ESTIMATION OF OBJECT ORIENTATION USING SENSOR FUSION AND VHDL

A THESIS

Presented to the University Honors Program

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the

University Honors Program Certificate

Michael Rozbicka-Goodheart

Spring 2021

I, THE UNDERSIGNED MEMBER OF THE COMMITTEE,

HAVE APPROVED THIS THESIS

ESTIMATION OF OBJECT ORIENTATION USING SENSOR FUSION AND VHDL

BY

Michael Rozbicka-Goodheart

I-Hung Khoo

I-Hung Khoo, Ph.D. (Thesis Advisor) **EE and BME Departments**

California State University, Long Beach

Spring 2021

ABSTRACT

ESTIMATION OF OBJECT ORIENTATION USING SENSOR FUSION AND VHDL

By

Michael Rozbicka-Goodheart

May 2021

This research aims to estimate the orientation of an object using sensor fusion by implementing the design through the programming language VHDL, Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language. An objects orientation may be estimated using an IMU, Inertial Measurement Unit, which is commonly used in aircrafts, drones, body movement studies as well as any application where an objects position and velocity is of importance. IMUs have many sensors that approximate the orientation of an object. However, like many devices each of these sensors has disadvantages and advantages that must weighed and chosen depending on the design requirements. Through using sensor fusion, which combines the readings of the sensors, the disadvantages of each individual sensor are minimized to get a more accurate estimation of an object's orientation. Completing the implementation of sensor fusion through VHDL and confirming the results are the contributions of this research.

ACKNOWLEDGEMENTS

I would like to thank Dr. I-Hung Khoo for being my Thesis Advisor and to allow me to learn from him during the two years I have done research under his lead. It is because of Dr. Khoo I have learned so much technical ability that has helped me differentiate myself from other students in projects and internships bettering myself for my future career.

I would like to thank Liza Bledsoe for the continual support and advise through my academic career that has brought me to graduating in 3 years. I would undoubtedly not have been able to accomplish this goal without Liza, and I am very thankful for her.

I would like to thank Dr. Henry Yeh along with the school organization the Institute of Electrical and Electronics Engineers (IEEE) who has provided me many opportunities throughout my Electrical Engineering development that I will remember for the rest of my life.

Finally, I must acknowledge my family and friends who have supported me through my career as a student and given me never-ending support through every stage of my life I have decided to pursue.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

IMU	Inertial Measurement Unit
FPGA	Field-Programmable Gate Array
IDE	Integrated Development Environment
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
SPI	Serial Peripheral Interface
I2C	Inter-integrated Circuit
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPP	General Purpose Processors
VCC	Voltage Common Collector
GND	Ground
SDA	Serial Data
SDI	Serial Data In
SDO	Serial Data Out
SCK	Serial Clock
IP	Intellectual Property

CHAPTER 1

BACKGROUND

1.1 INTRODUCTION

Microcontrollers have increasing popularity as a hobbyist's dream of automating his projects and simplifying his life. An increasing number of resources are established every day for the implementation of microcontrollers for various designs. However, is the microcontroller the path of the future? Are there other options and alternatives that could offer more advantages than the standard microcontrollers? The answer, of course, is dependent on each individual project, but one thing is for certain, integrated circuits consistently outperform microcontrollers in efficiency. The problem with integrated circuits is their inability to be reconfigured making them lack ease of use for the design process. This is not the case for Field Programmable Gate Arrays (FPGA), a reconfigurable integrated circuits board. In recent years the company Intel has made a $17 Billion investment in acquiring Altera a FPGA chip designer that offers benefits over standard use of microcontrollers. In this paper, the implementation of sensor fusion through VHDL is prepared for use on an FPGA. The results of sensor fusion on a standard microcontroller as well as an FPGA will be explored and compared with one another to see the benefits of how an IMU may be better used with an FPGA through VHDL over a microcontroller.

The implementation of sensor fusion is the combination of the different sensors that measure orientation on an Inertial Measurement Unit, an IMU. An IMU is most commonly composed of three sensors, an accelerometer, a gyroscope, and a magnetometer. The accelerometer measures linear acceleration, the gyroscope measures angular velocity, and the magnetometer

measures magnetic field strength. From these sensor readings the IMU can estimate an objects orientation and position along in three dimensions. The orientation along the three axis is also referred to as yaw, pitch, and roll. Yaw is the orientation around the z-axis, pitch is the orientation around the y-axis, and roll is the orientation along the x-axis.

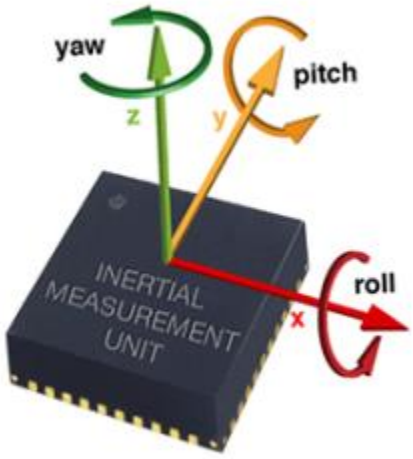


Figure 1. IMU Yaw, Pitch, and Roll depicted.

For the purposes of this research, pitch and roll are calculated while yaw is not needed. This is due to complications that the accelerometer has by measuring orientation in respect to the z-axis. The pitch and roll from the IMU may be calculated by using the accelerator and gyroscope, but there are tradeoffs to the orientation derived from each. Angle estimations from the gyroscope work very well with change in motion but can be unreliable in long term measurements due to drift in the sensor. Angle estimations from the accelerometer do not have the issue of drift like the gyroscope but are noisy in the short term due to external forces applied to the device. To reduce the effects of drift from the gyroscope and the effects of noise from the accelerometer combining the angle from both sensors produces a more accurate estimate and minimizes the disadvantages. This is sensor fusion.

Sensor fusion is often applied with microcontrollers as its implementation are not specific to any program or device but rather mathematical computations. The application of sensor fusion on an FPGA is to get the benefits of sensor fusion with the benefits that are offered while using this device. An FPGA can be thought of as an integrated circuit that is able to be configured and changed after it is manufactured. This is not the case with most microcontrollers such as Arduinos as they do not build an integrated circuit. Other options such as application-specific integrated circuits do not allow for the ease of reconfiguration that an FPGA has. The integrated circuit that is created on the FPGA is able to run much faster and more efficient than a microcontroller by getting rid of the need for the device to have to constantly check the states of inputs. As stated on Sparkfun's FPGA article "Using FPGAs", the FPGA's speed is faster because of this ability but also the power consumption is reduced due to lack of need to process a state constantly and read inputs indefinitely. FPGAs have optimal performance per watt and can be more than 3 to 4 times more efficient in power and speed than that of a GPU. Another factor that allows the FPGA to work faster than the alternative is the ability to process in parallel. This is because the integrated circuit is built within the FPGA and so it can simultaneously read and process information (SparkFun 1).

IMU's are used often in applications where orientation or position measurement is desired. This includes design of aircrafts, drones, robots that move, measuring body movement, etc. and it may be tempting to use IMUs with microcontrollers due to the abundance of resources available for implementation and use. However, this research may help cross the bridge of combining the IMU and the FPGA. The completion of this research provides more resources for implementation so that the benefits of an FPGA with an IMU can be used over standard microcontrollers.

In this research the Arduino Uno board and the corresponding Arduino Integrated Development Environment (IDE) software will be used as a reference point of running the IMU and sample data. As found on Digilent's Website, the Basys 3 Artix-7 FPGA Trainer Board and the corresponding program of Vivado will be used for programing and testing in VHDL. The IMU that will be used is PMOD Nav: 9-axis IMU Plus Barometer provided by Digilent, the same company that also sells the Basys 3 boards.

1.2 EQUIPMENT USED

The Basys 3 FPGA board that is used in this research is a part of the curriculum for EE301 as well as the corresponding program of Vivado. The use of the FPGA from course material provides CSULB students with previous experience to aid with the implementation for this research.

The IMU that is chosen has a 3-axis accelerometer, 3-axis gyroscope , 3-axis magnetometer, and a barometer providing 10 degrees-of-freedom functionality. However, for the purposes of this research only the 3-axis accelerometer and the 3-axis gyroscope will be used. The use of the other sensors allows for future development of this research and growth on the study that is achieved in this paper. The IMU utilizes the LSM9DS1 chip that will be used and may be referred to from the datasheet as this is a common chip for IMUs.

The common forms of communication between an IMU and a controller are SPI and I2C. Serial Peripheral Interface (SPI) and Inter-integrated Circuit (I2C) are communication protocols that allow communication between a controller and a device referred to as a master and a slave. In the case of the implementation for this paper the FPGA is the master and the IMU is the slave. SPI

and I2C have their own advantages and disadvantages for use that is not explored in this research. Both SPI and I2C will be prepared for implementation but the results in the Arduino will come from I2C due to reasons explored later and the results in VHDL will be simulated to prove implementation.

1.3 LITERATURE REVIEW

Other works have explored the use of sensor fusion with a controller as well as the use of FPGA to be used as a master with a slave device. However, there are little resources on the implementation of and IMU with sensor fusion and an FPGA together. In *Adaptive Linear Quadratic Attitude Tracking Control of a Quadrotor UAV Based on IMU Sensor Data Fusion* by Nasrettin Koksal, Mehdi Jalalmaab, and Baris Fidan a closed loop control system design is built with the addition of sensor fusion at the University of Waterloo. The use of sensor fusion in a closed loop control system design for a drone application allows for the increase in quality of orientation and position estimation. For projects such as a drone where orientation and position are very important, sensor fusion offers many benefits. In *High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU* by Xiang Tian and Khaled Benkrid the performance of an FPGA compared to that of a GPP and a GPU is studied in the application of Quasi-Monte Carlo simulation. This study found that FPGAs outperform GPP implementations by 2 orders of magnitude and GPU implementations with 3x the speed. This study also finds that in regard to power consumption FPGAs are 336 times more energy efficient than CPUs and 16 times more efficient than GPUs. While both studies offer insight into the hypothesis of this research and how an FPGA could offer many potential benefits over alternative devices both these studies

exclude sensor fusion combined with the FPGA. The combination of these two elements is the subject explored by this research.

As for the resources that will contribute to the implementation and completion of this work the datasheets for all components will be looked at and analyzed. The documentation offer specific instruction that become useful for any form of implementation with these devices. More specifically the documentation for the PMOD Nav: 9-axis IMU Plus Barometer and the Basys 3 Artix-7 FPGA Trainer Board will be relied on for technical specifications throughout this research.

For the implementation of the IMU on the Arduino there is a previous project created by Sparkfun Electronics that utilizes their breakout board IMU. While this is not the same IMU that will be used in this research the Sparkfun breakout board IMU utilizes the LSM9DS1 chip similar to the PMOD Nav. For this reason, the tutorial and example code may be a useful resource in applying the IMU on the Arduino.

For implementation of the of the IMU with the FPGA SPI and I2C will be used as communication protocols between the two devices. To contribute with the implementation of these protocols using the FPGA as the master, on there are previous examples and works provided on the Digikey Electronics forum. These may be a great beginning portion for code to build upon to make the master code for the FPGA.

CHAPTER 2

HYPOTHESIS/RESEARCH DESIGN

To achieve the estimation of orientation using sensor fusion in VHDL the results of this implementation must be compared to that a working example of sample data. It is the hypothesis of this research that the VHDL code to be implemented on an FPGA will lead to a much faster computation process and still accurately depict the results of sensor fusion. If the results of sensor fusion are achieved and compared to that of a working example with measured time, the result of this research is a success.

A known working example of sensor fusion must be created and analyzed using an Arduino Uno microcontroller. This implementation in the Arduino microcontroller will not only give data as a working example of sensor fusion but also provide a comparison point for the FPGA. The data for the time elapsed through the microcontroller can be analyzed and compared to that of the FPGA data and timing. Once this data is achieved, the protocols of SPI and I2C will be created, tested, and prepared for implementation in VHDL. This SPI and I2C creation and testing in VHDL allows for the expansion on this research as the next steps could include implementing the data straight from the hardware rather than testing it with sample data. Finally, the sensor fusion will be created in VHDL and then tested and compared to that of the original data achieved by the Arduino. If the sensor fusion results in the Arduino matches the sensor fusion in VHDL then the implementation in VHDL is successful.

CHAPTER 3 METHODOLOGY

3.1 IMPLEMENTATION ON ARDUINO

The benefit of using microcontrollers especially one as common as the Arduino is the number of resources that can be found from the multitude of users. As previously discussed, the PMOD Nav that is used for this research utilizes the LSM9DS1 microchip, luckily for the process of implementation on the Arduino there are many other IMUs that utilize this chip. There is even an accessible downloadable library that may be added to the Arduino IDE for the LSM9DS1. A library for the Arduino IDE is a set of built code and functions that allow for ease of use of that chip. In the case of the LSM9DS1, this library helps extract information from the sensors in the IMU and uses built in functions that simplify the process of use. In the resource referenced in the Literature Review, Sparkfun Electronics has already developed a working code that allows for the implementation of another IMU utilizing the LSM9DS1 and extracting and interpreting the data from this device. It is the first portion of this research to use this example and create a working code that can extract information from the PMOD Nav and be received in the Arduino Uno.

The SparkFun Electronics hook up and code is set up for the SparkFun 9DoF IMU Breakout Board but can be translated for the purposes of the PMOD Nav. The I2C implementation for the PMOD Nav with the SparkFun example code must first have all the proper connections so that the Arduino can extract the necessary information. I2C communication protocol utilizes 4 pins for communication, VCC, GND, SDA, and SCK. VCC is the Voltage Common Collector and is the power voltage hooked up to the board. The GND is the ground for the circuit. The SDA is the Serial Data connection between the IMU and the controller where data is received and sent. The

SCK is the Serial Clock which is used to carry the clock signal so that the two devices are synchronized. Below may be seen the PMOD Nav pin out from the documentation of the device so that it can be determined which connections may be needed for I2C communication.

Pinout Description Table

Header J1						Header J2		
Pin	Signal	Description	Pin	Signal	Description	Pin	Signal	Description
1	CS_A/G	Chip Select for Accel/Gyro	7	INT	Interrupt pin for all components	1	INT_M	Interrupt for the Magnetometer
2	SDI	Master-Out-Slave-In	8	DRDY_M	Data Ready for the Magnetometer	2	INT_ALT	Interrupt for the Altimeter
3	SDO	Master-In-Slave-Out	9	CS_M	Chip Select for the Magnetometer	Header JP1		
4	SCK	Serial Clock	10	CS_ALT	Chip Select for the Altimeter	1	INT1_A/G	Interrupt 1 for Altimeter/Gyroscope
5	GND	Power Supply Ground	11	GND	Power Supply Ground	2	DEN_A/G	Data Enable for Altimeter/Gyroscope
6	VCC	Power Supply (3.3V)	12	VCC	Power Supply (3.3V)			

Table 1. PMOD Nav Pin out Description Table

As seen from this pin out in Figure 2, the pins of VCC, GND, and SCK become clear as these are labeled as pins six, five, and four respectively. However, SDA is not as exact as seen above. Pin 2 is SDI and pin 3 is SDO. These pins are used more commonly in SPI communication and will be explored later. For the purposes of I2C communication after trial and error it is found that SDI or pin 2 is used for the connection to SDA. The VCC pin is connected to 3.3V on the Arduino as this is what the IMU operates on and the standard 5V would destroy the chip. The outputs from the IMU also come out as 3.3V but the digital ports of the Arduino read any voltage above 3V as a binary “1” or a “HIGH”. The GND port of the Arduino is connected to the ground port of the IMU. The SCK pin is connected to Analog Input pin 5 on the Arduino and the SDA pin is connected to Analog Input pin 4 on the Arduino. The connections may be seen below.

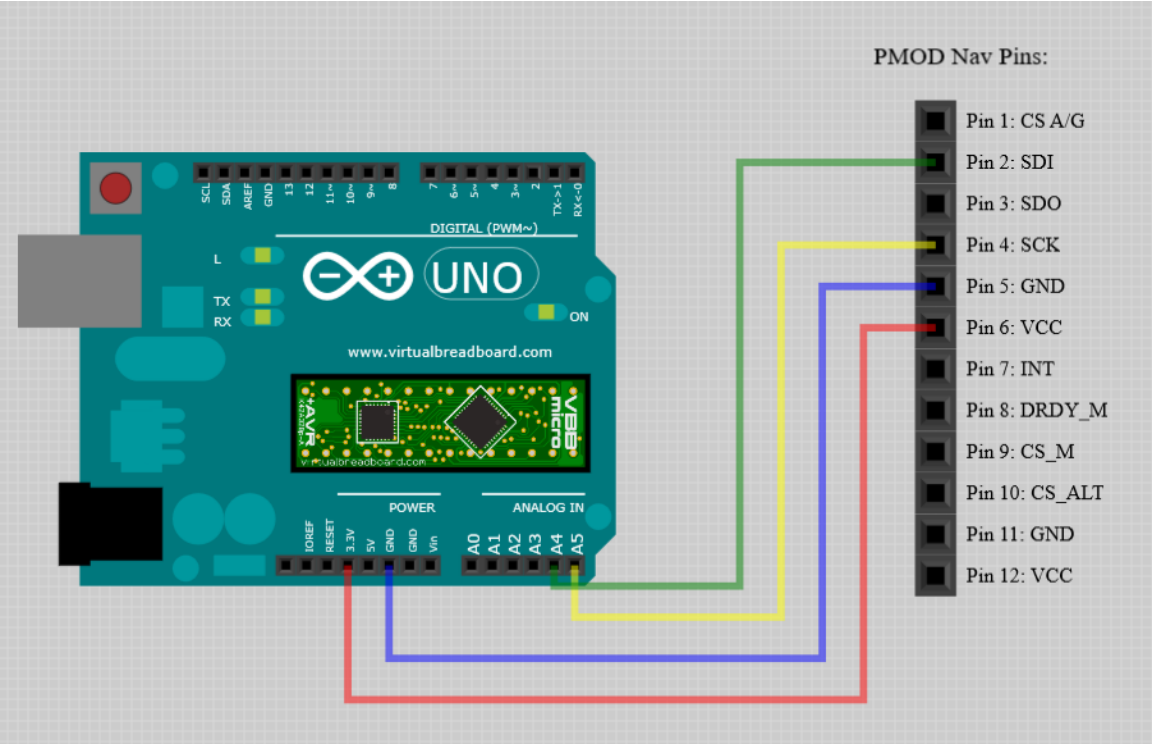
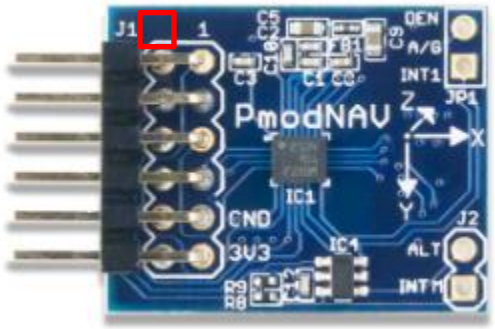


Figure 2. I2C Connections with PMOD NAV



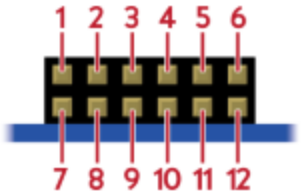


Figure 3. PMOD Nav for reference and the pins as shown above.

The code for the I2C communication with the PMOD Nav is heavily influenced by the example code provided by Sparkfun Electronics. This modified code may be found in Appendix A. The code that is used to get the data from the IMU deciphers the readings from the sensors to understandable values that measure needed variables. This is done by using the functions imu.calcGyro( ) and imu.calcAccel( ) functions provided by the Sparkfun library. In terms of the gyroscope, imu.calcGyro( ) returns the approximation of the angular velocity measurements ($\widetilde{\omega}_x$, $\widetilde{\omega}_y$, and $\widetilde{\omega}_z$) from the raw ADC values that the sensors provide to two decimal spot precision. As for the accelerometer this function returns the approximation of linear acceleration ($\tilde{a}_x$, $\tilde{a}_y$, and $\tilde{a}_z$) from the raw ADC values that the sensors provide with two decimal place precision. The major change and addition in this code is the orientation measurements from the accelerometer and gyroscope and the addition of sensor fusion rather than utilizing the sensors individually.

To achieve sensor fusion on the Arduino the orientation must first be acquired from the gyroscope and the accelerometer individually. The gyroscope in the IMU finds the angular acceleration of the object and then the orientation may be found by integration. The equations for this integration can be seen below.

$$\theta_x(t) = \theta_x(t - \Delta t) + \widetilde{\omega}_x * \Delta t$$

$$\theta_y(t) = \theta_y(t - \Delta t) + \widetilde{\omega}_y * \Delta t$$

Where:

$\theta_x(t)$ is the roll.

$\theta_y(t)$ is the pitch.

$\theta(t - \Delta t)$ is the previous angle estimate.

$\widetilde{\omega}$ is the measured angular velocity

Δt is the time change between measurements.

$\theta_x(t)$ is the orientation in respect to the x-axis, also known as the roll, and $\theta_y(t)$ is the orientation in respect to the y-axis, also known as the pitch. Yaw is not considered for this research so the orientation in respect to the z-axis is not necessary but would follow the same process for derivation. This formula is found from the formal definition of an integral as can be seen below.

$$\int f(x)dx = \lim_{n\to\infty} \sum_{i=0}^{n} f(x_i) * \Delta x$$

From this definition as the change in time becomes infinitely small the sum of all the terms become equal to the integral of the function. This means for the purposes of our application the smaller our change in time can be the better our results will be for an estimation of the orientation. This sensitivity in the change in time makes a faster controller a better integrator compared to a slower one. As for our equation with the gyroscope, the integral of velocity is equal to position. This is why the integral equation is used to obtain the angular position. As for the change in time in application, theoretically the time change needs be infinitely small, but this is not realistic and instead this equation produces an approximation of the integral.

Orientation from the accelerometer can be found by using trigonometric identities with the acceleration terms the sensor provides. The accelerometer provides three vectors in the 3 dimensions.

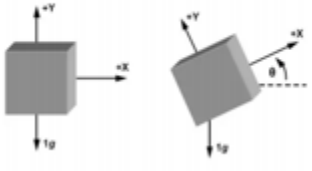


Figure 4. 3 Vectors from Accelerometer

The vectors can get the value of orientation by using the trigonometric identities associated with tangent. The equations can be seen below to derive orientation of pitch and roll.

$$\theta_x(t) = \tan^{-1}\left(\frac{\tilde{a}_y}{\sqrt{\tilde{a}_x{}^2 + \tilde{a}_z{}^2}}\right)$$

$$\theta_y(t) = \tan^{-1}\left(\frac{-\tilde{a}_x}{\sqrt{\tilde{a}_y{}^2 + \tilde{a}_z{}^2}}\right)$$

Where:

$\theta_x(t)$ is the roll.

$\theta_y(t)$ is the pitch.

$\tilde{a}$ are the linear acceleration vectors.

Recall that yaw is not calculated for this research, but for the case of the accelerometer yaw in unable to be calculated as the orientation of this term is affected by the acceleration due to gravity.

Now that the orientations for the gyroscope and the accelerometer have been determined, sensor fusion may occur by following the equation below by using a complementary filter.

$$\theta_{roll} = \theta_{Gyroscope_{roll}} * \alpha + \theta_{Accelerometer_{roll}} * (1 - \alpha)$$

$$\theta_{pitch} = \theta_{Gyroscope_{pitch}} * \alpha + \theta_{Accelerometer_{pitch}} * (1 - \alpha)$$

Where:

θ_{roll} is the orientation of device in respect to the x-axis.

θ_{pitch} is the orientation of the device in respect to the y-axis.

$\theta_{Gyroscope_{roll}}$ is the orientation in respect to the x-axis from the gyroscope.

$\theta_{Gyroscope_{pitch}}$ is the orientation in respect to the y-axis from the gyroscope.

$\theta_{Accelerometer_{roll}}$ is the orientation in respect to the x-axis from the accelerometer.

$\theta_{Accelerometer_{pitch}}$ is the orientation in respect to the y-axis from the accelerometer.

α is a number between 0 and 1 to mesh the two values.

The variable of α is found through trial and error with each individual device to get the most accurate results.

Using these equations for orientation acquisition and sensor fusion the Arduino can accurately obtain the values for pitch and roll and display these values to the user through the on-board serial monitor. The serial monitor from the Arduino is a pop-up screen that allows the board

to print and receive text from the computer acting as a user interface. The code for the Arduino can be found in Appendix A. The results for this portion of the research may be found in the Chapter 3 of this paper.

The process of developing code for sensor fusion with SPI communication on the Arduino is the same mathematical computation for sensor fusion. However, an issue arises with the PMOD Nav and the previously used LSM9DS1 library for the Arduino IDE. While there are examples found for SPI communication with the LSM9DS1 not all the ports needed for the SPI communication are attached to a pinout on the PMOD Nav. Commonly when a chip is integrated into a device, like the PMOD Nav, every output on the computing chip does not have an accessible pin out. While the chip on the device has outputs on them not all the outputs are connected and have a pin that is able to be easily connected to. This added some complication with implementation of SPI on the Arduino and further exploring I2C as the main form of communication protocol became the focus of this research. The I2C allows for the understanding of the IMU data and gives enough sample data for the implementation and testing of sensor fusion in VHDL.

3.2 SPI AND I2C FPGA MASTER CODE AND IMPLEMENTATION

The next portion of this research is dedicated to developing working code that may be implemented with the FPGA for communication with the PMOD Nav IMU. The code and communication protocols making the FPGA the “master” must be developed in VHDL so that it may be prepared to be uploaded to the Basys 3 FPGA Trainer Board. While I2C data is used and

simulated for the results of this research, this portion of the research sought to develop and test both the I2C and SPI protocols. As introduced earlier SPI and I2C are protocols of communication between a master device and a slave device. SPI does not have start and stop bits for communication like that of I2C, and instead relies on a Master-In Slave-Out (MISO) communication line and a Master-Out Slave-In (MOSI) communication line. Comparatively I2C uses only SDA as the communication line. Both protocols have their advantages and disadvantages and the decision to use one over another depends on the design specifications. For this purpose, in this research both I2C and SPI protocols were developed and tested in VHDL, however I2C was used to acquire and used as reference data in the Arduino.

When evaluating how to use SPI and I2C communication in VHDL, an example may be found on the Digikey Electronics forum that is referenced in the Literature Review with example code and an example waveform (Digikey Forum). These provided examples may act as a reference to build the needed interaction for the FPGA and test functionality by comparison. The example waveform can be used as test data and a model to prove functionality of this research's generated code.

The example code provided through the Digikey forum provides a strong base to use but quickly it may be seen that the data types that are used within this master code do not comply with Post Synthesis Simulations. In VHDL the data types are more sensitive to abnormalities than that of the C++ code that the Arduino IDE is based off of. The data types in VHDL that are standard for every application and for the purposes of this research are STD_LOGIC and STD_LOGIC_VECTOR. When implementing a design for VHDL the general process to follow is development of the code with synthesizing it in Vivado. Simulation of the code with test values and prove that the code is functional by using a testbench and simulate in Vivado. Create

constraints for the input and output variables that attach the variables to the actual hardware through a constraints file, and finally generate a bitstream and upload to the FPGA. While the code provided by the Digikey Forum does synthesize, it is unable to perform the simulations necessary for this research. For that reason the code must then be translated to STD_LOGIC and STD_LOGIC_VECTOR data types so that a Post-Synthesis Functional Simulation may be ran. This updated code may be found in Appendix B. The code developed must still follow the following block diagram found depicted from the Digikey forum that can be seen below.

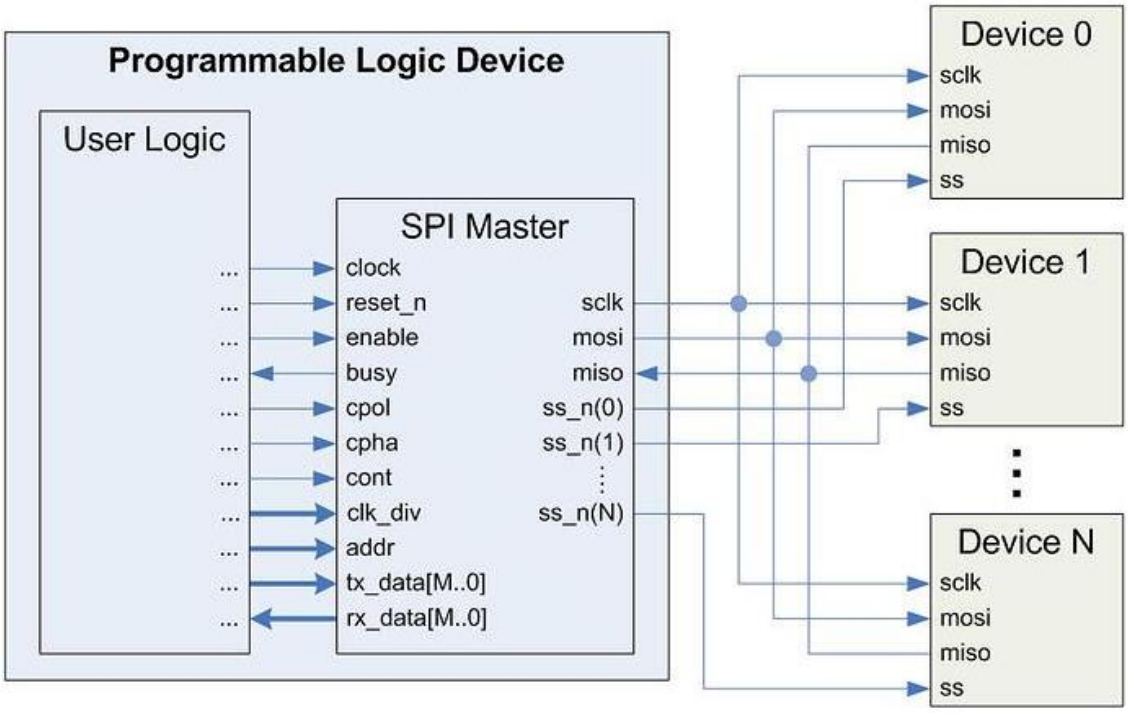


Figure 1. Example Application

Figure 5. 3 Structure of SPI Master and User Logic found on DigiKey Website

Once the code is developed a testbench code must be then created to simulate the inputs that the code may get and run a simulation in order to prove that the correct results were achieved. This simulation is compared to that of the results on the Digikey forum website. The Testbench code may also be found in Appendix B and the results may be found in the Results portion of this paper.

This same process is followed to develop the I2C VHDL code and test bench code to compare the simulation results with that of the Digikey forum. The results may be found in the results portion of this paper and the code for device and the testbench code may be found in Appendix B. The block diagram for the I2C communication and VHDL Code may be found below.

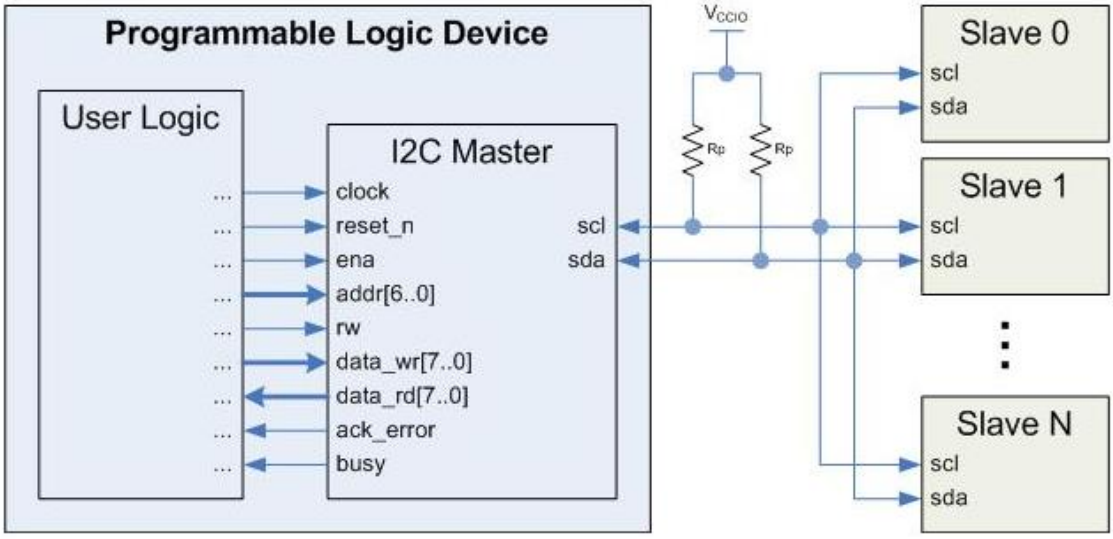


Figure 6. 3 Structure of I2C and User Logic.

3.3 TRANSLATION OF SENSOR FUSION TO VHDL

The concluding portion of this research is creating VHDL code to find the gyroscope pitch and roll, accelerometer pitch and roll, and then accomplish sensor fusion with the results. This code will be tested with sample data that is taken from the Arduino and then compared with the

Arduino results. The VHDL code may be generated using a similar process to that of the generation of the Arduino code. The equations that were derived for the pitch and roll in Chapter 2.1 may be used and directly implemented in VHDL code using the IEEE Library found in Vivado. However, there must be some changes with the VHDL code that did not have to be considered in Arduino.

As explored in the previous section Vivado can universally run with the STD_LOGIC and STD_LOGIC_VECTOR data types so these data types must be used in the implementation of the calculation code. These data types are represented by binary numbers so all numbers must be converted to their binary values. While the same formulas must be used to implement the orientation generation and sensor fusion, Vivado does not have a built in function to generate trigonometric identities. In order to implement this Vivado, the Intellectual Property Catalog (IP Catalog) must be used where preconfigured logic functions may be used in the design. To get the values for arctangent for the accelerometer orientation calculations the CORDIC IP must be utilized and set to calculate the arctangent value. This process and result will be displayed in the Results Chapter of this paper. The block diagram below depicts the method that will be followed in the development of the code for the IMU calculations.

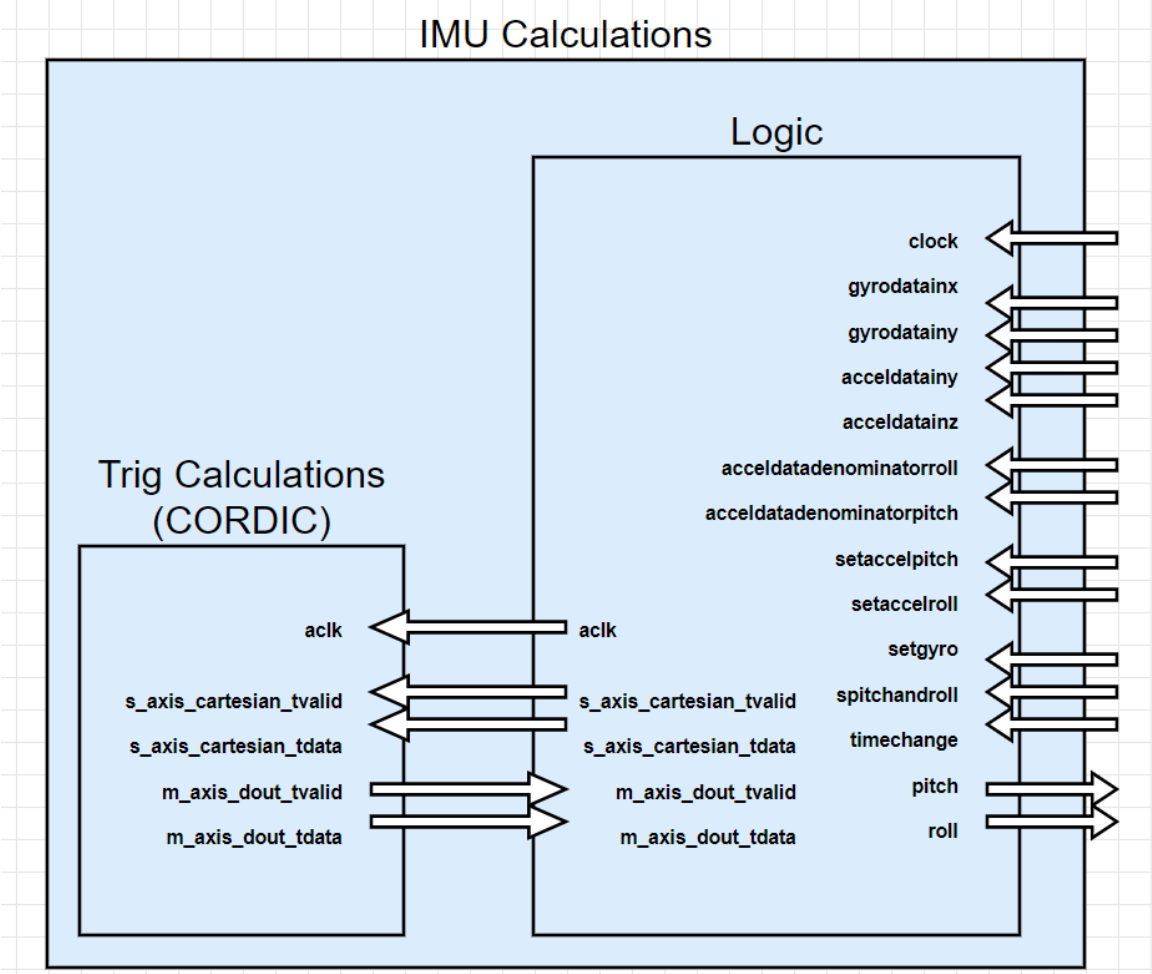


Figure 7. 3 This kind of block diagram for the implementation in VHDL

As seen from this block diagram the inputs for the VHDL code that is being designed is all the setting data as well as the readings from the IMU. The outputs for this code are the combined pitch and combined roll after sensor fusion. The connections between the blocks of "Logic" and "Trig Calculations" are the design specified inputs and outputs for the CORDIC IP that is being used to perform the trigonometric identities.

The specifications of these inputs for the trig identities are that the input's first two bits are before a decimal place and the rest of the bits are behind a decimal place. The two inputs are

combined and put into s_axis_cartesian_tdata variable. For the output, the first three bits are before the decimal place and then the rest are behind and put into m_axis_dout_tdata. For example, if the desired result for the arctangent is arctanget(0/1) where arctangent(Yout/Xout) assuming each input and output is 8 bits, then Yout = "00.000000" and Xout = "01.000000". This makes s_axis_cartesian_tdata = "0000000001000000" or "Yout & Xout" and m_axis_dout_tdata = "000.00000". The output of the arctangent function is in radians and this must be remembered for conversion in the results. To calculate the values of the binary output it can be handled just the same as any binary number. Each place away from the decimal increases or decreases the power of two that the number represents. For example, to calculate "011.10101" the process of the powers would be followed.

"011.10101" = $0*2^2+1*2^1+1*2^0+1*2^{-1}+0*2^{-2}+1*2^{-3}+0*2^{-4}+1*2^{-5}$ = 3.65625

Using this process all the numbers are then shifted so that the decimal part is always 3 bits after the first bit. This allows for the arithmetic to work smoothly with one another. The output of the arctangent function also utilizes two's complement to represent negative numbers. This is achieved by inverting all the bits and then adding one with the end bit. This may be seen in the table shown below.

2's complement table

Positive Decimal Number	Binary Representation	Binary Representation	Negative Decimal Number
0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

Table 2. Two's complement with 4 bits.

As seen in this table to get the value of -2 from the value of 2 or "0010", all the bits are first inverted to get "1101" and then an additional bit is added to get "1110". Two's complement offers the largest range of negative and positive numbers when negative numbers are needed. This two's complement will be used to represent negative numbers and a negative orientation in this research.

CHAPTER 4

RESULTS

4.1 RESULTS OF ARDUINO IMPLEMENTATION

The code that is used for the Arduino Implementation may be found in Appendix A. The result of the code is printed on the serial monitor and can be found below.

```
14:36:26.081 -> □G: 4.39, 5.22, 1.77 deg/s
14:36:27.703 -> A: -0.01, 0.08, 0.99 g
14:36:27.703 -> Denominator for Pitch Calculation:  1.00
14:36:27.703 -> Denominator for Roll Calculation:  0.99
14:36:27.703 -> Accel Pitch, Roll: -0.71, -4.40
14:36:27.703 -> Gyro Pitch, Roll: 0.03, 0.03
14:36:27.703 -> DeltaTime for Gyro Equation: 6
14:36:27.703 -> Combined Pitch, Combined Roll: -0.34, -2.19
14:36:27.703 ->
14:36:27.703 -> G: 22.54, 4.82, 1.44 deg/s
14:36:27.703 -> A: -0.05, 0.11, 1.05 g
14:36:27.703 -> Denominator for Pitch Calculation:  1.05
14:36:27.703 -> Denominator for Roll Calculation:  1.05
14:36:27.703 -> Accel Pitch, Roll: -2.98, -6.16
14:36:27.703 -> Gyro Pitch, Roll: 0.13, 0.48
14:36:27.703 -> DeltaTime for Gyro Equation: 20
14:36:27.749 -> Combined Pitch, Combined Roll: -1.43, -2.84
14:36:27.749 ->
14:36:27.749 -> G: 7.52, 1.23, 0.17 deg/s
14:36:27.749 -> A: 0.00, 0.10, 0.99 g
14:36:27.749 -> Denominator for Pitch Calculation:  0.99
14:36:27.749 -> Denominator for Roll Calculation:  0.99
14:36:27.749 -> Accel Pitch, Roll: 0.02, -5.77
14:36:27.749 -> Gyro Pitch, Roll: 0.16, 0.66
14:36:27.749 -> DeltaTime for Gyro Equation: 24
14:36:27.749 -> Combined Pitch, Combined Roll: 0.09, -2.56
```

Figure 8. Example Data that is generated from IMU using I2C on Arduino

As seen from the data above there is a column of numbers on the left with a "- >" pointing to letters and numbers. This column on the left is the timestamp in a 24 hour scale. As seen above these data samples were taken at approximately 2:30 P.M. To the left of the "- >" is what the code declares to be printed. In this figure there are three samples of printed code at different times.

The structure of the code first prints the character "G:" and then prints the values of $\tilde{\omega}_x$, $\tilde{\omega}_y$, and $\tilde{\omega}_z$ in that order followed by a "deg/s" as these values are angular velocity. Recall that the VHDL code does mathematical computations in radians so this will have to be translated. Following this is the term "A:" and then the terms for $\tilde{a}_x$, $\tilde{a}_y$, and $\tilde{a}_z$ followed by "g" as the acceleration is in the form of gravity. Following this is the Denominator values for the Pitch and the Roll or $\sqrt{\tilde{a}_y{}^2 + \tilde{a}_z{}^2}$ and $\sqrt{\tilde{a}_x{}^2 + \tilde{a}_z{}^2}$ respectively. This calculation is used for the test data in Chapter 4.3. Following this is the pitch calculations from the accelerometer and then the pitch calculations from the gyroscope. The change in time since the last reading is also printed to use as data for sensor fusion testing and then the combined pitch and roll after sensor fusion is printed.

The result above from the figure sets a baseline of what the VHDL sensor fusion code should produce and how it may be achieved. There are some errors to consider when evaluating this code. The built-in functions that are used to translate the raw ADC values from the sensors into the angular velocity and linear acceleration terms are an estimation and round when printing their values. For example, in the 3rd data sample set the $\tilde{a}_x$ term is 0.00 but the acceleration pitch is not 0 meaning that this $\tilde{a}_x$ is not truly the value of 0 but rather it does not print to a precision that the value is able to be seen. The arctangent result that would be achieved if also using the raw ADC values is slightly different than when using these converted values. These converted values

are preferred over the raw ADC values though due to the future implementation into VHDL and only being able to represent these values with two bits above the decimal spot. This means the highest number able to be represented in VHDL is less than four. This restricts for our purposes of this research to utilize the converted values, but recognition of errors is important. Also, the same approximation issues occur when the arctangent equation is done produces a slightly different result than a calculator or other computation software could produce.

When applying this code and determining the optimal value for alpha, it became noticeable that the polarities of the gyroscope and the accelerometer were flipped. When the gyroscope read a negative angle, the accelerometer read a positive one and vice versa. To get rid of this issue a negative polarity was multiplied to one of the values so that the two sensors read the same polarities when moving. After this change when the sensor was moved and rotated around the values were equivalent. The alpha that best fit this system was determined to be 0.5 for the most accurate results.

There is more error when the IMU does not start in a position of 0, this is because the gyroscope assumes that the initial position of the gyroscope is 0 so only when the accelerometer also reads 0 as the initial reading are the two readings synched. When this is achieved the code properly measures and prints all needed values and test values from the IMU using the Arduino.

4.2 RESULTS OF SPI AND I2C VHDL IMPLEMENTATION

The code for this section may be referred to in Appendix B of this paper. Recall that the methodology that was followed to produce this code was by changing the SPI and I2C master code that was provided by the Digikey Forum. Below can be seen the simulation results from the Digikey Forum of the SPI Master Code.

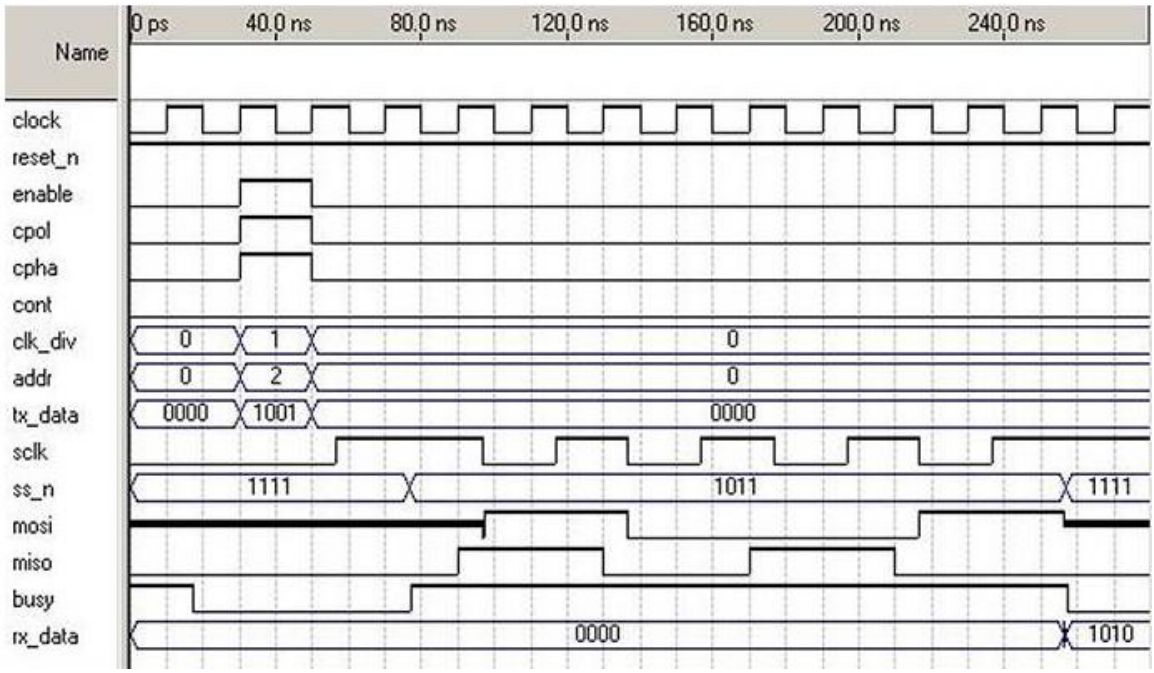


Figure 9. Reference Waveform from Digikey Forum

As seen from this reference waveform the signals names are listed on the left and their respective values at given times may be seen so that replication is possible. When the line is high this means the value is a "1" and when it is low it is a "0". This is not the case for signals that represent multiple bits, for example tx_data is represented in this by 4 bits where between 30 ns – 50ns the value is changed from "0000" to "1001" and then back to "0000". Tx_data is the data that is being transmitted while rx_data is the data that is simulated to be received.

After the master code was translated the resulting testbench code was created to simulate the desired values and the resulting waveform can be seen below. Rather than displaying 4 bits for a signal that represents 4 bits, Vivado represents that number in Hexadecimal. This can be seen for the values of tx_data and rx_data.

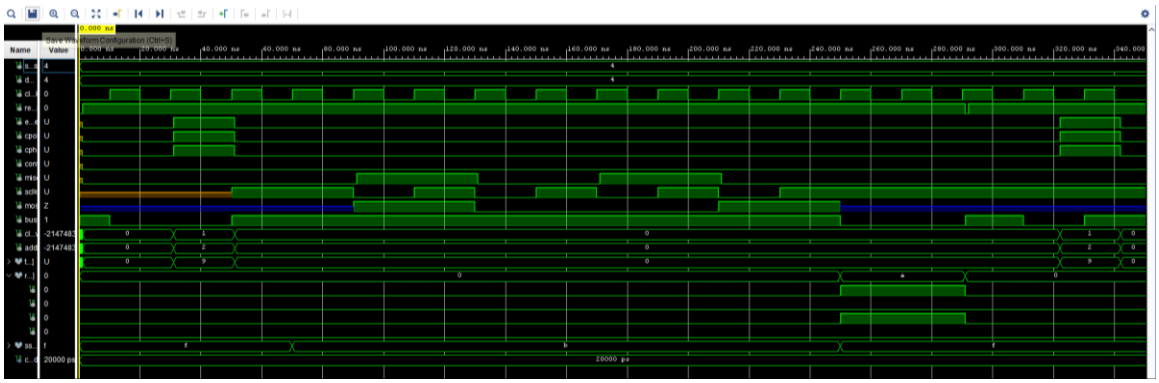


Figure 10. Simulated Results from Translated SPI Master Code in Vivado

As seen from this simulation the results match up exactly with the results of the SPI master reference waveform. The signals may be gone through one by one to view the similarities in them but by evaluating tx_data and rx_data the values are the same. Tx_data is an input signal and reads “9” or if represented by 4 bits is “1001” and then rx_data after a portion of time tx_data reads “a” or also known as “10” in decimal or “1010” in binary. This amount of time and how the signals operate between the transmitted data and received data is determined by the SPI protocol.

Following the SPI communication being confirmed and tested, it next the I2C code to be tested and compared. Below can be seen the Digikey Forum sample simulation to be used as reference to compare and prove functionality of our translated I2C master code. This I2C master code can be found in Appendix B.

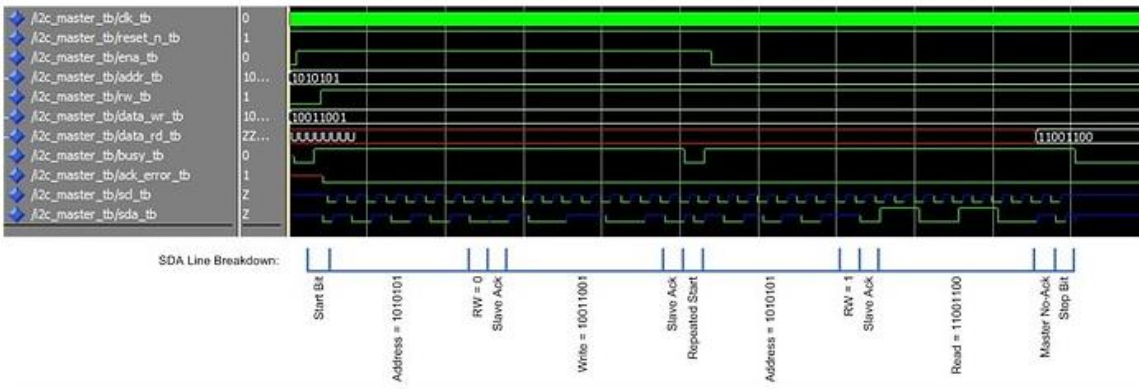


Figure 11. I2C Master Reference Simulation from Digikey Forum

Similarly to that of the SPI reference waveform this is used to compare the translated I2C code simulation. This waveform is slightly different than that of the SPI waveform as this one does not explicitly show time data. To account for this approximations of the time were made and applied in the testbench. The defining of the SDA line had to also be interpreted after multiple trials. The resulting waveform can be seen below from the translated code and simulated using a testbench code in Vivado.

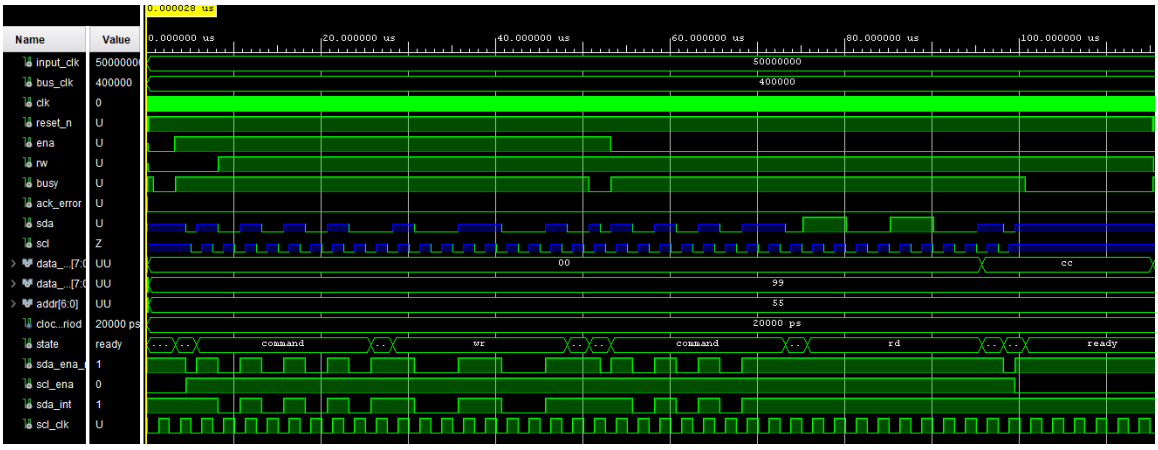


Figure 12. I2C Master Translated Code with generated Testbench in Vivado

When Comparing the results from the Vivado Simulation with the reference waveform they are very similar. There are slight differences in timing with some of the signals, but they hold the same shapes and these differences in timing could be accounted for by different times in the simulation of occurrence. Since the timing of the test bench was not known for the I2C simulation the timing data is generated by approximation. There was difficulty in applying this I2C signal because at many points the SDA line must be set as High Impedance or "X" in Vivado and it is only brought low for communication but then it must have values as the master must convey information to that of the slave. This was unable to be evaluated from the reference waveform and was rather added after trial and error to get the correct result.

Once these two waveforms were generated and confirmed to be accurate the configuration files were generated for SPI so that it could be uploaded to the FPGA. The file was uploaded and tested using an Arduino as the slave and the Basys 3 FPGA board as the master. The results can be seen below.

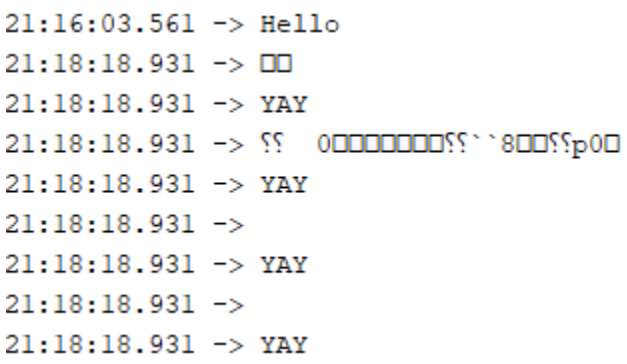


Figure 13. Implementation of SPI Master on FPGA with Arduino as slave and reading output from serial monitor.

In this portion of the implementation the Arduino had to be hooked up to a voltage level converter because signals from the Arduino are sent at 5V and the FPGA operates on 3.3V. The voltage level converter was set up and tested and then the two boards were hooked together with the Arduino running SPI “slave” code. The Arduino’s slave code would print “Hello” when the program began and then “YAY” when a transaction between the master and slave occurred along with the data that was transmitted. As seen in the figure above, the data that is translated is unintelligible. The data was meant to be sending a value representing the character “a” using the ASCII Table that represents characters from binary values. This character was not translated and the most likely culprit is differences in the different clocks in the Arduino and the Basys 3 Board. The Basys 3 board has a clock that operates at 100MHz while the Arduino board operates at 8MHz. This means the Basys 3 board is significantly faster than that of the Arduino board and most likely the unintelligible data being sent was due to the data being transmitted to fast and the Arduino unable to process it. The following process was not continued with I2C as the same issue would arise and testing the data straight with the IMU would not allow to make sure that the values were correct.

4.3 RESULTS OF VHDL SENSOR FUSION

As discussed in the methodologies chapter of this paper, the use of trigonometric identities in VHDL in Vivado was through using the IP resources that are available on the program. The resulting code for implementation and testbench code are found in Appendix C. The resulting data from Figure 8 shown in Chapter 4.1 has 3 sets of data and are the three sets of data used for sensor fusion in this portion and compared. It was later determined that VHDL cannot take the square

root of the function. To get around this the denominator that includes a square root was manually inputted. The arctangent IP that was implemented in this code is also used twice and because of this takes double the time as both need to be set separately. This does not utilize the proper parallel benefits of an FPGA has and to increase the speed another arctangent IP could be added .

To convert the data from the Arduino into VHDL data types conversions must be made and there is of course some error going to be achieved as the bits only show so much precision of a number. The first set of data as well as the corresponding waveform and waveform results may be seen in the table below and the waveform.

First Set of Data:

Gyroscope Converted to Radians	
Gx = 0.0766199542 rad/s	Gy = .091106187 rad/s
Gyroscope Radians Converted to Binary	
Gx = 000.00010011100111010110000000000 True Value: 0.0766201019287109375	Gy = 000.00010111010100101100000000000 True Value: 0.091106414794921875
Accelerometer Binary Values:	
Ax = -.01 +0.1: 00.00000010100011110110000000000 After Two's Complement: 11.11111010111000010100000000000	Ay = 0.08 Binary: 00.00010100011110101110000000000 True Value: 0.0799999237060546875

Denominator for Acceleration Roll	**Denominator For Acceleration Pitch**
Denominator for Acceleration Roll: .99 Binary: 00.11111101011100001010000000000 True Value: 0.9899997711181640625	Denominator Pitch : 1 Binary: 01.00000000000000000000000000000 True Value: 1
DeltaTime Value:	
DeltaTime = 0.006 Binary: 000.00000001100010010011000000000 True Value: 0.00599956512451171875	
Combined Pitch after Sensor Fusion from Arduino Code:	
Combined Pitch = -0.34 deg -.0059341195 rad	
Combined Roll after Sensor Fusion from Arduino Code:	
Combined Roll = -2.19 deg -.0382227106 rad	
Combined Pitch after Sensor Fusion from Waveform:	
Combined Pitch = -0.0052732452750205937 rad	
Combined Roll after Sensor Fusion from Waveform:	
Combined Roll = -0.04054625239223241806 rad	

Table 3. Conversions and Results from Waveform for First Data Set

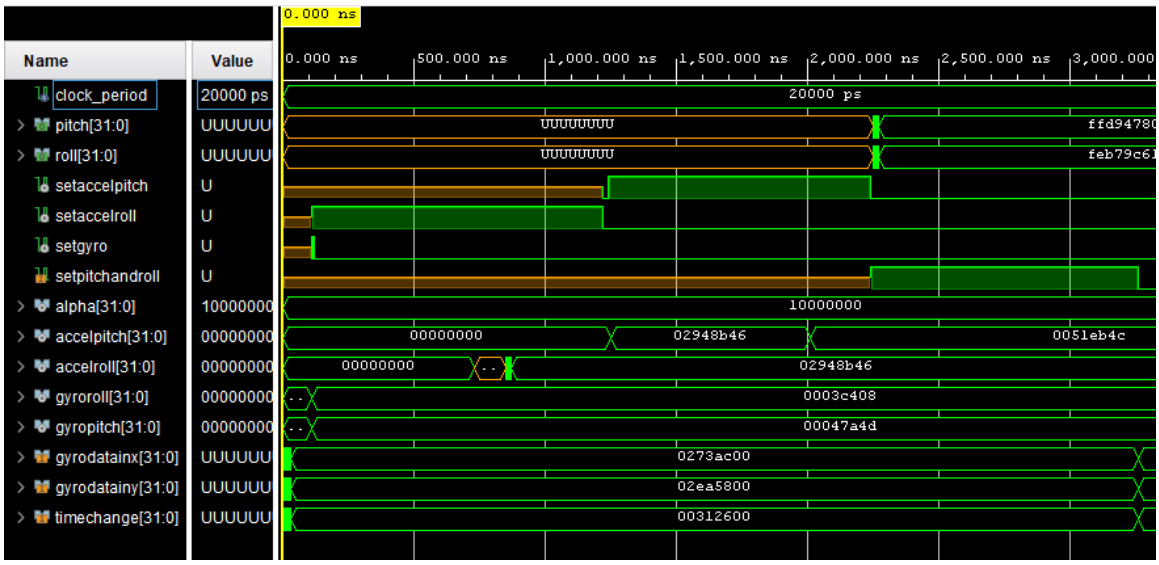


Figure 14. Sensor Fusion Waveform of First set of Data

As seen from the results in the table and the waveform above. The sensor fusion pitch and roll that is produced from VHDL is very close to that of the pitch and roll from the Arduino code. The discrepancies in answers come from the approximations that both processes have. The true value of each binary number is pasted below its binary representation in the box and it may be seen that while these numbers are close they are not exact. This along with the previously explored idea that Arduino makes approximations on their calculations produces slightly different results.

The second set of data as well as the corresponding waveform and waveform results may be seen in the table below and the waveform.

Second Set of Data:

Gyroscope Converted to Radians

Gx = 22.54 deg/s 0.3933972134 rad/s	Gy = 4.82 deg.s 0.0841248699 rad/s
Gyroscope Radians Converted to Binary	
Gx = 000.01100100101101011011000000000 True Value: 0.0766201019287109375	Gy = 000.00010101100010010011000000000 True Value: 0.091106414794921875
Accelerometer Binary Values:	
Ax = -.05 +0.5: 00.00001100110011001101000000000 After Two's Complement: 11.11110011001100110011000000000	Ay = 0.11 Binary: 00.00011100001010001111000000000 True Value: 0.10999965667724609375
Denominator for Acceleration Roll	**Denominator For Acceleration Pitch**
Denominator for Acceleration Roll: 1.05 Binary: 01.00001100110011001101000000000 True Value: 1.05000019073486328125	Denominator for Acceleration Pitch: 1.05 Binary: 01.00001100110011001101000000000 True Value: 1.05000019073486328125
DeltaTime Value:	
DeltaTime = 0.020 Binary: 000.00000101000111101100000000000	

True Value: 0.020000457763671875
Combined Pitch after Sensor Fusion from Arduino Code:
Combined Pitch = -1.43 deg -.0249582083 rad
Combined Roll after Sensor Fusion from Arduino Code:
Combined Roll = -2.84 deg -.0495673508 rad
Combined Pitch after Sensor Fusion from Waveform:
Combined Pitch = -0.02406493574380874634
Combined Roll after Sensor Fusion from Waveform:
Combined Roll = -0.05242024734616279602

Table 4. Conversions and Results from Waveform for Second Data Set

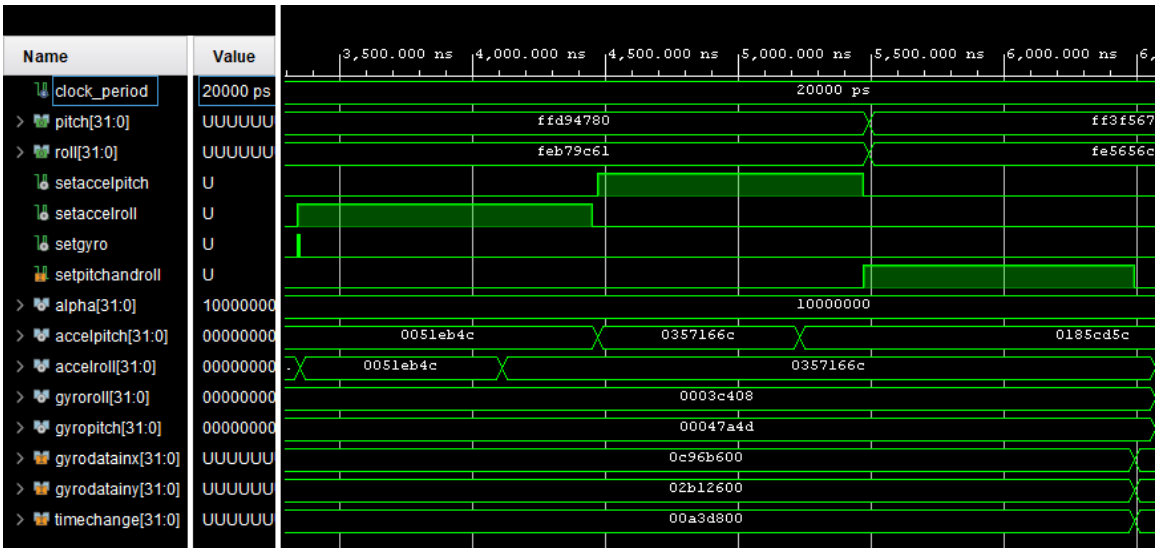


Figure 15. Sensor Fusion Waveform of Second set of Data

From this data like the first set the data is very similar but slightly different. These changes are expected from the translation from Arduino Code to VHDL. Below the final set of data can be seen in a table converting all the values to binary that were inputted to the testbench code that were inputted into the VHDL testbench code.

Third Set of Data:

Gyroscope Converted to Radians	
Gx = 7.52 deg/s 0.1312487597 rad/s	Gy = 1.23 deg.s 0.0214675498 rad/s
Gyroscope Radians Converted to Binary	
Gx = 000.00100001100110011000000000000 True Value: 0.13124847412109375	Gy = 000.00000101011111101110000000000 True Value: 0.0214672088623046875
Accelerometer Binary Values:	
Ax = 0.00 0.0: 00.00000000000000000000000000000	Ay = 0.10 Binary: 00.00011001100110011010000000000 True Value: 0.1000003814697265625
Denominator for Acceleration Roll	**Denominator For Acceleration Pitch**

Denominator for Acceleration Roll: .99 Binary: 00.11111101011100001010000000000 True Value: 0.9899997711181640625	Denominator for Acceleration Pitch: .99 Binary: 00.11111101011100001010000000000 True Value: 0.9899997711181640625
DeltaTime Value:	
DeltaTime = 0.024 Binary: 000.00000110001001001110000000000 True Value: 0.0240001678466796875	
Combined Pitch after Sensor Fusion from Arduino Code:	
Combined Pitch = 0.09 deg .0015707963 rad	
Combined Roll after Sensor Fusion from Arduino Code:	
Combined Roll = -2.56 deg -.0446804289 rad	
Combined Pitch after Sensor Fusion from Waveform:	
Combined Pitch = 0.00053090602159500122	
Combined Roll after Sensor Fusion from Waveform:	
Combined Roll = -0.05213936232030391693	

Table 5. Conversions and Results from Waveform for Third Data Set

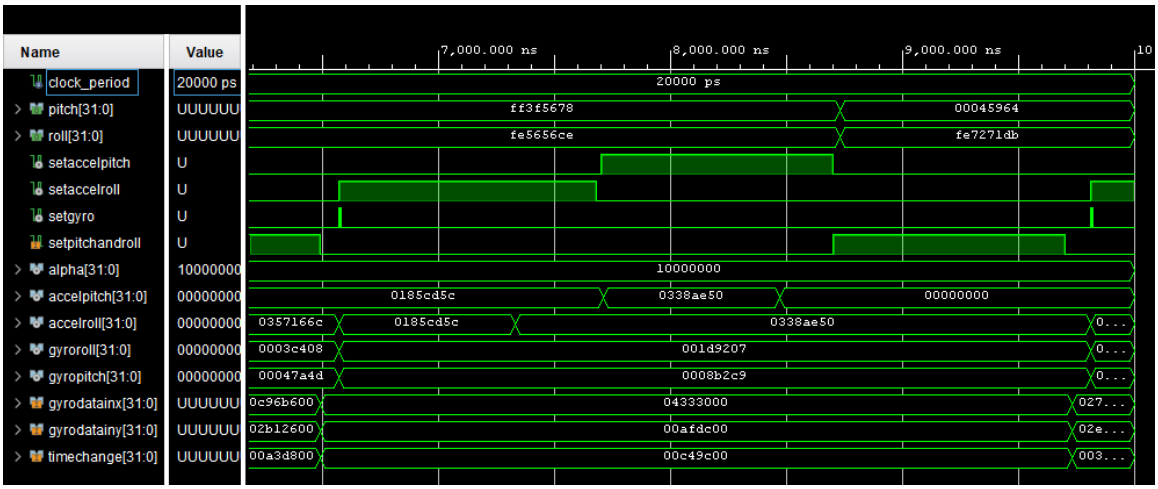


Figure 16. Sensor Fusion Waveform of Third set of Data

When viewing all the VHDL code together and comparing it to that of the reference example, it is seen the VHDL code produces an approximate estimate for the orientation of the object. The implementation of sensor fusion in VHDL is shown to be a success through the results of the waveform. While this proves functionality of sensor fusion in VHDL there is much that can be built on after the success of this implementation.

CHAPTER 5

CONCLUSION

The results from this research led to the estimation of orientation of an object using sensor fusion in VHDL. For this reason and the accuracy of the results this research is a success. The IMU was first implemented on an Arduino to achieve test data and then followed by VHDL implementation. This implementation started with the testing of working code and simulation to make the Basys 3 board the master in SPI and I2C communication. This was tested and then implemented on the board but due to the use of the Arduino as the slave the clocks did not match up and communication between the two devices was fruitless. The Arduino was the option of choice because it was already in possession for this research and provided a way to read the data coming out from the FPGA. The sample data from the Arduino was then taken to VHDL and simulated producing a very similar result. This proves the implementation of sensor fusion using VHDL.

The growth upon this research goes could expand upon the portion of the IMU calculations that were simulated and then getting the data straight from the IMU. In order to do this a way to see the data from the FPGA must be implemented. Another microcontroller may be used such as the process that was trying to be implemented in this research. However, a microcontroller with a faster clock must be utilized. To also improve upon the design of the simulated results the multiple arctangent CORDIC IPs may be implemented to take advantage of the parallel processing ability of the FPGA. Similarly, the square root IP that is also provided by CORDIC may be implemented instead of importing the data from the Arduino.

The purpose of this research sought out to prove the functionality and implementation of sensor fusion with VHDL so that it could be translated to that of an FPGA. In this data achieved by the Arduino it can be seen that the data computed in the range of approximately 20 milliseconds. While even with the inefficiencies of the VHDL code the time is measured in the range of 1 millisecond. The VHDL code is able to compute much faster than that of the Arduino. This research shows the benefits that may be achieved with an FPGA using VHDL and how with improvement an FPGA offers significant advantages over alternatives computation methods.

APPENDICES

APPENDIX A

ARDUINO CODE FOR IMU IMPLEMENTATION IN I2C

ARDUINO CODE FOR IMU IMPLEMENTATION IN I2C

```
#include <Wire.h> // Arduino Librarys added to make I2C work for LSM9DS1
#include <SPI.h>
#include <SparkFunLSM9DS1.h> //This is the libarary added from the SparkFun Tutorial
LSM9DS1 imu; //establishing the imu as a device using the builtin funciton.

#define PRINT_CALCULATED //This will be used later to print the converted values from the sensors
rather than the raw ADC values.
#define PRINT_SPEED 0 // 0 ms between prints
static unsigned long lastPrint = 0; // Keep track of print time
static unsigned long pastdeltat = 0; //Keep track of Last measurement time
static unsigned long readtime = 0; //Keep track of current measurement time
float pitchgyronew = 0; //establishing variables for pitch and roll
float rollgyronew = 0;

//Function definitions the funcitons may be found below
void printGyro();
void printAccel();
void printMag();
void printAttitude(float ax, float ay, float az, float mx, float my, float mz, float gx, float gy, float gz); //all
these floats are inputs

void setup()
{
  Serial.begin(115200); //Begins the Serial Monitor at 115200 BAUD.
  bool _autoCalc; // built in funciton in SparkFun Library to calibrate IMU
  Wire.begin(); // built in function from Wire.h
  if (imu.begin() == false) // The default address for the slave is used. If the arduino cannot connect to the
IMU then this if statement is triggered.
  {
    Serial.println("Failed to connect with PMOD Nav.");
    while (1);
  }
}
void loop()
{
  if ( imu.gyroAvailable() ) //if data is available then the sensor values are updated.
  {
    imu.readGyro(); //this function updates the values of gx, gy and gz
    readtime = millis(); //the time at the measurement is taken as integration is done for this portion and the
more accurate the reading the better the estimate

  }
  if ( imu.accelAvailable() )//if data is available then the sensor values are updated.
  {
    imu.readAccel(); //this function updates the values of ax, ay andaz
  }
```

```
  if ((lastPrint + PRINT_SPEED) < millis()) //This if statement occurs if the PRINT_SPEED wants to be
increased
  {
    printGyro();  // Print "G: gx, gy, gz"
    printAccel(); // Print "A: ax, ay, az"
    printAttitude(imu.calcAccel(imu.ax), imu.calcAccel(imu.ay), imu.calcAccel(imu.az), //Function sends
sensor data to do mathematical computation and print
                  imu.calcGyro(imu.gx), imu.calcGyro(imu.gy), imu.calcGyro(imu.gz));
    Serial.println();
    lastPrint = millis(); // Update lastPrint time
  }
}
void printGyro()
{
  Serial.print("G: ");
#ifdef PRINT_CALCULATED
  Serial.print(imu.calcGyro(imu.gx), 2); //imu.calcGyro prints the converted values rather than the raw
ADC values.
  Serial.print(", ");
  Serial.print(imu.calcGyro(imu.gy), 2); // angular acceleration in the y-axis is printed.
  Serial.print(", ");
  Serial.print(imu.calcGyro(imu.gz), 2); // angular acceleration in the z-axis is printed.
  Serial.println(" deg/s"); //this function prints in deg/s, this will have to be translated to rad/s
#endif
}
void printAccel()
{
  Serial.print("A: ");
#ifdef PRINT_CALCULATED
  Serial.print(imu.calcAccel(imu.ax), 2); //The same process as the gyroscope is used to print the linear
acceleration terms
  Serial.print(", ");
  Serial.print(imu.calcAccel(imu.ay), 2);
  Serial.print(", ");
  Serial.print(imu.calcAccel(imu.az), 2);
  Serial.println(" g"); //The values are printed in g's
#endif
}
void printAttitude(float ax, float ay, float az, float gx, float gy, float gz)
{
  Serial.print("Denominator for Pitch Calculation:  ");
  Serial.println(sqrt(ay * ay + az * az)); // This is used as sqrt was not used in the implementation for VHDL
so this value was manually inputted.
  Serial.print("Denominator for Roll Calculation:  ");
  Serial.println(sqrt(ax * ax + az * az)); // This is used as sqrt was not used in the implementation for VHDL
so this value was manually inputted.
  float pitchaccelnew = atan2(-ax, sqrt(ay * ay + az * az)); // the accel pitch is calculated
  float rollaccelnew = atan2(ay, sqrt(ax * ax + az * az)); // the accel roll is calcualted
  pitchaccelnew *= -180.0 / PI; // converted to radians with this conversion but also multiplied by -1
because it is in a different
  rollaccelnew  *= -180.0 / PI;       //orientation than the gyroscope readings so to match them the negative
was added.
  Serial.print("Accel Pitch, Roll: "); //Accel pitch and roll in degrees are printed.
  Serial.print(pitchaccelnew);
  Serial.print(", ");
```

```
  Serial.println(rollaccelnew);
  float pitchcombined; //pitch combined and roll combined are the sensor fusion variables and will be
defined.
  float rollcombined;
  float alpha = 0.5; //alpha = 0.5
  pitchgyronew = pitchgyronew + imu.calcGyro(imu.gy)*(readtime-pastdeltat)*.001; //equation to calculate
gyroscope values.
  rollgyronew = rollgyronew + imu.calcGyro(imu.gx)*(readtime-pastdeltat)*.001;
  Serial.print("Gyro Pitch, Roll: "); //print gyroscope values.
  Serial.print(pitchgyronew);
  Serial.print(", ");
  Serial.println(rollgyronew);
  Serial.print("DeltaTime for Gyro Equation: ");
  Serial.println(readtime-pastdeltat); //print the change in time from the gyroscope equation
  pitchcombined = pitchgyronew*alpha + pitchaccelnew * (1-alpha); //calculate the pitch sensor fusion
  rollcombined = rollgyronew*alpha + rollaccelnew * (1-alpha); //calculate the roll sensor fusion
  Serial.print("Combined Pitch, Combined Roll: ");  //print sensor fusion values
  Serial.print(pitchcombined);
  Serial.print(", ");
  Serial.println(rollcombined);
  pastdeltat = readtime;
}
```

APPENDIX B

SPI AND I2C MASTER CODE IN VHDL WITH TEST BENCH

SPI MASTER CODE:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY spi_master IS
  GENERIC(
    slaves  : INTEGER := 1;  --number of spi slaves
    d_width : INTEGER := 16); --data bus width
  PORT(
    clock   : IN     STD_LOGIC;                     --system clock
    reset_n : IN     STD_LOGIC;                     --asynchronous reset
    enable  : IN     STD_LOGIC;                     --initiate transaction
    cpol    : IN     STD_LOGIC;                     --spi clock polarity
    cpha    : IN     STD_LOGIC;                     --spi clock phase
    cont    : IN     STD_LOGIC;                     --continuous mode command
    clk_div : IN     STD_LOGIC_VECTOR(0 DOWNTO 0);                  --system clock cycles per
1/2 period of sclk
    addr    : IN     STD_LOGIC_VECTOR(1 DOWNTO 0);                  --address of slave
    tx_data : IN     STD_LOGIC_VECTOR(d_width-1 DOWNTO 0);  --data to transmit
    miso    : IN     STD_LOGIC;                     --master in, slave out
    sclk    : INOUT STD_LOGIC;                      --spi clock
    ss_n    : INOUT STD_LOGIC_VECTOR(slaves-1 DOWNTO 0);   --slave select
    mosi    : OUT    STD_LOGIC;                     --master out, slave in
    busy    : OUT    STD_LOGIC;                     --busy / data ready signal
    rx_data : OUT    STD_LOGIC_VECTOR(d_width-1 DOWNTO 0)); --data received
END spi_master;

ARCHITECTURE logic OF spi_master IS
  TYPE machine IS(ready, execute);                  --state machine data type
  SIGNAL state       : machine;                     --current state
  SIGNAL slave       : STD_LOGIC_VECTOR(1 DOWNTO 0);                --slave selected for
current transaction
  SIGNAL clk_ratio   : STD_LOGIC_VECTOR(3 DOWNTO 0);                --current clk_div
  SIGNAL count       : STD_LOGIC_VECTOR(3 DOWNTO 0 );               --counter to trigger sclk
from system clock
  SIGNAL clk_toggles : STD_LOGIC_VECTOR(3 DOWNTO 0);     --count spi clock toggles
  SIGNAL assert_data : STD_LOGIC;                   --'1' is tx sclk toggle, '0' is rx sclk toggle
  SIGNAL continue    : STD_LOGIC;                   --flag to continue transaction
  SIGNAL rx_buffer   : STD_LOGIC_VECTOR(d_width-1 DOWNTO 0); --receive data buffer
  SIGNAL tx_buffer   : STD_LOGIC_VECTOR(d_width-1 DOWNTO 0); --transmit data buffer
```

```
  SIGNAL last_bit_rx : STD_LOGIC_VECTOR(3 DOWNTO 0) ;         --last rx data bit location
  signal clkvariable1 : STD_LOGIC_VECTOR(3 DOWNTO 0 ) :=  "0001";
  signal clkvariable2 : STD_LOGIC_VECTOR(3 DOWNTO 0 ) :=  "0001";
  signal check1, check2, check3, check4, check5, check6, check7, check8, check9, check10, check11 :
STD_LOGIC;
  signal check12, check13, check14, check15, check16, check17, check18, check19 : STD_LOGIC;
BEGIN
  PROCESS(clock, reset_n)
  BEGIN
    IF(reset_n = '0') THEN        --reset system
      busy <= '1';                --set busy signal
      ss_n <= (OTHERS => '1');    --deassert all slave select lines
      mosi <= 'Z';                --set master out to high impedance
      rx_data <= (OTHERS => '0'); --clear receive data port
      state <= ready;             --go to ready state when reset is exited
    ELSIF(clock'EVENT AND clock = '0') THEN
      CASE state IS               --state machine
        WHEN ready =>
          check12 <= '0';
           check13 <= '0';
          check14 <= '0';
          check15 <= '0';
          check16 <= '0';
          check17 <= '0';
          check18 <= '0';
          check19 <= '0';
        busy <= '0';              --clock out not busy signal
        ss_n <= (OTHERS => '1'); --set all slave select outputs high
        mosi <= 'Z';              --set mosi output high impedance
        continue <= '0';          --clear continue flag
        --user input to initiate transaction
        IF(enable = '1') THEN
          check13 <= '1';
          busy <= '1';            --set busy signal
          IF(addr < slaves) THEN    --check for valid slave address
            check14 <= '1';
            slave <= addr;          --clock in current slave selection if valid
          ELSE
            check12 <= '1';
            slave <= "00";            --set to first slave if not valid
          END IF;
          IF(clk_div = "00") THEN     --check for valid spi speed
            check15 <= '1';
            clk_ratio <= "0001";        --set to maximum speed if zero
            count <= "0001";            --initiate system-to-spi clock counter
          ELSE
            check16 <= '1';
            clk_ratio <= clk_div;  --set to input selection if valid
            clkvariable1 <= clk_div;
            count <= clkvariable1;      --initiate system-to-spi clock counter
          END IF;
          check17 <= '1';
          sclk <= cpol;             --set spi clock polarity
          assert_data <= NOT cpha; --set spi clock phase
          tx_buffer <= tx_data;    --clock in data for transmit into buffer
          clk_toggles <= "0000";        --initiate clock toggle counter
```

```
          last_bit_rx <= "1000" + cpha - "1"; --set last rx data bit d_width*2 + conv_integer(cpha) - 1
          state <= execute;        --proceed to execute state
        ELSE
          check18 <= '1';
          state <= ready;          --remain in ready state
        END IF;
      WHEN execute =>
        check1 <= '0';
        check2 <= '0';
        check3 <= '0';
        check4 <= '0';
        check5 <= '0';
        check6 <= '0';
        check7 <= '0';
        check8 <= '0';
        check9 <= '0';
        check10 <= '0';
        check11 <= '0';
        busy <= '1';        --set busy signal
        ss_n(conv_integer(slave)) <= '0'; --set proper slave select output
        --system clock to sclk ratio is met
        clkvariable2 <= clk_ratio;
        IF(count = clkvariable2) THEN
        check1 <= '1';
          count <= "0001";                    --reset system-to-spi clock counter
          assert_data <= NOT assert_data; --switch transmit/receive indicator
          IF(clk_toggles = "1001") THEN --was 101
            check2 <= '1';
            clk_toggles <= "0000";              --reset spi clock toggles counter
          ELSE
            check3 <= '1';
            clk_toggles <= clk_toggles + 1; --increment spi clock toggles counter
          END IF;
          --spi clock toggle needed
          IF(clk_toggles <= "1000" AND ss_n(conv_integer(slave)) = '0') THEN
            check4 <= '1';
            sclk <= NOT sclk; --toggle spi clock
          END IF;
          --receive spi clock toggle
          IF(assert_data = '0' AND clk_toggles < last_bit_rx + '1' AND ss_n(conv_integer(slave)) = '0')
THEN
            check5 <= '1';
            rx_buffer <= rx_buffer(d_width-2 DOWNTO 0) & miso; --shift in received bit
          END IF;
          --transmit spi clock toggle
          IF(assert_data = '1' AND clk_toggles < last_bit_rx) THEN
            check6 <= '1';
            mosi <= tx_buffer(d_width-1);                   --clock out data bit
            tx_buffer <= tx_buffer(d_width-2 DOWNTO 0) & '0'; --shift data transmit buffer
          END IF;
          --last data receive, but continue
          IF(clk_toggles = last_bit_rx AND cont = '1') THEN
            check7 <= '1';
            tx_buffer <= tx_data;                   --reload transmit buffer
            clk_toggles <= last_bit_rx - d_width*2 + 1; --reset spi clock toggle counter
            continue <= '1';                        --set continue flag
```

```
          END IF;
          --normal end of transaction, but continue
          IF(continue = '1') THEN
            check8 <= '1';
            continue <= '0';      --clear continue flag
            busy <= '0';          --clock out signal that first receive data is ready
            rx_data <= rx_buffer; --clock out received data to output port
          END IF;
          --end of transaction
          IF((clk_toggles = d_width*2 + 1) AND cont = '0') THEN
            check9 <= '1';
            busy <= '0';            --clock out not busy signal
            ss_n <= (OTHERS => '1'); --set all slave selects high
            mosi <= 'Z';            --set mosi output high impedance
            rx_data <= rx_buffer;    --clock out received data to output port
            state <= ready;         --return to ready state
          ELSE                      --not end of transaction
            check10 <= '1';
            state <= execute;       --remain in execute state
          END IF;
        ELSE       --system clock to sclk ratio not met
          check11 <= '1';
          state <= execute;   --remain in execute state
        END IF;
      END CASE;
    END IF;
  END PROCESS;
END logic;
```

USER LOGIC CODE:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
entity User_Logic is --MODELED AFTER TESTBENCH
  GENERIC(
    slaves  : INTEGER := 1;  --number of spi slaves
    d_width : INTEGER := 16); --data bus width
  PORT(
    clock   : INOUT     STD_LOGIC;                      --system clock
    reset_n : OUT     STD_LOGIC :=  '0';                      --asynchronous reset
    enable  : OUT     STD_LOGIC;                     --initiate transaction
    cpol    : OUT     STD_LOGIC;                     --spi clock polarity
    cpha    : OUT     STD_LOGIC;                     --spi clock phase
    cont    : OUT     STD_LOGIC;                     --continuous mode command
    clk_div : OUT     STD_LOGIC_VECTOR(3 DOWNTO 0);                          --system clock cycles per
1/2 period of sclk
    addr    : OUT     STD_LOGIC_VECTOR(1 DOWNTO 0);                         --address of slave
    tx_data : OUT     STD_LOGIC_VECTOR(d_width-1 DOWNTO 0);  --data to transmit
    busy    : IN    STD_LOGIC;                      --busy / data ready signal
    rx_data : IN    STD_LOGIC_VECTOR(d_width-1 DOWNTO 0)); --data received
```

```
end User_Logic;
architecture Behavioral of User_Logic is
signal clkrise  : INTEGER := 0;
signal clkfall  : INTEGER := 0;
signal count : INTEGER := 0;
signal busy_state: std_logic :='0';
begin
spi_transactions : process(clock, clkrise, clkfall, rx_data)
begin
if clock'EVENT AND clock = '1' then
        clkrise <= clkrise + 1;
end if;
if clock'EVENT AND clock = '0' then
        clkfall <= clkfall + 1;
end if;
      if clkrise = 32 then
      reset_n <= '0';
      end if;
      if clkrise = 128 then
            reset_n <= '1';
            enable <= '0';
            cpol <= '0';
            cpha <= '0';
            cont <= '0';
            clk_div <= "0000";
            addr <= "00";
            tx_data <= "0000000000000000";
       end if;
      if clkfall =448 then
            cont <= '1';
            enable <= '1';
            cpol <= '1';
            cpha <= '0';
            clk_div <= "0001";
            addr <= "00";
--            tx_data <= rx_data;
      end if;
      if clkfall = 16000 then
            clkrise <= 2;
            clkfall <= 2;
      end if;
      if (clock'event and clock='1') then
         if clkfall>640 then
            if (busy='1' and busy_state='0') then    -- need to declare an internal signal
               busy_state<='1';
            elsif (busy='0' and busy_state='1') then
               busy_state<='0';
               tx_data<=rx_data; --rx_data
            end if;
         end if;
      end if;
end process;
end Behavioral;
```

TEST BENCH:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
entity spi_master_test is
end spi_master_test;
architecture Behavioral of spi_master_test is
   component spi_master
    PORT(
      clock   : IN     STD_LOGIC;                       --system clock
      reset_n : IN     STD_LOGIC;                        --asynchronous reset
      enable  : IN     STD_LOGIC;                        --initiate transaction
      cpol    : IN     STD_LOGIC;                       --spi clock polarity
      cpha    : IN     STD_LOGIC;                       --spi clock phase
      cont    : IN     STD_LOGIC;                       --continuous mode command
      clk_div : IN     STD_LOGIC;                          --system clock cycles per 1/2 period of sclk
      addr    : IN     STD_LOGIC_VECTOR(1 DOWNTO 0);                          --address of slave
      tx_data : IN     STD_LOGIC_VECTOR(3 DOWNTO 0);  --data to transmit
      miso    : IN     STD_LOGIC;                       --master in, slave out
      sclk    : INOUT STD_LOGIC;                          --spi clock
      ss_n    : INOUT STD_LOGIC_VECTOR(0 DOWNTO 0);   --slave select
      mosi    : OUT    STD_LOGIC;                          --master out, slave in
      busy    : OUT    STD_LOGIC;                          --busy / data ready signal
      rx_data : OUT    STD_LOGIC_VECTOR(3 DOWNTO 0)); --data received
   END component;
signal slaves  : INTEGER := 1;
signal d_width : INTEGER := 4;
signal clock, reset_n, enable, cpol, cpha, cont, miso, sclk, mosi, busy : STD_LOGIC;
signal clk_div : STD_LOGIC;
signal addr : STD_LOGIC_VECTOR(1 DOWNTO 0);
signal tx_data, rx_data : STD_LOGIC_VECTOR(d_width-1 DOWNTO 0);
signal ss_n : STD_LOGIC_VECTOR(slaves-1 DOWNTO 0);
signal clk_chk : STD_LOGIC;
constant clock_period : time := 20 ns;
begin
uut: spi_master port map (
      clock => clock,
      reset_n => reset_n,
      enable => enable,
      cpol => cpol,
      cpha => cpha,
      cont => cont,
      clk_div => clk_div,
      addr => addr,
      tx_data => tx_data,
      miso => miso,
      sclk => sclk,
      ss_n => ss_n,
      mosi => mosi,
```

```
        busy => busy,
        rx_data => rx_data
);
clock_process : process
begin
    clock <= '0';
    wait for clock_period/2;
    clock <= '1';
    wait for clock_period/2;
end process;
spi_transactions : process
begin
wait until clock'EVENT AND clock = '0';
reset_n <= '0';
wait until clock'EVENT AND clock = '1';
reset_n <= '1';
wait for 20 ns;
enable <= '0';
cpol <= '0';
cpha <= '0';
cont <= '0';
clk_div <= '0';
addr <= "00";
tx_data <= "0000";
miso <= '0';
wait for 80 ns;
wait for 31 ns;
enable <= '1';
cpol <= '1';
cpha <= '1';
clk_div <= '1';
addr <= "10";
tx_data <= "1001";
wait for 20 ns;
enable <= '0';
cpol <= '0';
cpha <= '0';
addr <= "00";
tx_data <= "0000";
clk_div <= '0';
wait for 1000 ns;
end process;
end Behavioral;
```

SPI CONSTRAINTS CODE:

```
set_property BITSTREAM.STARTUP.STARTUPCLK JTAGCLK [current_design]
set_property PACKAGE_PIN K17 [get_ports {ss_n}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ss_n}]
set_property PACKAGE_PIN M18 [get_ports {mosi}]
    set_property IOSTANDARD LVCMOS33 [get_ports {mosi}]
set_property PACKAGE_PIN N17 [get_ports {miso}]
```

```
    set_property IOSTANDARD LVCMOS33 [get_ports {miso}]
set_property PACKAGE_PIN P18 [get_ports {sclk}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sclk}]
set_property PACKAGE_PIN W5 [get_ports {clock}]
    set_property IOSTANDARD LVCMOS33 [get_ports {clock}]
```

APPENDIX C

VHDL SENSOR FUSION CODE AND TESTBENCH

IMU SYSTEM CODE:

```
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
entity IMU_system is
  PORT(
          clock : IN STD_LOGIC;
          gyrodatainx : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          gyrodatainy : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          gyrodatainz : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldatainx : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldatainy : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldatainz : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldataindenominatorroll : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldataindenominatorpitch : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          setaccelpitch : IN STD_LOGIC;
          setaccelroll : IN STD_LOGIC;
          setgyro : IN STD_LOGIC;
          setpitchandroll : IN STD_LOGIC;
          timechange : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          pitch : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
          roll : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
          );
end IMU_system;
architecture Behavioral of IMU_system is
       COMPONENT cordic_0
         PORT (
          aclk : IN STD_LOGIC;
          s_axis_cartesian_tvalid : IN STD_LOGIC;
          s_axis_cartesian_tdata : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
          m_axis_dout_tvalid : OUT STD_LOGIC;
          m_axis_dout_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
         );
       END COMPONENT;
    COMPONENT IMU_Calculations is
     Port (
          clock : IN STD_LOGIC;
          gyrodatainx : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          gyrodatainy : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          gyrodatainz : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldatainx : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldatainy : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldatainz : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldataindenominatorroll : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldataindenominatorpitch : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          setaccelpitch : IN STD_LOGIC;
```

```
            setaccelroll : IN STD_LOGIC;
            setgyro : IN STD_LOGIC;
            setpitchandroll : IN STD_LOGIC;
            timechange : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            m_axis_dout_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            m_axis_dout_tvalid : IN STD_LOGIC;
            s_axis_cartesian_tdata : OUT STD_LOGIC_VECTOR(63 DOWNTO 0);
            s_axis_cartesian_tvalid : OUT STD_LOGIC;
            pitch : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
            roll : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
            );
    end COMPONENT;
signal m_axis_dout_tdata : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal s_axis_cartesian_tvalid : STD_LOGIC;
signal s_axis_cartesian_tdata : STD_LOGIC_VECTOR(63 downto 0);
signal m_axis_dout_tvalid : STD_LOGIC;
constant clock_period : time := 20 ns;
begin
Arctangent : cordic_0
  PORT MAP (
    aclk => clock,
    s_axis_cartesian_tvalid => s_axis_cartesian_tvalid,
    s_axis_cartesian_tdata => s_axis_cartesian_tdata,
    m_axis_dout_tvalid => m_axis_dout_tvalid,
    m_axis_dout_tdata => m_axis_dout_tdata
  );
IMU_Calc: IMU_Calculations port map (
    clock => clock,
    gyrodatainx => gyrodatainx,
    gyrodatainy => gyrodatainy,
    gyrodatainz => gyrodatainz,
    acceldatainx => acceldatainx,
    acceldatainy => acceldatainy,
    acceldatainz => acceldatainz,
    acceldataindenominatorroll => acceldataindenominatorroll,
    acceldataindenominatorpitch => acceldataindenominatorpitch,
    setaccelpitch => setaccelpitch,
    setaccelroll => setaccelroll,
    setgyro => setgyro,
    setpitchandroll => setpitchandroll,
    timechange => timechange,
    m_axis_dout_tdata => m_axis_dout_tdata,
    m_axis_dout_tvalid => m_axis_dout_tvalid,
    s_axis_cartesian_tdata => s_axis_cartesian_tdata,
    s_axis_cartesian_tvalid => s_axis_cartesian_tvalid,
    pitch => pitch,
    roll => roll
);
end Behavioral;
```

IMU CALCULATIONS CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity IMU_Calculations is
  Port (
          clock : IN STD_LOGIC;
          gyrodatainx : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          gyrodatainy : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          gyrodatainz : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldatainx : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldatainy : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldatainz : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldataindenominatorroll : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          acceldataindenominatorpitch : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          setaccelpitch : IN STD_LOGIC;
          setaccelroll : IN STD_LOGIC;
          setpitchandroll : IN STD_LOGIC;
          setgyro : IN STD_LOGIC;
          timechange : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          m_axis_dout_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
          m_axis_dout_tvalid : IN STD_LOGIC;
          s_axis_cartesian_tdata : OUT STD_LOGIC_VECTOR(63 DOWNTO 0);
          s_axis_cartesian_tvalid : OUT STD_LOGIC;
          pitch : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
          roll : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
end IMU_Calculations;
architecture Behavioral of IMU_Calculations is
signal yout : STD_LOGIC_VECTOR(31 downto 0) := "00100000000000000000000000000000";
signal xout : STD_LOGIC_VECTOR(31 downto 0) := "00101000000000000000000000000000";
signal alpha : STD_LOGIC_VECTOR(31 downto 0) := "00010000000000000000000000000000";--
"00100000000000000000000000000000";
signal gyroroll : STD_LOGIC_VECTOR(31 downto 0) := "00000000000000000000000000000000";
signal gyropitch: STD_LOGIC_VECTOR(31 downto 0) := "00000000000000000000000000000000";
signal accelroll : STD_LOGIC_VECTOR(31 downto 0) := "00000000000000000000000000000000";
signal accelpitch : STD_LOGIC_VECTOR(31 downto 0) := "00000000000000000000000000000000";
signal temperarydata1 : STD_LOGIC_VECTOR(63 downto 0) :=
"0010000000000000000000000000000000000000000000000000000000000000";
signal temperarydata2 : STD_LOGIC_VECTOR(63 downto 0) :=
"0010000000000000000000000000000000000000000000000000000000000000";
signal temperarydata3 : STD_LOGIC_VECTOR(63 downto 0) :=
"0010000000000000000000000000000000000000000000000000000000000000";
signal temperarydata4 : STD_LOGIC_VECTOR(63 downto 0) :=
"0010000000000000000000000000000000000000000000000000000000000000";
signal temperarydata5 : STD_LOGIC_VECTOR(63 downto 0) :=
"0010000000000000000000000000000000000000000000000000000000000000";
signal temperarydata6 : STD_LOGIC_VECTOR(63 downto 0) :=
"0010000000000000000000000000000000000000000000000000000000000000";
```

```
signal temperarydata7 : STD_LOGIC_VECTOR(31 downto 0) :=
"00000000000000000000000000000000";
signal temperarydata8 : STD_LOGIC_VECTOR(31 downto 0) :=
"00000000000000000000000000000000";
signal internalpitch : STD_LOGIC := '1';
signal internalroll : STD_LOGIC := '1';
begin
process(setaccelpitch, setaccelroll, clock)
begin
if clock'EVENT AND clock = '1' then
if setaccelpitch = '1' then
   s_axis_cartesian_tvalid <= '1';
   yout <= 0-acceldatainx;
   xout <= acceldataindenominatorpitch;
   s_axis_cartesian_tdata <= yout & xout;
   s_axis_cartesian_tvalid <= '1';
   accelpitch <= m_axis_dout_tdata;
end if;
if setaccelroll = '1' then
   s_axis_cartesian_tvalid <= '1';
   yout <= acceldatainy;
   xout <= acceldataindenominatorroll;
   s_axis_cartesian_tdata <= yout & xout;
   s_axis_cartesian_tvalid <= '1';
   accelroll <= m_axis_dout_tdata;
end if;
temperarydata1 <= gyrodatainy * timechange;
temperarydata2 <= gyrodatainx * timechange;
if setgyro = '1' then
   gyropitch <= gyropitch + temperarydata1(60 downto 29);--gyropitch + temperarydata1(63 downto 32);
   gyroroll <= gyroroll + temperarydata2(60 downto 29);--gyroroll + temperarydata2(63 downto 32);
end if;
if setpitchandroll = '1' then
      temperarydata3 <= accelpitch*("00100000000000000000000000000000"-alpha);
      temperarydata4 <= gyropitch * alpha;
      pitch <= temperarydata4(60 downto 29) - temperarydata3(60 downto 29);
      temperarydata5 <= accelroll*("00100000000000000000000000000000"-alpha);
      temperarydata6 <= gyroroll * alpha;
      roll <= temperarydata6(60 downto 29) - temperarydata5(60 downto 29);
end if;
end if;
end process;
end Behavioral;
```

IMU CALCULATIONS TEST BENCH:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

```
USE ieee.std_logic_unsigned.all;
entity arctan_test is
end arctan_test;
architecture Behavioral of arctan_test is
COMPONENT IMU_system is
  PORT(
        clock : IN STD_LOGIC;
        gyrodatainx : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        gyrodatainy : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        gyrodatainz : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        acceldatainx : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        acceldatainy : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        acceldatainz : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        acceldataindenominatorroll : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        acceldataindenominatorpitch : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        setaccelpitch : IN STD_LOGIC;
        setaccelroll : IN STD_LOGIC;
        setgyro : IN STD_LOGIC;
        setpitchandroll : IN STD_LOGIC;
        timechange : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        pitch : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        roll : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
end COMPONENT;
signal clock : STD_LOGIC;
signal gyrodatainx : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal gyrodatainy : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal gyrodatainz : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal acceldatainx : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal acceldatainy : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal acceldatainz : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal setaccelpitch :STD_LOGIC;
signal setaccelroll : STD_LOGIC;
signal setgyro: STD_LOGIC;
signal setpitchandroll : STD_LOGIC;
signal pitch : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal roll : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal timechange : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal acceldataindenominatorroll : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal acceldataindenominatorpitch : STD_LOGIC_VECTOR(31 DOWNTO 0);
constant clock_period : time := 20 ns;
begin
uut : IMU_System
  PORT MAP (
    clock => clock,
    gyrodatainx => gyrodatainx,
    gyrodatainy => gyrodatainy,
    gyrodatainz => gyrodatainz,
    acceldatainx => acceldatainx,
    acceldatainy => acceldatainy,
    acceldatainz => acceldatainz,
    acceldataindenominatorroll => acceldataindenominatorroll,
    acceldataindenominatorpitch => acceldataindenominatorpitch,
    setaccelpitch => setaccelpitch,
    setaccelroll => setaccelroll,
    setgyro => setgyro,
```

```
    setpitchandroll => setpitchandroll,
    timechange => timechange,
    pitch => pitch,
    roll => roll
  );
clock_process : process
begin
    clock <= '0';
    wait for clock_period/2;
    clock <= '1';
    wait for clock_period/2;
end process;
spi_transactions : process
begin
wait until clock'EVENT AND clock = '0';
wait until clock'EVENT AND clock = '1';
wait for 20 ns;
acceldatainx <= "11111111010111000010100000000000";
acceldatainy <= "00000101000111101011100000000000";
acceldatainz <= "00111100110011001100110000000000";
acceldataindenominatorpitch <= "01000000000000000000000000000000";
acceldataindenominatorroll <= "00111111010111000010100000000000";
gyrodatainy <=  "00000010111010100101100000000000";
gyrodatainx <=  "00000010011100111010110000000000";
gyrodatainz <= "00000001010001111011000000000000";
timechange <=   "00000000001100010010011000000000";
wait for 80 ns;
setaccelroll <= '1';
setgyro <= '1';
wait for 10 ns;
setgyro <= '0';
wait for 100 ns;
wait for 1000 ns;
setaccelpitch <= '0';
setaccelroll <= '0';
setgyro <= '0';
wait for 20 ns;
setaccelpitch <= '1';
wait for 1000 ns;
setaccelpitch <= '0';
setaccelroll <= '0';
setgyro <= '0';
setpitchandroll <= '1';
wait for 1000 ns;
wait for 20 ns;
setpitchandroll <= '0';
acceldatainx <= "11111100110011001100110000000000";
acceldatainy <= "00000111000010100011110000000000";
acceldatainz <= "00111100110011001100110000000000";
acceldataindenominatorpitch <= "01000011001100110011010000000000";
acceldataindenominatorroll <= "01000011001100110011010000000000";
gyrodatainy <=  "00000010101100010010011000000000";
gyrodatainx <=  "00001100100101101011011000000000";
gyrodatainz <= "00000001010001111011000000000000";
timechange <=   "00000000101000111101100000000000";
wait for 80 ns;
```

```
setaccelroll <= '1';
setgyro <= '1';
wait for 10 ns;
setgyro <= '0';
wait for 100 ns;
wait for 1000 ns;
setaccelpitch <= '0';
setaccelroll <= '0';
setgyro <= '0';
wait for 20 ns;
setaccelpitch <= '1';
wait for 1000 ns;
setaccelpitch <= '0';
setaccelroll <= '0';
setgyro <= '0';
setpitchandroll <= '1';
wait for 1000 ns;
wait for 20 ns;
setpitchandroll <= '0';
acceldatainx <= "00000000000000000000000000000000";
acceldatainy <= "00000110011001100110100000000000";
acceldatainz <= "00111100110011001100110000000000";
acceldataindenominatorpitch <= "00111111010111000010100000000000";
acceldataindenominatorroll <= "00111111010111000010100000000000";
gyrodatainy <=  "00000000101011111101110000000000";
gyrodatainx <=  "00000100001100110011000000000000";
gyrodatainz <= "00000001010001111011000000000000";
timechange <=   "00000000110001001001110000000000";
wait for 80 ns;
setaccelroll <= '1';
setgyro <= '1';
wait for 10 ns;
setgyro <= '0';
wait for 100 ns;
wait for 1000 ns;
setaccelpitch <= '0';
setaccelroll <= '0';
setgyro <= '0';
wait for 20 ns;
setaccelpitch <= '1';
wait for 1000 ns;
setaccelpitch <= '0';
setaccelroll <= '0';
setgyro <= '0';
setpitchandroll <= '1';
wait for 1000 ns;
setpitchandroll <= '0';
end process;
end Behavioral;
```

REFERENCES

Benkrid, Khaled. Tian, Xiang. “High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU” (2010). https://dl-acm-org.csulb.idm.oclc.org/doi/abs/10.1145/1862648.1862656.

K, Sam. “Basys 3.” Basys 3 - Digilent Reference, reference.digilentinc.com/reference/programmable-logic/basys-3/start.

Koksal, N., M. Jalalmaab, and B. Fidan. “Adaptive Linear Quadratic Attitude Tracking Control of a Quadrotor UAV Based on IMU Sensor Data Fusion.” Sensors 19.1 (2018): 46. Crossref. Web.

LSM9DS1 Breakout Hookup Guide, learn.sparkfun.com/tutorials/lsm9ds1-breakout-hookup-guide/all.

“Pmod NAV Reference Manual.” Pmod NAV Reference Manual - Digilent Reference, reference.digilentinc.com/reference/pmod/pmodnav/reference-manual.

Scott_1767. “I2C Master (VHDL).” Engineering and Component Solution Forum - TechForum │ Digi-Key, 17 Mar. 2021, https://forum.digikey.com/t/i2c-master-vhdl/12797.

Scott_1767. “SPI Master (VHDL).” Engineering and Component Solution Forum - TechForum │ Digi-Key, 16 Mar. 2021, https://forum.digikey.com/t/spi-master-vhdl/12717.

“Using FPGAs.” *Using FPGAs - SparkFun Electronics*, www.sparkfun.com/fpga.

Wertz, Duane. “Pmod NAV: 9-Axis IMU Plus Barometer.” Digilent, store.digilentinc.com/pmod-nav-9-axis-imu-plus-barometer/.