Have a Safe Flight: Bon Voyage!

Mariela E. Buchin, WonRon Cho, Scott Fisher

6.111 Digital Systems Laboratory – Spring 2006
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
May 18, 2006

Abstract

This project creates an interface between the movement of a human body and the flight path of an airplane. It uses a "smart flight vest" to detect the movements of a pilot's upper body and translates them into parameters representing the pitch and roll of an airplane in flight. In addition to the pitch and roll, the throttle of the airplane is set and adjusted through a throttle device that replicates the motion of an actual throttle lever. The physics of the flight mimics the real physics of an actual airplane, taking into account lift, drag, weight, and thrust. The state of the flight can be viewed from a monitor that displays the main features of an airplane console, including an attitude indicator, a compass, and an altitude and vertical velocity display.

Table of Contents

List of Figures/ Tables

Overview

The main hardware component of the system is found in the flight vest where two angular rate sensors are mounted onto circuit boards perpendicular to each other. The output of the sensors are sent through analog to digital converters that are also attached to the circuit boards, then they are sent to the main labkit via a long ribbon cable. A sensor module programmed into the Xilinx chip interfaces with the converter to make digital values that can then be used to determine the angles that the pilot is tilting at. A throttle lever is also interfaced with the kit though the use of a AD converter the user I/O pins. The three output signals from the sensor module represent the vertical and horizontal tilt of the "yoke" of the airplane, and the amount of throttle. These 10 bit signals are then sent to the physics module for conversion into actual physical aspects of an airplane.

Mariela Buchin

The goal of the physics engine is to take in three user inputs, the throttle, the yoke's pitch, and the yoke's roll, and convert these signals into a realistically controlled simulation of flight. All calculations are done within the modules within the physics engine. To create realistic flight there are a few considerations to account for when coming up with physics equations. One main consideration is the forces on the plane.

While in flight, the plane experiences four main forces: Lift, Thrust, Drag, and Weight. These are primarily controlled by the value of the throttle and the orientation of the plane. Lift is always perpendicular to the orientation of the wings. Thrust and Drag always work to counteract each other in the direction of the plane's motion. And Weight is always directed down. Thrust is proportional to the percentage of the throttle; Drag and Lift are proportional to the square of the velocity of the plane as well as experimentally determined coefficients. During the course of flight, these four forces act together to create the basic motion of the plane through space. As the plane rotates and pitches, these forces act in different directions, thus these forces are calculated and combined by their component vectors.

Another large part of a plane's behavior in the air is the nose's orientation. The pilots control yoke (or in our case, the orientation of the user's body) is used to roll and pitch the plane. In reality, there are many forces and complicated equations involved to create a very realizable attitude (pitch and roll) behavior, however, for this simulation, the plane's attitude will be controlled simply by the position of the controls, not subject to drift or other ambient behaviors. As it turns out, using these approximations do not affect the realizable aspect in any large factor when the plane is under control. Also, with the lack of precision guaranteed by the gyros, this approximation enables the pilot much better control over the plane.

One last major consideration for the physics, is the fact that this is a digital system that is updated periodically. Thus, a discrete system is trying to simulate a continuous one. Therefore, throughout the physics engine's many modules, there are frequent bit shifts in order to reach fractional values for velocities and altitudes. These are known as the delta values, ultimately calculating a rough Euler approximation to the actual flight of the plane.

All of these considerations have been analyzed and calculations carefully written to realistically simulate all of these forces, angles, and discrete approximations working

together. After calculating everything, outputs for Speed, Altitude, Pitch, Roll, Ascent Rate, and Direction are output to the display modules to give our pilot something to look at.

Scott Fisher

The visual component of the user interface of this system is the LCD screen with 640x480 (VGA) resolution. The user will be able to track parameters such as speed, altitude, ascent rate, and heading (on a compass) of the airplane on a simplified version of an aircraft console. The user can also see the horizon on an attitude indicator. In addition, the position of the plane's throttle is displayed on two vertical bars on both ends of the screen. When the plane flies too high or too low, the game ends with corresponding "Game Over" screens.

WonRon Cho

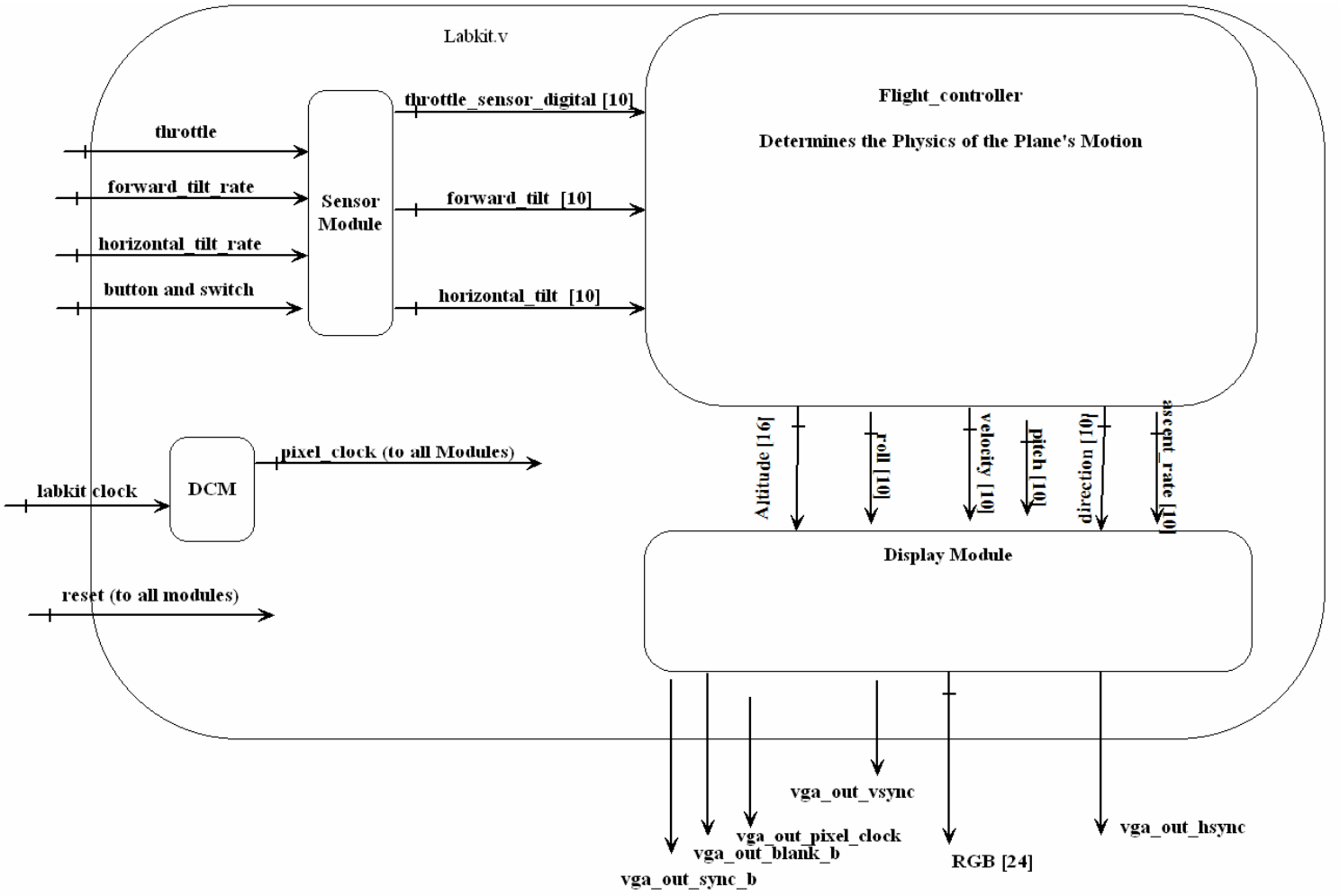

Figure 1: Block diagram of top level view of system

From Body Movements to Digital Values – Mariela Buchin

Interfacing the Analog Sensors to the Labkit

Attached to the back of the flight vest is a 50 wire ribbon cable that carries all of the data output from the sensors to a connector on the labit. From there, wires are sent to the user I/O pins of the labkit, interfacing each of them to an analog to digital FSM that will sample their angular rate values. From the FSM, the rates are synchronized to the main clock frequency of the entire project, then they are converted to angle values and sent to the physics module. The throttle lever is connected to the labkit via an AD converter and the user I/O pins. The input from the throttle into the labkit is also interfaced with an FSM and then synchronized with the main clock. All three signals outputted to the physics module are 10 bits wide. A block diagram of the sensor system is shown in Figure #.

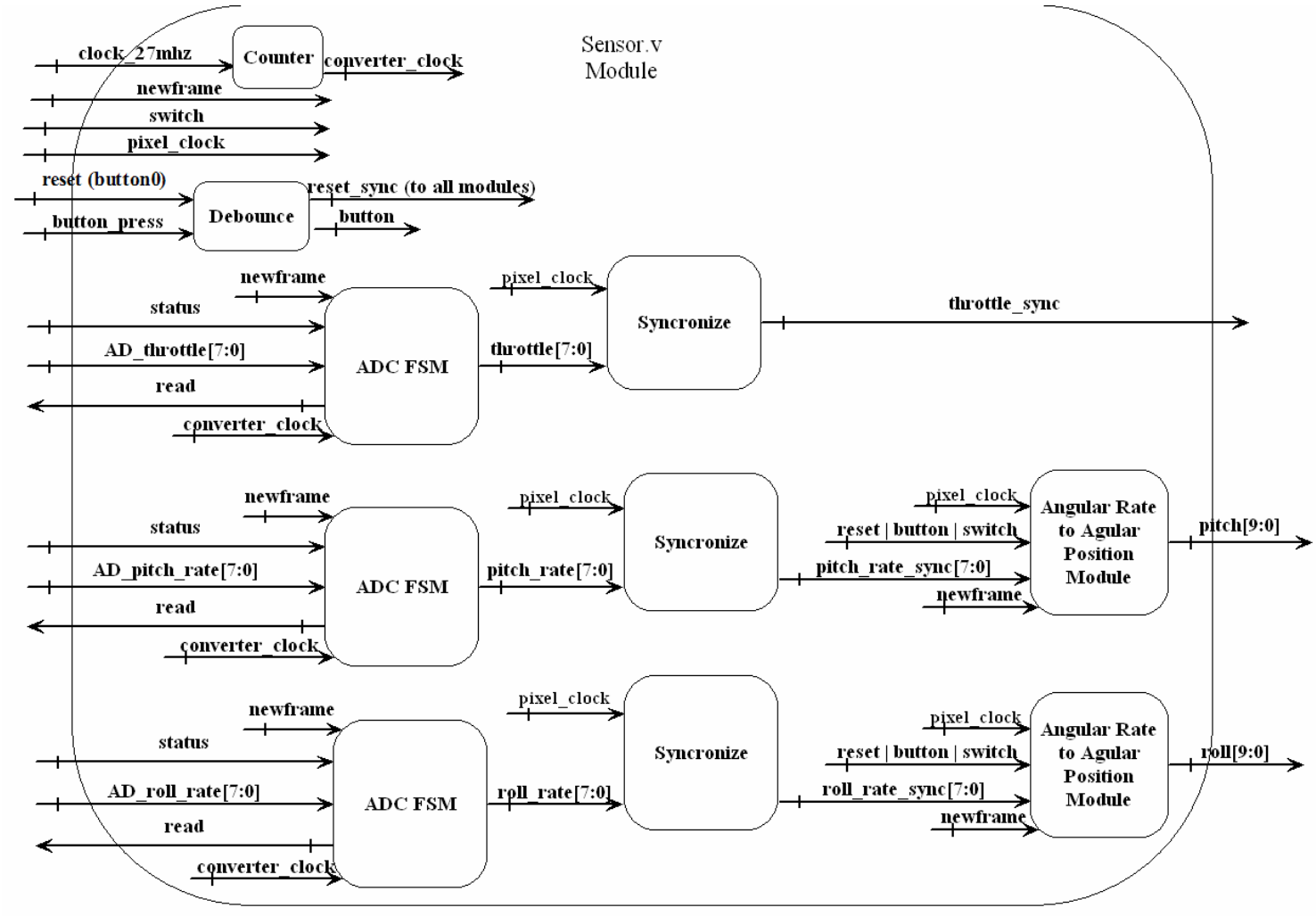

Figure 2: Block diagram of the Sensor System

Detecting Pilot's Movements

There are two angular rate sensors mounted onto the upper back of the flight vest. The sensors are mounted perpendicular to each other to create two axis of rotation corresponding to the horizontal and vertical tilt of the pilot. The rate sensor being used is the ADXRS150EB, which is an evaluation board containing a yaw rate gyro. The output of the sensor is an analog signal between 0 and 5 volts, where a voltage of 2.5 represents a zero angular rate. Rotating the gyro clockwise increases the output voltage from the 2.5 mark, and rotating it counterclockwise decreases the output voltage. The maximum rate that the gyro can detect is +/- 300 degrees per second; however, for the intents of this project, the angular rates being detected will have magnitudes much smaller than the specified maximum. The bandwidth of the gyro is 400 Hz so the Nyquist rate is 800 Hz. We decided to over-sample the data with a sampling frequency of 3.375MHz.

Throttle Lever

A 10k potentiometer is used to create an analog output that represents the throttle of the airplane. The output of the potentiometer is sent through an opAmp in a LM358AN chip, with the output of the opAmp wired to its negative input to provide negative feedback. Turning the potentiometer arm clockwise causes the output voltage to increase at a near linear rate, and turning it counter-clockwise causes the voltage to decrease. The arm is glued to a wooden lever that looks similar to a throttle lever in an airplane. The output of the opAmp is an analog signal between 0 and approximately 2.5 volts. This signal is sent to an AD converter and then sent to the FPGA labkit through wires connected to the user4 I/O pins.

Analog to Digital Converter

The AD670 analog to digital converter is used to convert the analog output of the gyro to an 8 bit digital input to the labkit. The voltage input range that the AD670 can take is between 0 and 2.5 volts, and the converter has an internal resistance of 10k ohms. Thus, the output of the two gyros is connected to the input of the AD converter through a 10k resistor, creating a voltage divider so that the input voltage to the converter is in between 0 and 2.5 volts instead of 0 and 5 volts. The AD670 has the capability of outputting a bipolar value and formatting its output bits through the use of the BPO and Format pins; however, none of these features are needed for the purpose of the project so both the BPO and the Format pins are grounded. The ~CE and ~CS pins are also wired to ground since the AD converter is the only chip driving the bus. This leaves only two pins, the status and read pins, to determine the state of the converter. The data comes out of the converter through 8 parallel bit output pins. For the case of the gyro AD converters, the output pins are sent to the labkit through a long 50 wire ribbon cable with a connecter on both ends. The output of the ribbon cable is then sent to the user1 and user3 I/O pins of the labkit through two separate sets of wires soldered to header pins.

Creating a 3.375 MHz Clock

To make a clock that is slow enough for the AD converter to function properly, a 3 bit counter was made that continuously counted from zero to eight. The high order bit of the counter was taken to represent the 3.375 MHz clock, and this signal was sent through a buffer with the final output being called converter_clock.

AD670 FSM

The process for converting from an analog signal to an 8 bit digital signal requires the use of an 8 state finite state machine. The FSM module takes in a sample signal that forces the FSM to cycle through its eight states. When the FSM finishes its cycle, it waits in an idle state until the sample signal goes high again. The sample signal is an enable signal coming in from the main module that pulses every frame. When the sample signal pulses, it sends the FSM first into a cycle of three states that set the "read" signal low. When the read signal goes low, the converter has 700 ns to begin its conversions on the analog input. When conversion begins, the chip forces the status signal high, then, when it comes out of conversion, it sets the status signal low. There is a time delay from the time when conversion ends to the time when the data output becomes valid; therefore, there are two read delay cycles that come before the read cycle. In the read cycle, the data output of the FSM gets updated with the new values coming from the AD converter. Then the FSM moves back to the Idle state to wait for sample to go high again.

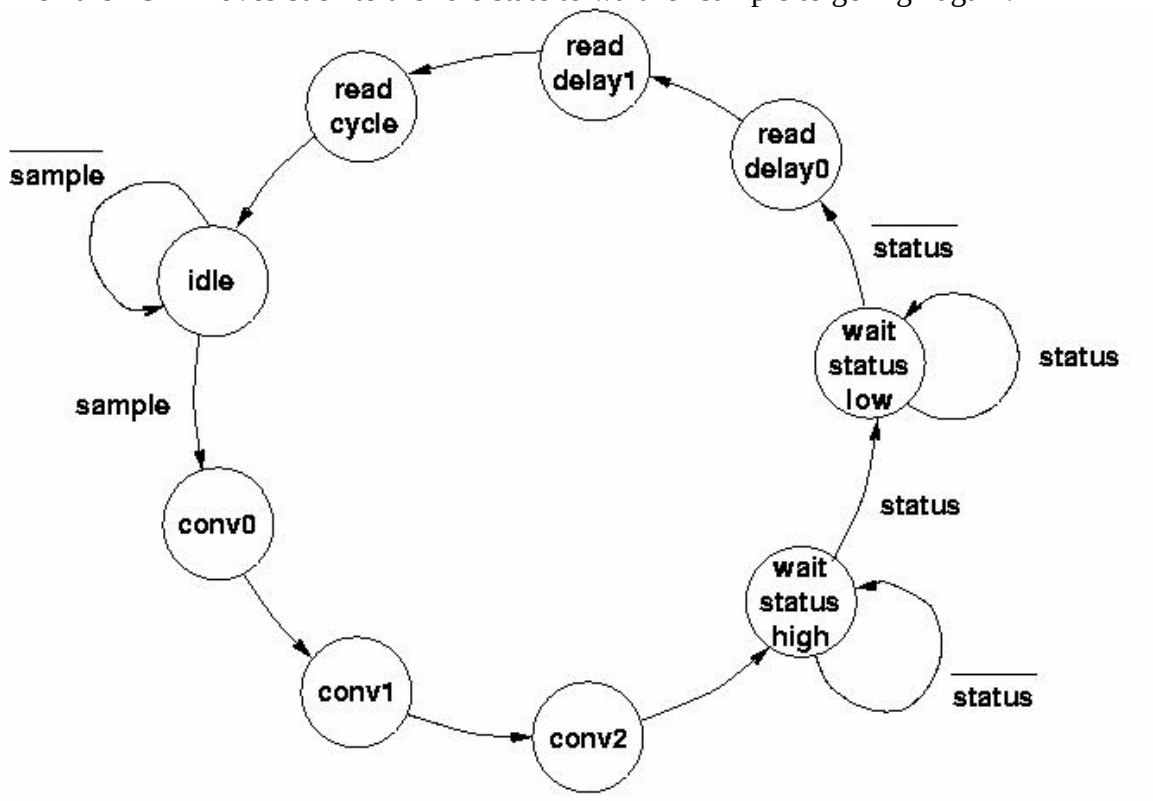

Figure 3 : FSM representing the cycle of the converter for taking a sample of data *(Anantha Chandrakasan, Lecture 10)*

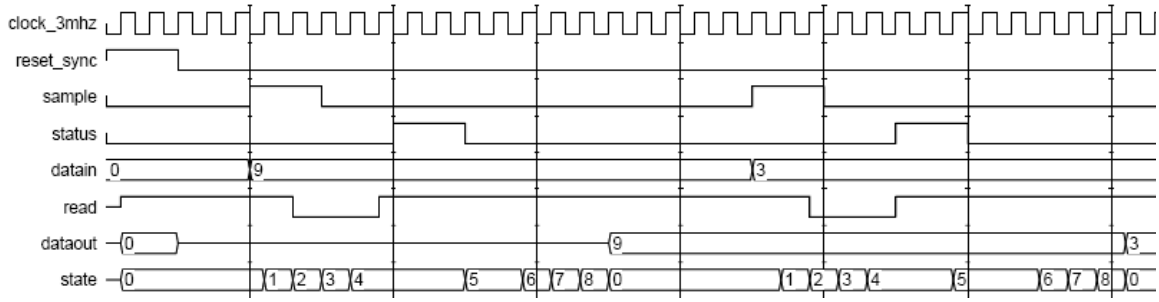

Figure 4 : Shows how the data output is set using the read and status signals.

Synchronizer

Since all of the signals coming out of the AD670 FSM are running at the 3.375 MHz clock, they all need to be synchronized with main clock that all the other modules of the project are running at. The main clock that the modules are using is called pixel_clock, which is determined by parameters related to the Display module. To synchronize the signals, at every positive edge of pixel_clock, the outputs of the converters go through two registers so that the chances of being stuck in the metastable state are extremely small. Two of these outputs are then sent to the angle-to-rate module so that the angular rates from the gyro can be translated in to actual angle values that are similar to the output of a yoke of an airplane. The output from the throttle however, is ready to be sent to the physics module since it does not need any further conversions.

Rate to Angle Module

The two digital signals coming in from the gyros are voltages that represent an angular rate in degrees per second. This value needs to be converted first into an angular rate and then from there converted into an angle value. To get an angular rate, the zero movement voltage of the gyro is subtracted from the current voltage value being sent out of the ADC. The resulting value is multiplied by a K factor that is a constant with the units of Degs/sec/volt. For moving counterclockwise, the K value was one; however, because of a noticeable slack in movement when the gyros were turned clockwise, the K value for clockwise motion was chosen to be 17/16. To get the angle from the rate, the rate signal is integrated to give the equation Angle = Angle + AngleRate*deltaT. Delta T is given the value of one, and it is good to note that the Angle is only incremented on the positive edge of an enable signal that pulses every new frame of the display monitor. Even with the enable signal, the angle still increments to fast to be realistic, so the angle variable is made to be 14 bits long, with the top ten bits being assigned to the output. The zero voltage of both gyros varies from time to time; therefore, a button an additional button is used for the purposed of recalibrating the zero voltage value. The output coming from this module is twelve bits long and is ready to be sent directly to the physics module.

Debugging

The main problems that were encountered in implementing this section of the project were mainly due to clocking errors. Initially, the ADC FSM was not working properly due to a false impression that the DCM is capable of making a 3.375 MHz clock. It was later found that the DCM module can not make clocks with frequencies that low, therefore the use of a counter was needed to implement the lower frequency clock. Upon integrating all the modules together, it was noticed that some modules were being synchronized with the 27mhz clock instead of pixel_clock, this was creating an output on the monitor that looked random and ecstatic.

Physics of an Aiplane in Flight – Scott Fisher

Module PlanePhysics.v & Flight.v

The Flight.v is the main module that holds all of the physics and increments all the values every time the "calc_enable" signal goes high. In order to update the graphics, the values the display uses need to be updated once every frame, thus all calculations are based on a discretized system that increments at 75 Hz. The Flight module implements all of the sub-modules, described below, to calculate the many physical values surrounding the plane's flight. Only the modules calculating alpha and rho are clocked. Thus the modules for alpha, rho, and flight in a bigger module: PlanePhysics. Alpha and rho were treated as constant valued inputs to the Flight module. The other modules were un-clocked within the Flight module because, with all the calculations that had to be done, there was an uncertainty whether enough time would be given to each module to do the calculations each one required. There were many modules requiring sine and cosine calculations and these were based off of look up tables in roms. Figure 5 shows a block diagram of the physics modules and how they are connected.

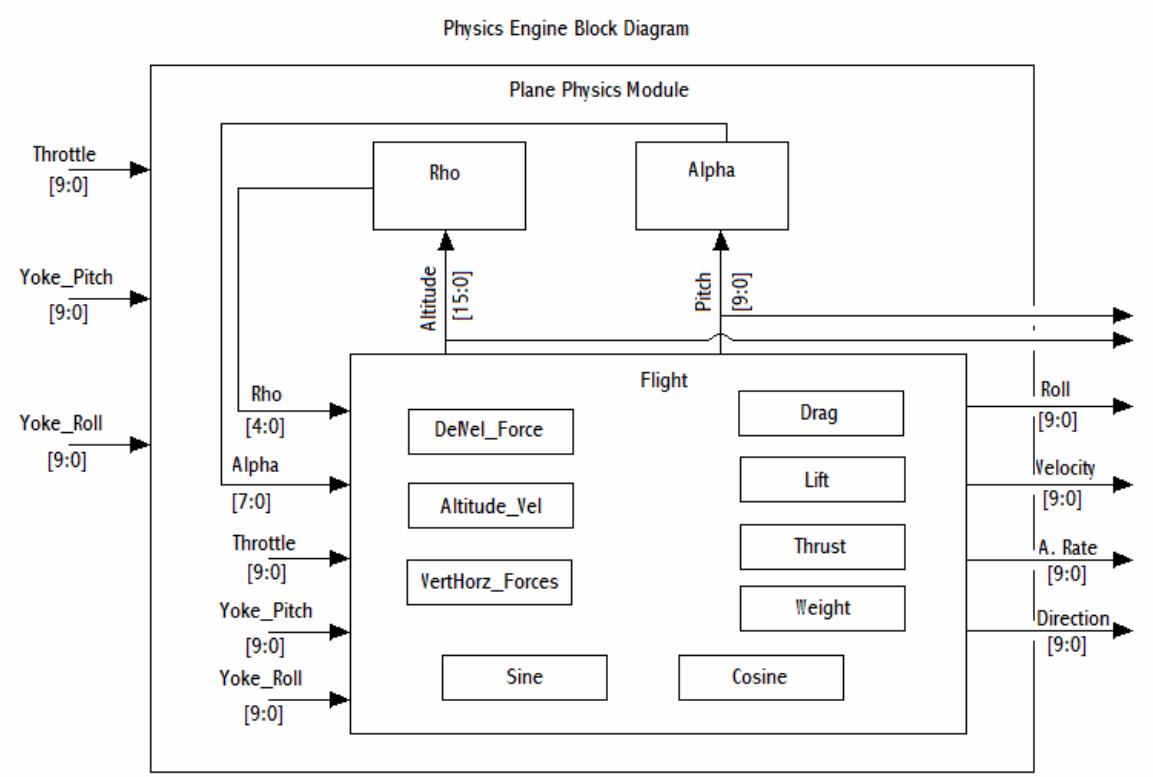

Figure 5: Block diagram of the Physics Engine showing the modules and their connections.

Within the clocked portion of the Flight module, there is a check for a reset and the two endgame scenarios before letting the system increment everything if the "calc_enable" is high. If it is, the module relies on the un-clocked outputs from the many modules to have settled down to give the many different force, delta velocity, delta altitude, and delta angle values to add to the current values.

The outputs from the Flight module are attacked to the upper bits of the corresponding values that are used in the calculations within the module. An example of this is the output for altitude "alt_out" is connected to the top 15 bits of the 23 bit value "altitude" which is the value that is actually incremented during the "calc_enable." This gives 8 bits to treat as a fractional value for each output value.

Each module within these two modules will be explained below.
Code for PlanePhysics.v can be found in Appendix #.
Code for Flight.v can be found in Appendix #.

Module Alpha1.v

One of the main forces present on an airplane is the lift force. This force makes the airplane what it is. The wings of the plane are what generate the lift force based on their shape and the presence of fast moving air over and under them. The pressure differential in the two streams of air is what creates the lift force. The actual calculations

for the strength of the force are explained for the module actually calculating that force, however, one fundamental part of the equation for lift is an experimentally derived constant usually denoted as “alpha.” The value of “alpha” ranges from roughly -1.2 to 1.6 and is a scalar for the lift force. The value of “alpha” to scale the lift force changes as the “angle of attack” changes. This relationship can be seen in Figure 6.

Image removed due to copyright restrictions.
Please see http://home.comcast.net/~clipper-108/Image20.gif

Figure 6: Lift (proportional to alpha) vs. angle of attack in degrees

The relationship is roughly symmetrical for negative angles. The coefficient alpha is basically undefined for angles higher than about 20 degrees and planes have a hard time flying at angles of attack higher than 20 degrees because what is known as a stall ensues from flying at those higher angles. The stall can be seen in the figure as the value for lift sharply falls. The “angle of attack” that alpha is based off of is the angle the wing is above the air flow. A picture of this can be seen in Figure 7.

Images removed due to copyright restrictions. Please see
http://home.comcast.net/~clipper-108/Image14.gif and
http://home.comcast.net/~clipper-108/Image15.gif

Figure 7: A small change in the “Angle of Attack” with a significant change in the airflow.

Calculating the true angle of attack can be very tricky as it involves vectors and direction of the ambient airflow around the plane. For this physics simulation, the angle of attack is approximated as simply the pitch of the plane. The pitch of the plane can go higher than 20 degrees, so values for alpha had to be approximated for angles greater than 20 degrees. Because the angle of attack is approximated as pitch, the stall angle has been increased to about 30 degrees. The final piecewise linear approximation that can be seen in Figure 8.

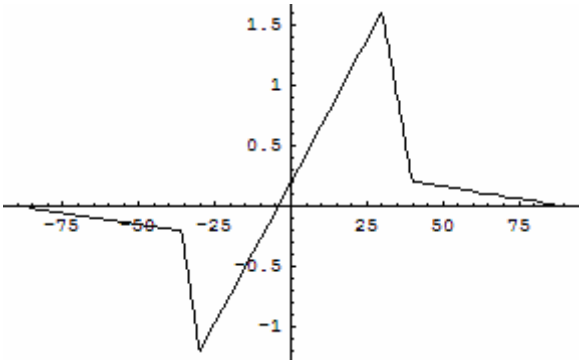

Figure 8: Piecewise Linear approximation for alpha for angles -90 through 90.

To calculate the value at each angle, a .coe file was created to load into memory with an eight bit scaled value for alpha. Thus, the eight bits can represent a value from -64 through 63. When alpha is equal to one, the output value is 32. Thus, possible values can range from -2 to 2 after scaling. How these 8 bit values actually work to scale the lift force will be discussed in that module's description. Figure # shows a wave form of the Alpha1 module working.

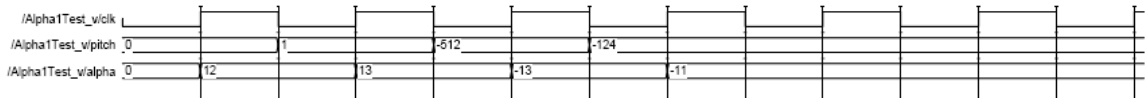

Figure 9: Alpha values for various values of pitch.

Module Altitude_Vel.v

To fly an airplane, it helps to keep it in the air. The physics simulation approximates the motion of the plane by adding small percentages of the altitude every frame of the display. This module takes in the current vertical velocity value and computes a small change in altitude to be added to the altitude to simulate the height change between the current frame and next frame of the display. The value for velocity is originally in miles per hour. The velocity is multiplied by 22 and divided by 1024 to get it roughly to a value in feet per frame. Del_Altitude is assigned to the top 11 bits of the velocity value in feet per frame. The bottom 8 bits of values in the major physics engine, and the auxiliary modules such as this, are basically treated as a fractional value for the value represented by the higher bits. Del_Altitude is then sign extended by assigning its higher bits to the sign bit of the velocity value. A short wave printout for this module can be seen in Figure 10.

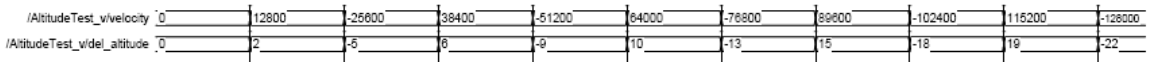

Figure 10: Wave form for the Altitude_Vel.v module.

Module Cos.v & Sin.v

The coregen generated by xilinx to handle the the cosine and sine functions don't actually output values between 0 and 1 and don't even take in values for angles on a 360 degree scale. The angles used in the physics engine are all 10 bits and are treated as one complete circle, thus the circles have 1024 degrees. A value of 512 corresponds to an angle of 180 in the conventional understanding. The output from the cosine and sine roms were 9 bits and scaled similar to how the alpha was scaled for the lift equation's coefficient. Thus a value of -128 corresponds to -1. The CosTest and SinTest modules take care of the actual scaling of the input value by the sine or cosine of the input angle. Thus, after getting the 9 bit scaled value from the rom, that number is then multiplied by the input, and then divided by 128 before being set to the output of the module.

Module DelVel_Force.v

Similar to the Altitude_Vel module, the velocity increments as a function of force, analogous to altitude incrementing as a function of velocity. The well known physics equation F = MA can also be discretized to the equation:

$$\frac{\hat{a}t * F}{M} = \hat{a}V$$

The delta t is simply the time between frames on the VGA. M, referring to the mass of the plane, has simply been held to 1 for the physics simulation to simply the calculations. The delta t is roughly 13 milliseconds or $\frac{1}{75}$ seconds. This value is approximated to $\frac{1}{64}$ seconds to allow for a power of 2 in the denominator. Thus, delta V values are simply the value for Force bit shifted by 6 and subsequently sign extended. A wave form can be seen in Figure 11 showing various input values for Force and the resulting delta Velocity.

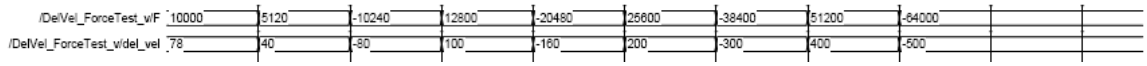

Figure 11: DelVel_Force wave form showing delta Velocity values resulting from different forces.

Module Drag.v

One of the most annoying factors to any object that goes fast is drag. It is an inevitability from living in a fluid. The force of drag is dependant on a couple factors. There is a drag coefficient and a similar value representing the surface area of the object in the direction the drag is being applied. These values were held to 1 to simplify the calculations. Drag is also dependent on the air density, or "rho", however for this physics engine, rho is hardwired to a value of 6 (the highest possible value in the physics world) in order to get a dynamic relationship between the drag force and the lift and thrust forces. Thus, as the plane travels up and down, terminal velocities change as would be expected in normal flight. Once all the many values are multiplied together, the possible bits as a result of the multiplications is 42 bits, however, the value of force needs to be

kept down to 17 bits, so the actual drag force is assigned to only the top 17 bits of the 42 bit number. The drag force also must be signed because if the plane ever has a vector component of velocity in a negative direction (such as down when descending), then drag would be pushing up on the aircraft, rather than down. A wave form of the drag force as a function of different velocities can be seen in Figure 12.

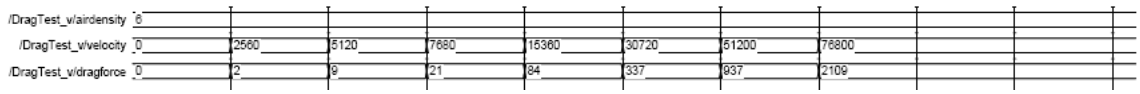

Figure 12: Drag showing values for Drag as a function of different Velocities.

Module Lift.v

The lift of the aircraft makes the plane what it is and this force acts similar to how the force of Drag works. It is proportional to the square of the velocity and some coefficients. The main difference is the variable value of alpha. The alpha value varies with the pitch of the aircraft and is calculated as mentioned in the description of the Alpha1.v module. Alpha is simply an input to this system, so as far as this module is concerned, it is simply a multiplication of all the values and then a bit shift division to cut the value of lift down to 17 bits. Although the plane can go upside down and thus generate a negative lift, this with not be handled in this module. A separate module will take care of taking from each force the components in the directions as a result of the pitch and roll angles: forward, lateral, and vertical. An example of the Lift module working correctly can be seen in the wave form in Figure 13.

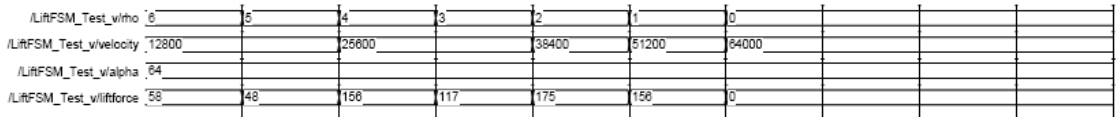

Figure 13: Lift Module showing values for Lift as a function of different Velocities, rho, and alpha. For simplicity, alpha is held at value representing a pitch of roughly 25 degrees.

Module PitchOmega.v

The pitch of the aircraft is controlled by the yoke or, in our case, the forward and backward motion of the individual in the flight jacket. As the pitch of the individual increases (simulating pulling back on the yoke) the angular velocity of the aircraft should increase. Thus, the angular velocity of the rotation of the length of the aircraft is proportional to the angle of the yoke. This axis however changes direction with the roll of the aircraft. A clear example of this effect can be seen if the aircraft is flying perpendicular to the ground. If the pilot pulls back on the yoke, the plane will no longer increase to point more towards the sky, but rather turn in the direction its facing. Thus the compass direction will change, not the nose's angle above the horizon.

Therefore, when the pilot pulls back on the controls, both the angular velocity of pitch and of yaw (or compass direction) will be affected proportional to sine and cosine

values of the aircraft's roll. The values for pitch and yaw angular velocities as a result of changes in the control's pitch and roll angle can be seen in Figure 14's wave form.

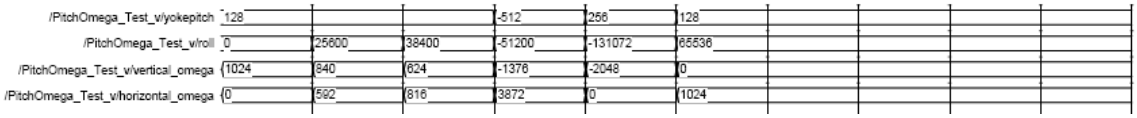

Figure 14: PitchOmega displaying both a vertical (pitch) omega and a horizontal (yaw) omega as the pitch of the yoke and the roll of the aircraft change.

Module RollOmega.v

The roll of the aircraft, like pitch, is controlled by the yoke. This angle is controlled by the individual's side to side motion in the flight jacket. Unlike pitch, the aircraft roll's angular velocity is directly proportional to the angle of the yoke; it does not depend on any other angles or forces. Thus, the output of this module, the resulting angular velocity as a result of the pilot's side to side motion is simply a multiplication of the current pilot's position with a constant that was determined experimentally to ensure the best simulation of flight. The values for the roll's angular velocity as a function of the input can be seen in the wave form in Figure 15.

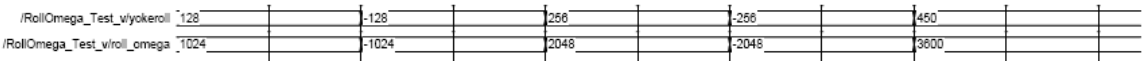

Figure 15: RollOmega takes in a value for the yoke's roll angle and gives a subsequent roll omega or angular velocity.

Module Thrust.v

Thrust is what makes the plane go. The thrust of the plane is proportional to the rotation of the throttle. If the throttle is all the way over, then the thrust should be at 100%. The thrust of the plane is produced by the flow of air through the engine and then it is accelerated out the back of the engine. Both Drag and Thrust then depend on the density of the air. Because of complications with the drag's dependence on the air density rho, the value of rho is hardwired within the Drag module. Therefore, to maintain a relationship of increasing terminal velocities (decreasing drag) as the plane increased in altitude and the air density decreased, an inverse dependence of the value of thrust on rho was added. Therefore, as the plane gets higher, for the same input of throttle, the thrust force increases. This relationship can be seen in Figure 16 showing a wave form of the simulation with different throttle values and rho values.

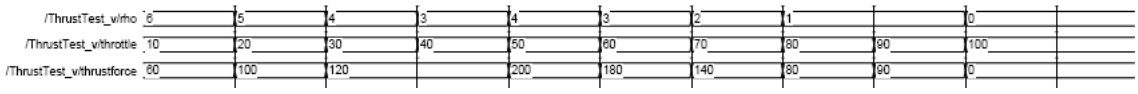

Figure 16: Thrust calculating values for thrust off of the throttle value input and the value for air density, rho.

Module Weight.v

This module computes the weight of the object. Assuming a constant value for gravity, which is reasonable between 0 and 30,000 feet, the force of weight acting on the plane is constant. This weight could have simply been hardwired inside the major module calculating all the physics, however, for debugging purposes, it was easy having a separate module that took care of this value.

Module VertHorzForces.v

There are four main forces that act on the airplane and each of those has been discussed in each module in which they are calculated, however, when dealing with a simple simulation, it is beneficial to break these four main forces up into components in the forward, lateral, and vertical directions. To do this, the thrust force, lift force, and drag force were broken up into two or three vectors each.

The thrust force can be in both the vertical direction and forward direction. The two vectors are related by the sine and cosine values of the pitch angle of the plane. The lift force can act in all three directions, however, by only using the forward velocity to calculate the lift force, this limits the lift force to only the vertical and lateral directions. Thus, to get these vertical and lateral components, sine and cosine calculations must be made based on the roll angle.

Drag is experienced in every direction and it is relatively difficult to get all of the vector values simply based on the current angles. Thus, instead the Drag module is instantiated three times, calculating explicitly each component based on the components of the plane's velocity in each direction.

Once all the components were calculated, simple addition was able to give the total values for force in each of the three vector directions. In Figure 17, the relationships between drag, thrust, lift, weight, pitch, and roll can be seen as the two angles are changed in the wave form.

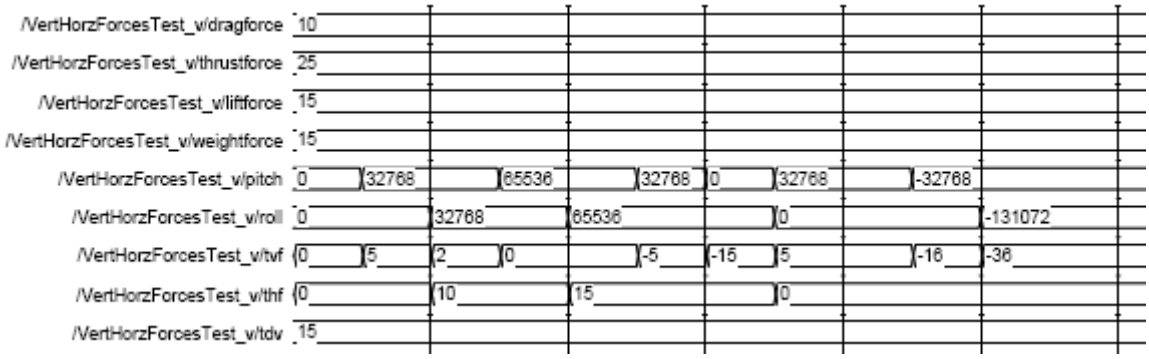

Figure 17: VertHorzForces calculates the total forces in 3 separate vector directions based on input forces and the attitude (pitch and roll) of the airplane.

Module Rho.v

Many of the forces that act on the aircraft are dependant on the density of the air at the altitude the plane is at that time. This value is usually a decimal value that changes as the altitude changes. In order to not have to deal with the decimal precision, rho was held between 0 and 5 and compensated for in the constants in the calculations for thrust, lift, and drag.

This module takes in a value for altitude and depending on that value, rho is output as a value between 0 and 5. Because Rho.v is a true finite state machine, it needs to be clocked and thus the pixel clock is one of its inputs. An example of this finite state machine working correctly can be seen in the wave form in Figure 18.

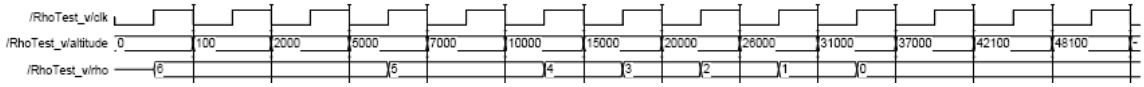

Figure 18: The values for Rho at the different values for altitude.

Physics Debugging

The physics required a fair amount of debugging, however most of the debugging had to deal with getting the plane to act physically realizable. Even if all the modules are working correctly, if the value for the weight force is much bigger than any of the other force's values, the plane will fall and nothing can stop it. Thus, much time was spent tweaking powers of 2 and incrementing and decrementing the many coefficient for the many forces and angular velocities.

One big problem occurred in the calculation of the force vectors and how they act on the plane. Originally the forces were simply four forces. However after miscalculating the vector values from the angles numerous times, the physics engine was changed to simply deal with components of these four forces. Thus, inevitably there were more values to keep track of, but the simplification of the calculations ultimately saved the day and allowed for realistic force relationships.

There were no main hardware issues to debug. To do this again, deciding to break everything up into discrete vectors earlier on would have proved very helpful. Being more careful to keep the many forces to similar values with the constants early on would have also proved helpful.

Displaying the State of an Airplane in Flight – WonRon Cho

Display

The LCD screen is the visual part of this system's user interface. When the user is flying the plane, the screen shows a simplified version of a flight-console. The attitude indicator, in which the ground is represented as brown and the sky is shown as blue, shows the horizon. Different values of roll and pitch change the position of the horizon in the attitude indicator. The compass informs the user about the heading of the plane. Two bars on the far left and far right of the screen show how much power (throttle) is being

put into the system. ASCII characters are rendered as they are needed to display various words numbers on the flight console.

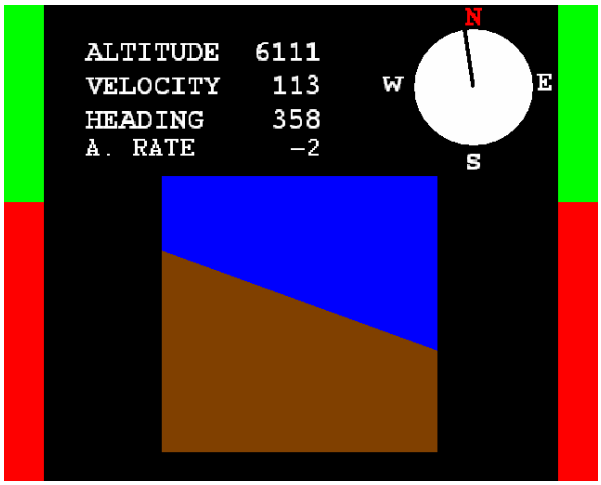

Figure 19: The user sees the flight console during flight.

When the user flies too high or too low, the system enters crash mode. At this point, a “Game Over” screen replaces the image of the flight console, signaling that the game is over.

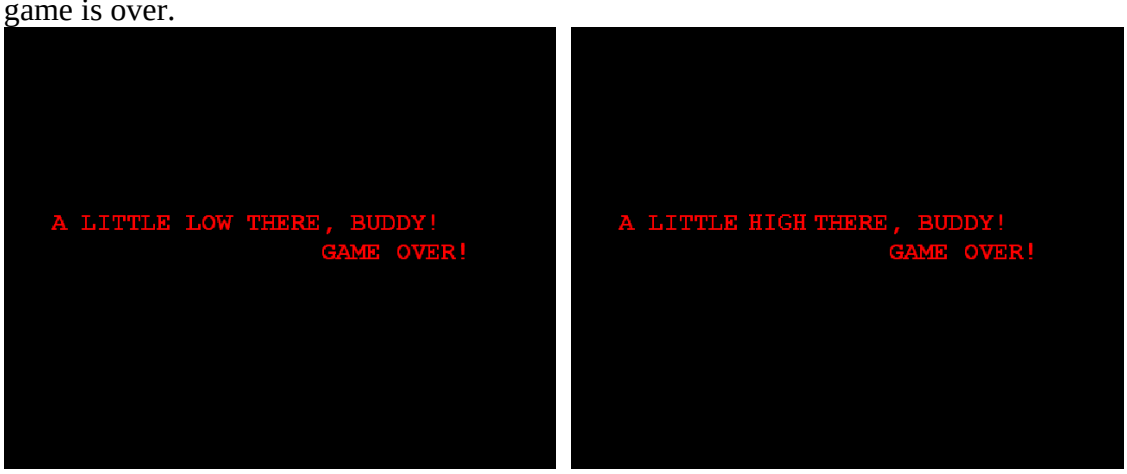

Figure 20: These two screen show up when the game is over.

The display is set up in 640x480 VGA and is clocked by a 31.5MHz pixel clock. The pixel clock is also used to time modules outside of the display block. The various modules work together to produce the RGB values of each pixel on the screen and sends them to the 6.111 labkit.

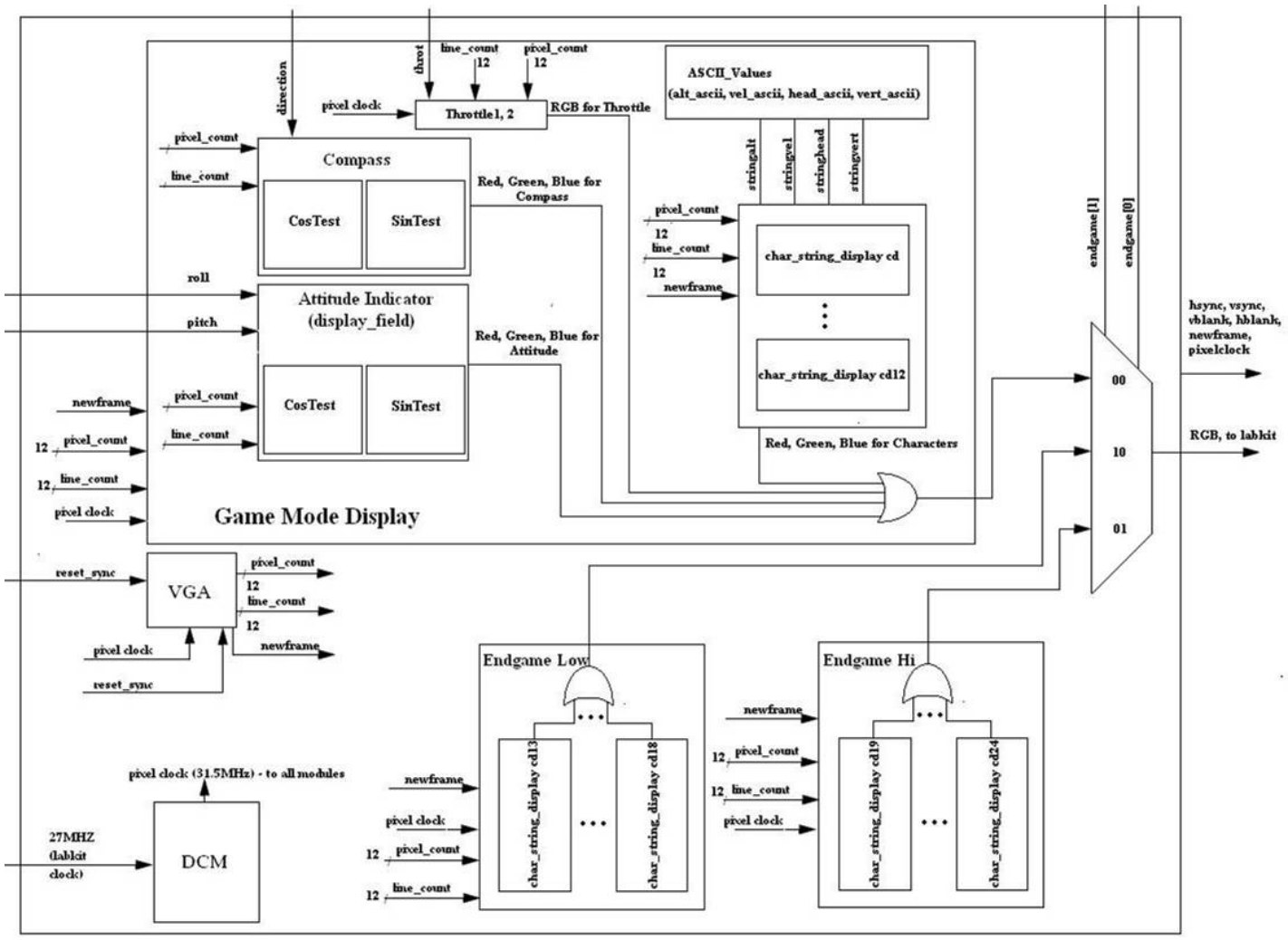

Figure 21: The display module takes values from the physics and sensor modules and outputs appropriate RGB values to the screen.

VGA

The VGA module is a controller for VGA video generation. The specified display for this system is 640-pixels by 480-pixels at a refresh rate of 75 frames per second. In effect, the VGA controller is a finite state machine that determines the values of various sync and blank signals at different line_count and pixel_count values. The pixel_count determines what horizontal point on the screen is currently being outputted. The line_count determines its vertical position. When hsync and hblank are both high, the pixel_count value is between 0 and 639, both hsync and hblank are high. These signals tell the screen to display a color on the screen. When the pixel_count is between 640 and 655, hsync is high and hblank is low. This period is called the front porch, which is not displayed on the LCD. When pixel_count is between 656 and and 751, both hsync

and hblank are low. This period, the sync pulse, is used to retrace to the left edge of the screen and switch to the next line of pixels. When the pixel_count is between 753 and 799, hsync is high and hblank is low. This period is the back porch and the pixels drawn are not displayed to the screen. After the back porch, the process starts over again. At this point, the line_count is incremented by one. The vsync and vblank signals also change according to the line_count. When the line_count is between 0 and 479, both vsync and vblank are high. The front porch of the vertical signal is from line_count 480 to 490. During the synch pulse period, diagonal retrace (to get a new frame) from the lower right corner to the upper left corner of the screen occurs. After the vertical back porch from line_count 493 to 525, the process begins all over again.

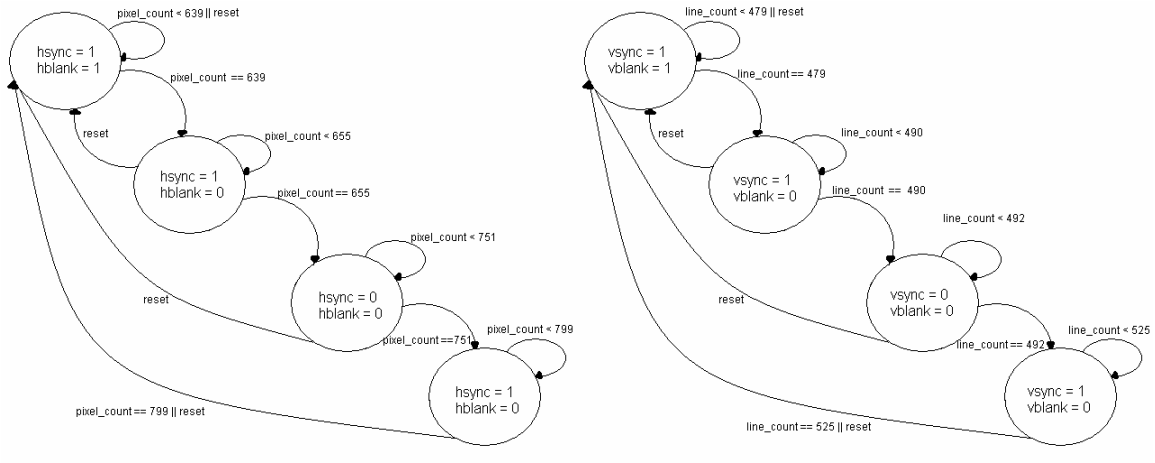

Figure 22: The figures above show how to sync and blank signals are generated according the current line_count and pixel_count.

DCM

The Digital Clock Manager (DCM) is module built into to the 6.111 labkit. In this laboratory exercise, the DCM serves as a multiplier that takes in the 27-MHz labkit clock and generates a new clock signal with a frequency that is a multiple of the input frequency. In this exercise, the VGA format is 640X480 at 75 frames per second.Because 800*525*75Hz is 31.5MHz, the DCM converts the 27-MHz labkit clock to a 31.5MHz pixel clock that is used to clock all the modules of the Pong system.

Attitude Indicator (display_field)

The Attitude Indicator takes in as input roll and pitch from the physics module and pixel_count and line_count from the VGA module. This module is responsible for drawing the horizon and filling in the ground and sky with their respective colors. If pitch is positive, the user should see more sky than ground. When the pitch is negative, the user sees more ground. The horizon should be a line with slope equal to the current roll of the plane. In order to create a line at arbitrary angles, the simple equation line_count = m*pixel_count + b is not substantial because using Verilog, this formula only deals with integer valued slopes. The algorithm that is eventually adopted to draw the horizon takes

advantage of the fact that drawing such a line is like drawing a line passes through the center of a circle. Knowing Cosine (roll) and Sine (roll), the coordinates of the points {(x1, y1), (x2, y2)} at which this line touches the circle can be found. Cosine (roll) and Sine (roll) can be calculated with Corgen modules built into Xilinx. If x1 and x2 are radius* sin (roll) and –radius*sin (roll) and y1 and y2 are radius*cos (roll) and radius*cos (roll), equation (x-x1)/(y-y1) = (x-x2)/(y-y2) holds. Now a line with slope equal to any integer angle between 0 and 359 can be drawn.

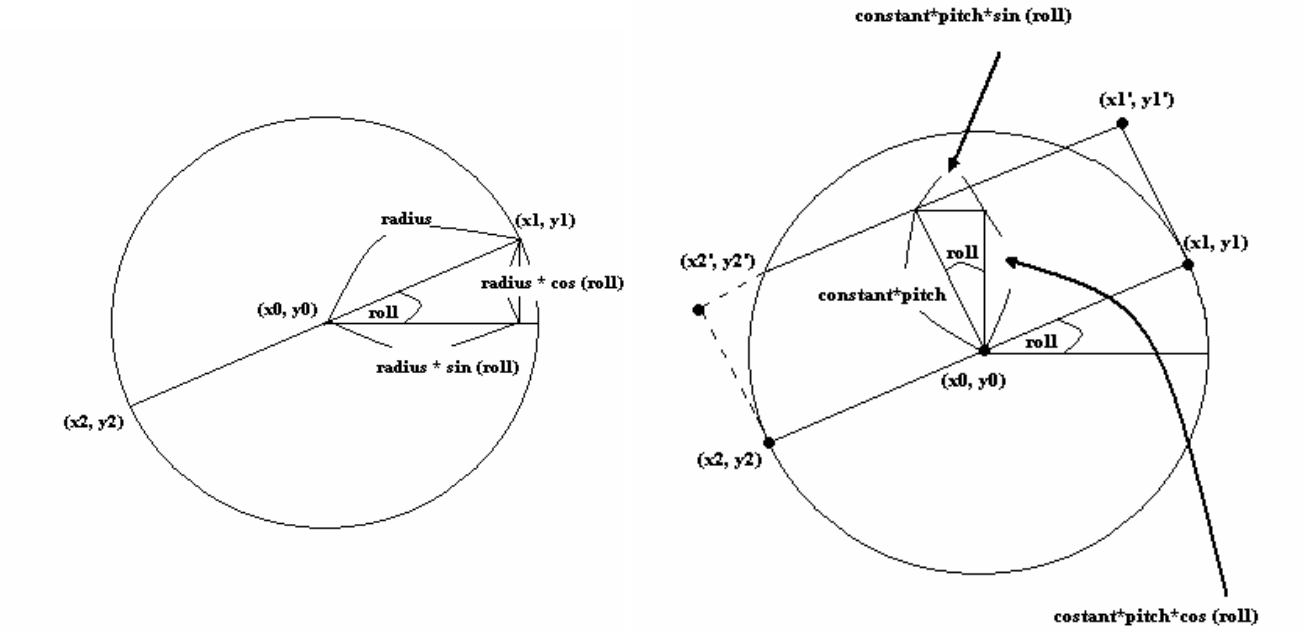

Figure 23: Pictures representing the calculations involved in displaying the attitude indicator

To take care of how the horizon moves as the pitch changes, the line has to be shifted around the coordinate plane accordingly. And the fact that on the LCD screen, the y-axis gets more positive going down must also be taken into account. Therefore, when the pitch changes, the horizon moves a distance constant*pitch in direction perpendicular to the line itself. The new positions (x1', y1') and (x2', y2') will be

{x1+constant*pitch*sin(roll), y1-constant*pitch*cos(roll)}

and

{x2+constant*pitch*sin(roll), y2-constant*pitch*cos(roll)} .

The complete equation of the line will be (x-x1')/(y-y1')=(x-x2')/(y-y2'). To color in the sky and ground with their corresponding colors, the module checks to see if this statement is true: (x-x1')*(y-y2') < (x-x2')*(y-y1'). If it is true, the module checks if the pitch is between -90 and 90 degrees. If this is true as well, the space is colored brown. Else, it is colored blue. If (x-x1')*(y-y2') > (x-x2')*(y-y1') is true instead and the pitch is between -90 and 90 degrees, the space is colored blue. Otherwise, it is colored brown. As a result, two horizons are drawn, allowing the plane to fly upside down, seeing the earth above the sky.

Compass

The compass uses an algorithm very similar to the one used to implement the attitude indicator. Two parallel lines are drawn through the origin of a white circle and the space between them is filled with black. Another line is drawn to cover up half of the black line with white. What results is what looks like a compass with a needle. When a different angle is given to the module, it can draw the black bar with the given slope and act as a compass for the flight console.

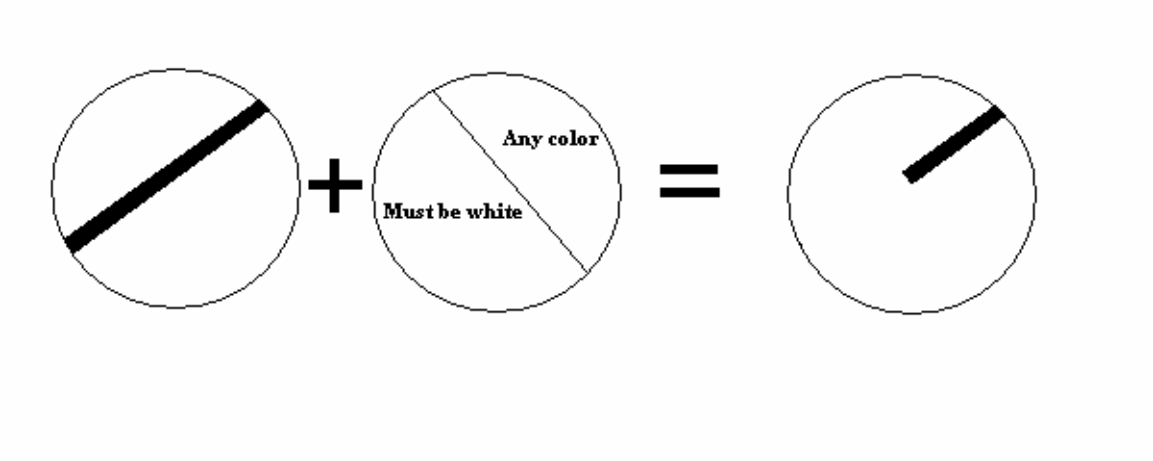

Figure 24: The compass is implemented in 2 steps.

Char_String_Display

To display the ASCII Characters on the screen, a module written by Professors C. Terman and I. Chuang was used. The module takes a string of 8 ASCII characters and looks up the bitmap of the character in a ROM. The module is responsible for rendering these characters on the screen. The font for the ASCII characters is stored in ROM using the method on Lecture 9, Slide 13 fro 6.111 Fall, 2005. The memory contents are created in notepad and loaded into Coregen ROM as a coefficients file.

Throttle

The throttle is implemented as a simple horizontal line that divides a long bar into a red and a green part. The module takes a number directly from the throttle potentiometer readings and shifts the line up and down on the screen. Because the y-axis of the LCD screen increases going down, the position of the line is the height of the bar minus the throttle.

ASCII_Values

Since the values (Altitude, Speed, Heading, Ascent Rate) that the physics modules produce are in two's complement, there must be a module that takes these two's

complement numbers and converts them into strings of characters. In effect, the module is looking to see how any ones, tens, hundreds, thousands, and ten thousands the two's complement number has if it were to be represented in decimal. The module first checks to see if the incoming number is positive or negative. If the number is negative, the module remembers that fact and will proceed to do other calculations with the negated version of the number. If the number is positive, the module proceeds with the calculations with the number as it is. When a new number is received, the module calculates the difference between the new number and the number that the module had been dealing with before. If the difference were positive, the module will count up, keeping track of the number of ones, tens, hundreds, thousands, and ten thousands. If the difference were negative, the module will count down, also keeping track of the number of ones, tens, hundreds, thousands, and ten thousands. What will be shown on the screen will depend on where the two's complement number falls on the number line. The appropriate string is then sent to Char_String_Display.

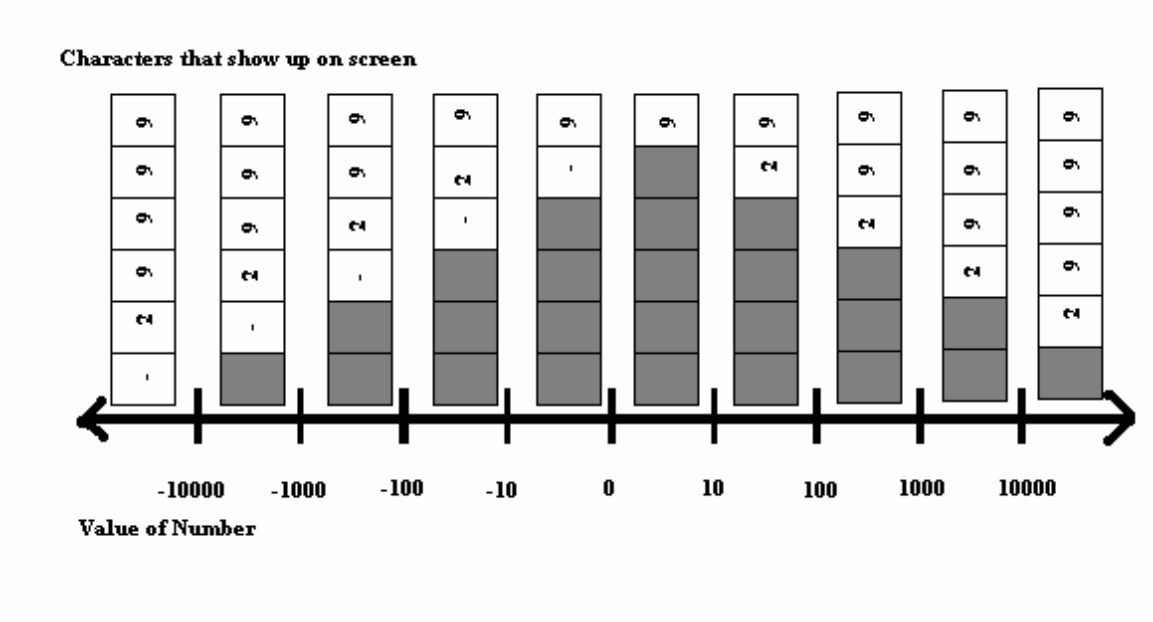

Figure 25: Depending on where the number falls on the number line, the negative sign moves.

Debugging

The VGA was debugged in the 6.111 Pong Lab. A test bench is included in the appendix. The following waveform results:

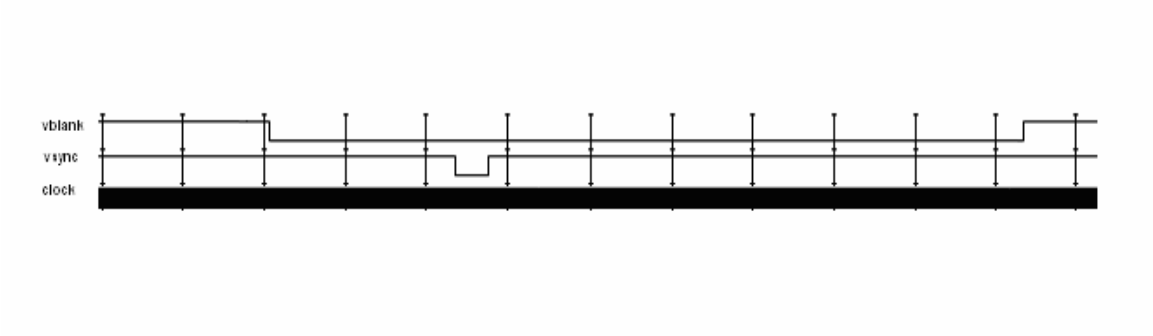

Figure 26: This shows the vertical blank and sync signals.

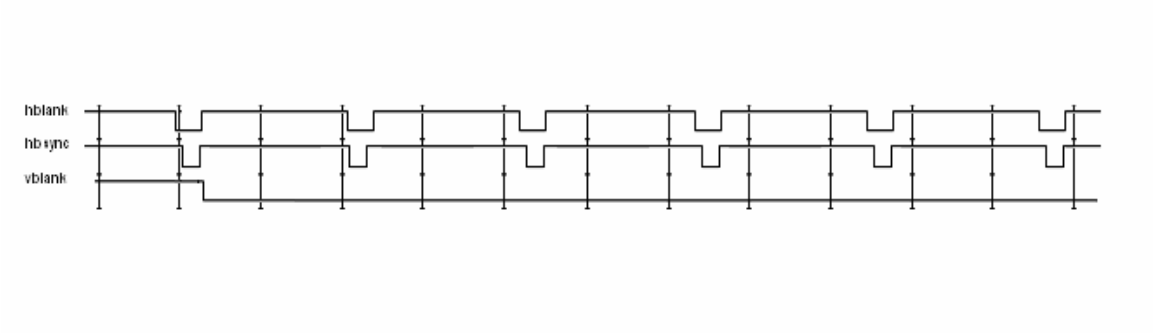

Figure 27: This shows the horizontal blank and sync signals.

To test the attitude indicator, compass, throttle, and various numbers were hooked up to counters Roll.v and Pitch.v that incremented and decremented when corresponding buttons are pushed. The functionality of the Attitude Indicator is verified by watching the horizon scroll with changing pitch and rotate with varying roll. By feeding both the compass and “Heading” with the output of Roll.v, the number that shows up next to the entry “Heading” should be the angle at which the compass needle points. Similarly, by feeding the throttle and “Speed” entry with the output of the Pitch.v, the fullness of the throttle should go up as “Speed” increases.

Conclusion

Implementing blocks individually ensures modularity of the digital system's various components. Highly modular systems are easier to debug. Furthermore, because module-to-module interfaces are the only places where the blocks interact, design changes within each module do not affect the internals of other modules. Since this project involves both analog and digital components that must all communicate and be synchronized with each other, building the sensor blocks, physics modules, and display field separately simplified what might have been a daunting task. Additionally, the modular structure of this system ensures minimal propagation of errors and allows easy pinpointing of modules that are out of synchronization.

After finishing each individual part and interfacing them successfully, a fully functional, body activated, physically realizable flight simulator was created. Bon Voyage!