dont_touch = "true" *) reg [17:0] out_real;
    (* dont_touch = "true" *) reg [17:0] out_imag;

    //Complex Addition/Subtraction
    //                Real     Imag
    //(a+jb)+(c+jd) = a + c  ,  j(b+d)
    //(a+jb)-(c+jd) = a - c  ,  j(b-d)
    assign Xqout_real = out_real; //Real Component
    assign Xqout_imag = out_imag; //Imag Component

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

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

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/19/2017 04:41:54 PM
// Design Name:
// Module Name: pingpong
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

//   for N=8
//k = 0, N/2, 1, N/2+1, 2, N/2+2, 3, N/2+3;;
//k = (0, 4, 1, 5, 2, 6, 3, 7)

//////////////
//Ping//Pong//   //One Register is being utilized for real component
// 0  // 0  //   //One Register is being utilized for imag component
///////////////   //i.e. Ping has a real and imaginery register
// 1  // 1  //   //Pong has a real and imaginery register
///////////////   //
// 2  // 2  //   //Ping-pong fill order explained
///////////////   //X[000] = x[000]  or X[0] => x[0]
// 3  // 3  //   //X[001] = x[100]  or X[1] => x[4]
///////////////   //X[010] = x[001]  or X[2] => x[1]
// 4  // 4  //   //X[011] = x[101]  or X[3] => x[5]
///////////////   //X[100] = x[010]  or X[4] => x[2]
// 5  // 5  //   //X[101] = x[110]  or X[5] => x[6]
///////////////   //X[110] = x[011]  or X[6] => x[3]
// 6  // 6  //   //X[111] = x[111]  or X[7] => x[7]
///////////////   //
// 7  // 7  //   //
///////////////   //

// complex input = (x + jz)
// complex output = (x + jz)

module pingpongK(
    input wire  [17:0] XqIn_PingPong_Real,    // input Xq real component
    input wire  [17:0] XqIn_PingPong_Imag,    // input Xq imag component
    input wire         clk,                   // input clock
    input wire  [3:0]  counter,               // input counter
    output wire [17:0] XqOut_PingPongw_Real,  // output Xq real component on a wire
    output wire [17:0] XqOut_PingPongw_Imag   // output Xq imag component on a wire
    );

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


assign XqOut_PingPongw_Real = (buffer_counter[3]==1'b1) ? XqPing_Real[buffer_counter[2:0]] : XqPong_Real[buffer_counter[2:0]]; //XqOut_PingPong_Real;  // register for output Xq rea
assign XqOut_PingPongw_Imag = (buffer_counter[3]==1'b1) ? XqPing_Imag[buffer_counter[2:0]] : XqPong_Imag[buffer_counter[2:0]]; //XqOut_PingPong_Imag;  // register for output Xq ima
//assign select = buffer_counter + 1;
//always statement that occurs at every rising
//edge of the clock

initial
    begin

    end

always@(posedge clk)
    begin
        //The Pong buffer is outputed when the Ping buffer is
        //being filled up.  Ping and Pong then alternate
        //The MSB of counter is utilized as
        //my switch bit.  Buffers are complex numbers.

        // Input code to Ping and Pong buffers
        // based on k = 0, N/2, 1, N/2+1, 2, N/2+2, 3, N/2+3;
        // k = (0, 4, 1, 5, 2, 6, 3, 7);
        // Results in 16 cycle period.

            case(counter)
            3'b0000:  //Xq(0)
```

```
91              begin
92              XqPong_Real[2] <= XqIn_PingPong_Real;
93              XqPong_Imag[2] <= XqIn_PingPong_Imag;
94              buffer_counter = 4'd12;
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**********

```
## This file is a general .xdc for the Nexys4 DDR Rev. C
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

## Clock signal
set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports CLK100MHZ]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];


##Switches

#set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { SW[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
#set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
#set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
#set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
#set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
#set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { SW[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
#set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
#set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
#set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
#set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
#set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
#set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
#set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
#set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]


## LEDs

#set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
#set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
#set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
#set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
#set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { LED[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
#set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
#set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

#set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 } [get_ports { LED16_B }]; #IO_L5P_T0_D06_14 Sch=led16_b
#set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 } [get_ports { LED16_G }]; #IO_L10P_T1_D14_14 Sch=led16_g
#set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 } [get_ports { LED16_R }]; #IO_L11P_T1_SRCC_14 Sch=led16_r
#set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { LED17_B }]; #IO_L15N_T2_DQS_ADV_B_15 Sch=led17_b
#set_property -dict { PACKAGE_PIN R11   IOSTANDARD LVCMOS33 } [get_ports { LED17_G }]; #IO_0_14 Sch=led17_g
#set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { LED17_R }]; #IO_L11N_T1_SRCC_14 Sch=led17_r


##7 segment display

#set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
#set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { CB }]; #IO_25_14 Sch=cb
#set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { CC }]; #IO_25_15 Sch=cc
#set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { CD }]; #IO_L17P_T2_A26_15 Sch=cd
#set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { CE }]; #IO_L13P_T2_MRCC_14 Sch=ce
#set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { CF }]; #IO_L19P_T3_A10_D26_14 Sch=cf
#set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { CG }]; #IO_L4P_T0_D04_14 Sch=cg

#set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp

#set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
#set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
#set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
#set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
#set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
#set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
#set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
#set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]


##Buttons

#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTNC }]; #IO_L9P_T1_DQS_14 Sch=btnc
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { BTNU }]; #IO_L4N_T0_D05_14 Sch=btnu
#set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { BTNL }]; #IO_L12P_T1_MRCC_14 Sch=btnl
#set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { BTNR }]; #IO_L10N_T1_D15_14 Sch=btnr
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { BTND }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

```
##Pmod Headers


##Pmod Header JA

#set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 } [get_ports { JA[1] }]; #IO_L20N_T3_A19_15 Sch=ja[1]
#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { JA[2] }]; #IO_L21N_T3_DQS_A18_15 Sch=ja[2]
#set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports { JA[3] }]; #IO_L21P_T3_DQS_15 Sch=ja[3]
#set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports { JA[4] }]; #IO_L18N_T2_A23_15 Sch=ja[4]
#set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports { JA[7] }]; #IO_L16N_T2_A27_15 Sch=ja[7]
#set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports { JA[8] }]; #IO_L16P_T2_A28_15 Sch=ja[8]
#set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports { JA[9] }]; #IO_L22N_T3_A16_15 Sch=ja[9]
#set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports { JA[10] }]; #IO_L22P_T3_A17_15 Sch=ja[10]


##Pmod Header JB

#set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 } [get_ports { JB[1] }]; #IO_L1P_T0_AD0P_15 Sch=jb[1]
#set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports { JB[2] }]; #IO_L14N_T2_SRCC_15 Sch=jb[2]
#set_property -dict { PACKAGE_PIN G16   IOSTANDARD LVCMOS33 } [get_ports { JB[3] }]; #IO_L13N_T2_MRCC_15 Sch=jb[3]
#set_property -dict { PACKAGE_PIN H14   IOSTANDARD LVCMOS33 } [get_ports { JB[4] }]; #IO_L15P_T2_DQS_15 Sch=jb[4]
#set_property -dict { PACKAGE_PIN E16   IOSTANDARD LVCMOS33 } [get_ports { JB[7] }]; #IO_L11N_T1_SRCC_15 Sch=jb[7]
#set_property -dict { PACKAGE_PIN F13   IOSTANDARD LVCMOS33 } [get_ports { JB[8] }]; #IO_L5P_T0_AD9P_15 Sch=jb[8]
#set_property -dict { PACKAGE_PIN G13   IOSTANDARD LVCMOS33 } [get_ports { JB[9] }]; #IO_0_15 Sch=jb[9]
#set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports { JB[10] }]; #IO_L13P_T2_MRCC_15 Sch=jb[10]

#set_property -dict { PACKAGE_PIN F6    IOSTANDARD LVCMOS33 } [get_ports { JC[2] }]; #IO_L19N_T3_VREF_35 Sch=jc[2]
#set_property -dict { PACKAGE_PIN J2    IOSTANDARD LVCMOS33 } [get_ports { JC[3] }]; #IO_L22N_T3_35 Sch=jc[3]
#set_property -dict { PACKAGE_PIN G6    IOSTANDARD LVCMOS33 } [get_ports { JC[4] }]; #IO_L19P_T3_35 Sch=jc[4]
#set_property -dict { PACKAGE_PIN E7    IOSTANDARD LVCMOS33 } [get_ports { JC[7] }]; #IO_L6P_T0_35 Sch=jc[7]
#set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 } [get_ports { JC[8] }]; #IO_L22P_T3_35 Sch=jc[8]
#set_property -dict { PACKAGE_PIN J4    IOSTANDARD LVCMOS33 } [get_ports { JC[9] }]; #IO_L21P_T3_DQS_35 Sch=jc[9]
#set_property -dict { PACKAGE_PIN E6    IOSTANDARD LVCMOS33 } [get_ports { JC[10] }]; #IO_L5P_T0_AD13P_35 Sch=jc[10]


##Pmod Header JD

#set_property -dict { PACKAGE_PIN H4    IOSTANDARD LVCMOS33 } [get_ports { JD[1] }]; #IO_L21N_T3_DQS_35 Sch=jd[1]
#set_property -dict { PACKAGE_PIN H1    IOSTANDARD LVCMOS33 } [get_ports { JD[2] }]; #IO_L17P_T2_35 Sch=jd[2]
#set_property -dict { PACKAGE_PIN G1    IOSTANDARD LVCMOS33 } [get_ports { JD[3] }]; #IO_L17N_T2_35 Sch=jd[3]
#set_property -dict { PACKAGE_PIN G3    IOSTANDARD LVCMOS33 } [get_ports { JD[4] }]; #IO_L20N_T3_35 Sch=jd[4]
#set_property -dict { PACKAGE_PIN H2    IOSTANDARD LVCMOS33 } [get_ports { JD[7] }]; #IO_L15P_T2_DQS_35 Sch=jd[7]
#set_property -dict { PACKAGE_PIN G4    IOSTANDARD LVCMOS33 } [get_ports { JD[8] }]; #IO_L20P_T3_35 Sch=jd[8]
#set_property -dict { PACKAGE_PIN G2    IOSTANDARD LVCMOS33 } [get_ports { JD[9] }]; #IO_L15N_T2_DQS_35 Sch=jd[9]
#set_property -dict { PACKAGE_PIN F3    IOSTANDARD LVCMOS33 } [get_ports { JD[10] }]; #IO_L13N_T2_MRCC_35 Sch=jd[10]


##Pmod Header JXADC

#set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVDS     } [get_ports { XA_N[1] }]; #IO_L9N_T1_DQS_AD3N_15 Sch=xa_n[1]
#set_property -dict { PACKAGE_PIN A13   IOSTANDARD LVDS     } [get_ports { XA_P[1] }]; #IO_L9P_T1_DQS_AD3P_15 Sch=xa_p[1]
#set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVDS     } [get_ports { XA_N[2] }]; #IO_L8N_T1_AD10N_15 Sch=xa_n[2]
#set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVDS     } [get_ports { XA_P[2] }]; #IO_L8P_T1_AD10P_15 Sch=xa_p[2]
#set_property -dict { PACKAGE_PIN B17   IOSTANDARD LVDS     } [get_ports { XA_N[3] }]; #IO_L7N_T1_AD2N_15 Sch=xa_n[3]
#set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVDS     } [get_ports { XA_P[3] }]; #IO_L7P_T1_AD2P_15 Sch=xa_p[3]
#set_property -dict { PACKAGE_PIN A18   IOSTANDARD LVDS     } [get_ports { XA_N[4] }]; #IO_L10N_T1_AD11N_15 Sch=xa_n[4]
#set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVDS     } [get_ports { XA_P[4] }]; #IO_L10P_T1_AD11P_15 Sch=xa_p[4]


##VGA Connector

#set_property -dict { PACKAGE_PIN A3    IOSTANDARD LVCMOS33 } [get_ports { VGA_R[0] }]; #IO_L8N_T1_AD14N_35 Sch=vga_r[0]
#set_property -dict { PACKAGE_PIN B4    IOSTANDARD LVCMOS33 } [get_ports { VGA_R[1] }]; #IO_L7N_T1_AD6N_35 Sch=vga_r[1]
#set_property -dict { PACKAGE_PIN C5    IOSTANDARD LVCMOS33 } [get_ports { VGA_R[2] }]; #IO_L1N_T0_AD4N_35 Sch=vga_r[2]
#set_property -dict { PACKAGE_PIN A4    IOSTANDARD LVCMOS33 } [get_ports { VGA_R[3] }]; #IO_L8P_T1_AD14P_35 Sch=vga_r[3]

#set_property -dict { PACKAGE_PIN C6    IOSTANDARD LVCMOS33 } [get_ports { VGA_G[0] }]; #IO_L1P_T0_AD4P_35 Sch=vga_g[0]
#set_property -dict { PACKAGE_PIN A5    IOSTANDARD LVCMOS33 } [get_ports { VGA_G[1] }]; #IO_L3N_T0_DQS_AD5N_35 Sch=vga_g[1]
#set_property -dict { PACKAGE_PIN B6    IOSTANDARD LVCMOS33 } [get_ports { VGA_G[2] }]; #IO_L2N_T0_AD12N_35 Sch=vga_g[2]
#set_property -dict { PACKAGE_PIN A6    IOSTANDARD LVCMOS33 } [get_ports { VGA_G[3] }]; #IO_L3P_T0_DQS_AD5P_35 Sch=vga_g[3]

#set_property -dict { PACKAGE_PIN B7    IOSTANDARD LVCMOS33 } [get_ports { VGA_B[0] }]; #IO_L2P_T0_AD12P_35 Sch=vga_b[0]
#set_property -dict { PACKAGE_PIN C7    IOSTANDARD LVCMOS33 } [get_ports { VGA_B[1] }]; #IO_L4N_T0_35 Sch=vga_b[1]
#set_property -dict { PACKAGE_PIN D7    IOSTANDARD LVCMOS33 } [get_ports { VGA_B[2] }]; #IO_L6N_T0_VREF_35 Sch=vga_b[2]
#set_property -dict { PACKAGE_PIN D8    IOSTANDARD LVCMOS33 } [get_ports { VGA_B[3] }]; #IO_L4P_T0_35 Sch=vga_b[3]

#set_property -dict { PACKAGE_PIN B11   IOSTANDARD LVCMOS33 } [get_ports { VGA_HS }]; #IO_L4P_T0_15 Sch=vga_hs
#set_property -dict { PACKAGE_PIN B12   IOSTANDARD LVCMOS33 } [get_ports { VGA_VS }]; #IO_L3N_T0_DQS_AD1N_15 Sch=vga_vs


##Micro SD Connector

#set_property -dict { PACKAGE_PIN E2    IOSTANDARD LVCMOS33 } [get_ports { SD_RESET }]; #IO_L14P_T2_SRCC_35 Sch=sd_reset
#set_property -dict { PACKAGE_PIN A1    IOSTANDARD LVCMOS33 } [get_ports { SD_CD }]; #IO_L9N_T1_DQS_AD7N_35 Sch=sd_cd
#set_property -dict { PACKAGE_PIN B1    IOSTANDARD LVCMOS33 } [get_ports { SD_SCK }]; #IO_L9P_T1_DQS_AD7P_35 Sch=sd_sck
#set_property -dict { PACKAGE_PIN C1    IOSTANDARD LVCMOS33 } [get_ports { SD_CMD }]; #IO_L16N_T2_35 Sch=sd_cmd
```

```
#set_property -dict { PACKAGE_PIN C2    IOSTANDARD LVCMOS33 } [get_ports { SD_DAT[0] }]; #IO_L16P_T2_35 Sch=sd_dat[0]
#set_property -dict { PACKAGE_PIN E1    IOSTANDARD LVCMOS33 } [get_ports { SD_DAT[1] }]; #IO_L18N_T2_35 Sch=sd_dat[1]
#set_property -dict { PACKAGE_PIN F1    IOSTANDARD LVCMOS33 } [get_ports { SD_DAT[2] }]; #IO_L18P_T2_35 Sch=sd_dat[2]
#set_property -dict { PACKAGE_PIN D2    IOSTANDARD LVCMOS33 } [get_ports { SD_DAT[3] }]; #IO_L14N_T2_SRCC_35 Sch=sd_dat[3]


##Accelerometer

#set_property -dict { PACKAGE_PIN E15   IOSTANDARD LVCMOS33 } [get_ports { ACL_MISO }]; #IO_L11P_T1_SRCC_15 Sch=acl_miso
#set_property -dict { PACKAGE_PIN F14   IOSTANDARD LVCMOS33 } [get_ports { ACL_MOSI }]; #IO_L5N_T0_AD9N_15 Sch=acl_mosi
#set_property -dict { PACKAGE_PIN F15   IOSTANDARD LVCMOS33 } [get_ports { ACL_SCLK }]; #IO_L14P_T2_SRCC_15 Sch=acl_sclk
#set_property -dict { PACKAGE_PIN D15   IOSTANDARD LVCMOS33 } [get_ports { ACL_CSN }]; #IO_L12P_T1_MRCC_15 Sch=acl_csn
#set_property -dict { PACKAGE_PIN B13   IOSTANDARD LVCMOS33 } [get_ports { ACL_INT[1] }]; #IO_L2P_T0_AD8P_15 Sch=acl_int[1]
#set_property -dict { PACKAGE_PIN C16   IOSTANDARD LVCMOS33 } [get_ports { ACL_INT[2] }]; #IO_L20P_T3_A20_15 Sch=acl_int[2]


##Temperature Sensor

#set_property -dict { PACKAGE_PIN C14   IOSTANDARD LVCMOS33 } [get_ports { TMP_SCL }]; #IO_L1N_T0_AD0N_15 Sch=tmp_scl
#set_property -dict { PACKAGE_PIN C15   IOSTANDARD LVCMOS33 } [get_ports { TMP_SDA }]; #IO_L12N_T1_MRCC_15 Sch=tmp_sda
#set_property -dict { PACKAGE_PIN D13   IOSTANDARD LVCMOS33 } [get_ports { TMP_INT }]; #IO_L6N_T0_VREF_15 Sch=tmp_int
#set_property -dict { PACKAGE_PIN B14   IOSTANDARD LVCMOS33 } [get_ports { TMP_CT }]; #IO_L2N_T0_AD8N_15 Sch=tmp_ct

##Omnidirectional Microphone

#set_property -dict { PACKAGE_PIN J5    IOSTANDARD LVCMOS33 } [get_ports { M_CLK }]; #IO_25_35 Sch=m_clk
###PWM Audio Amplifier

#set_property -dict { PACKAGE_PIN A11   IOSTANDARD LVCMOS33 } [get_ports { AUD_PWM }]; #IO_L4N_T0_15 Sch=aud_pwm
#set_property -dict { PACKAGE_PIN D12   IOSTANDARD LVCMOS33 } [get_ports { AUD_SD }]; #IO_L6P_T0_15 Sch=aud_sd


##USB-RS232 Interface

#set_property -dict { PACKAGE_PIN C4    IOSTANDARD LVCMOS33 } [get_ports { UART_TXD_IN }]; #IO_L7P_T1_AD6P_35 Sch=uart_txd_in
#set_property -dict { PACKAGE_PIN D4    IOSTANDARD LVCMOS33 } [get_ports { UART_RXD_OUT }]; #IO_L11N_T1_SRCC_35 Sch=uart_rxd_out
#set_property -dict { PACKAGE_PIN D3    IOSTANDARD LVCMOS33 } [get_ports { UART_CTS }]; #IO_L12N_T1_MRCC_35 Sch=uart_cts
#set_property -dict { PACKAGE_PIN E5    IOSTANDARD LVCMOS33 } [get_ports { UART_RTS }]; #IO_L5N_T0_AD13N_35 Sch=uart_rts

##USB HID (PS/2)

#set_property -dict { PACKAGE_PIN F4    IOSTANDARD LVCMOS33 } [get_ports { PS2_CLK }]; #IO_L13P_T2_MRCC_35 Sch=ps2_clk
#set_property -dict { PACKAGE_PIN B2    IOSTANDARD LVCMOS33 } [get_ports { PS2_DATA }]; #IO_L10N_T1_AD15N_35 Sch=ps2_data


##SMSC Ethernet PHY

#set_property -dict { PACKAGE_PIN C9    IOSTANDARD LVCMOS33 } [get_ports { ETH_MDC }]; #IO_L11P_T1_SRCC_16 Sch=eth_mdc
#set_property -dict { PACKAGE_PIN A9    IOSTANDARD LVCMOS33 } [get_ports { ETH_MDIO }]; #IO_L14N_T2_SRCC_16 Sch=eth_mdio
#set_property -dict { PACKAGE_PIN B3    IOSTANDARD LVCMOS33 } [get_ports { ETH_RSTN }]; #IO_L10P_T1_AD15P_35 Sch=eth_rstn
#set_property -dict { PACKAGE_PIN D9    IOSTANDARD LVCMOS33 } [get_ports { ETH_CRSDV }]; #IO_L6N_T0_VREF_16 Sch=eth_crsdv
#set_property -dict { PACKAGE_PIN C10   IOSTANDARD LVCMOS33 } [get_ports { ETH_RXERR }]; #IO_L13N_T2_MRCC_16 Sch=eth_rxerr
#set_property -dict { PACKAGE_PIN C11   IOSTANDARD LVCMOS33 } [get_ports { ETH_RXD[0] }]; #IO_L13P_T2_MRCC_16 Sch=eth_rxd[0]
#set_property -dict { PACKAGE_PIN D10   IOSTANDARD LVCMOS33 } [get_ports { ETH_RXD[1] }]; #IO_L19N_T3_VREF_16 Sch=eth_rxd[1]
#set_property -dict { PACKAGE_PIN B9    IOSTANDARD LVCMOS33 } [get_ports { ETH_TXEN }]; #IO_L11N_T1_SRCC_16 Sch=eth_txen
#set_property -dict { PACKAGE_PIN A10   IOSTANDARD LVCMOS33 } [get_ports { ETH_TXD[0] }]; #IO_L14P_T2_SRCC_16 Sch=eth_txd[0]
#set_property -dict { PACKAGE_PIN A8    IOSTANDARD LVCMOS33 } [get_ports { ETH_TXD[1] }]; #IO_L12N_T1_MRCC_16 Sch=eth_txd[1]
#set_property -dict { PACKAGE_PIN D5    IOSTANDARD LVCMOS33 } [get_ports { ETH_REFCLK }]; #IO_L11P_T1_SRCC_35 Sch=eth_refclk
#set_property -dict { PACKAGE_PIN B8    IOSTANDARD LVCMOS33 } [get_ports { ETH_INTN }]; #IO_L12P_T1_MRCC_16 Sch=eth_intn


##Quad SPI Flash

#set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports { QSPI_DQ[0] }]; #IO_L1P_T0_D00_MOSI_14 Sch=qspi_dq[0]
#set_property -dict { PACKAGE_PIN K18   IOSTANDARD LVCMOS33 } [get_ports { QSPI_DQ[1] }]; #IO_L1N_T0_D01_DIN_14 Sch=qspi_dq[1]
#set_property -dict { PACKAGE_PIN L14   IOSTANDARD LVCMOS33 } [get_ports { QSPI_DQ[2] }]; #IO_L2P_T0_D02_14 Sch=qspi_dq[2]
#set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { QSPI_DQ[3] }]; #IO_L2N_T0_D03_14 Sch=qspi_dq[3]
#set_property -dict { PACKAGE_PIN L13   IOSTANDARD LVCMOS33 } [get_ports { QSPI_CSN }]; #IO_L6P_T0_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. Schafer, *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 demostration 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 numberical 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