FPGA Implementation of a Digital Controller for a Small VTOL UAV

Final Project Report   13 December 2014

Gregory Kravit   gkravit@mit.edu

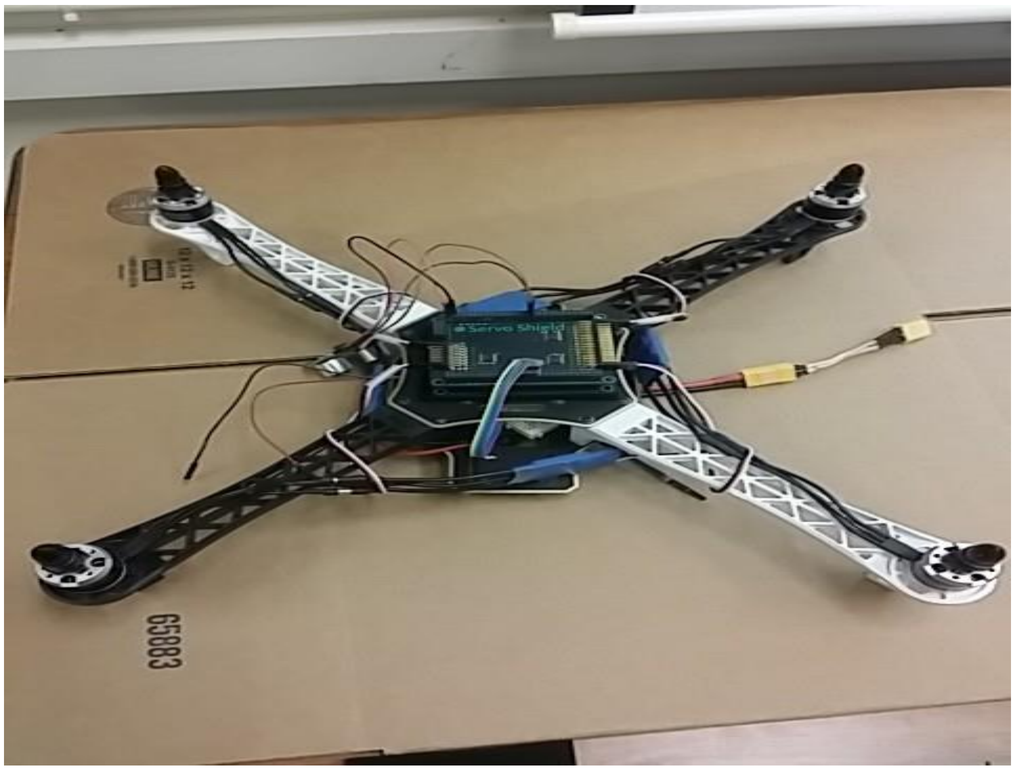

Figure 1: Picture of the assembled quadrotor.

Abstract

This project attempted to design and implement a digital flight controller on a FPGA prototype board for stabilizing a small quadcopter unmanned aerial vehicle. The purpose of the project was to assess the feasibility of using an FPGA in the stabilized control of an underactuated aerial robotic system.   Unfortunately, the project was not successfully implemented due to difficulties interfacing with the proprietary sensor components and underestimation of the risk exposure in the project scope.   The project was able to determine that the FPGA is an inadequate platform for implementing quadcopter control compared to widely used microcontrollers that are more than adequate and have a large support base. The FPGA platform was found to pose good properties for integrating multiple sensors and performing parallel tasks such as motor control. Future projects should consider the viability of pairing a microcontroller handling basic control functionality with the FPGA handling sensor interface tasks and performing large parallel computational tasks such as optical flow sensing and enabling self-tuning feedback.

TABLE OF CONTENTS

TABLE OF FIGURES

0 Introduction

The goal of this project was to assess the viability of the FPGA as a digital controller for a small vertical takeoff and land (VTOL) unmanned aerial vehicle (UAV). To assess the viability, I attempted to implement a digital PID controller that could interface with an inertial measurement unit (IMU) and an ultrasonic range sensor in order to calculate control inputs for the four brushless DC motors of a small quadcopter. The flight controller would allow for the stabilized flight of the quadcopter and maintain a constant desired altitude.

A Mojo v3 FPGA prototype board by Embedded Micro was chosen for the project. The Mojo was to send pulse width modulated (PWM) signals to four iPower 2212Q 1000 Kv rated motors through an accessory servo shield, as well as, interface with the sensors through its GPIO pins. Power (12 volt-18 ampere) was delivered via tethered speaker wire to a power distribution board on the quadcopter and supplied by a modified computer desktop power supply (PSU).

Despite my best efforts, I was not successful in completing a flying prototype in time. I encountered numerous issues related to interfacing with proprietary sensors, developing a usable power source, and working with corrupted ISE project files that caused large schedule setbacks I could not overcome. Highlights I was able to accomplish were: developed working drivers for interfacing with the sensors including an SPI mode 0 master; created a useful MATLAB simulation for developing the control scheme; and assembled a viable quadcopter with frame, motors, electronic speed controllers (ESC) and power supply that only needed a working flight controller to fly. The Verilog modules that worked in simulation and on the 6.111 labkit did not work properly on the Mojo, indicating issues that I did not account for with developing across FPGA platforms (the labkit contained an older Xilinx Virtex II while the Mojo had a newer, but less powerful Spartan 6).

1 Hardware Integration

A major aspect of this project were issues concerning interfacing the digital world of the FPGA with the physical world of motors, quadcopter frame and sensors. Considering I had never before built a quadcopter or any RC aircraft or had much experience with tasks such as soldering, there was a large amount of risk in the project that I had not anticipated. Major concerns arose with key initial design decisions related to developing on two different FPGA platforms, proprietary sensor formats, and hacking together the power system.

1.1 FPGA

There were two FPGA prototype systems used in the development of this project: the Xilinx Virtex II located on the 6.111 labkit and the Xilinx Spartan 6 on the Mojo V3 prototype board. The labkit was used for the rapid development and test of Verilog modules because it had buttons, switches and a led hex display useful for verifying modules are working properly that the Mojo lacked. The Mojo V3 FPGA prototype board from Embedded Micro was intended to be the final flight control platform and located on the quadrotor during flight. The Mojo had been chosen for this project due to its form factor, its low price ($70) and that it can easily be connected to an accessory servo shield ($40) to operate the motors.

The Mojo could take anywhere between 3V-12V because of its onboard voltage regulator and could be powered from either its DC bullet connector or micro USB port. It also had the ability to accept a power source from its servo accessory which was ideal because the ESCs used had a 5V battery elimination circuit (BEC) that could supply the necessary power to the Mojo. This avoided having to design a voltage regulator to convert down the 12V power from the power supply used.

I ran into trouble with developing on two platforms because modules and state machines that worked properly on the labkit Virtex II did not transfer to the Mojo without major issues. In hindsight, I should have developed a working serial out capability connected to my laptop first on the Mojo so I would be able to only develop on one platform. This would have saved me a good amount of frustration and time that I lost having to transfer between the two platforms. It would also would have proven useful later for sending desired control inputs and reading data from the Mojo during flight.

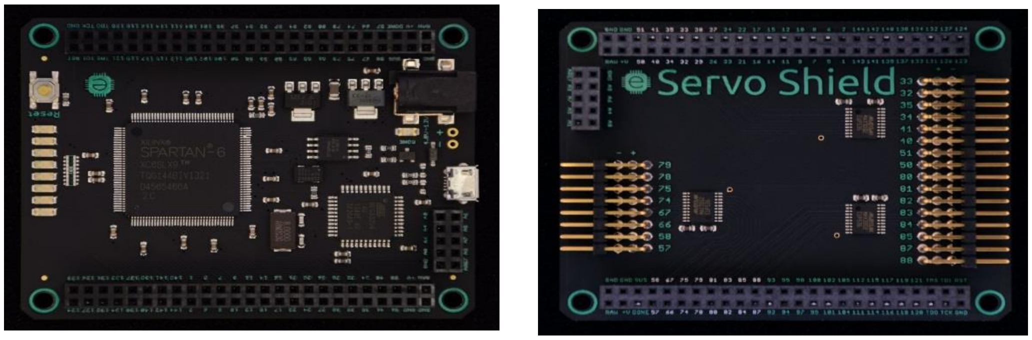

Figure 2: Mojo V3 and servo shield

1.2 iFlight 450 mm Quadcopter Kit

For this project, I purchased an almost-ready-to-fly (ARF) quadcopter kit from AltitudeHobbies.com. The $150 kit came with a sturdy and light 450 mm frame with center platform, four powerful iPower M2212Q 1000 Kv rated brushless DC motors, four max 30 ampere electronic speed controllers (ESCs) with 5V-3A battery elimination circuits (BEC), two sets of 10 inch x 4.5 pitch plastic propellers, and the necessary assembly materials.

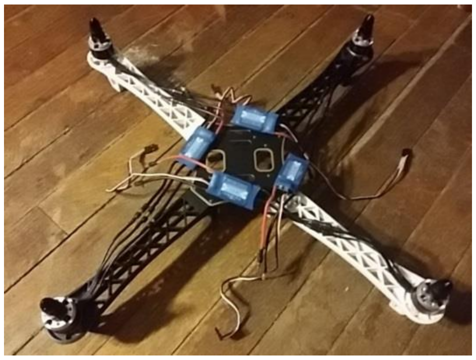
Figure 3: Picture of the assembled quadcopter kit (without propellers)

From the details on the webpage, it seemed to meet my requirements that it be low cost, easily assembled and be able to easily carry the FPGA and sensors onboard. The ESCs in the kit were also ideal because they became preloaded with the SimonK ESC firmware. The SimonK firmware is open source and heavily documented which will allow me to understand how to interface with the motors through a custom motor controller implemented in the FPGA.

When it arrived, I found out that it was just a cheaper Chinese Dji Flamewheel 450 clone with sparse documentation and performance specs. The quality of the material was adequate, and if the project flew, the quadcopter would have been a stable platform. However, the lack of easily available performance information caused issues because it made it harder to calculate system parameters for accurately modeling the quadcopter dynamics. I spent a significant amount of time researching and deriving estimates that would not have been necessary if I had used a more well-known kit.

In addition, I ran into issues with mounting sensors due to the limited amount of space in the center platforms. There was limited clearance available underneath for storing parts. A quadcopter with extended clearance possibly with landing legs would have been ideal. I was able to overcome this by placing the Mojo with servo shield on the top platform, the IMU sensor on a breadboard on the lower platform along the right orientation, and mounting the ultrasonic sensor on the arms, but as far away from the propellers as possible. There were definitely tradeoffs with the limited space.

1.3 Power System

The power system was a key implementation task on the project that required a good deal of assembly tasks to create a power system from scratch. One of the key initial design choices was to tether the quadrotor to eliminate the use of Lithium-Polymer (LiPo) batteries and increase the safety factor in the lab. However, existing power supplies in the lab were not adequate to meet the power requirements of the motors. The 2212Q motors could draw up to 17 amps each under max throttle. For hover applications at least an 18 amp power supply was needed. The motors also needed between a 9-14 volts for proper operation of the motors and the ESCs needed at least the 9 volts in order to produce the 5 volt power for the Mojo.

After a bit of research, I ended up determining that a computer power supply unit (PSU) could provide the necessary power requirements with a few modifications. With a PSU in hand, I added a switch between the green power line and ground, a 5 ohm, 20 watt power resistor to a 5 volt line, and ground to deliver a basic load when turning on the PSU, an red led indicator light on another 5 volt line and a male XC60 power connector to a 12 volt-18 amp line out for the quadcopter power.

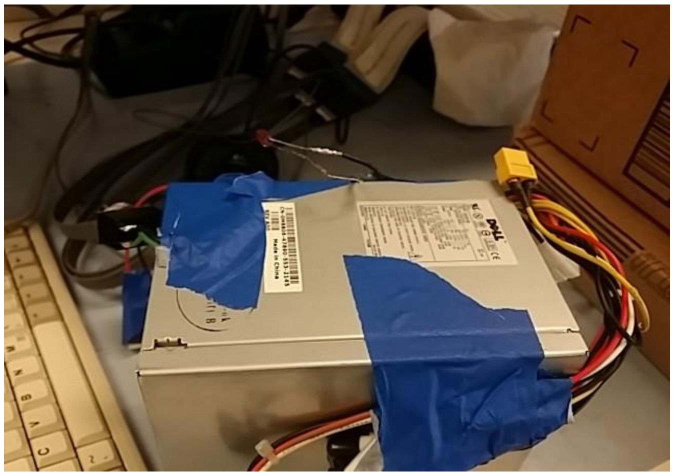
Figure 4: The modified PSU. (From left to right) The switch, led indicator, and female XC60 power connector can be seen in the photo.

To complete the power system, the power tether was made out of 16 gauge copper wire of sufficient length. The power tether connected the PSU to a power distribution board on the quadcopter with soldered male and female XC60 power connectors, respectively. The power distribution board on the quadcopter was then connected to each of the ESCs via bullet connectors.

Because I had never soldered before, I experienced a major learning curve in soldering connectors to the power system wires. I spent much of the initial two weeks of the project in the soldering lab preparing the power distribution system and soldering header pins on the sensor boards. It was a good learning experience, but developing the power system certainly contributed to schedule delays. Unfortunately, I needed a working power supply in order to develop important components of the system and ultimately to enable the integrated the quadcopter system to fly.

1.4 Ultrasonic Sensor

In order to obtain sensor data for altitude, a Devantech SRF05 ($23) sold by Acroname was chosen for measuring the height of the UAV. The sensor works by sending out a series of ultrasonic pulses and then measuring the time delay average in the return signals. The operation of the digital sensor is relatively simple. The system returns a positive transistor-transistor logic (TTL) level signal that is proportional to the range sensed. Using the separate trigger and echo mode of operation, the sensor is sent a minimum 10µs trigger high signal and then listens for an echo pulse to return. The length (between 100µS to 25 mS) of the return echo pulse is then proportional to the height.

1.5 IMU Sensor Module

For stabilizing the quadcopter, onboard sensors that can read attitude orientation and position information are needed for the flight controller. These inertial measurement units (IMU) often consist of gyroscopes (angular rates), accelerometers (forces), magnetometers (magnetic heading) and GPS beacons (geographic position). For the project, I needed an IMU that could determine body attitude (roll, pitch, yaw) and angular rates for the control scheme. I had two options: 1) buy an off the shelf IMU that computes flight attitudes; or 2) integrate gyroscopes and accelerometers myself and implement an attitude calculation through the use of a Kalman Filter or Direction Cosine matrix. To avoid the complexity of implementing my own IMU, I chose to obtain an off-the-shelf option. Luckily, the price of consumer IMU breakout boards has come down considerably over the last few years and affordable options are available and easy to obtain.

1.5.1 JB IMU

After dealing with proprietary issues with the initial IMU breakout board I selected, I was able to locate a more suitable alternative. The JB IMU from JB Robotics was a fully integrated IMU breakout board that communicates over SPI. The JB IMU produces attitude angle estimates with a quaternion-based Extended Kalman Filter (EKF) that avoids singularities associated with Euler angle based calculations. For the purposes of the project, the board provided angle, angular rate and force measurements as 16 bit signed integers that could be readily used by the flight control module.

There were a few issues I encountered working with the IMU. Documentation was rather sparse with regards to the SPI protocol used. A specification document was provided for the onboard microcontroller, but that did not provide enough information because multiple modes and register widths were possible. As it was designed to be used by Arduinos, there was some sample code provided, but that was rather minimal as well. From studying the code, I was able to determine all the default settings were left intact with Arduino SPI library commands indicating that Mode 0 was being used with a 4 MHz SCK frequency. A follow-up email with the manufacturer confirmed that the JB IMU was a Mode 0 slave and that an 8 bit output SPI bus was used. From that information, I was then able to write an SPI module.

Another major issue was I did not receive the IMU breakout board until after Thanksgiving due to the late switch of sensor boards. That left only a week remaining in the project to troubleshoot all the issues and successfully integrate the sensor board. I was successful in developing an SPI module and reading the output data from the board in the time remaining, but that left little time for much else.

1.5.2 Proprietary Issues

As mentioned previously, I initially selected a different IMU breakout board but ran into severe proprietary issues. I originally selected a pretty cheap ($10) breakout board containing the Invensense MPU6050 6DOF chip. I had chosen the MPU6050 because it not only contains both an accelerometer and gyroscope, but most importantly has an onboard sensor fusion digital signal processor (DSP) that incorporates the gyroscope and accelerometer readings to make an accurate estimate on current angular orientation.

Unfortunately, initializing and interfacing with the DSP to obtain the angle states is quite difficult because the manufacturer of the chip Invensense only provides for a proprietary software package to communicate with the chip. While I did find several open source communities that were able to reverse engineer the chip and make available a series of Arduino code packages to interface with the chip, the initialization procedure required configuring the volatile memory dump with over 20Kb and proved very complicated to implement and beyond my current skill level.

If I would have been able to initialize the DSP properly, I would then have had to deal with complex trigonometric functions to convert the quaternion to angles for my control calculations. With the DSP properly initialized, the IMU would have sent out 42 bit data packet FIFO (first in, first out) queue on the chip that contains the accelerometer (g forces), gyroscope(degrees per second) and angular readings in quaternions. I could have implemented a CORDIC module for the trigonometry but that would have required a lot more logic slices on the Spartan 6 FPGA compared to the using the IMU board I ended up with.

2 Quadcopter Control

Over the last decade, the market for small UAVs have rapidly exceeded its origins in the hobbyist and military communities to become a reality for the broader consumer and commercial space. Much of this growth has been enabled by the near universal availability of cheap, fast microprocessors that can perform all the regulation needed to allow humans or machine interfaces to fly the UAVs safely. These microprocessors are more than fast enough to accommodate simple control schemes, but struggle with tasks that require more parallel execution of tasks. For a vertical takeoff and landing (VTOL) vehicle such as a quadcopter, four controls inputs are ideally needed to be executed in parallel to operate the vehicle. A microcontroller overcomes this need for parallelism by operating at high enough frequency such that to the physical system, the control inputs are applied almost instantaneously. FPGAs provide an attractive option to accomplish true parallelism for controls as well as signal processing without sacrificing the qualities that make microprocessors attractive development platforms (i.e. affordability, scalability and adaptability).

2.1 Digital Control Background

Controllers for small unmanned aerial vehicles are necessary in order to stabilize the aircraft and allow for the system to be controllable by a human or machine interface. In general, controllers act to regulate systems, also referred to as *plants*, through state feedback in which system responses are used to calibrate the future response of the system. Controllers can be mechanical, analog, digital or biological in nature. As shown in **Figure 4,** a digital controller uses computer processors and/or digital integrated circuits to calculate control inputs and provide the regulation of the system. For physical systems, a digital controller will need to use analog to digital converters (A2D) and digital to analog converters (D2A) in order to interact with and observe the physical plant and its environment.

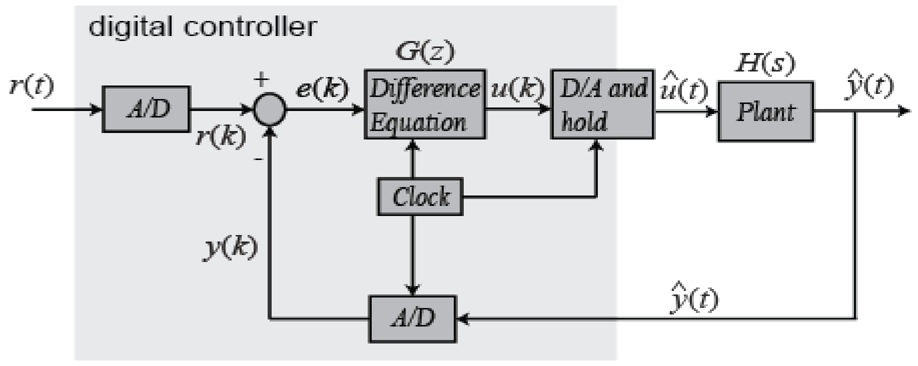

Figure 5: A generalized diagram of digital control system
Source: University of Michigan Engineering

The quadcoptor digital controller that was to be implemented in the FPGA follows a similar pattern. As displayed in **Figure 5**, the FPGA contains a flight controls module that calculates the control inputs and subsequently provides a throttle setting to the motor control module. The motor control module then emits a pulse-width-modulation (PWM) signal to the electronic speed controllers (ESCs) to operate the motors. In this way, the motor control module and ESCs act as digital to analog converters (DACs) to the plant. The system responses and state feedback is then reintroduced to the controller via sensor modules (the ADCs of the system) that communicate with the sensors.

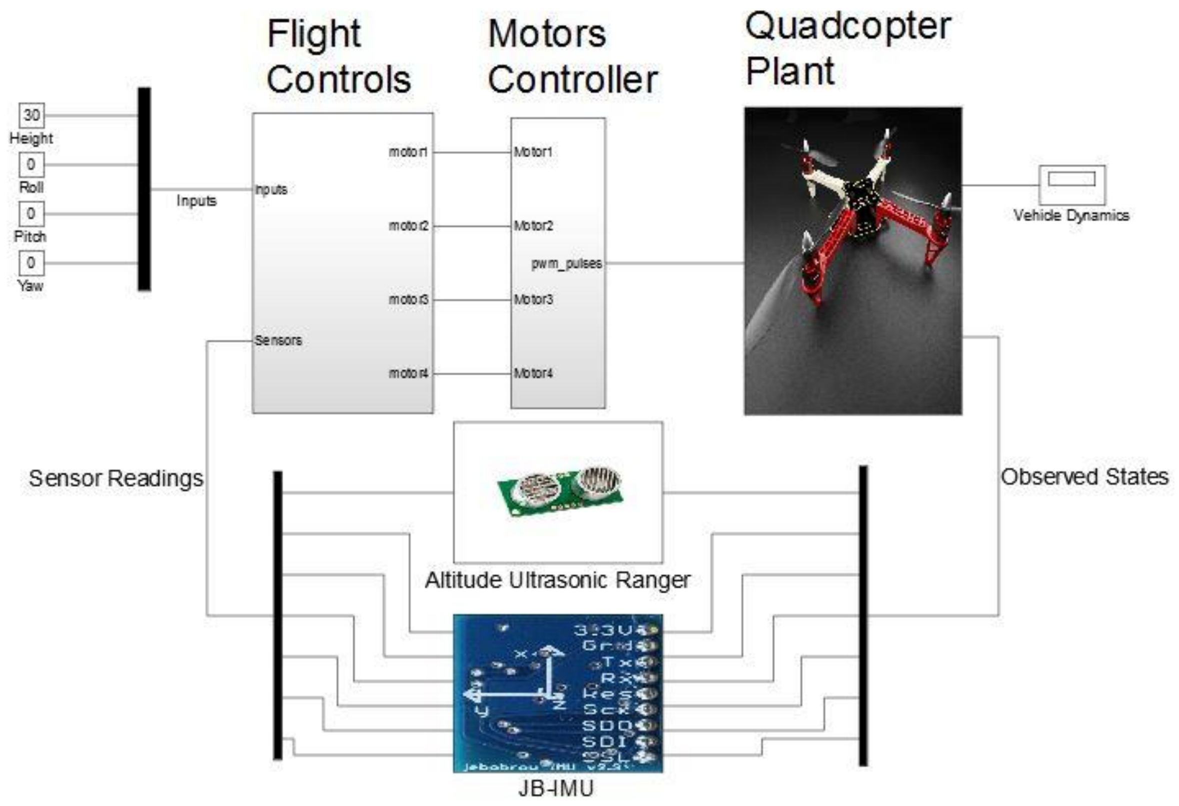

Figure 6: Simulink model of the quadcoptor digital controller

2.2 Modeling the Quadcopter

It was necessary that a model of the quadcopter dynamics be determined in order to derive control equations and for simulation purposes. As I had no prior experience in system identification, I needed a working simplified dynamics model in order to design the controller. In selecting a model and control scheme, I referred to a number of academic papers and reports.

I originally based my work on a paper and series of MATLAB scripts by Andrew Gibiansky. It initially seemed that Gibianksy's work could readily be adapted to my purposes and thus allow me to avoid having to develop my own MATLAB and

Simulink simulation from scratch. However, it became clear that **Gibiansky** was inadequate for the needs of the project. The model in **Gibiansky** relied on precise estimates of system parameters in calculating the controls that I could not accurately measure given available resources and the chosen hardware platform. The control scheme also relied on divisions and trigonometry functions that would have been too resource heavy for implementation on the FPGA.

I eventually settled on using a dynamic model and control scheme derived in **Mellinger, et al** from the University of Pennsylvania. Much like **Gibiansky**, the paper had a full derivation of a quadrotor model and control scheme. The control scheme was much more simplified and did not rely on having accurate estimates of system parameters to work properly. The paper also gave instructions for implementing hover control and most importantly, had documented success implementing the system on a digital feedback controller.

2.2.1 Dynamics of Quadcopters

A quadcopter is what is referred to as an underactuated plant. An underactuated system is one in which there are fewer independent inputs than there are degrees of freedom. Like all bodies in space, quadcopters have six degrees of freedom (three translational and three rotational), however, the quadcopter only has four independent inputs that can be used for movement. The quadcopter is able to move in space through altering the rotor speeds of its four motors to generate moments. Inducing a moment alters the rotational attitude of the aircraft and changes the thrust vector producing translational motion. The rotational and translational motion of the quadcopter is highly coupled so the dynamics of the quadcopter are highly nonlinear.

From **Mellinger et al**, the following state space model of the quadcopter was derived (**see Appendix A** for the complete derivation):

$$I\begin{bmatrix}\dot{p}\\ \dot{q}\\ \dot{r}\end{bmatrix} = \begin{bmatrix}L(F_2 - F_4)\\ L(F_3 - F_1)\\ M_1 - M_2 + M_3 - M_4\end{bmatrix} - \begin{bmatrix}p\\ q\\ r\end{bmatrix} \times I \begin{bmatrix}p\\ q\\ r\end{bmatrix}$$

*Figure 7: Simplified state space dynamics of quadcopter (**Mellinger, et al: 4).***

2.2.2 Motor Model

The connection between force (**F_i**) and moments (**M_i**) and the rotor speeds (**ω_i**) of each motor was also derived by **Mellinger, et al**. The relationship can be modeled as

$$F_i = k_F{\omega_i}^2$$
$$M_i = k_M{\omega_i}^2$$

, where k_F and k_M are constant coefficients that relate rotor speed to thrust and torque respectively. Rough estimates were calculated based on available performance data, but would ideally be matched to real performance data from testing. Estimates for the motor coefficients were calculated to be:

$$k_F = 9.18 \times 10^{-8} \frac{N}{rpm^2}$$

$$k_M = 3.01 \times 10^{-5} \frac{Nm}{rpm^2}$$

In hindsight, it would have been wise to have selected a more well-known and tested motor for the project, instead of the cheaper Chinese motor clone I went with. I was able to find some performance data for the iPower 2212Q motor selected for the project, but I would have saved a good amount of time and effort if I had been more prudent in my choice.

2.3 Developing Control Equations

2.3.1 Discrete PID Control Equations

As mentioned previously, I desired to implement a PID digital control scheme. PIDs (also P, PI and PD) are a control scheme where proportional gains are applied to the derivative, integral and present feedback error to compute control inputs. PID controllers are most commonly used in industrial control applications or where single set points are desired. PID controllers have been popular for the hobbyist UAV flight controllers because they are rather simple to program compared to more modern methods. Advantages of PID controllers are they do not rely on having very accurate dynamics models, can be tuned manually using proven methods, and can be readily converted for discrete applications.

The classic form of the PID error equation is

$$u(t) = K_d \frac{d}{dt}\left(x^{des} - x(t)\right) + K_P\left(x^{des} - x(t)\right) + K_i \int \left(x^{des} - x(t)\right)$$

, where K_d, K_i, K_p are the gains for the derivative, integral and present error respectively. To be useful for digital control, the discrete form of PID control (derived from backwards difference) is needed which is

$$u[k] = u[k-1] + \frac{K_d}{T_s}(e[k] - 2e[k-1] + e[k-2]) + K_P(e[k] - e[k-1]) + K_i T_s e[k]$$

, where

k	=	sample number
u[k]	=	controller input value at sample k
e[k]	=	the error term (x^{des}-x[k])
T[s]	=	Sample period

This discrete form is also known as the Type A equation. Type B and Type C equations involve removing the error term from the proportional and both the proportional and derivative terms respectively. Moving the set point from terms lessens the impact of rapid changes in the state and makes it easier to tune the gains.

$$\boldsymbol{Type\ B}: u[k] = u[k-1] + \frac{K_d}{T_s}(-x[k] + 2x[k-1] + x[k-2]) + K_P(e[k] - e[k-1]) + K_i T_s e[k]$$

$$\boldsymbol{Type\ C}: u[k] = u[k-1] + \frac{K_d}{T_s}(-x[k] + 2x[k-1] + x[k-2]) + K_P(-x[k] + x[k-1]) + K_i T_s e[k]$$

2.3.2 Attitude PD Control

The control scheme detailed in **Mellinger et al**, incorporates two controllers in cascade feedback: an inner loop attitude controller and an outer loop trajectory controller. As shown in **Figure 5** below, the simplified control scheme relates attitude error and required rotor speed to desired input rotor speed. The term ω_h is calculated to be the required rotor speed for hover. A proportional derivative control law was given for calculating the nominal vectors for roll ($\Delta\omega_\phi$), pitch ($\Delta\omega_\theta$), and yaw ($\Delta\omega_\psi$) as shown in **Figure 6**. Due to selecting the correct IMU board, the PD equations in **Figure 6** can be used directly in the digital controller because the IMU returns both present angle and angular rate.
Hover Control

$$\begin{bmatrix} \omega_1^{des} \\ \omega_2^{des} \\ \omega_3^{des} \\ \omega_4^{des} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 1 & 1 & 0 & -1 \\ 1 & 0 & 1 & 1 \\ 1 & -1 & 0 & -1 \end{bmatrix} \begin{bmatrix} \omega_h + \Delta\omega_F \\ \Delta\omega_\phi \\ \Delta\omega_\theta \\ \Delta\omega_\psi \end{bmatrix}$$

Figure 9: State space control scheme relating desired rotor rpm to PD difference errors (Mellinger: 4)

$$\Delta\omega_\phi = k_{p,\phi}(\phi^{des} - \phi) + k_{d,\phi}(p^{des} - p)$$
$$\Delta\omega_\theta = k_{p,\theta}(\theta^{des} - \theta) + k_{d,\theta}(q^{des} - q)$$
$$\Delta\omega_\psi = k_{p,\psi}(\psi^{des} - \psi) + k_{d,\psi}(r^{des} - r)$$

Figure 8: PD difference equations for attitude control (Mellinger:5)

2.3.3 Hover Control

For the purposes of this project, only hover control was desired so the desired roll and pitch angles were desired to be constant at level (equal to zero) and the PID feedback was only implemented on the height measurement. Using the type C equations from above, the PID control for height (z) with slight modification is

$$u_Z[k] = \frac{K_d}{T_s}(-z[k] + 2z[k-1] + z[k-2]) + K_P(-z[k] + z[k-1]) + K_i T_s(z^{des} - z[k])$$

, and then for hover the remaining terms of the control laws are $\phi^{des} = 0$, $\theta^{des} = 0$, $\psi^{des} = \psi_0$(some arbitrary yaw orientation), and $\Delta\omega_h = \frac{m}{8k_F\omega_h}u_Z[k]$.

3 IMPLEMENTATION

3.1 IMPLEMENTATION OVERVIEW

The core of the project consisted of several finite state machine (FSM) modules consisting of a **flight control module (FC), motor control module (MC), an IMU interface module (JB_IMU), and an altitude range sensor module (SR05)** implemented on the Xilinx Spartan 6 FPGA**.** A high level diagram of the modules is shown in **Figure 7** below.

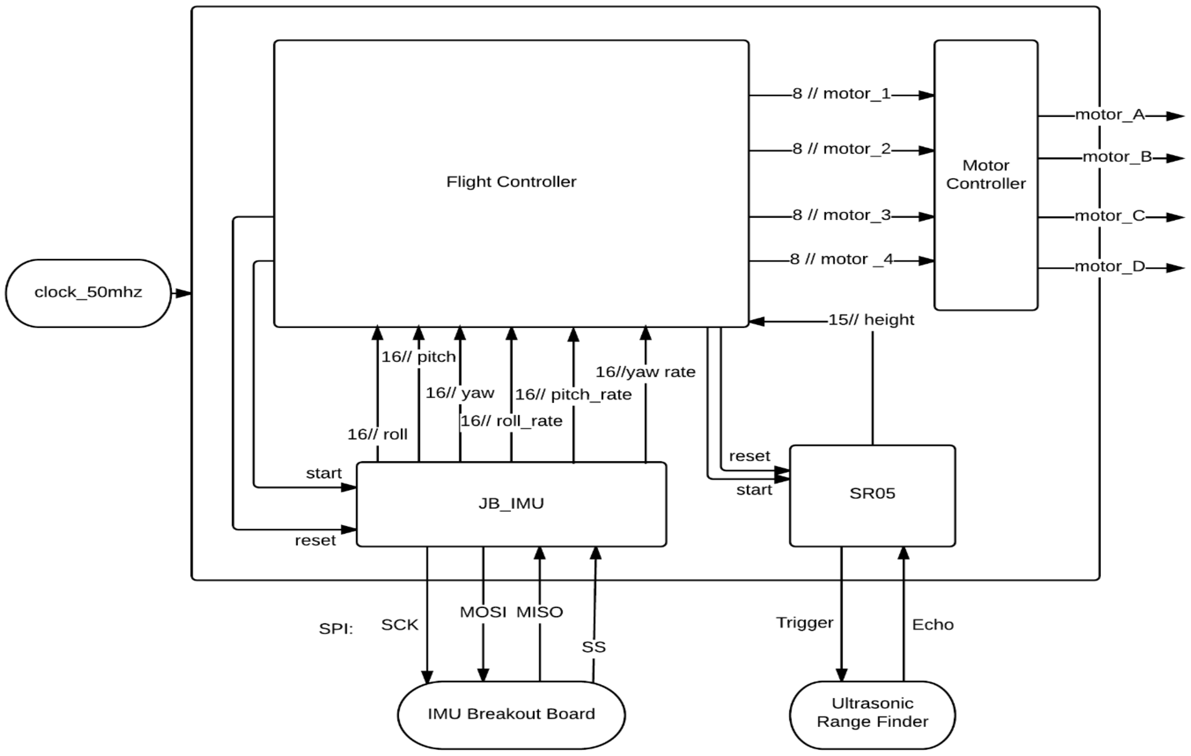

Figure 10: Module Level FPGA Block Diagram

The flight control module (FC) performs the flight arithmetic functions explained in **Section 2.3**, sends throttle settings to the motor controller, and interfaces with the sensor modules. The flight control module is also where the system level state machine resides. The system level state machine controls system function modes such as initialization and flying.

The motor controller takes in an 8 bit throttle setting (0-255) for each motor and converts that setting to a PWM signal using a look up table and PWM module for each motor. The refresh rate of the motor controllers is 400 Hz, which is the maximum refresh rate of the electronic speed controllers.

The JB_IMU is a driver for connecting to the JB Robotics IMU breakout board. The module communicates over SPI with the board and retrieves a 16 bit signed integer values for current attitude (roll, pitch, yaw), angular rates, and body accelerations. The module then outputs the attitude and angular rate information to the flight control module. The refresh rate is limited to 250 Hz because the extended Kalman filter (EKF) algorithm implemented on the JB IMU is 250 Hz.

The SR05 module is a driver for the control of the SR05 ultrasonic ranger. The module triggers an ultrasonic pulse and then reads the active high signal time in microseconds of the echo pulse to determine the range. The module then outputs the distance reading as a 15 bit unsigned integer with a cycle time of 20 Hz. The sensor itself needs time to reset and can only operate at a minimum of 20 Hz.

All modules are connected to the same system clock of 50 megahertz. The Mojo has a 50 Mhz crystal oscillator which is more than fast enough to perform all operations necessary given the relatively slow update rates of the modules required. Synchronizing all modules on the same clock allows for oversampling of sensor inputs and preventing any metastability issues due to different clock domains.

3.2 Flight Controller (FC) Module

3.2.1 System Level FSM

A system level FSM was intended to be included in the system to control functionality of the FPGA modules, but was ***not fully implemented by the end of project deadline.*** Only partial functionality was implemented in Verilog module *flight_controller.v*.

The quadrotor controller was designed to be in several states: **Power On Reset**, **Initialization**, **Fail, Idle**, and **Fly**. When the system is first supplied power, the system will be in **Power On Reset** mode for 2 seconds to complete hardware initialization. The system will then move to **Initialization** mode to calibrate the ESCs, and to self-test and calibrate the digital sensors. If the **Initialization** is not successful, the system will move to **Fail** mode and flash the LEDs on the FPGA in a specific pattern to indicate that the system is not safe for flight. If the **Initialization** is successful, the system will move to **Idle** mode and wait for operator input to begin flying. When the operator green lights for takeoff, the system will change to **Fly** mode, which is the normal operating mode. The motors will initially operate at 1% throttle setting to indicate that the system is ready to fly. The system can return to **Idle** mode if the system senses it is on the ground and after being directed by the operator.

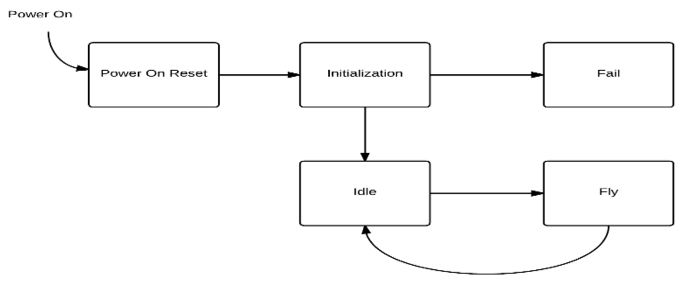

Figure 11: System state flow diagram

3.2.2 Initialize

When the system is in the **initialization** state, an initialization process occurs. This initialization process was implemented in a standalone Verilog module *flight_control_initialize.v*. However, the module was not tested to be working by the project deadline due to compiler issues that were unfortunately left unresolved.

The initialization procedure focused on three tasks: 1) Calibrate the ESCs and make sure they are operational, 2) start the SR05 sensor module and validate that proper signal returns are valid, and 3) start the JBIMU sensor module, validate that the sensor driver is working, and record initial offset state readings to be used by the flight arithmetic module. After each task was completed, the motors were then commanded to spin at a low RPM setting for visual indication that initialization was successful. If the initialization process was successful, then a finished signal was sent to the parent module. If an error occurred, the parent module is notified that there is an error with a separate signal.

3.2.3 FC_Arithmetic

The FC_Arithmetic module was intended to be the core module at the heart of the flight controller that would operate when the system is in FLY mode. **This module was not implemented by the project deadline.** This module was to perform the control calculations described in **Section 2.3** in fixed point integer arithmetic with proper attention to the scaling of the system state readings from the sensors. For example, the angle estimates given by the JB IMU as 16 bit signed integers were scaled by 100 so that would then need to be accounted for in the fixed point math.

The arithmetic module will then output four independent inputs for RPM speed scaled down to a 16 bit unsigned integers. The 16 bit RPM settings would then be sent to a look up table that would convert the RPM setting to an 8 bit throttle setting to be read by the motor controller.
The arithmetic module would operate on a 400 Hz rate to match the refresh rate of the motor controller. The module would also only calculate new values when new data was available from the JB_IMU or SR05 module that are operating at a 250Hz and 20 Hz cycle time, respectively.

3.3 Motor Controller (MC) Module

The motor controller module acts as the interface between the flight controller module and the signals sent to the ESCs. The module contains look up tables in series with a pulse width modulation (PWM) generator that converts the throttle setting from the flight controller and outputs the PWM signal with the desired duty time. For interfacing with the SimonK ESCs, a PWM active high signal is needed for anywhere between 1060 us (for Idle) and 1860 us (for Full Throttle).

3.3.1 Throttle2PWM

This module implements a look up table that converts the 8 bit unsigned integer throttle setting (0-255) to a valid 12 bit comparison time to be used by a subsequent PWM module. The module can also take input for desired idle. If *idle* is high, then the signal output to the pwm module is less than 1060 us. Anything that is less 1060 us is read by the ESCs as idle. This just provides extra maneuvering ability between modules in case the flight controller wants to cutoff the motors instantly.

3.3.2 PWM

The PWM.v module takes a control signal of some parameter length and outputs a pwm signal of the desired duty time. For the purposes of this project, the module works on a refresh rate assigned as a parameter in microseconds. For the 400 Hz refresh rate desired, that rate is 2500 microseconds. Default parameters values for the control length is 12 bits long and for the refresh rate 12 bit integer equal to 2499 (400 Hz).

The module operates through the use of a 1 microsecond divider. Upon each 1 us enable signal, a counter is incremented. If the counter is less the comparison control signal, the pwm output signal is high. If the counter is greater than the comparison signal, than the pwm output signal is low. The counter resets when the refresh rate is reached as given by the parameter.

3.4 JB IMU Module (JB_IMU)

As mentioned previously in **Section 1.5**, the JB IMU can provide state orientation data through a SPI Mode 0 slave interface. **A SPI master module was implemented that successfully communicates to the IMU slave** with Mode 0 enabled. A driver module was also implemented to control the SPI transactions and to obtain all the data from the JB IMU. Simulation was used to validate the implementations and the actual operation was demonstrated on the labkit.

3.4.1 SPI

An SPI master mode 0 module was implemented in Verilog as *spi.v*. SPI is a 4 wire communication protocol between a single master and its slaves. The master generates a SCK clock signal to be used in the transaction and also transmits bits serially on the Master Out-Slave In wire (MOSI). The master reads in bits serially from the slave on the Master In-Slave Out wire (MISO). The master can select a slave to communicate with by pulling the Slave Select (SS) wire low.

The JB IMU operates as a Mode 0 slave. Of the four possible modes in the SPI specification, Mode 0 indicates that clock polarity when idle is low (CPHOL =0), and data is transmitted on the negative edge of the clock signal (CPHA = 0). The SPI module correspondingly samples new data on the rising edge of the clock signal.

The *spi.v* module operates in three states: IDLE, WAIT_HALF, and TRANSFER. When a start signal arrives, the module pulls the slave select low and then enters WAIT_HALF state. The module waits half a SCK cycle before starting TRANSFER mode. An SCK signal is generated by assigning the last bit of the clock divider with bit width set by a parameter. The master SPI then proceeds with the data transfer: transmitting each bit (MSB to LSB) on MOSI on the negative edge of SCK and then sampling the MISO until filling the output data bus. When the output data bus is filled, the state is returned to IDLE, SS is reasserted active high, and the *new_data* output signal is assigned high.

An interesting feature implemented is that the SPI bus width can be set arbitrarily through parameters. An internal function implements log base 2 to calculate the number of bits needed to count as a local parameter.

3.4.1.1 SPI Testbench

The Verilog testbench *test_spi.v* was successfully simulated using ModelSim to verify that the spi.v module worked as expected. A mode 0 slave Verilog module was used to test the interface. The spi_slave.v was included with starter Mojo code.

3.4.2 JB IMU FSM

As previously mentioned in **Section 1.5**, I relied on sample Arduino code for understanding how to interface with the board. The Verilog code *jb_imu.v* is loose interpretation of that code. The sample Arduino code sends one dummy signal to begin the transaction and then performs 9x2 = 18 transactions to obtain the attitude, angular rate, and accelerometer data. Each 16 bit signed integer is thus divided into two 8 bit transactions.

Each cycle, the JB_IMU sends 19 transmissions of all zeros to the slave. The first 8 bit transaction is discarded by the master. Thereafter, all transactions are shifted onto an internal 144 bit (9*16) data register. When all transmissions are finished, outputs are assigned in order whereas the first 48 bits are the three 16 bit signed attitude angles, the next 48 bits are the three 16 bit signed angular rates, and the last 48 bits are the three 16 bit signed accelerations.

After completing the round of transactions, the JB_IMU waits in IDLE state until receiving an enable signal high from a 250 microsecond divider.

3.4.2.1 JB_IMU Testbench

Similar to the SPI test bench, the JB_IMU testbench, *test_jbimu.v*, was used to verify proper operation with the Mode 0 slave module, *spi_slave.v*. The test bench sends incrementing 8 bit integers from the slave to the master. Correct operation is observed when the unit under test properly outputs values with incrementing hexadecimal values.

3.5 SR05 MODULE (SR05)

The SR05 operates through a trigger and echo cycle and requires 20 Hz cycle in order to reset properly. The SR05 was successfully implemented, verified in simulation and tested on the labkit to be working properly.

3.5.1 Trigger and Echo

The Verilog module *srf05_trigger_and_echo* is just a dumb driver that upon a start signal sends a 10 microsecond pulse on the trigger output wire and then listens for a high signal on the echo wire input. The module then counts the time length of the TTL signal. If the time signal recorded is not of invalid length (less than 100 microseconds or more than 30 milliseconds), the time recorded is then assigned to the 15 bit unsigned output distance. There is error detection and time outs if the sensor malfunctions.

3.5.1.1 *Testbench for SR05*

A test bench, test_srf05.v, was implemented to verify the proper functionality of the trigger and echo low level driver. The final module successfully passed the simulation test.

3.5.2 SRF05 FSM

The finite state machine module, *srf05.v*, operates the trigger and echo low level driver on a 20 Hz refresh rate. The FSM first triggers the low level driver. The FSM then waits for a ready signal. If the ready signal is pulled high, the FSM then moves to the store the value from the low level driver to the 15 bit unsigned distance output. The FSM then waits to receive a high enable signal from the 50 ms/20Hz divider to begin a new trigger and receive cycle.

4 Lessons Learned

Although the project was not successful, it became clear that for just basic flight controls, the FPGA was not a useful platform and the microcontroller is more than adequate for use. Other than incorporated into a novelty project, implementing a PID controller on an FPGA is just overkill and an enormous of effort for not much gain. Possible areas where FPGA would be suitable for more advanced control schemes incorporating inverse learning networks or for providing additional sensor functionality such as optical flow sensing for position tracking. Generally, use case for microcontrollers breakdown where parallelism is necessary and this is exactly where the FPGA would be best suited for.

To conclude this report, the following is a non-exhaustive list of topics and tasks that I was able to learn and demonstrate in this project.

- Digital Systems
- Communication Protocols (SPI/I2C)
- Interfaces between digital and physical
- R/C aircraft technology
- Suitable Uses for FPGA
- IMU (proprietary vs composite)
- Sensor Filtering (Kalman, Direction Cosine Matrix, EKF, Complementary Filter)
- Fixed Point vs Floating Point
- PID Digital Control (Types, Designs, Pros/Cons)
- Soldering/Powering Connections
- Quadrotor Modeling
- Underactuated Control
- Verilog vs VHDL
- Latches vs Shift Registers in HDL Design
- UCF constraint files
- Using FPGA very different from Labkit
- PWM

5 References

5.1 Internet Sources (informal Citations)

Devantech SRTF05 Ultrasonic Ranger http://www.acroname.com/products/R271-SRF05.html

"I2Cdev library collection by Jeff Rowberg" https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050

Mojo V3 by Embedded Micro https://embeddedmicro.com/products/mojo-v3.html

Servo Shield by Embedded Micro https://embeddedmicro.com/servo-shield.html

5.2 Textual Sources (Formal Citations

A. Gibianksy., "Quadcopter Dynamics, Simulation, and Control" Nov. 2014https://github.com/gibiansky/experiments/tree/master/quadcopter

B K.P. Horn and B. G. Schunck., "Determining Optical Flow," in *Artificial Intelligence* 17(1981), Mar 1980, pp. 185-203. 0004-3702/81/0000-0000 © North-Holland

D. Mellinger, et al. "Trajectory Generation and Control for Precise Aggressive Manuevers with Quadrotors" GRASP Laboratory, University of Pennsylvania. 2010 http://www.seas.upenn.edu/~dmel/mellingerISER2010.pdf

G. Shapiro. "Commercial Drones Are Ready For Takeoff." *Washington Post*. The Washington Post, 4 July 2014. Web. 14 Nov. 2014.

P.D. Jameson and A Cooke, "Developing Real-Time System Identification for UAVs" in *UKACC International Conference on Control 2012*, Cardiff, UK, 3-5 September 2012, 978-1-4673-1560-9/12 ©2012 IEEE

P. Schad and D. Carney, Plexus. "Case Study of PID control in an FPGA" [jan 17, 2011] http://www.embedded.com/design/configurable-systems/4212241/Case-Study-of-PID-Control-in-an-FPGA-

S. G. Tzafestas "Digital PID and Self-Tuning Control" Applied Digital Control. ed. Tzafestas. Elsevier Science Publishers B.V. (North-Holland), 1985.

6 Glossary

FPGA	field programmable gate array
UAV	unmanned aerial vehicle
VTOL	vertical takeoff and land
PID	A feedback control scheme that uses derivative, integral and proportional constants to provide control inputs
IMU	inertial measurement unit
DC	direct current
DAC	digital to analog converter
ADC	analog to digital converter
SPI	serial periphereal interface
TTL	Analog signal that is similar nature of a digital signal, and can be recognized as a digital signal.
ESC	Electronic speed controller

7 Appendix A: Derivation of Quadcopter Model

The following is an excerpt from **Mellinger et al,** pp. 3-6. The following is a thorough derivation of the quadcopter dynamics and control scheme.

1.2 Modeling

1.2.1 Dynamic Model

The coordinate systems and free body diagram for the quadrotor are shown in Fig. 1.1(b). The world frame, $\mathcal{W}$, is defined by axes x_W, y_W, and z_W with z_W pointing upward. The body frame, $\mathcal{B}$, is attached to the center of mass of the quadrotor with x_B coinciding with the preferred forward direction and z_B perpendicular to the plane of the rotors pointing vertically up during perfect hover (see Fig. 1.1(b)). Rotor 1 is on the positive x_B-axis, 2 on the positive y_B-axis, 3 on the negative x_B-axis, 4 on the negative y_B-axis. We use Z-X-Y Euler angles to model the rotation of the quadrotor in the world frame. To get from $\mathcal{W}$ to $\mathcal{B}$, we first rotate about z_W by the yaw angle, ψ, then rotate about the intermediate x-axis by the roll angle, ϕ, and finally rotate about the y_B axis by the pitch angle, θ. The rotation matrix for transforming coordinates from $\mathcal{B}$ to $\mathcal{W}$ is given by

$$R = \begin{bmatrix} c\psi c\theta - s\phi s\psi s\theta & -c\phi s\psi & c\psi s\theta + c\theta s\phi s\psi \\ c\theta s\psi + c\psi s\phi s\theta & c\phi c\psi & s\psi s\theta - c\psi c\theta s\phi \\ -c\phi s\theta & s\phi & c\phi c\theta \end{bmatrix},$$

where $c\theta$ and $s\theta$ denote $\cos(\theta)$ and $\sin(\theta)$, respectively, and similarly for ϕ and ψ. The position vector of the center of mass in the world frame is denoted by $\mathbf{r}$. The forces on the system are gravity, in the $-z_W$ direction, and the forces from each of the rotors, F_i, in the z_B direction. The equations governing the acceleration of the center of mass are

$$m\ddot{\mathbf{r}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R \begin{bmatrix} 0 \\ 0 \\ \Sigma F_i \end{bmatrix}. \tag{1.1}$$

The components of angular velocity of the robot in the body frame are p, q, and r. These values are related to the derivatives of the roll, pitch, and yaw angles according to

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} c\theta & 0 & -c\phi s\theta \\ 0 & 1 & s\phi \\ s\theta & 0 & c\phi c\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}.$$

In addition to forces, each rotor produces a moment perpendicular to the plane of rotation of the blade, M_i. Rotors 1 and 3 rotate in the $-z_B$ direction while 2 and 4 rotate in the z_B direction. Since the moment produced on the quadrotor is opposite the direction of rotation of the blades, M_1 and M_3 act in the z_B direction while M_2 and M_4 act in the $-z_B$ direction. We let L be the distance from the axis of rotation of the rotors to the center of the quadrotor. The moment of inertia matrix referenced to the center of mass along the $x_B - y_B - z_B$ axes, I, is found by weighing individual components of the quadrotor and building a physically accurate model in SolidWorks. [1] The angular acceleration determined by the Euler equations is

[1] The off-diagonal terms of I are nearly zero since $x_B - y_B - z_B$ are close to the principal axes of the quadrotor.

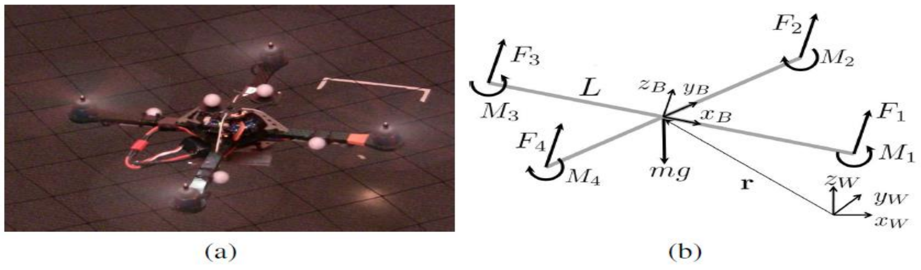

Fig. 1.1. (a) Quadrotor used in experiments. (b) Coordinate systems and forces/moments acting on the quadrotor.

$$I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} L(F_2 - F_4) \\ L(F_3 - F_1) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix}. \tag{1.2}$$

1.2.2 Motor Model

Each rotor has an angular speed ω_i and produces a vertical force F_i according to

$$F_i = k_F \omega_i^2. \tag{1.3}$$

Experimentation with a fixed rotor at steady state shows that $k_F \approx 6.11 \times 10^{-8} \frac{N}{rpm^2}$. The rotors also produce a moment according to

$$M_i = k_M \omega_i^2. \tag{1.4}$$

The constant, k_M, is determined to be about $1.5 \times 10^{-9} \frac{Nm}{rpm^2}$ by matching the performance of the simulation to the real system.

The results of a system identification exercise suggest that the rotor speed is related to the commanded speed by a first-order differential equation

$$\dot{\omega}_i = k_m(w_i^{des} - w_i).$$

This motor gain, k_m, is found to be about $20\,\mathrm{s}^{-1}$ by matching the performance of the simulation to the real system. The desired angular velocities, ω_i^{des}, are limited to a minimum and maximum value determined through experimentation to be approximately 1200 rpm and 7800 rpm.

1.3 Control

Here we present methods used to control the robot (a) to a desired attitude; (b) to hover in place; and (c) along a three-dimensional trajectory with specified position and linear velocity. We design a sequence of these controllers to generate the desired trajectories in the next section. The controllers presented in this section are also discussed in [7].

1.3.1 Attitude Control

The goal of this controller is to reach a desired attitude with a specified angular velocity. The vector of desired rotor speeds can be written as a linear combination of four terms

$$\begin{bmatrix} \omega_1^{des} \\ \omega_2^{des} \\ \omega_3^{des} \\ \omega_4^{des} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 1 & 1 & 0 & -1 \\ 1 & 0 & 1 & 1 \\ 1 & -1 & 0 & -1 \end{bmatrix} \begin{bmatrix} \omega_h + \Delta\omega_F \\ \Delta\omega_\phi \\ \Delta\omega_\theta \\ \Delta\omega_\psi \end{bmatrix}, \tag{1.5}$$

where the nominal rotor speed required to hover in steady state is ω_h, and the deviations from this nominal vector are $\Delta\omega_F$, $\Delta\omega_\phi$, $\Delta\omega_\theta$, and $\Delta\omega_\psi$. $\Delta\omega_F$ results in a net force along the z_B axis, while $\Delta\omega_\phi$, $\Delta\omega_\theta$, and $\Delta\omega_\psi$ produce moments causing roll, pitch, and yaw, respectively. We use proportional derivative control laws that take the form

$$\begin{aligned} \Delta\omega_\phi &= k_{p,\phi}(\phi^{des} - \phi) + k_{d,\phi}(p^{des} - p) \\ \Delta\omega_\theta &= k_{p,\theta}(\theta^{des} - \theta) + k_{d,\theta}(q^{des} - q) \\ \Delta\omega_\psi &= k_{p,\psi}(\psi^{des} - \psi) + k_{d,\psi}(r^{des} - r). \end{aligned} \tag{1.6}$$

Substituting (1.6) into (1.5) yields the desired rotor speeds.

1.3.2 Hover Controller

The goal of this controller is to reach a desired position and yaw angle with zero linear and angular velocities. Here we use pitch and roll angle to control position in the x_W and y_W plane, $\Delta\omega_\psi$ to control yaw angle, and $\Delta\omega_F$ to control position along z_W. We let $\mathbf{r}_T(t)$ and $\psi_T(t)$ be the trajectory and yaw angle we are trying to track. Note that $\psi_T(t) = \psi_0$ for the hover controller. The command accelerations, $\ddot{\mathbf{r}}_i^{des}$, are calculated from PID feedback of the position error, $e_i = (\mathbf{r}_{i,T} - \mathbf{r}_i)$, as

$$(\ddot{\mathbf{r}}_{i,T} - \ddot{\mathbf{r}}_i^{des}) + k_{d,i}(\dot{\mathbf{r}}_{i,T} - \dot{\mathbf{r}}_i) + k_{p,i}(\mathbf{r}_{i,T} - \mathbf{r}_i) + k_{i,i}\int(\mathbf{r}_{i,T} - \mathbf{r}_i) = 0,$$

where $\dot{\mathbf{r}}_{i,T} = \ddot{\mathbf{r}}_{i,T} = 0$ for hover. We linearize (1.1) to get the relationship between the desired accelerations and roll and pitch angles

$$\begin{aligned}
\ddot{\mathbf{r}}_1^{des} &= g(\theta^{des}\cos\psi_T + \phi^{des}\sin\psi_T) \\
\ddot{\mathbf{r}}_2^{des} &= g(\theta^{des}\sin\psi_T - \phi^{des}\cos\psi_T) \\
\ddot{\mathbf{r}}_3^{des} &= \frac{8k_F\omega_h}{m}\Delta\omega_F.
\end{aligned}$$

These relationships are inverted to compute the desired roll and pitch angles for the attitude controller as well as $\Delta\omega_F$ from the desired accelerations

$$\phi^{des} = \frac{1}{g}(\ddot{\mathbf{r}}_1^{des}\sin\psi_T - \ddot{\mathbf{r}}_2^{des}\cos\psi_T) \tag{1.8a}$$

$$\theta^{des} = \frac{1}{g}(\ddot{\mathbf{r}}_1^{des}\cos\psi_T + \ddot{\mathbf{r}}_2^{des}\sin\psi_T) \tag{1.8b}$$

$$\Delta\omega_F = \frac{m}{8k_F\omega_h}\ddot{\mathbf{r}}_3^{des}. \tag{1.8c}$$

8 APPENDIX B: VERILOG

8.1 FLIGHT_CONTROLLER.V

```
`timescale 1ns / 1ps

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

// Flight Control Module Major FSM

//

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

module flight_controller #(parameter CALIBRATE_ESC = 1'b0)

(

        input clock,

        input reset,

//      output [7:0] led,

//      input fly,
        //whether should go to fly mode

//      input idle,
        //wheter should go back to IDLE

        //JB IMU interface

        output imu_start,                                                //start
imu signal

        output imu_reset,
        //reset imu signal

        input imu_new_data,                                              //new
data available

        input signed [15:0] roll,

        input signed [15:0] pitch,

        input signed [15:0] yaw,

        input signed [15:0] roll_rate,

        input signed [15:0] pitch_rate,

        input signed [15:0] yaw_rate,

//      //Arithmetic Inputs
//      input signed [15:0] desired_roll,
//      input signed [15:0] desired_pitch,
//      input signed [15:0] desired_yaw,
//      input signed [15:0] desired_roll_rate,
//      input signed [15:0] desired_pitch_rate,
//      input signed [15:0] desired_yaw_rate,
        //SRF05 interface
        output srf05_start,                                      //start
height sensor
        output srf05_reset,                                      //reset
height sensor
        input srf05_new_data,                                    //new data
available
        input [14:0] distance,
        //Motor Controller interface
        output motors_start,
        output motors_idle,
        output motors_reset,
        output [7:0] throttle1,
        output [7:0] throttle2,
        output [7:0] throttle3,
        output [7:0] throttle4
        );
/////////////////////////////////////////////////////////////////////////////////////
//////
// 1 Second Divider
        wire reset_div1, en_sec;
        divider_sec #(.CLK_FRQ_MHZ(50))
timer_sec(.clock(clock),.reset(reset_div1),.en_sec(en_sec));
```

```
/////////////////////////////////////////////////////////////////////////////////////////
//////
///	Initialization Components


	wire signed [15:0] init_roll, init_pitch, init_yaw, init_P, init_Q, init_R;
	wire [14:0] init_distance;
	reg init_reset, init_start;
	wire init_finished;
	wire init_reset_div1;
	wire error_srf05, error_imu;
	wire calibrate;
	assign calibrate = CALIBRATE_ESC;
	wire [31:0] init_throttles;
	wire init_imu_start, init_imu_reset;
	wire init_srf_start, init_srf_reset;
	wire init_motors_start, init_motors_reset;


	//Initialization Module
	flight_control_initialize init(
	.clock(clock),
	//clock
	.reset(init_reset),
	//reset initialize
	.start(init_start),
	//start initialization
	.done(init_finished),
	//indicates initialization is done
	.error_srf05(error_srf05),
	.error_imu(error_imu),
	.reset_1sec(init_reset_div1),
	//reset 1 sec divider
```

```
        .en_sec(en_sec),
        //divider signal from 1 second divider


        .calibrate(calibrate),
        //if high, calibrate motors

        .throttles(init_throttles),                        //throttle settings
for motors

        .motors_start(init_motors_start),
        //start motor controller

        .motors_reset(init_motors_reset),
        //reset motor controller

        .srf05_start(init_srf_start),
        //start height sensor

        .srf05_reset(init_srf_reset),
        //reset height sensor

        .srf05_new_data(srf05_new_data),

        .distance(distance),
        //height reading from sensor (in microseconds)

        .offset_distance(init_distance),                //offset initial height
reading (in microseconds)

        .imu_start(init_imu_start),
        //start imu signal

        .imu_reset(init_imu_reset),
        //reset imu signal

        .imu_new_data,                                         //new data
available

        .cur_roll(roll),                                 //roll angle: sensor
reading from imu

        .cur_pitch(pitch),                           //pitch angle: sensor reading
from imu

        .cur_yaw(yaw),                                     //yaw angle:
sensor reading from imu

        .cur_roll_rate(roll_rate),              //roll rate: sensor reading from imu

        .cur_pitch_rate(pitch_rate),            //pitch rate: sensor reading from imu
        .cur_yaw_rate(yaw_rate),                        //yaw rate: sensor reading from
imu
        .offset_roll(init_roll),                //roll angle: offset for control
arithmetic
        .offset_pitch(init_pitch),              //pitch angle: offset for control
arithmetic
        .offset_yaw(init_yaw),                          //yaw angle: offset for control
arithmetic
        .offset_roll_rate(init_P),              //roll rate: offset for control arithmetic
        .offset_pitch_rate(init_Q),             //pitch rate: offset for control
arithmetic
        .offset_yaw_rate(init_R)                //yaw rate: offset for control
arithmetic
    );


////////////////////////////////////////////////////////////////////////////////////////////
//////
///     FLY Modules




////////////////////////////////////////////////////////////////////////////////////////////
//////
///     Major FSM states
        localparam POWER_ON_RESET = 2'd0,
                                INITIALIZE = 2'd1,
                                IDLE = 2'd2,
        //                      FLY = 3'd3,
                                FAIL = 2'd3;
        reg [1:0] state, next_state;


        //Assign I/Os
```

```
	assign srf05_start		= (state == INITIALIZE) ?	init_srf_start
	: 1'b0;								//start height sensor

	assign srf05_reset		= (state == INITIALIZE) ?	init_srf_reset
	: 1'b1;								//reset height sensor

	assign imu_start			= (state == INITIALIZE) ?
	init_imu_start		: 1'b0;
					//start imu signal

	assign imu_reset			= (state == INITIALIZE) ?
	init_imu_reset		: 1'b1;
					//reset imu signal

	assign motors_start			= (state == INITIALIZE) ?
	init_motors_start	: 1'b0;

	assign motors_reset			= (state == INITIALIZE) ?
	init_motors_reset	: 1'bz;

	assign motors_idle		= 1'b0;

	assign reset_div1			= (state == INITIALIZE) ?
	init_reset_div1		: 1'b0;

	wire [32:0] throttles;

	assign throttles = (state == INITIALIZE) ? init_throttles : 32'h0;

	assign throttle1 = throttles[31:24];

	assign throttle2 = throttles[23:16];

	assign throttle3 = throttles[15:8];

	assign throttle4 = throttles[7:0];

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


	always @(posedge clock) begin

		if(reset) begin

			state <= POWER_ON_RESET;

			init_reset <= 1'b1;

			init_start <= 1'b0;

			//Flight Arithmetic states
		end
		else begin
			state <= next_state;
			if(state == POWER_ON_RESET)begin
					init_reset <= 1'b0;
			end
			else if(INITIALIZE)		begin
					init_start <= 1'b1;
					init_reset <= 1'b0;
			end
		end
	end

	always @(*) begin
		case(state)
			POWER_ON_RESET:		next_state = INITIALIZE;
			INITIALIZE:			next_state = (~init_finished) ?
INITIALIZE :

	((error_srf05 | error_imu) ? FAIL :

	IDLE);
			IDLE:					next_state = IDLE;
		//	FLY:
			FAIL:					next_state = FAIL;
			default:			next_state = IDLE;
		endcase
	end

endmodule
```

8.2 FLIGHT_CONTROL_INITIALIZE.V

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

//

// Module Name:    flight_control_initialize

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

module flight_control_initialize(

        input clock,
        //clock

        input reset,
        //reset initialize

        input start,                                                    //start
initialization

        output done,
        //indicates initialization is done

        output reg error_srf05,

        output reg error_imu,

        output reset_1sec,                                         //reset 1 sec
divider

        input en_sec,
        //divider signal from 1 second divider


        input calibrate,                                                //if
high, calibrate motors

        output reg [31:0] throttles,                               //throttle
settings for motors

        output reg motors_start,                                        //start
motor controller

        output reg motors_reset,
        //reset motor controller

        output reg srf05_start,                                         //start
height sensor

        output reg srf05_reset,
        //reset height sensor
        input srf05_new_data,

        input [14:0] distance,
        //height reading from sensor (in microseconds)

        output reg [14:0] offset_distance,                    //offset initial
height reading (in microseconds)

        output reg imu_start,
        //start imu signal

        output reg imu_reset,
        //reset imu signal

        input imu_new_data,                                              //new
data available

        input signed [15:0] cur_roll,                         //roll angle: sensor
reading from imu

        input signed [15:0] cur_pitch,                    //pitch angle: sensor
reading from imu

        input signed [15:0] cur_yaw,                          //yaw angle:
sensor reading from imu

        input signed [15:0] cur_roll_rate,                //roll rate: sensor
reading from imu

        input signed [15:0] cur_pitch_rate,               //pitch rate: sensor
reading from imu

        input signed [15:0] cur_yaw_rate,                 //yaw rate: sensor
reading from imu

        output reg signed [15:0] offset_roll,             //roll angle: offset for
control arithmetic

        output reg signed [15:0] offset_pitch,            //pitch angle: offset for
control arithmetic

        output reg signed [15:0] offset_yaw,                  //yaw angle: offset
for control arithmetic

        output reg signed [15:0] offset_roll_rate,        //roll rate: offset for
control arithmetic

        output reg signed [15:0] offset_pitch_rate,   //pitch rate: offset for control
arithmetic

        output reg signed [15:0] offset_yaw_rate          //yaw rate: offset for
control arithmetic
```

```
);

     //Initialization Steps
     // 1. Calibrate Motors (if enabled)
     //             -Throttle High
     //             -Throttle Idle
     //
     // 2. Start Range Sensor
     //             -Check no error recorded
     //             -Averaging Filter
     //             -Record Initial Value
     // 3. Start IMU Board
     //             -Record Initial Values for 5 seconds
     //             -Averaging Filter
     //             -Record Offset Values
     // Spin Rotors for Confirmation
    reg [4:0] state, next_state;
    localparam RESET = 4'd0;
    localparam ERROR_SRF05 = 4'd1, ERROR_IMU = 4'd2;
    localparam CALIBRATE = 4'd3,
                              CALIBRATE_COMPLETE = 4'd4;
    localparam RANGE_SENSOR1 = 4'd5,
                                     RANGE_SENSOR2 = 4'd6,
                                     RANGE_SENSOR3 = 4'd7;
    localparam IMU1 = 4'd8,
                             IMU2 = 4'd9,
                             IMU3  = 4'd10;
    localparam END1 = 4'd11,
                                     END2 = 4'd12;
```

```
	localparam DONE = 4'd13;



	assign done = (state == DONE);


	reg [3:0] timer_q, timer_d;



	assign reset_1sec = (state != next_state);




	//Ring Buffer for SRF05-Range Sensor
	reg  [14:0] buffer_dist[31:0];
	reg  [19:0] sum_dist = 0;
	reg  [14:0] avg_dist = 0;
	reg [5:0] index = 0;

	integer i;
//	initial begin
//		for(i = 0; i < 32;i = i+1) begin
//			buffer_dist[i] = 15'd0;
//		end
//	end

	//Ring Buffer for
	///AVeraging Filter for IMU outputs
	//Ring Buffer
	reg signed  [15:0] buffer_roll[63:0];
	reg signed [15:0] buffer_pitch[63:0];
	reg signed [15:0] buffer_yaw[63:0];
	reg signed [15:0] buffer_roll_rate[63:0];
	reg signed [15:0] buffer_pitch_rate[63:0];
	reg signed [15:0] buffer_yaw_rate[63:0];
	reg signed [21:0] sum_roll = 0;
	reg signed [21:0] sum_pitch = 0;
	reg signed [21:0] sum_yaw = 0;
	reg signed [21:0] sum_roll_rate = 0;
	reg signed [21:0] sum_pitch_rate = 0;
	reg signed [21:0] sum_yaw_rate		= 0;
	reg signed [15:0] avg_roll;
	reg signed [15:0] avg_pitch;
	reg signed [15:0] avg_yaw;
	reg signed [15:0] avg_roll_rate;
	reg signed [15:0] avg_pitch_rate;
	reg signed [15:0] avg_yaw_rate;
	integer i2;
//	initial begin
//		for(i2 = 0; i2 < 64;i2 = i2+1) begin
//			buffer_roll[i2] = 16'sd0;
//			buffer_pitch[i2] = 16'sd0;
//			buffer_yaw[i2] = 16'sd0;
//			buffer_roll_rate[i2] = 16'sd0;
//			buffer_pitch_rate[i2] = 16'sd0;
//			buffer_yaw_rate[i2] = 16'sd0;
//		end
//	end
```

```
/////////////////////////////////////////////////////////////////////////////////////////
// State Machine
	always @(posedge clock) begin
		if(reset) begin
			state				<= 4'd0;
			error_srf05		<= 1'b0;
			error_imu				<= 1'b0;
			motors_start			<= 1'b0;
			motors_reset			<= 1'b1;
			throttles				<=	32'hFF_FF_FF_FF;
			srf05_start			<= 1'b0;
			srf05_reset			<= 1'b1;
			offset_distance	<= {15{1'b0}};
			imu_start				<= 1'b0;
			imu_reset				<= 1'b1;
			timer_q					<= 3'd0;
			sum_dist				<= 20'd0;
			avg_dist				<= 15'd0;
			index					<= 6'd0;
			sum_roll				<= 22'sd0;
			sum_pitch				<= 22'sd0;
			sum_yaw					<= 22'sd0;
			sum_roll_rate			<= 22'sd0;
			sum_pitch_rate	<= 22'sd0;
			sum_yaw_rate			<= 22'sd0;
			avg_roll						<= 16'sd0;
			avg_pitch					<= 16'sd0;
			avg_yaw						<= 16'sd0;
```

```
                avg_roll_rate                   <= 16'sd0;
                avg_pitch_rate                  <= 16'sd0;
                avg_yaw_rate                    <= 16'sd0;
        end
        else begin
                state <= next_state;
                timer_q <= (state != next_state) ? 4'd0 : timer_d;

                //Case Specific Shift Registers
                case(state)
                                RESET:begin
                                        state                           <= 4'd0;
                                        error_srf05             <= 1'b0;
                                        error_imu                       <= 1'b0;
                                        motors_start                    <= 1'b0;
                                        motors_reset                    <= 1'b1;
                                        throttles                       <=
        32'hFF_FF_FF_FF;
                                        srf05_start                     <= 1'b0;
                                        srf05_reset                     <= 1'b1;
                                        offset_distance         <= {15{1'b0}};
                                        imu_start                       <= 1'b0;
                                        imu_reset                       <= 1'b1;
                                        timer_q                                 <= 3'd0;
                                end
                                CALIBRATE: begin
                                                motors_start                    <=
1'b1;
                                                motors_reset                    <=
1'b0;
```

```
                                        throttles                  <=
    (timer_d < 2) ? 32'hFF_FF_FF_FF : 32'd0;  //High then low throttle to
calibrate
                            end
                            CALIBRATE_COMPLETE: begin
                                        motors_start               <=
1'b0;
                                        motors_reset               <=
(timer_d < 3) ? 1'd0 : 1'd1;
                                        throttles                  <=
(timer_d < 3) ? 32'h01_02_03_04 : 32'd0;   //Spin Rotors to indicate calibration
complete
                            end
                            RANGE_SENSOR1: begin
                                        srf05_reset                <=
1'b0;
                                        srf05_start                <=
1'b1;            //start range sensor
                            end
                            RANGE_SENSOR2: begin
                                if(srf05_new_data) begin
                                    sum_dist                       <=
sum_dist + (distance) - (buffer_dist[index]);
                                    buffer_dist[index]  <= distance;
                                    index                          <=
(index == 6'd31) ? 5'd0 : index + 1'b1; //overflow at 32
                                end
                                avg_dist                           <=
sum_dist>>5;
                            end
                            RANGE_SENSOR3: begin
                                    offset_distance            <=
avg_dist;     //set initial offset for height
```

```
                                //      srf05_reset                 <= 1'b1;
        //shut off sensor
                                        srf05_start                      <=
1'b0;
                            end
                            IMU1: begin
                                        imu_reset                        <=
1'b0;
                                        imu_start                        <=
1'b1;
                            end
                            IMU2: begin
                                if(imu_new_data) begin
                                        sum_roll <= sum_roll + (cur_roll) -
(buffer_roll[index]);
                                        buffer_roll[index] <= cur_roll;
                                        sum_pitch <= sum_pitch +
(cur_pitch) - (buffer_pitch[index]);
                                        buffer_pitch[index] <= cur_pitch;
                                        sum_yaw <= sum_yaw + (cur_yaw) -
(buffer_yaw[index]);
                                        buffer_yaw[index] <= cur_yaw;
                                        sum_roll_rate <= sum_roll_rate +
(cur_roll_rate) - (buffer_roll_rate[index]);
                                        buffer_roll_rate[index] <=
cur_roll_rate;
                                        sum_pitch_rate <= sum_pitch_rate +
(cur_pitch_rate) - (buffer_pitch_rate[index]);
                                        buffer_pitch_rate[index] <=
cur_pitch_rate;
                                        sum_yaw_rate <= sum_yaw_rate +
(cur_yaw_rate) - (buffer_yaw_rate[index]);
```

```
                                        buffer_yaw_rate[index] <=
cur_yaw_rate;
                                        index <= index + 1'b1;
                                end
                                avg_roll <= sum_roll/64;
                                avg_pitch <= sum_pitch/64;
                                avg_yaw <= sum_yaw/64;
                                avg_roll_rate <= sum_roll_rate/64;
                                avg_pitch_rate <= sum_pitch_rate/64;
                                avg_yaw_rate <= sum_yaw_rate/64;
                        end
                        IMU3: begin
                                        offset_roll                <=
avg_roll;       //set initial offsets for height
                                        offset_pitch               <=
avg_pitch;    //set initial offset for height
                                        offset_yaw
        <= avg_yaw;         //set initial offset for height
                                        offset_roll_rate           <=
avg_roll_rate;        //set initial offset for height
                                        offset_pitch_rate       <=
avg_pitch_rate;     //set initial offset for height
                                        offset_yaw_rate            <=
avg_yaw_rate;       //set initial offset for height
                                        //Shut off Sensor
                                        srf05_start
        <= 1'b0;


                        end
                        //Spin Rotors to Finish Intialization
                        END1: begin
                                motors_start            <= 1'b1;
```

```
                                        motors_reset            <= 1'b0;
                                        throttles               <=
32'h01_02_03_04;
                                end
                                END2: begin
                                        motors_start            <= 1'b0;
                                        motors_reset            <= (timer_d < 3)
? 1'd0 : 1'd1;
                                        throttles               <= (timer_d < 3)
? 32'h01_02_03_04 : 32'd0;
                                end
                                ERROR_SRF05:    error_srf05     <= 1'b1;
                                ERROR_IMU:              error_imu       <= 1'b1;
                        endcase
                end
        end


        always @(*)  begin
                timer_d = timer_q;


                //Case Specific Latches
                case(state)
                        RESET:                  next_state = (~start) ? RESET :
(calibrate) ? CALIBRATE : RANGE_SENSOR1;
                        CALIBRATE:      begin
                                                timer_d = (en_sec) ? timer_q +
1'b1 : timer_q;
                                                next_state = (timer_q == 4) ?
CALIBRATE_COMPLETE : CALIBRATE;         //elapsed time = 4 seconds
                        end
                        CALIBRATE_COMPLETE:             begin
```

```
                                        timer_d = (en_sec) ? timer_q +
1'b1 : timer_q;
                                        next_state = (timer_q == 3) ?
RANGE_SENSOR1: CALIBRATE_COMPLETE;   //elapsed time = 3 seconds
            end
            //SRF05 Initialize Statements
            RANGE_SENSOR1: next_state = RANGE_SENSOR2;
            RANGE_SENSOR2: begin
                timer_d = (en_sec) ? timer_q + 1'b1 : timer_q;
                next_state = (timer_q == 3) ? RANGE_SENSOR3:
RANGE_SENSOR2;          //elapsed time = 3 seconds
            end
            RANGE_SENSOR3: next_state = (avg_dist < 100) ?
ERROR_SRF05 : IMU1;
            //IMU Initialize Statements
            IMU1:                   next_state = IMU2;
            IMU2:       begin
                timer_d = (en_sec) ? timer_q + 1'b1 : timer_q;
                next_state = (timer_q == 3) ? IMU3: IMU2;
    //elapsed time = 3 seconds
            end
            //IF Roll is especially erroneous, report an error
            IMU3:               next_state = ((avg_roll > 15'sd4500 &&
avg_roll < 15'sd13500) || (avg_roll < -15'sd4500 && avg_roll > -15'sd13500)) ?

    ERROR_IMU : END1;
            //End spin props to signal success
            END1: next_state = END2;
            END2: begin
                timer_d = (en_sec) ? timer_q + 1'b1 : timer_q;
                next_state = (timer_q == 3) ? DONE: END2;   //elapsed
time = 3 seconds
```

```
                end
                ERROR_SRF05: next_state = DONE;
                ERROR_IMU: next_state = DONE;
                DONE:         next_state = DONE;
                default: next_state = RESET;
            endcase
        end


endmodule
```

8.3 Mojo_top.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//
// Module Name:    flight_control_initialize
////////////////////////////////////////////////////////////////////////////////
module flight_control_initialize(
        input clock,
        //clock
        input reset,
        //reset initialize
        input start,                                                   //start
initialization
        output done,
        //indicates initialization is done
        output reg error_srf05,
        output reg error_imu,
        output reset_1sec,                                         //reset 1 sec
divider
        input en_sec,
        //divider signal from 1 second divider

        input calibrate,                                               //if
high, calibrate motors
        output reg [31:0] throttles,                               //throttle
settings for motors
        output reg motors_start,                                       //start
motor controller
        output reg motors_reset,
        //reset motor controller
        output reg srf05_start,                                        //start
height sensor
        output reg srf05_reset,
        //reset height sensor
        input srf05_new_data,
        input [14:0] distance,
        //height reading from sensor (in microseconds)
        output reg [14:0] offset_distance,                     //offset initial
height reading (in microseconds)
        output reg imu_start,
        //start imu signal
        output reg imu_reset,
        //reset imu signal
        input imu_new_data,                                            //new
data available
        input signed [15:0] cur_roll,                      //roll angle: sensor
reading from imu
```

```
        input signed [15:0] cur_pitch,                         //pitch angle: sensor
reading from imu
        input signed [15:0] cur_yaw,                                   //yaw angle:
sensor reading from imu
        input signed [15:0] cur_roll_rate,                     //roll rate: sensor
reading from imu
        input signed [15:0] cur_pitch_rate,                    //pitch rate: sensor
reading from imu
        input signed [15:0] cur_yaw_rate,                      //yaw rate: sensor
reading from imu
        output reg signed [15:0] offset_roll,                  //roll angle: offset for
control arithmetic
        output reg signed [15:0] offset_pitch,                 //pitch angle: offset for
control arithmetic
        output reg signed [15:0] offset_yaw,                           //yaw angle: offset
for control arithmetic
        output reg signed [15:0] offset_roll_rate,             //roll rate: offset for
control arithmetic
        output reg signed [15:0] offset_pitch_rate,   //pitch rate: offset for control
arithmetic
        output reg signed [15:0] offset_yaw_rate               //yaw rate: offset for
control arithmetic
   );

        //Initialization Steps
        // 1. Calibrate Motors (if enabled)
        //             -Throttle High
        //             -Throttle Idle
        //
        // 2. Start Range Sensor
        //             -Check no error recorded
        //             -Averaging Filter
        //             -Record Initial Value
        // 3. Start IMU Board
        //             -Record Initial Values for 5 seconds
        //             -Averaging Filter
        //             -Record Offset Values
        // Spin Rotors for Confirmation
        reg [4:0] state, next_state;
        localparam RESET = 4'd0;
        localparam ERROR_SRF05 = 4'd1, ERROR_IMU = 4'd2;
        localparam CALIBRATE = 4'd3,
                           CALIBRATE_COMPLETE = 4'd4;
        localparam RANGE_SENSOR1 = 4'd5,
                                 RANGE_SENSOR2 = 4'd6,
                                 RANGE_SENSOR3 = 4'd7;
        localparam IMU1 = 4'd8,
                          IMU2 = 4'd9,
                        IMU3  = 4'd10;
        localparam END1 = 4'd11,
                            END2 = 4'd12;
        localparam DONE = 4'd13;


        assign done = (state == DONE);

        reg [3:0] timer_q, timer_d;


        assign reset_1sec = (state != next_state);



        //Ring Buffer for SRF05-Range Sensor
        reg  [14:0] buffer_dist[31:0];
        reg  [19:0] sum_dist = 0;
        reg  [14:0] avg_dist = 0;
        reg [5:0] index = 0;

        integer i;
//      initial begin
//              for(i = 0; i < 32;i = i+1) begin
//                      buffer_dist[i] = 15'd0;
//              end
//      end

        //Ring Buffer for
        ///AVeraging Filter for IMU outputs
        //Ring Buffer
        reg signed  [15:0] buffer_roll[63:0];
        reg signed [15:0] buffer_pitch[63:0];
        reg signed [15:0] buffer_yaw[63:0];
        reg signed [15:0] buffer_roll_rate[63:0];
        reg signed [15:0] buffer_pitch_rate[63:0];
        reg signed [15:0] buffer_yaw_rate[63:0];
        reg signed [21:0] sum_roll = 0;
        reg signed [21:0] sum_pitch = 0;
        reg signed [21:0] sum_yaw = 0;
        reg signed [21:0] sum_roll_rate = 0;
        reg signed [21:0] sum_pitch_rate = 0;
        reg signed [21:0] sum_yaw_rate        = 0;
        reg signed [15:0] avg_roll;
        reg signed [15:0] avg_pitch;
        reg signed [15:0] avg_yaw;
        reg signed [15:0] avg_roll_rate;
        reg signed [15:0] avg_pitch_rate;
```

```
	reg signed [15:0] avg_yaw_rate;
	integer i2;
//	initial begin
//		for(i2 = 0; i2 < 64;i2 = i2+1) begin
//			buffer_roll[i2] = 16'sd0;
//			buffer_pitch[i2] = 16'sd0;
//			buffer_yaw[i2] = 16'sd0;
//			buffer_roll_rate[i2] = 16'sd0;
//			buffer_pitch_rate[i2] = 16'sd0;
//			buffer_yaw_rate[i2] = 16'sd0;
//		end
//	end


////////////////////////////////////////////////////////////////////////////////
// State Machine
	always @(posedge clock) begin
		if(reset) begin
			state				<= 4'd0;
			error_srf05		<= 1'b0;
			error_imu			<= 1'b0;
			motors_start			<= 1'b0;
			motors_reset			<= 1'b1;
			throttles			<=	32'hFF_FF_FF_FF;
			srf05_start			<= 1'b0;
			srf05_reset			<= 1'b1;
			offset_distance	<= {15{1'b0}};
			imu_start			<= 1'b0;
			imu_reset			<= 1'b1;
			timer_q					<= 3'd0;
			sum_dist			<= 20'd0;
			avg_dist			<= 15'd0;
			index				<= 6'd0;
			sum_roll			<= 22'sd0;
			sum_pitch			<= 22'sd0;
			sum_yaw					<= 22'sd0;
			sum_roll_rate			<= 22'sd0;
			sum_pitch_rate	<= 22'sd0;
			sum_yaw_rate			<= 22'sd0;
			avg_roll						<= 16'sd0;
			avg_pitch					<= 16'sd0;
			avg_yaw						<= 16'sd0;
			avg_roll_rate				<= 16'sd0;
			avg_pitch_rate				<= 16'sd0;
			avg_yaw_rate				<= 16'sd0;
		end
		else begin
			state <= next_state;
```

```
                    timer_q <= (state != next_state) ? 4'd0 : timer_d;

                    //Case Specific Shift Registers
                    case(state)
                            RESET:begin
                                    state                   <= 4'd0;
                                    error_srf05         <= 1'b0;
                                    error_imu               <= 1'b0;
                                    motors_start            <= 1'b0;
                                    motors_reset            <= 1'b1;
                                    throttles               <=
        32'hFF_FF_FF_FF;
                                    srf05_start             <= 1'b0;
                                    srf05_reset             <= 1'b1;
                                    offset_distance     <= {15{1'b0}};
                                    imu_start               <= 1'b0;
                                    imu_reset               <= 1'b1;
                                    timer_q                         <= 3'd0;
                            end
                            CALIBRATE: begin
                                            motors_start            <=
1'b1;
                                            motors_reset            <=
1'b0;
                                            throttles               <=
        (timer_d < 2) ? 32'hFF_FF_FF_FF : 32'd0;  //High then low throttle to
calibrate
                            end
                            CALIBRATE_COMPLETE: begin
                                            motors_start            <=
1'b0;
                                            motors_reset            <=
(timer_d < 3) ? 1'd0 : 1'd1;
                                            throttles               <=
(timer_d < 3) ? 32'h01_02_03_04 : 32'd0;   //Spin Rotors to indicate calibration
complete
                            end
                            RANGE_SENSOR1: begin
                                            srf05_reset             <=
1'b0;
                                            srf05_start             <=
1'b1;               //start range sensor
                            end
                            RANGE_SENSOR2: begin
                                    if(srf05_new_data) begin
                                            sum_dist                <=
sum_dist + (distance) - (buffer_dist[index]);
                                            buffer_dist[index]  <= distance;
```

```
                                        index                           <=
(index == 6'd31) ? 5'd0 : index + 1'b1; //overflow at 32
                                    end
                                    avg_dist                            <=
sum_dist>>5;
                                end
                                RANGE_SENSOR3: begin
                                        offset_distance             <=
avg_dist;     //set initial offset for height
                                    //      srf05_reset             <= 1'b1;
            //shut off sensor
                                        srf05_start                     <=
1'b0;
                                end
                                IMU1: begin
                                        imu_reset                       <=
1'b0;
                                        imu_start                       <=
1'b1;
                                end
                                IMU2: begin
                                    if(imu_new_data) begin
                                        sum_roll <= sum_roll + (cur_roll) -
(buffer_roll[index]);
                                        buffer_roll[index] <= cur_roll;
                                        sum_pitch <= sum_pitch +
(cur_pitch) - (buffer_pitch[index]);
                                        buffer_pitch[index] <= cur_pitch;
                                        sum_yaw <= sum_yaw + (cur_yaw) -
(buffer_yaw[index]);
                                        buffer_yaw[index] <= cur_yaw;
                                        sum_roll_rate <= sum_roll_rate +
(cur_roll_rate) - (buffer_roll_rate[index]);
                                        buffer_roll_rate[index] <=
cur_roll_rate;
                                        sum_pitch_rate <= sum_pitch_rate +
(cur_pitch_rate) - (buffer_pitch_rate[index]);
                                        buffer_pitch_rate[index] <=
cur_pitch_rate;
                                        sum_yaw_rate <= sum_yaw_rate +
(cur_yaw_rate) - (buffer_yaw_rate[index]);
                                        buffer_yaw_rate[index] <=
cur_yaw_rate;
                                        index <= index + 1'b1;
                                    end
                                    avg_roll <= sum_roll/64;
                                    avg_pitch <= sum_pitch/64;
                                    avg_yaw <= sum_yaw/64;
```

```
                                        avg_roll_rate <= sum_roll_rate/64;
                                        avg_pitch_rate <= sum_pitch_rate/64;
                                        avg_yaw_rate <= sum_yaw_rate/64;
                                end
                                IMU3: begin
                                                offset_roll                     <=
avg_roll;       //set initial offsets for height
                                                offset_pitch                    <=
avg_pitch;      //set initial offset for height
                                                offset_yaw
        <= avg_yaw;             //set initial offset for height
                                                offset_roll_rate                <=
avg_roll_rate;          //set initial offset for height
                                                offset_pitch_rate               <=
avg_pitch_rate;         //set initial offset for height
                                                offset_yaw_rate                 <=
avg_yaw_rate;           //set initial offset for height
                                                //Shut off Sensor
                                                srf05_start
        <= 1'b0;

                                end
                                //Spin Rotors to Finish Intialization
                                END1: begin
                                        motors_start            <= 1'b1;
                                        motors_reset            <= 1'b0;
                                        throttles               <=
32'h01_02_03_04;
                                end
                                END2: begin
                                        motors_start            <= 1'b0;
                                        motors_reset            <= (timer_d < 3)
? 1'd0 : 1'd1;
                                        throttles               <= (timer_d < 3)
? 32'h01_02_03_04 : 32'd0;
                                end
                                ERROR_SRF05:    error_srf05     <= 1'b1;
                                ERROR_IMU:              error_imu       <= 1'b1;
                        endcase
                end
        end

        always @(*)  begin
                timer_d = timer_q;

                //Case Specific Latches
                case(state)
```

```
                    RESET:                    next_state = (~start) ? RESET :
(calibrate) ? CALIBRATE : RANGE_SENSOR1;
                    CALIBRATE:          begin
                                                  timer_d = (en_sec) ? timer_q +
1'b1 : timer_q;
                                                  next_state = (timer_q == 4) ?
CALIBRATE_COMPLETE : CALIBRATE;           //elapsed time = 4 seconds
                    end
                    CALIBRATE_COMPLETE:           begin
                                                  timer_d = (en_sec) ? timer_q +
1'b1 : timer_q;
                                                  next_state = (timer_q == 3) ?
RANGE_SENSOR1: CALIBRATE_COMPLETE;   //elapsed time = 3 seconds
                    end
                    //SRF05 Initialize Statements
                    RANGE_SENSOR1: next_state = RANGE_SENSOR2;
                    RANGE_SENSOR2: begin
                            timer_d = (en_sec) ? timer_q + 1'b1 : timer_q;
                            next_state = (timer_q == 3) ? RANGE_SENSOR3:
RANGE_SENSOR2;          //elapsed time = 3 seconds
                    end
                    RANGE_SENSOR3: next_state = (avg_dist < 100) ?
ERROR_SRF05 : IMU1;
                    //IMU Initialize Statements
                    IMU1:                     next_state = IMU2;
                    IMU2:         begin
                            timer_d = (en_sec) ? timer_q + 1'b1 : timer_q;
                            next_state = (timer_q == 3) ? IMU3: IMU2;
        //elapsed time = 3 seconds
                    end
                    //IF Roll is especially erroneous, report an error
                    IMU3:               next_state = ((avg_roll > 15'sd4500 &&
avg_roll < 15'sd13500) || (avg_roll < -15'sd4500 && avg_roll > -15'sd13500)) ?

        ERROR_IMU : END1;
                    //End spin props to signal success
                    END1: next_state = END2;
                    END2: begin
                            timer_d = (en_sec) ? timer_q + 1'b1 : timer_q;
                            next_state = (timer_q == 3) ? DONE: END2;   //elapsed
time = 3 seconds
                    end
                    ERROR_SRF05: next_state = DONE;
                    ERROR_IMU: next_state = DONE;
                    DONE:         next_state = DONE;
                    default: next_state = RESET;
            endcase
        end
```

```
endmodule
```

8.4 Mojo.UCF

```
#Created by Constraints Editor (xc6slx9-tqg144-3) - 2012/11/05
NET "clk" TNM_NET = clk;
TIMESPEC TS_clk = PERIOD "clk" 50 MHz HIGH 50%;

NET "clk" LOC = P56 | IOSTANDARD = LVTTL;
NET "rst_n" LOC = P38 | IOSTANDARD = LVTTL;

NET "cclk" LOC = P70 | IOSTANDARD = LVTTL;

NET "led<0>" LOC = P134 | IOSTANDARD = LVTTL;
NET "led<1>" LOC = P133 | IOSTANDARD = LVTTL;
NET "led<2>" LOC = P132 | IOSTANDARD = LVTTL;
NET "led<3>" LOC = P131 | IOSTANDARD = LVTTL;
NET "led<4>" LOC = P127 | IOSTANDARD = LVTTL;
NET "led<5>" LOC = P126 | IOSTANDARD = LVTTL;
NET "led<6>" LOC = P124 | IOSTANDARD = LVTTL;
NET "led<7>" LOC = P123 | IOSTANDARD = LVTTL;

NET "spi_mosi" LOC = P44 | IOSTANDARD = LVTTL;
NET "spi_miso" LOC = P45 | IOSTANDARD = LVTTL;
NET "spi_ss" LOC = P48 | IOSTANDARD = LVTTL;
NET "spi_sck" LOC = P43 | IOSTANDARD = LVTTL;
NET "avr_flags<0>" LOC = P46 | IOSTANDARD = LVTTL;
NET "avr_flags<1>" LOC = P61 | IOSTANDARD = LVTTL;
NET "avr_flags<2>" LOC = P62 | IOSTANDARD = LVTTL;
NET "avr_flags<3>" LOC = P65 | IOSTANDARD = LVTTL;

NET "avr_tx" LOC = P55 | IOSTANDARD = LVTTL;
NET "avr_rx" LOC = P59 | IOSTANDARD = LVTTL;
NET "avr_rx_busy" LOC = P39 | IOSTANDARD = LVTTL;



# JB IMU

NET "imu[0]" LOC = P92;
NET "imu[1]" LOC = P94;
NET "imu[2]" LOC = P97;
NET "imu[3]" LOC = P99;

# SRF05
```

```
NET "srf05[0]" LOC = P8 | IOSTANDARD = LVTTL;
NET "srf05[1]" LOC = P7 | IOSTANDARD = LVTTL;

# Motors

NET "motor<0>" LOC = P79 | IOSTANDARD = LVTTL;
NET "motor<1>" LOC = P57 | IOSTANDARD = LVTTL;
NET "motor<2>" LOC = P88 | IOSTANDARD = LVTTL;
NET "motor<3>" LOC = P33 | IOSTANDARD = LVTTL;
```

8.5 MOTOR_CONTROLLER.V

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Motor Control Module
//
//      Acts as a black box to flight control module
//                      -- Two States On or Off
//                      -- Flight Control Module Must Tell it To Stop, or go IDLE
//////////////////////////////////////////////////////////////////////////////////
module motor_controller(
        input clock,
        input start,
        input idle,
        input reset,
        input [7:0] throttle1,
        input [7:0] throttle2,
        input [7:0] throttle3,
        input [7:0] throttle4,
        output [3:0] motor_signals
 );


/////////////////////////////////////////////////////////////////////////////////////////
////////
// Motor Controller State Machine
        localparam IDLE = 1'b0;
        localparam ON = 1'b1;
        reg state, next_state;

        wire pwm_reset;
        reg pwm_on_q, pwm_on_d;
        assign pwm_reset = ~pwm_on_q;

        always @(posedge clock) begin
                if(reset) begin
```

```
				state <= IDLE;
				pwm_on_q <= 1'b0;
			end
			else	begin
				state <= next_state;
				pwm_on_q <= pwm_on_d;
			end
		end
		always @(*) begin
			case(state)
				IDLE: begin
								next_state = (start) ? ON : IDLE;
								pwm_on_d = 1'b0;
				end
				ON: begin
								next_state = ON;
								pwm_on_d = 1'b1;
				end
			endcase
		end

////////////////////////////////////////////////////////////////////////////////////////
//////////////
//		Throttle 2 PWM Lookup Tables
//

		wire [11:0] pwm_signal[3:0];
		wire [47:0] compare;

		throttle2pwm throttleLUT1(
   .clock(clock),
	.reset(pwm_reset),
	.idle(idle),
   .throttle_setting(throttle1),
   .pwm_signal_time(compare[47:36])
   );

	throttle2pwm throttleLUT2(
   .clock(clock),
	.reset(pwm_reset),
	.idle(idle),
   .throttle_setting(throttle2),
   .pwm_signal_time(compare[35:24])
   );

	throttle2pwm throttleLUT3(
   .clock(clock),
	.reset(pwm_reset),
        .idle(idle),
    .throttle_setting(throttle3),
    .pwm_signal_time(compare[23:12])
    );

        throttle2pwm throttleLUT4(
    .clock(clock),
        .reset(pwm_reset),
        .idle(idle),
    .throttle_setting(throttle4),
    .pwm_signal_time(compare[11:0])
    );



//////////////////////////////////////////////////////////////////////////////////
//////////
// PWM Modules
//////////////////////////////////////////////////////////////////////////////////
//////////

        wire pwm1,pwm2,pwm3,pwm4;
        assign motor_signals = {pwm1,pwm2,pwm3,pwm4};
        //output of motor controller

        pwm motor1 //400 hz = 2500 us period
        (
    .clock(clock),          //clock 50 Mhz
    .reset(pwm_reset),                          //reset wire
    .compare(compare[47:36]),  //compare value in microseconds
    .pwm(pwm1)              //pwm signal out
    );

         pwm motor2 //400 hz = 2500 us period
        (
    .clock(clock),          //clock 50 Mhz
    .reset(pwm_reset),                          //reset wire
    .compare(compare[35:24]),  //compare value in microseconds
    .pwm(pwm2)              //pwm signal out
    );

         pwm motor3 //400 hz = 2500 us period
        (
    .clock(clock),          //clock 50 Mhz
    .reset(pwm_reset),                          //reset wire
    .compare(compare[23:12]),  //compare value in microseconds
    .pwm(pwm3)              //pwm signal out
    );
```

```
     pwm motor4 //400 hz = 2500 us period
    (
  .clock(clock),           //clock 50 Mhz
  .reset(pwm_reset),                           //reset wire
  .compare(compare[11:0]),   //compare value in microseconds
  .pwm(pwm4)               //pwm signal out
  );

endmodule
```

8.6 PWM.V

```
`timescale 1ns / 1ps

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

// Engineer:

//

// Create Date:    17:06:44 11/18/2014

// Design Name: Digital Controller for VTOL UAV

// Module Name:    pwm

// Project Name:

// Target Devices: Mojov V3 Spartan 6 xclx9

// Tool versions:

// Description: Sends a PWM pulse up to the compare value on a 400 hz refresh

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//                              Based off code examples from
embbededmicro.com/tutorials/mojo/pulse-width-modulation

//

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

module pwm #(parameter CTR_LEN = 12,
                        parameter REFRESH_RATE_US = 12'd2499) //400 hz =
2500 us period
        (
   input clock,             //clock 50 Mhz
   input reset,                          //reset wire
   input [CTR_LEN-1:0] compare,     //compare value in microseconds
   output pwm               //pwm signal out
   );


        wire one_us;
        divider_1us one_us_div(.clock(clock),.reset(reset),.one_us_enable(one_us));


        reg pwm_d, pwm_q;                                       //pulse out
registers
        reg [CTR_LEN-1:0] ctr_d, ctr_q;               //counters for pulse


        assign pwm = pwm_q & ~reset;




        always @(*) begin
                ctr_d = (one_us) ? ctr_q + 1'b1 : ctr_q;  //if one ms passed,
increment q register
                pwm_d = (compare > ctr_q) ? 1'b1 : 1'b0;  //set pulse high
        end


        always @(posedge clock) begin
                if(reset) ctr_q <= 1'b0;
                else ctr_q <= (ctr_d == REFRESH_RATE_US) ? 1'b0 :  ctr_d;
                //if exceed refresh rate, return to 0. Otherwise except register value.
			//shift register value
			pwm_q <= pwm_d;
		end
endmodule
```

8.7 THROTTLE2PWM.V

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: 
// Engineer: 
// 
// Create Date:    18:01:43 11/18/2014 
// Design Name: 
// Module Name:    throttle2pwm 
// Project Name: 
// Target Devices: 
// Tool versions: 
// Description: 
//
// Dependencies: 
//
// Revision: 
// Revision 0.01 - File Created
// Additional Comments: 
//
//////////////////////////////////////////////////////////////////////////////////
module throttle2pwm #(parameter PWM_LEN = 12)
	(
input clock,
    input reset,
    input idle,
input [7:0] throttle_setting,
output [11:0] pwm_signal_time
);

    reg [11:0] signal_time;
    assign pwm_signal_time = (idle || reset) ? 12'd900 : signal_time;

    always @(posedge clock) begin
        case(throttle_setting)
            8'd0:                   signal_time =   12'd1064;
            8'd1:                   signal_time =   12'd1067;
            8'd2:                   signal_time =   12'd1070;
            8'd3:                   signal_time =   12'd1073;
            8'd4:                   signal_time =   12'd1077;
            8'd5:                   signal_time =   12'd1080;
            8'd6:                   signal_time =   12'd1083;
            8'd7:                   signal_time =   12'd1086;
            8'd8:                   signal_time =   12'd1089;
            8'd9:                   signal_time =   12'd1092;
            8'd10:              signal_time =   12'd1095;
            8'd11:              signal_time =   12'd1099;
            8'd12:              signal_time =   12'd1102;
            8'd13:              signal_time =   12'd1105;
            8'd14:              signal_time =   12'd1108;
            8'd15:              signal_time =   12'd1111;
            8'd16:              signal_time =   12'd1114;
        8'd17:          signal_time =   12'd1117;
        8'd18:          signal_time =   12'd1120;
        8'd19:          signal_time =   12'd1124;
        8'd20:          signal_time =   12'd1127;
        8'd21:          signal_time =   12'd1130;
        8'd22:          signal_time =   12'd1133;
        8'd23:          signal_time =   12'd1136;
        8'd24:          signal_time =   12'd1139;
        8'd25:          signal_time =   12'd1142;
        8'd26:          signal_time =   12'd1146;
        8'd27:          signal_time =   12'd1149;
        8'd28:          signal_time =   12'd1152;
        8'd29:          signal_time =   12'd1155;
        8'd30:          signal_time =   12'd1158;
        8'd31:          signal_time =   12'd1161;
        8'd32:          signal_time =   12'd1164;
        8'd33:          signal_time =   12'd1168;
        8'd34:          signal_time =   12'd1171;
        8'd35:          signal_time =   12'd1174;
        8'd36:          signal_time =   12'd1177;
        8'd37:          signal_time =   12'd1180;
        8'd38:          signal_time =   12'd1183;
        8'd39:          signal_time =   12'd1186;
        8'd40:          signal_time =   12'd1189;
        8'd41:          signal_time =   12'd1193;
        8'd42:          signal_time =   12'd1196;
        8'd43:          signal_time =   12'd1199;
        8'd44:          signal_time =   12'd1202;
        8'd45:          signal_time =   12'd1205;
```

```
8'd46:          signal_time =   12'd1208;
8'd47:          signal_time =   12'd1211;
8'd48:          signal_time =   12'd1215;
8'd49:          signal_time =   12'd1218;
8'd50:          signal_time =   12'd1221;
8'd51:          signal_time =   12'd1224;
8'd52:          signal_time =   12'd1227;
8'd53:          signal_time =   12'd1230;
8'd54:          signal_time =   12'd1233;
8'd55:          signal_time =   12'd1237;
8'd56:          signal_time =   12'd1240;
8'd57:          signal_time =   12'd1243;
8'd58:          signal_time =   12'd1246;
8'd59:          signal_time =   12'd1249;
8'd60:          signal_time =   12'd1252;
8'd61:          signal_time =   12'd1255;
8'd62:          signal_time =   12'd1259;
8'd63:          signal_time =   12'd1262;
8'd64:          signal_time =   12'd1265;
8'd65:          signal_time =   12'd1268;
8'd66:          signal_time =   12'd1271;
8'd67:          signal_time =   12'd1274;
8'd68:          signal_time =   12'd1277;
8'd69:          signal_time =   12'd1280;
8'd70:          signal_time =   12'd1284;
8'd71:          signal_time =   12'd1287;
8'd72:          signal_time =   12'd1290;
8'd73:          signal_time =   12'd1293;
8'd74:          signal_time =   12'd1296;
```

```
8'd75:          signal_time =   12'd1299;
8'd76:          signal_time =   12'd1302;
8'd77:          signal_time =   12'd1306;
8'd78:          signal_time =   12'd1309;
8'd79:          signal_time =   12'd1312;
8'd80:          signal_time =   12'd1315;
8'd81:          signal_time =   12'd1318;
8'd82:          signal_time =   12'd1321;
8'd83:          signal_time =   12'd1324;
8'd84:          signal_time =   12'd1328;
8'd85:          signal_time =   12'd1331;
8'd86:          signal_time =   12'd1334;
8'd87:          signal_time =   12'd1337;
8'd88:          signal_time =   12'd1340;
8'd89:          signal_time =   12'd1343;
8'd90:          signal_time =   12'd1346;
8'd91:          signal_time =   12'd1349;
8'd92:          signal_time =   12'd1353;
8'd93:          signal_time =   12'd1356;
8'd94:          signal_time =   12'd1359;
8'd95:          signal_time =   12'd1362;
8'd96:          signal_time =   12'd1365;
8'd97:          signal_time =   12'd1368;
8'd98:          signal_time =   12'd1371;
8'd99:          signal_time =   12'd1375;
8'd100:              signal_time =   12'd1378;
8'd101:              signal_time =   12'd1381;
8'd102:              signal_time =   12'd1384;
8'd103:              signal_time =   12'd1387;
```

```
8'd104:        signal_time =   12'd1390;
8'd105:        signal_time =   12'd1393;
8'd106:        signal_time =   12'd1397;
8'd107:        signal_time =   12'd1400;
8'd108:        signal_time =   12'd1403;
8'd109:        signal_time =   12'd1406;
8'd110:        signal_time =   12'd1409;
8'd111:        signal_time =   12'd1412;
8'd112:        signal_time =   12'd1415;
8'd113:        signal_time =   12'd1419;
8'd114:        signal_time =   12'd1422;
8'd115:        signal_time =   12'd1425;
8'd116:        signal_time =   12'd1428;
8'd117:        signal_time =   12'd1431;
8'd118:        signal_time =   12'd1434;
8'd119:        signal_time =   12'd1437;
8'd120:        signal_time =   12'd1440;
8'd121:        signal_time =   12'd1444;
8'd122:        signal_time =   12'd1447;
8'd123:        signal_time =   12'd1450;
8'd124:        signal_time =   12'd1453;
8'd125:        signal_time =   12'd1456;
8'd126:        signal_time =   12'd1459;
8'd127:        signal_time =   12'd1462;
8'd128:        signal_time =   12'd1466;
8'd129:        signal_time =   12'd1469;
8'd130:        signal_time =   12'd1472;
8'd131:        signal_time =   12'd1475;
8'd132:        signal_time =   12'd1478;
```

```
8'd133:        signal_time =  12'd1481;
8'd134:        signal_time =  12'd1484;
8'd135:        signal_time =  12'd1488;
8'd136:        signal_time =  12'd1491;
8'd137:        signal_time =  12'd1494;
8'd138:        signal_time =  12'd1497;
8'd139:        signal_time =  12'd1500;
8'd140:        signal_time =  12'd1503;
8'd141:        signal_time =  12'd1506;
8'd142:        signal_time =  12'd1509;
8'd143:        signal_time =  12'd1513;
8'd144:        signal_time =  12'd1516;
8'd145:        signal_time =  12'd1519;
8'd146:        signal_time =  12'd1522;
8'd147:        signal_time =  12'd1525;
8'd148:        signal_time =  12'd1528;
8'd149:        signal_time =  12'd1531;
8'd150:        signal_time =  12'd1535;
8'd151:        signal_time =  12'd1538;
8'd152:        signal_time =  12'd1541;
8'd153:        signal_time =  12'd1544;
8'd154:        signal_time =  12'd1547;
8'd155:        signal_time =  12'd1550;
8'd156:        signal_time =  12'd1553;
8'd157:        signal_time =  12'd1557;
8'd158:        signal_time =  12'd1560;
8'd159:        signal_time =  12'd1563;
8'd160:        signal_time =  12'd1566;
8'd161:        signal_time =  12'd1569;
```

```
8'd162:    signal_time =   12'd1572;
8'd163:    signal_time =   12'd1575;
8'd164:    signal_time =   12'd1579;
8'd165:    signal_time =   12'd1582;
8'd166:    signal_time =   12'd1585;
8'd167:    signal_time =   12'd1588;
8'd168:    signal_time =   12'd1591;
8'd169:    signal_time =   12'd1594;
8'd170:    signal_time =   12'd1597;
8'd171:    signal_time =   12'd1600;
8'd172:    signal_time =   12'd1604;
8'd173:    signal_time =   12'd1607;
8'd174:    signal_time =   12'd1610;
8'd175:    signal_time =   12'd1613;
8'd176:    signal_time =   12'd1616;
8'd177:    signal_time =   12'd1619;
8'd178:    signal_time =   12'd1622;
8'd179:    signal_time =   12'd1626;
8'd180:    signal_time =   12'd1629;
8'd181:    signal_time =   12'd1632;
8'd182:    signal_time =   12'd1635;
8'd183:    signal_time =   12'd1638;
8'd184:    signal_time =   12'd1641;
8'd185:    signal_time =   12'd1644;
8'd186:    signal_time =   12'd1648;
8'd187:    signal_time =   12'd1651;
8'd188:    signal_time =   12'd1654;
8'd189:    signal_time =   12'd1657;
8'd190:    signal_time =   12'd1660;
```

```
8'd191:          signal_time =   12'd1663;
8'd192:          signal_time =   12'd1666;
8'd193:          signal_time =   12'd1669;
8'd194:          signal_time =   12'd1673;
8'd195:          signal_time =   12'd1676;
8'd196:          signal_time =   12'd1679;
8'd197:          signal_time =   12'd1682;
8'd198:          signal_time =   12'd1685;
8'd199:          signal_time =   12'd1688;
8'd200:          signal_time =   12'd1691;
8'd201:          signal_time =   12'd1695;
8'd202:          signal_time =   12'd1698;
8'd203:          signal_time =   12'd1701;
8'd204:          signal_time =   12'd1704;
8'd205:          signal_time =   12'd1707;
8'd206:          signal_time =   12'd1710;
8'd207:          signal_time =   12'd1713;
8'd208:          signal_time =   12'd1717;
8'd209:          signal_time =   12'd1720;
8'd210:          signal_time =   12'd1723;
8'd211:          signal_time =   12'd1726;
8'd212:          signal_time =   12'd1729;
8'd213:          signal_time =   12'd1732;
8'd214:          signal_time =   12'd1735;
8'd215:          signal_time =   12'd1739;
8'd216:          signal_time =   12'd1742;
8'd217:          signal_time =   12'd1745;
8'd218:          signal_time =   12'd1748;
8'd219:          signal_time =   12'd1751;
```

```
8'd220:            signal_time =  12'd1754;
8'd221:            signal_time =  12'd1757;
8'd222:            signal_time =  12'd1760;
8'd223:            signal_time =  12'd1764;
8'd224:            signal_time =  12'd1767;
8'd225:            signal_time =  12'd1770;
8'd226:            signal_time =  12'd1773;
8'd227:            signal_time =  12'd1776;
8'd228:            signal_time =  12'd1779;
8'd229:            signal_time =  12'd1782;
8'd230:            signal_time =  12'd1786;
8'd231:            signal_time =  12'd1789;
8'd232:            signal_time =  12'd1792;
8'd233:            signal_time =  12'd1795;
8'd234:            signal_time =  12'd1798;
8'd235:            signal_time =  12'd1801;
8'd236:            signal_time =  12'd1804;
8'd237:            signal_time =  12'd1808;
8'd238:            signal_time =  12'd1811;
8'd239:            signal_time =  12'd1814;
8'd240:            signal_time =  12'd1817;
8'd241:            signal_time =  12'd1820;
8'd242:            signal_time =  12'd1823;
8'd243:            signal_time =  12'd1826;
8'd244:            signal_time =  12'd1829;
8'd245:            signal_time =  12'd1833;
8'd246:            signal_time =  12'd1836;
8'd247:            signal_time =  12'd1839;
8'd248:            signal_time =  12'd1842;
```

```
                    8'd249:                 signal_time =   12'd1845;
                    8'd250:                 signal_time =   12'd1848;
                    8'd251:                 signal_time =   12'd1851;
                    8'd252:                 signal_time =   12'd1855;
                    8'd253:                 signal_time =   12'd1858;
                    8'd254:                 signal_time =   12'd1861;
                    8'd255:                 signal_time =   12'd1864;
                endcase
        end
endmodule
```

8.8 JB_IMU.V

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: 
// Engineer: 
// 
// Create Date:    18:01:43 11/18/2014 
// Design Name: 
// Module Name:    throttle2pwm 
// Project Name: 
// Target Devices: 
// Tool versions: 
// Description: 
//
// Dependencies: 
//
// Revision: 
// Revision 0.01 - File Created
// Additional Comments: 
//
//////////////////////////////////////////////////////////////////////////////////
module throttle2pwm #(parameter PWM_LEN = 12)
        (
   input clock,
         input reset,
         input idle,
   input [7:0] throttle_setting,
   output [11:0] pwm_signal_time
   );
```

```
reg [11:0] signal_time;
assign pwm_signal_time = (idle || reset) ? 12'd900 : signal_time;

always @(posedge clock) begin
        case(throttle_setting)
                8'd0:                   signal_time =   12'd1064;
                8'd1:                   signal_time =   12'd1067;
                8'd2:                   signal_time =   12'd1070;
                8'd3:                   signal_time =   12'd1073;
                8'd4:                   signal_time =   12'd1077;
                8'd5:                   signal_time =   12'd1080;
                8'd6:                   signal_time =   12'd1083;
                8'd7:                   signal_time =   12'd1086;
                8'd8:                   signal_time =   12'd1089;
                8'd9:                   signal_time =   12'd1092;
                8'd10:          signal_time =   12'd1095;
                8'd11:          signal_time =   12'd1099;
                8'd12:          signal_time =   12'd1102;
                8'd13:          signal_time =   12'd1105;
                8'd14:          signal_time =   12'd1108;
                8'd15:          signal_time =   12'd1111;
                8'd16:          signal_time =   12'd1114;
                8'd17:          signal_time =   12'd1117;
                8'd18:          signal_time =   12'd1120;
                8'd19:          signal_time =   12'd1124;
                8'd20:          signal_time =   12'd1127;
                8'd21:          signal_time =   12'd1130;
                8'd22:          signal_time =   12'd1133;
                8'd23:          signal_time =   12'd1136;
                8'd24:          signal_time =   12'd1139;
                8'd25:          signal_time =   12'd1142;
                8'd26:          signal_time =   12'd1146;
                8'd27:          signal_time =   12'd1149;
                8'd28:          signal_time =   12'd1152;
                8'd29:          signal_time =   12'd1155;
                8'd30:          signal_time =   12'd1158;
                8'd31:          signal_time =   12'd1161;
                8'd32:          signal_time =   12'd1164;
                8'd33:          signal_time =   12'd1168;
                8'd34:          signal_time =   12'd1171;
                8'd35:          signal_time =   12'd1174;
                8'd36:          signal_time =   12'd1177;
                8'd37:          signal_time =   12'd1180;
                8'd38:          signal_time =   12'd1183;
                8'd39:          signal_time =   12'd1186;
                8'd40:          signal_time =   12'd1189;
                8'd41:          signal_time =   12'd1193;
                8'd42:          signal_time =   12'd1196;
```

```
8'd43:          signal_time =   12'd1199;
8'd44:          signal_time =   12'd1202;
8'd45:          signal_time =   12'd1205;
8'd46:          signal_time =   12'd1208;
8'd47:          signal_time =   12'd1211;
8'd48:          signal_time =   12'd1215;
8'd49:          signal_time =   12'd1218;
8'd50:          signal_time =   12'd1221;
8'd51:          signal_time =   12'd1224;
8'd52:          signal_time =   12'd1227;
8'd53:          signal_time =   12'd1230;
8'd54:          signal_time =   12'd1233;
8'd55:          signal_time =   12'd1237;
8'd56:          signal_time =   12'd1240;
8'd57:          signal_time =   12'd1243;
8'd58:          signal_time =   12'd1246;
8'd59:          signal_time =   12'd1249;
8'd60:          signal_time =   12'd1252;
8'd61:          signal_time =   12'd1255;
8'd62:          signal_time =   12'd1259;
8'd63:          signal_time =   12'd1262;
8'd64:          signal_time =   12'd1265;
8'd65:          signal_time =   12'd1268;
8'd66:          signal_time =   12'd1271;
8'd67:          signal_time =   12'd1274;
8'd68:          signal_time =   12'd1277;
8'd69:          signal_time =   12'd1280;
8'd70:          signal_time =   12'd1284;
8'd71:          signal_time =   12'd1287;
8'd72:          signal_time =   12'd1290;
8'd73:          signal_time =   12'd1293;
8'd74:          signal_time =   12'd1296;
8'd75:          signal_time =   12'd1299;
8'd76:          signal_time =   12'd1302;
8'd77:          signal_time =   12'd1306;
8'd78:          signal_time =   12'd1309;
8'd79:          signal_time =   12'd1312;
8'd80:          signal_time =   12'd1315;
8'd81:          signal_time =   12'd1318;
8'd82:          signal_time =   12'd1321;
8'd83:          signal_time =   12'd1324;
8'd84:          signal_time =   12'd1328;
8'd85:          signal_time =   12'd1331;
8'd86:          signal_time =   12'd1334;
8'd87:          signal_time =   12'd1337;
8'd88:          signal_time =   12'd1340;
8'd89:          signal_time =   12'd1343;
8'd90:          signal_time =   12'd1346;
```

```
8'd91:          signal_time =   12'd1349;
8'd92:          signal_time =   12'd1353;
8'd93:          signal_time =   12'd1356;
8'd94:          signal_time =   12'd1359;
8'd95:          signal_time =   12'd1362;
8'd96:          signal_time =   12'd1365;
8'd97:          signal_time =   12'd1368;
8'd98:          signal_time =   12'd1371;
8'd99:          signal_time =   12'd1375;
8'd100:                 signal_time =   12'd1378;
8'd101:                 signal_time =   12'd1381;
8'd102:                 signal_time =   12'd1384;
8'd103:                 signal_time =   12'd1387;
8'd104:                 signal_time =   12'd1390;
8'd105:                 signal_time =   12'd1393;
8'd106:                 signal_time =   12'd1397;
8'd107:                 signal_time =   12'd1400;
8'd108:                 signal_time =   12'd1403;
8'd109:                 signal_time =   12'd1406;
8'd110:                 signal_time =   12'd1409;
8'd111:                 signal_time =   12'd1412;
8'd112:                 signal_time =   12'd1415;
8'd113:                 signal_time =   12'd1419;
8'd114:                 signal_time =   12'd1422;
8'd115:                 signal_time =   12'd1425;
8'd116:                 signal_time =   12'd1428;
8'd117:                 signal_time =   12'd1431;
8'd118:                 signal_time =   12'd1434;
8'd119:                 signal_time =   12'd1437;
8'd120:                 signal_time =   12'd1440;
8'd121:                 signal_time =   12'd1444;
8'd122:                 signal_time =   12'd1447;
8'd123:                 signal_time =   12'd1450;
8'd124:                 signal_time =   12'd1453;
8'd125:                 signal_time =   12'd1456;
8'd126:                 signal_time =   12'd1459;
8'd127:                 signal_time =   12'd1462;
8'd128:                 signal_time =   12'd1466;
8'd129:                 signal_time =   12'd1469;
8'd130:                 signal_time =   12'd1472;
8'd131:                 signal_time =   12'd1475;
8'd132:                 signal_time =   12'd1478;
8'd133:                 signal_time =   12'd1481;
8'd134:                 signal_time =   12'd1484;
8'd135:                 signal_time =   12'd1488;
8'd136:                 signal_time =   12'd1491;
8'd137:                 signal_time =   12'd1494;
8'd138:                 signal_time =   12'd1497;
```

```
8'd139:         signal_time =  12'd1500;
8'd140:         signal_time =  12'd1503;
8'd141:         signal_time =  12'd1506;
8'd142:         signal_time =  12'd1509;
8'd143:         signal_time =  12'd1513;
8'd144:         signal_time =  12'd1516;
8'd145:         signal_time =  12'd1519;
8'd146:         signal_time =  12'd1522;
8'd147:         signal_time =  12'd1525;
8'd148:         signal_time =  12'd1528;
8'd149:         signal_time =  12'd1531;
8'd150:         signal_time =  12'd1535;
8'd151:         signal_time =  12'd1538;
8'd152:         signal_time =  12'd1541;
8'd153:         signal_time =  12'd1544;
8'd154:         signal_time =  12'd1547;
8'd155:         signal_time =  12'd1550;
8'd156:         signal_time =  12'd1553;
8'd157:         signal_time =  12'd1557;
8'd158:         signal_time =  12'd1560;
8'd159:         signal_time =  12'd1563;
8'd160:         signal_time =  12'd1566;
8'd161:         signal_time =  12'd1569;
8'd162:         signal_time =  12'd1572;
8'd163:         signal_time =  12'd1575;
8'd164:         signal_time =  12'd1579;
8'd165:         signal_time =  12'd1582;
8'd166:         signal_time =  12'd1585;
8'd167:         signal_time =  12'd1588;
8'd168:         signal_time =  12'd1591;
8'd169:         signal_time =  12'd1594;
8'd170:         signal_time =  12'd1597;
8'd171:         signal_time =  12'd1600;
8'd172:         signal_time =  12'd1604;
8'd173:         signal_time =  12'd1607;
8'd174:         signal_time =  12'd1610;
8'd175:         signal_time =  12'd1613;
8'd176:         signal_time =  12'd1616;
8'd177:         signal_time =  12'd1619;
8'd178:         signal_time =  12'd1622;
8'd179:         signal_time =  12'd1626;
8'd180:         signal_time =  12'd1629;
8'd181:         signal_time =  12'd1632;
8'd182:         signal_time =  12'd1635;
8'd183:         signal_time =  12'd1638;
8'd184:         signal_time =  12'd1641;
8'd185:         signal_time =  12'd1644;
8'd186:         signal_time =  12'd1648;
```

```
8'd187:          signal_time =   12'd1651;
8'd188:          signal_time =   12'd1654;
8'd189:          signal_time =   12'd1657;
8'd190:          signal_time =   12'd1660;
8'd191:          signal_time =   12'd1663;
8'd192:          signal_time =   12'd1666;
8'd193:          signal_time =   12'd1669;
8'd194:          signal_time =   12'd1673;
8'd195:          signal_time =   12'd1676;
8'd196:          signal_time =   12'd1679;
8'd197:          signal_time =   12'd1682;
8'd198:          signal_time =   12'd1685;
8'd199:          signal_time =   12'd1688;
8'd200:          signal_time =   12'd1691;
8'd201:          signal_time =   12'd1695;
8'd202:          signal_time =   12'd1698;
8'd203:          signal_time =   12'd1701;
8'd204:          signal_time =   12'd1704;
8'd205:          signal_time =   12'd1707;
8'd206:          signal_time =   12'd1710;
8'd207:          signal_time =   12'd1713;
8'd208:          signal_time =   12'd1717;
8'd209:          signal_time =   12'd1720;
8'd210:          signal_time =   12'd1723;
8'd211:          signal_time =   12'd1726;
8'd212:          signal_time =   12'd1729;
8'd213:          signal_time =   12'd1732;
8'd214:          signal_time =   12'd1735;
8'd215:          signal_time =   12'd1739;
8'd216:          signal_time =   12'd1742;
8'd217:          signal_time =   12'd1745;
8'd218:          signal_time =   12'd1748;
8'd219:          signal_time =   12'd1751;
8'd220:          signal_time =   12'd1754;
8'd221:          signal_time =   12'd1757;
8'd222:          signal_time =   12'd1760;
8'd223:          signal_time =   12'd1764;
8'd224:          signal_time =   12'd1767;
8'd225:          signal_time =   12'd1770;
8'd226:          signal_time =   12'd1773;
8'd227:          signal_time =   12'd1776;
8'd228:          signal_time =   12'd1779;
8'd229:          signal_time =   12'd1782;
8'd230:          signal_time =   12'd1786;
8'd231:          signal_time =   12'd1789;
8'd232:          signal_time =   12'd1792;
8'd233:          signal_time =   12'd1795;
8'd234:          signal_time =   12'd1798;
```

```
                    8'd235:                     signal_time =   12'd1801;
                    8'd236:                     signal_time =   12'd1804;
                    8'd237:                     signal_time =   12'd1808;
                    8'd238:                     signal_time =   12'd1811;
                    8'd239:                     signal_time =   12'd1814;
                    8'd240:                     signal_time =   12'd1817;
                    8'd241:                     signal_time =   12'd1820;
                    8'd242:                     signal_time =   12'd1823;
                    8'd243:                     signal_time =   12'd1826;
                    8'd244:                     signal_time =   12'd1829;
                    8'd245:                     signal_time =   12'd1833;
                    8'd246:                     signal_time =   12'd1836;
                    8'd247:                     signal_time =   12'd1839;
                    8'd248:                     signal_time =   12'd1842;
                    8'd249:                     signal_time =   12'd1845;
                    8'd250:                     signal_time =   12'd1848;
                    8'd251:                     signal_time =   12'd1851;
                    8'd252:                     signal_time =   12'd1855;
                    8'd253:                     signal_time =   12'd1858;
                    8'd254:                     signal_time =   12'd1861;
                    8'd255:                     signal_time =   12'd1864;
            endcase
        end
endmodule
```

8.9 SPI.V

```
`timescale 1ns / 1ps

//Gregory Kravit
// gkravit@mit.edu
//Adapted from embeddedmicro.com
/////////////////////////////////////////////////////////////////////////////////
// SPI Master:  Mode 0 CKE = 1; CKP = 0;
//Sources Used:
// http://www.rosseeld.be/DRO/PIC/SPI_Timing.htm
// http://www.elecdude.com/2013/09/spi-master-slave-verilog-code-spi.html
// dsPIC (JBimu Microcontroller) p. 372
//
//Mode 0
// Parameters:
//          SPI_CLK_DIV:
//                  -Divides System Clock for relevant SCK speed
//                  SPI_CLK_DIV = C:  SCK_FREQ = CLK_FREQ/2^C
//                          C = 2: SCK_FREQ = CLK_FREQ /4
//                                 3: SCK_FREQ = CLK_FREQ /8
//                                 4: SCK_FREQ = CLK_FREQ /16
//                                    5: SCK_FREQ = CLK_FREQ /32

//              SPI_BUS_WIDTH:
//                      -Number of Bits In Communication expected to transmit and
receive
//                      SPI_BUS_WIDTH = W:

////////////////////////////////////////////////////////////////////////////////
module spi #(parameter SPI_CLK_DIV = 4, parameter SPI_BUS_WIDTH = 8)(
       input clk,
       input rst,
       input miso,
       output mosi,
       output sck,
       input start,
       input [SPI_BUS_WIDTH-1:0] data_in,
       output[SPI_BUS_WIDTH-1:0] data_out,
                  output ss,
       output busy,
       output new_data
   );
        //num bits needed to count to bus width
        localparam NUM_BITS = log2(SPI_BUS_WIDTH);

        localparam STATE_SIZE = 2;
        localparam IDLE = 2'd0,
                        WAIT_HALF = 2'd1,
                        TRANSFER = 2'd2;

        reg [STATE_SIZE-1:0] state_d, state_q;

        reg [SPI_BUS_WIDTH-1:0] data_d, data_q;
        reg [SPI_CLK_DIV-1:0] sck_d, sck_q;
        // wire sck_old;
        reg mosi_d, mosi_q;
        reg [NUM_BITS-1:0] ctr_d, ctr_q;
        reg new_data_d, new_data_q;
        reg [7:0] data_out_d, data_out_q;

        assign mosi = mosi_q;
        assign ss = state_q == IDLE;
        assign sck = (sck_q[SPI_CLK_DIV-1]) & (state_q == TRANSFER);
        assign busy = state_q != IDLE;
        assign data_out = data_out_q;
        assign new_data = new_data_q;

        always @(*) begin
          sck_d = sck_q;
```

```
    data_d = data_q;
    mosi_d = mosi_q;
    ctr_d = ctr_q;
    new_data_d = 1'b0;
    data_out_d = data_out_q;
    state_d = state_q;

    case (state_q)
            IDLE: begin
                    sck_d = 4'b0;
                    ctr_d = 3'b0;
                    mosi_d = 1'b1;
                    if (start == 1'b1) begin
                            data_d = data_in;
                            state_d = WAIT_HALF;
                    end
            end
            WAIT_HALF: begin
                    sck_d = sck_q + 1'b1;
                    if (sck_q == {1'b0,{SPI_CLK_DIV-1{1'b1}}}) begin
                            sck_d = 1'b0; //go right to transfer
                            state_d = TRANSFER;
                    end
            end
            TRANSFER: begin
                    sck_d = sck_q + 1'b1;
                    if (sck_q == 0) begin //transmit on falling edge
                            mosi_d = data_q[7];
                    end else if (sck_q == {1'b0,{SPI_CLK_DIV-1{1'b1}}})
begin //sample on rising edge
                            data_d = {data_q[6:0], miso};
                    end else if (sck_q == {SPI_CLK_DIV{1'b1}}) begin //
change bits between sck
                            ctr_d = ctr_q + 1'b1;
                            if (ctr_q == {NUM_BITS-1{1'b1}}) begin
                                    state_d = IDLE;
                                    data_out_d = data_q;
                                    new_data_d = 1'b1;
                            end
                    end
            end
    endcase
  end

  always @(posedge clk) begin
    if (rst) begin
            ctr_q <= {NUM_BITS-1{1'b0}};
            data_q <= {SPI_BUS_WIDTH{1'b0}};
```

```
                sck_q <= {SPI_CLK_DIV{1'b0}};
                mosi_q <= 1'b1;
                state_q <= IDLE;
                data_out_q <= {SPI_BUS_WIDTH{1'b0}};
                new_data_q <= 1'b0;
        end else begin
                ctr_q <= ctr_d;
                data_q <= data_d;
                sck_q <= sck_d;
                mosi_q <= mosi_d;
                state_q <= state_d;
                data_out_q <= data_out_d;
                new_data_q <= new_data_d;
        end
        end

        function integer log2;
                input [31:0] value;
                begin
                        for (log2=0; value>0; log2=log2+1)
                                begin
                                        value = value>>1;
                                end
                end
        endfunction


endmodule
```

8.10 TEST_JBIMU.V

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
//test_jbimu.v

module test_jbimu;

        // Inputs
        reg clks;
        reg clock;
        reg reset;
        reg start;
        wire miso;

        // Outputs
        wire [15:0] roll;
```

```
	wire [15:0] pitch;
	wire [15:0] yaw;
	wire [15:0] roll_rate;
	wire [15:0] pitch_rate;
	wire [15:0] yaw_rate;
	wire [15:0] accel_x;
	wire [15:0] accel_y;
	wire [15:0] accel_z;
	wire done;
	wire mosi;
	wire sck;
	wire ss;

	// Instantiate the Unit Under Test (UUT)
	jb_imu uut (
		.clock(clock),
		.reset(reset),
		.start(start),
		.roll(roll),
		.pitch(pitch),
		.yaw(yaw),
		.roll_rate(roll_rate),
		.pitch_rate(pitch_rate),
		.yaw_rate(yaw_rate),
		.accel_x(accel_x),
		.accel_y(accel_y),
		.accel_z(accel_z),
		.done(done),
		.miso(miso),
		.mosi(mosi),
		.sck(sck),
		.ss(ss)
	);

	wire slave_done;
	reg [7:0] din;
	wire [7:0] slave_dout;
	spi_slave slave(
.clk(clks),
.rst(reset),
.ss(ss),
.mosi(mosi),
.miso(miso),
.sck(sck),
.done(slave_done),
.din(din),
.dout(slave_dout)
);
```

```
	always #10 clock = ~clock; //50Mhz = 20 ns period

	always #20 clks = ~clks; //25 Mhz slave clock

	always @(slave_done) begin
		if(slave_done) begin
			din = din + 1'b1;
		end
	end

	initial begin
		// Initialize Inputs
		clock = 0;
		clks=0;
		reset = 0;
		start = 0;
		din = 0;

		// Wait 100 ns for global reset to finish
		reset = 1;
		#100;
		reset = 0;

		// Add stimulus here
		din = 8'h00;
		#20;

		start = 1;
		#20;
		start = 0;
		#10000;
	end

endmodule
```

8.11 TEST_SPI.V

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
//test_jbimu.v

module test_jbimu;

	// Inputs
	reg clks;
	reg clock;
	reg reset;
	reg start;
	wire miso;

	// Outputs
	wire [15:0] roll;
	wire [15:0] pitch;
	wire [15:0] yaw;
	wire [15:0] roll_rate;
	wire [15:0] pitch_rate;
	wire [15:0] yaw_rate;
	wire [15:0] accel_x;
	wire [15:0] accel_y;
	wire [15:0] accel_z;
	wire done;
	wire mosi;
	wire sck;
	wire ss;

	// Instantiate the Unit Under Test (UUT)
	jb_imu uut (
		.clock(clock),
		.reset(reset),
		.start(start),
		.roll(roll),
		.pitch(pitch),
		.yaw(yaw),
		.roll_rate(roll_rate),
		.pitch_rate(pitch_rate),
		.yaw_rate(yaw_rate),
		.accel_x(accel_x),
		.accel_y(accel_y),
		.accel_z(accel_z),
		.done(done),
		.miso(miso),
		.mosi(mosi),
		.sck(sck),
		.ss(ss)
	);

	wire slave_done;
	reg [7:0] din;
	wire [7:0] slave_dout;
	spi_slave slave(
.clk(clks),
.rst(reset),
.ss(ss),
    .mosi(mosi),
    .miso(miso),
    .sck(sck),
    .done(slave_done),
    .din(din),
    .dout(slave_dout)
    );

        always #10 clock = ~clock; //50Mhz = 20 ns period

        always #20 clks = ~clks; //25 Mhz slave clock

        always @(slave_done) begin
                if(slave_done) begin
                        din = din + 1'b1;
                end
        end

        initial begin
                // Initialize Inputs
                clock = 0;
                clks=0;
                reset = 0;
                start = 0;
                din = 0;

                // Wait 100 ns for global reset to finish
                reset = 1;
                #100;
                reset = 0;

                // Add stimulus here
                din = 8'h00;
                #20;

                start = 1;
                #20;
                start = 0;
                #10000;
        end

endmodule
```

8.12 SPI_SLAVE.V (EMBEDDEDMICRO.COM)

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

```
///////////////////////////////////////////////////////////////////////////////
//test_jbimu.v

module test_jbimu;

	// Inputs
	reg clks;
	reg clock;
	reg reset;
	reg start;
	wire miso;

	// Outputs
	wire [15:0] roll;
	wire [15:0] pitch;
	wire [15:0] yaw;
	wire [15:0] roll_rate;
	wire [15:0] pitch_rate;
	wire [15:0] yaw_rate;
	wire [15:0] accel_x;
	wire [15:0] accel_y;
	wire [15:0] accel_z;
	wire done;
	wire mosi;
	wire sck;
	wire ss;

	// Instantiate the Unit Under Test (UUT)
	jb_imu uut (
		.clock(clock),
		.reset(reset),
		.start(start),
		.roll(roll),
		.pitch(pitch),
		.yaw(yaw),
		.roll_rate(roll_rate),
		.pitch_rate(pitch_rate),
		.yaw_rate(yaw_rate),
		.accel_x(accel_x),
		.accel_y(accel_y),
		.accel_z(accel_z),
		.done(done),
		.miso(miso),
		.mosi(mosi),
		.sck(sck),
		.ss(ss)
	);
```

```
    wire slave_done;
    reg [7:0] din;
    wire [7:0] slave_dout;
    spi_slave slave(
.clk(clks),
.rst(reset),
.ss(ss),
.mosi(mosi),
.miso(miso),
.sck(sck),
.done(slave_done),
.din(din),
.dout(slave_dout)
);

    always #10 clock = ~clock; //50Mhz = 20 ns period

    always #20 clks = ~clks; //25 Mhz slave clock

    always @(slave_done) begin
            if(slave_done) begin
                    din = din + 1'b1;
            end
    end

    initial begin
            // Initialize Inputs
            clock = 0;
            clks=0;
            reset = 0;
            start = 0;
            din = 0;

            // Wait 100 ns for global reset to finish
            reset = 1;
            #100;
            reset = 0;

            // Add stimulus here
            din = 8'h00;
            #20;

            start = 1;
            #20;
            start = 0;
            #10000;
    end
```

```
endmodule
```

8.13 SRF05.V

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
//test_jbimu.v

module test_jbimu;

	// Inputs
	reg clks;
	reg clock;
	reg reset;
	reg start;
	wire miso;

	// Outputs
	wire [15:0] roll;
	wire [15:0] pitch;
	wire [15:0] yaw;
	wire [15:0] roll_rate;
	wire [15:0] pitch_rate;
	wire [15:0] yaw_rate;
	wire [15:0] accel_x;
	wire [15:0] accel_y;
	wire [15:0] accel_z;
	wire done;
	wire mosi;
	wire sck;
	wire ss;

	// Instantiate the Unit Under Test (UUT)
	jb_imu uut (
		.clock(clock),
		.reset(reset),
		.start(start),
		.roll(roll),
		.pitch(pitch),
		.yaw(yaw),
		.roll_rate(roll_rate),
		.pitch_rate(pitch_rate),
		.yaw_rate(yaw_rate),
		.accel_x(accel_x),
		.accel_y(accel_y),
		.accel_z(accel_z),
		.done(done),
		.miso(miso),
		.mosi(mosi),
		.sck(sck),
		.ss(ss)
	);

	wire slave_done;
	reg [7:0] din;
	wire [7:0] slave_dout;
	spi_slave slave(
.clk(clks),
.rst(reset),
.ss(ss),
.mosi(mosi),
.miso(miso),
.sck(sck),
.done(slave_done),
.din(din),
.dout(slave_dout)
);

	always #10 clock = ~clock; //50Mhz = 20 ns period

	always #20 clks = ~clks; //25 Mhz slave clock

	always @(slave_done) begin
		if(slave_done) begin
			din = din + 1'b1;
		end
	end

	initial begin
		// Initialize Inputs
		clock = 0;
		clks=0;
		reset = 0;
		start = 0;
		din = 0;

		// Wait 100 ns for global reset to finish
		reset = 1;
		#100;
		reset = 0;

		// Add stimulus here
		din = 8'h00;
		#20;
		start = 1;
		#20;
		start = 0;
		#10000;
	end

endmodule
```

8.14 TESST_SRF05.V

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
//test_jbimu.v

module test_jbimu;

	// Inputs
	reg clks;
	reg clock;
	reg reset;
	reg start;
	wire miso;

	// Outputs
	wire [15:0] roll;
	wire [15:0] pitch;
	wire [15:0] yaw;
	wire [15:0] roll_rate;
	wire [15:0] pitch_rate;
	wire [15:0] yaw_rate;
	wire [15:0] accel_x;
	wire [15:0] accel_y;
	wire [15:0] accel_z;
	wire done;
	wire mosi;
	wire sck;
	wire ss;

	// Instantiate the Unit Under Test (UUT)
	jb_imu uut (
		.clock(clock),
		.reset(reset),
		.start(start),
		.roll(roll),
		.pitch(pitch),
		.yaw(yaw),
		.roll_rate(roll_rate),
		.pitch_rate(pitch_rate),
		.yaw_rate(yaw_rate),
		.accel_x(accel_x),
		.accel_y(accel_y),
		.accel_z(accel_z),
		.done(done),
		.miso(miso),
		.mosi(mosi),
		.sck(sck),
		.ss(ss)
	);

	wire slave_done;
	reg [7:0] din;
	wire [7:0] slave_dout;
	spi_slave slave(
.clk(clks),
.rst(reset),
.ss(ss),
.mosi(mosi),
.miso(miso),
.sck(sck),
.done(slave_done),
.din(din),
.dout(slave_dout)
);

	always #10 clock = ~clock; //50Mhz = 20 ns period

	always #20 clks = ~clks; //25 Mhz slave clock

	always @(slave_done) begin
		if(slave_done) begin
			din = din + 1'b1;
		end
	end

	initial begin
		// Initialize Inputs
		clock = 0;
		clks=0;
		reset = 0;
		start = 0;
		din = 0;

		// Wait 100 ns for global reset to finish
```

```
            reset = 1;
            #100;
            reset = 0;

            // Add stimulus here
            din = 8'h00;
            #20;

            start = 1;
            #20;
            start = 0;
            #10000;
        end

endmodule
```

8.15 LABKIT.V

```
`default_nettype none
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
```

```
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green"
//              and "vga_out_blue". (Was 10'h0, now 8'h0.)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////

module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
               ac97_bit_clock,

               vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
               vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
               vga_out_vsync,

               tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
               tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
               tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,
               tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
               tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
               tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
               tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

               ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
               ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

               ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
               ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

               clock_feedback_out, clock_feedback_in,

               flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
               flash_reset_b, flash_sts, flash_byte_b,

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

               disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_in,

               button0, button1, button2, button3, button_enter, button_right,
               button_left, button_down, button_up,

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

               systemace_data, systemace_address, systemace_ce_b,
               systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

               analyzer1_data, analyzer1_clock,
               analyzer2_data, analyzer2_clock,
               analyzer3_data, analyzer3_clock,
               analyzer4_data, analyzer4_clock);

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
```

```
  output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
         vga_out_hsync, vga_out_vsync;

  output [9:0] tv_out_ycrcb;
  output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
         tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
         tv_out_subcar_reset;

  input  [19:0] tv_in_ycrcb;
  input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
         tv_in_hff, tv_in_aff;
  output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
         tv_in_reset_b, tv_in_clock;
  inout  tv_in_i2c_data;

  inout  [35:0] ram0_data;
  output [18:0] ram0_address;
  output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b,
ram0_we_b;
  output [3:0] ram0_bwe_b;

  inout  [35:0] ram1_data;
  output [18:0] ram1_address;
  output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b,
ram1_we_b;
  output [3:0] ram1_bwe_b;

  input  clock_feedback_in;
  output clock_feedback_out;

  inout  [15:0] flash_data;
  output [23:0] flash_address;
  output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
  input  flash_sts;

  output rs232_txd, rs232_rts;
  input  rs232_rxd, rs232_cts;

  input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

  input  clock_27mhz, clock1, clock2;

  output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
  input  disp_data_in;
  output  disp_data_out;

  input  button0, button1, button2, button3, button_enter, button_right,
         button_left, button_down, button_up;
```

```
input  [7:0] switch;
output [7:0] led;

inout [31:0] user1, user2, user3, user4;

inout [43:0] daughtercard;

inout  [15:0] systemace_data;
output [6:0]  systemace_address;
output systemace_ce_b, systemace_we_b, systemace_oe_b;
input  systemace_irq, systemace_mpbrdy;

output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
              analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

////////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////////////////////////////////////////////

// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// VGA Output
assign vga_out_red = 8'h0;
assign vga_out_green = 8'h0;
assign vga_out_blue = 8'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
assign vga_out_vsync = 1'b0;

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
```

```
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

//   // LED Displays
//   assign disp_blank = 1'b1;
//   assign disp_clock = 1'b0;
//   assign disp_rs = 1'b0;
//   assign disp_ce_b = 1'b1;
//   assign disp_reset_b = 1'b0;
//   assign disp_data_out = 1'b0;
   // disp_data_in is an input

   // Buttons, Switches, and Individual LEDs
 //  assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
   assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   //assign user3 = 32'hZ;
  //assign user4 = 32'hZ;

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

//   // Logic Analyzer
//   assign analyzer1_data = 16'h0;
//   assign analyzer1_clock = 1'b1;
//   assign analyzer2_data = 16'h0;
//   assign analyzer2_clock = 1'b1;
//   assign analyzer3_data = 16'h0;
//   assign analyzer3_clock = 1'b1;
//   assign analyzer4_data = 16'h0;
//   assign analyzer4_clock = 1'b1;

////////////////////////////////////////////////////////////////////////////
```

```
  //
  // Reset Generation
  //
  // A shift register primitive is used to generate an active-high reset
  // signal that remains high for 16 clock cycles after configuration finishes
  // and the FPGA's internal clocks begin toggling.
  //
  ////////////////////////////////////////////////////////////////////////////
  wire reset;
  SRL16 reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(reset),
                 .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
  defparam reset_sr.INIT = 16'hFFFF;

        // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
////   wire clock_50mhz_unbuf,clock_50mhz;
////   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_50mhz_unbuf));
////   // synthesis attribute CLKFX_DIVIDE of vclk1 is 27
////   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 50
////   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
////   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
////////   BUFG vclk2(.O(clock_50mhz),.I(clock_50mhz_unbuf));

        wire sensor_reset, sensor_reset_debounced;
        assign sensor_reset = reset | sensor_reset_debounced;
        wire sensor_start;



        debounce reset_debounce(.reset(reset), .clock(clock_27mhz),
.noisy(~button0),
                .clean(sensor_reset_debounced));

        debounce sensor_start_deb(.reset(reset),.clock(clock_27mhz),

        .noisy(~button3),.clean(sensor_start));


        wire echo;
        assign echo = user3[30];
        wire trigger;
        assign user3[31] = trigger;
        assign user3[29:5] = 25'hZ;
        wire [14:0] distance_out;
        wire [14:0] distance;
        wire error;
        wire sensor_ready;
```

```
//      srf05_trigger_and_echo srf05(
//    .clock(clock_27mhz),                          //50 MHz clock signal
//    .reset(height_reset),                                        //reset signal
//    .start(height_start),                                        //start signal to begin
sensor reading
//      .echo(echo),                                               //echo read in line from
sensor
//    .distance(distance_out),              //distance output (as a factor distance/29=
cm distance)
//      .trigger(trigger),                                  //trigger line out to sensor
//      .ready(sensor_ready)                                              //when finished
measuring distance
//    );



        srf05 height_sensor(
    .clock(clock_27mhz),                          //50 MHz clock signal
    .reset(sensor_reset),                                          //reset signal from
fpga/flight controller
//    input init,                                        //intialize signal from flight
controller
        .start(sensor_start),                                             //start signal from
flight controller to begin normal operation
        .echo(echo),                                               //output echo line from
sensor
    .distance(distance),              //Kept as a factor of 29 (for centimeter height) for
simplified interface (assumed signed)
        .trigger(trigger),                                  //output trigger line to sensor
        .error(error),                                             //HIGH if error is
detected
        .ready(sensor_ready)
    );

//      wire signed [15:0] distance_sync;
//      synchronize16
sync_avg(.clk(clock_27mhz),.in(distance),.out(distance_sync));
//      wire fir_nfd, fir_ready, fir_nd;
//      assign fir_nd = sensor_ready;
//      wire [15:0] fir_dout;
//      wire [14:0] fir_din;
//      assign fir_din = distance[14:0];
//
//      fir_compiler_v5_0 fir_sr50(
//              .rfd(nfd), .rdy(fir_ready), .nd(fir_nd), .clk(clock_27mhz),
.dout(fir_dout), .din(fir_din));
////
        //Ring Buffer
        reg  [14:0] buffer[31:0];
        reg  [19:0] sum = 0;
        reg  [14:0] avg = 0;
        reg [4:0] offset = 0;

        integer i;
        initial begin
                for(i = 0; i < 32;i = i+1) begin
                        buffer[i] = 15'sd0;
                end
        end


        always @(posedge clock_27mhz) begin
                        if(sensor_ready) begin
                                sum <= sum + (distance) - (buffer[offset]);
                                buffer[offset] <= distance;
                                offset <= offset + 1'b1;
                                avg <= sum/32;
                        end
        end

        //Jbimu

        //hardware reset
//      wire imu_reset,imu_reset_deb;
        //assign imu_reset = switch[0];
//      assign user4[0] = imu_reset;



        wire signed [15:0]
roll,pitch,yaw,roll_rate,yaw_rate,pitch_rate,accx,accy,accz;
        wire [143:0] data_out2;
        wire done;
        wire miso,mosi,sck,ss;
        assign user3[0] = sck;
        assign user3[1] = sck;
        assign miso = user3[2];
        assign user3[3] = mosi;
        assign user3[4] = ss;
        assign user4[31:0] = 32'hZ;
        wire [7:0] spi_data;
        assign led = {1'b1,
miso,switch[1],switch[2],switch[3],~sensor_reset,~sensor_start,~sensor_ready};

        jb_imu imu(
   .clock(clock_27mhz),                          //50 Mhz clock
   .reset(sensor_reset),                                        //reset signal
        .start(sensor_start),                                          //start normal
operations
   .roll(roll),                  //Roll Angle *100 (deg)
   .pitch(pitch),                //Pitch Angle *100 (deg)
   .yaw(yaw),                            //Yaw Angle *100 (deg)
   .roll_rate(roll_rate),        //Roll Rate *10 (deg/sec)
   .pitch_rate(pitch_rate),              //Pitch Rate *10 (deg/sec)
   .yaw_rate(yaw_rate),          //Yaw Rate * 10 (deg/sec)
   .accel_x(accx),               //Accleration gs*1000
   .accel_y(accy),               //Acceleration gs*1000
   .accel_z(accz),               //Acceleration gs*1000
        .data_out_raw(data_out2),
        .spi_data(spi_data),
        .done(done),                                            //IMU Finished Reading
        .miso(miso),                                            //MISO Master In/Slave
Out
   .mosi(mosi),                                       //MOSI Master OUt/Slave IN
   .sck(sck),                                 //SClock out to device
        .ss(ss)                                                 //SPI Select Bit
   );

       ///AVeraging Filter for IMU outputs
       //Ring Buffer
       reg signed  [15:0] buffer_roll[63:0];
       reg signed [15:0] buffer_pitch[63:0];
       reg signed [15:0] buffer_yaw[63:0];
       reg signed [15:0] buffer_roll_rate[63:0];
       reg signed [15:0] buffer_pitch_rate[63:0];
       reg signed [15:0] buffer_yaw_rate[63:0];
       reg signed [15:0] buffer_accx[63:0];
       reg signed [15:0] buffer_accy[63:0];
       reg signed [15:0] buffer_accz[63:0];
       reg signed [21:0] sum_roll = 0;
       reg signed [21:0] sum_pitch = 0;
       reg signed [21:0] sum_yaw = 0;
       reg signed [21:0] sum_roll_rate = 0;
       reg signed [21:0] sum_pitch_rate = 0;
       reg signed [21:0] sum_yaw_rate        = 0;
       reg signed [21:0] sum_accx = 0;
       reg signed [21:0] sum_accy = 0;
       reg signed [21:0] sum_accz = 0;
       reg signed [15:0] avg_roll;
       reg signed [15:0] avg_pitch;
       reg signed [15:0] avg_yaw;
       reg signed [15:0] avg_roll_rate;
       reg signed [15:0] avg_pitch_rate;
       reg signed [15:0] avg_yaw_rate;
```

```
	reg signed [15:0] avg_accx;
	reg signed [15:0] avg_accy;
	reg signed [15:0] avg_accz;
	reg [4:0] offset2 = 0;


	integer i2;
	initial begin
		for(i2 = 0; i2 < 64;i2 = i2+1) begin
			buffer_roll[i2] = 16'sd0;
			buffer_pitch[i2] = 16'sd0;
			buffer_yaw[i2] = 16'sd0;
			buffer_roll_rate[i2] = 16'sd0;
			buffer_pitch_rate[i2] = 16'sd0;
			buffer_yaw_rate[i2] = 16'sd0;
			buffer_accx[i2] = 16'sd0;
			buffer_accy[i2] = 16'sd0;
			buffer_accz[i2] = 16'sd0;
		end
	end

	reg [4:0] buffer_cnt = 0;
	always @(posedge clock_27mhz) begin
			if(done) begin
				sum_roll <= sum_roll + (roll) - (buffer_roll[offset2]);
				buffer_roll[offset2] <= roll;
				avg_roll <= sum_roll/64;

				sum_pitch <= sum_pitch + (pitch) -
(buffer_pitch[offset2]);
				buffer_pitch[offset2] <= pitch;
				avg_pitch <= sum_pitch/64;

				sum_yaw <= sum_yaw + (yaw) - (buffer_yaw[offset2]);
				buffer_yaw[offset2] <= yaw;
				avg_yaw <= sum_yaw/64;

				sum_roll_rate <= sum_roll_rate + (roll_rate) -
(buffer_roll_rate[offset2]);
				buffer_roll_rate[offset2] <= roll_rate;
				avg_roll_rate <= sum_roll_rate/64;

				sum_pitch_rate <= sum_pitch_rate + (pitch_rate) -
(buffer_pitch_rate[offset2]);
				buffer_pitch_rate[offset] <= pitch_rate;
				avg_pitch_rate <= sum_pitch_rate/64;
```

```
                                        sum_yaw_rate <= sum_yaw_rate + (yaw_rate) -
(buffer_yaw_rate[offset2]);
                                        buffer_yaw_rate[offset2] <= yaw_rate;
                                        avg_yaw_rate <= sum_yaw_rate/64;

                                        sum_accx <= sum_accx + (accx) -
(buffer_accx[offset2]);
                                        buffer_accx[offset2] <= accx;
                                        avg_accx <= sum_accx/64;

                                        sum_accy <= sum_accy + (accy) -
(buffer_accy[offset2]);
                                        buffer_accy[offset2] <= accy;
                                        avg_accy <= sum_accy/64;

                                        sum_accz <= sum_accz + (accz) -
(buffer_accz[offset2]);
                                        buffer_accz[offset2] <= accz;
                                        avg_accz <= sum_accz/64;
                              end
        end




        wire [47:0] imu_info;
//      assign imu_info = (switch[1]) ? {roll,pitch,yaw} :
//                                                   (switch[2]) ?
{roll_rate,pitch_rate,yaw_rate} :
//                                                   (switch[3]) ? {accx,accy,accz} :
//                                                   48'hFF_FF_FF_FF_FF_FF;
        assign imu_info = (switch[1]) ? {avg_roll,avg_pitch,avg_yaw} :
                                                     (switch[2]) ?
{avg_roll_rate,avg_pitch_rate,avg_yaw_rate} :
                                                     (switch[3]) ?
{avg_accx,avg_accy,avg_accz} :
                                                     48'hFF_FF_FF_FF_FF_FF;
//      assign imu_info = {roll,pitch,yaw};


        wire [15:0] data_out;
        assign data_out = {1'b0,avg};


        //hex display out
        wire [63:0] data;
        assign data = {imu_info,data_out};
```

```
	display_16hex hexdisplay(.reset(reset),.clock_27mhz(clock_27mhz),
								 .data(data),
.disp_blank(disp_blank),

	.disp_clock(disp_clock),.disp_rs(disp_rs),

	.disp_ce_b(disp_ce_b),.disp_reset_b(disp_reset_b),

	.disp_data_out(disp_data_out)
								 );


	// Logic Analyzer
   assign analyzer1_data = roll;][=
   assign analyzer1_clock = 1'b0;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = {3'd0,done,ss,sck,miso,mosi,spi_data};
   assign analyzer3_clock = {clock_27mhz};
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

endmodule

//
//module synchronize16 (input clk,input [15:0] in,
//                     output reg [15:0] out);
//
//  reg [15:0] sync;
//
//  always @ (posedge clk)
//  begin
//    {out,sync} <= {sync,in};
//  end
//endmodule

module debounce #(parameter DELAY=270000)   // .01 sec with a 27Mhz clock
	        (input reset, clock, noisy,
	         output reg clean);

   reg [18:0] count;
   reg new;

   always @(posedge clock)
     if (reset)
       begin
          count <= 0;
          new <= noisy;
          clean <= noisy;
      end
    else if (noisy != new)
      begin
          new <= noisy;
          count <= 0;
      end
    else if (count == DELAY)
      clean <= new;
    else
      count <= count+1;

endmodule
```