Virtual Softball

6.111 - Fall 2017
Katie Shade and Melinda Szabo

1 Overview 2

2 Goals 3
- 2.1 First Base (Baseline) 3
- 2.2 Second Base (Expected) 4
- 2.3 Third Base (Stretch) 4
- 2.4 Home Run 5

3 Higher Level Block Diagram 5

4 Subsystems 5
- 4.1 Finite State Machine (FSM) 5
- 4.2 Bat Physics 7
  - 4.2.1 Position 7
  - 4.2.2 Slope 8
- 4.3 Bat Signals 9
- 4.4 Serial Communication 10
- 4.5 Wireless Communication 11
- 4.6 Gameplay Graphics 11
  - 4.6.1 Shapes 12
  - 4.6.2 Ball Movement 13
  - 4.6.3 Swing Timing Representation 13
- 4.7 Post Swing Graphics 14
- 4.8 Trajectory 14
- 4.9 Background Picture 15
- 4.10 Adjustable Ball Speed 16
- 4.11 LED strip 16
- 4.12 Scoring 17
- 4.13 Externals 17
- 4.14 Physical Setup 18

5 Testing and Debugging 19
- 5.1 Simulations 19
- 5.2 ILA and Oscilloscope 19
- 5.3 Manual Mode 19
- 5.4 Indicator Displays 19

5.5 Calibrating Parameters 20
6 Challenges 20
6.1 Bat Angle Calculation 20
6.2 Bat Angle Representation 21
6.3 Background Picture 21
6.4 LEDs 22
7 Design Decisions 23
7.1 Nexys 4 vs Labkit 23
7.2 IMU vs Camera 23
7.3 Communication Protocol 24
7.4 User Interface 24
7.5 LED 25
8 Reflections 25
8.1 Katie 25
8.2 Melinda 26
9 Conclusion 26
9.1 Start Early and Ask for Help 26
9.2 Modular Design 27
10 Acknowledgements 27
11 Appendix 27
11.1 User Experience 27
11.2 PC Code 27
11.3 Verilog Code 28

1 Overview

Softball is a team sport played by people of all ages. It relies on both strategy and skill in order to succeed. In the winter months, it is often difficult to practice outside due to cold weather. This virtual softball game allows users to swing a real bat and practice their hitting timing from the comfort of the indoors. It also gives non-softball players the opportunity to experience a life-like version of the game with real equipment.

Users will stand with a bat in front of a screen. When the game begins, the user will signal that they are ready for a swing and the pitch initiates with the ball appearing at a random location in the strike zone. As the ball approaches the batter, this will be visually cued by the ball changing

color and growing larger. Additional visual cues to assist the user will be a strip of LEDs that indicate the changing distance of the batter to the ball as it approaches.

The user is expected to swing the bat when the ball is in a hittable position. By interfacing with an IMU, real bat position and angle from an accelerometer and gyroscope can be recorded, and a virtual representation of the bat will be reflected onto the screen in front of the user after the swing. The goal is to give the user feedback on both their hitting timing as well as their batting mechanics in the form of the bat angle while swinging. The screen will indicate approximately where the bat hit the ball (up, down, left, right, center) or if they missed the ball due to timing or incorrect bat angle. If the ball is hit, a future plausible trajectory of the hit ball will be shown in order to make the game more fun and give better visual feedback to the user.

There are a few additional features added to this game to make it more enjoyable for the user. A real picture of a softball pitcher will be the background to the game, and the full size game will be playable on a TV. The score will also be displayed on the labkit, denoting the total hits and misses for each user. Users also have the option to choose their difficulty. The more difficult game has a faster ball speed and less time for the user to react. Additionally, a foot pedal is used instead of a button press to signal that the batter is ready for the pitch.

2 Goals

The following list of goals were submitted as the project checklist.

2.1 First Base (Baseline)

1. Calculate angle of a physical bat swing using an IMU's accelerometer readings. Also determine timing of swing from gyroscopic readings.

2. Connect FPGA to IMU via a microcontroller transmitting a stream of accelerometer data using UART over a direct wired connection. Deserialize this input to get real-time values.

3. Graphically represent an approaching ball by blending color and increasing size of a circle.

4. Display a bat object at any angle provided by a real bat orientation.

5. Signal a new pitch and calibration with a button press. Trigger graphical response based on swing timing.

2.2 Second Base (Expected)

1. Randomly change the ball location in the strike zone between pitches. Correctly detect a hit or miss taking into account the ball's location and bat swing.

2. Detect that the user wants the next pitch when they raise the bat by their shoulder. Graphically pitch the ball at this time.

3. Based on the bat angle and timing, show on the ball where the bat would have likely contacted the ball.

4. Determine the angle of the physical bat as it comes through the strike zone using a combination of accelerometer and gyroscope readings, in order to accurately determine the bat angle at high speed circular swinging motions.

2.3 Third Base (Stretch)

1. Transmit IMU data over bluetooth connection to wirelessly communicate with the FPGA.

2. Improved gameplay interface in at least two of the following areas.
    - Include softball-themed backdrop on screen.
    - Add sound effects for ball hits and misses.
    - Make game playable on TV screen
    - Adjustable difficulty can be set by the user before each swing.
    - Provide scoring system

3. Utilize an addressable RGB LED strip to indicate the approaching ball, which is appropriately in sync with the FPGA.

2.4 Home Run

1. Have fun!

3 Higher Level Block Diagram

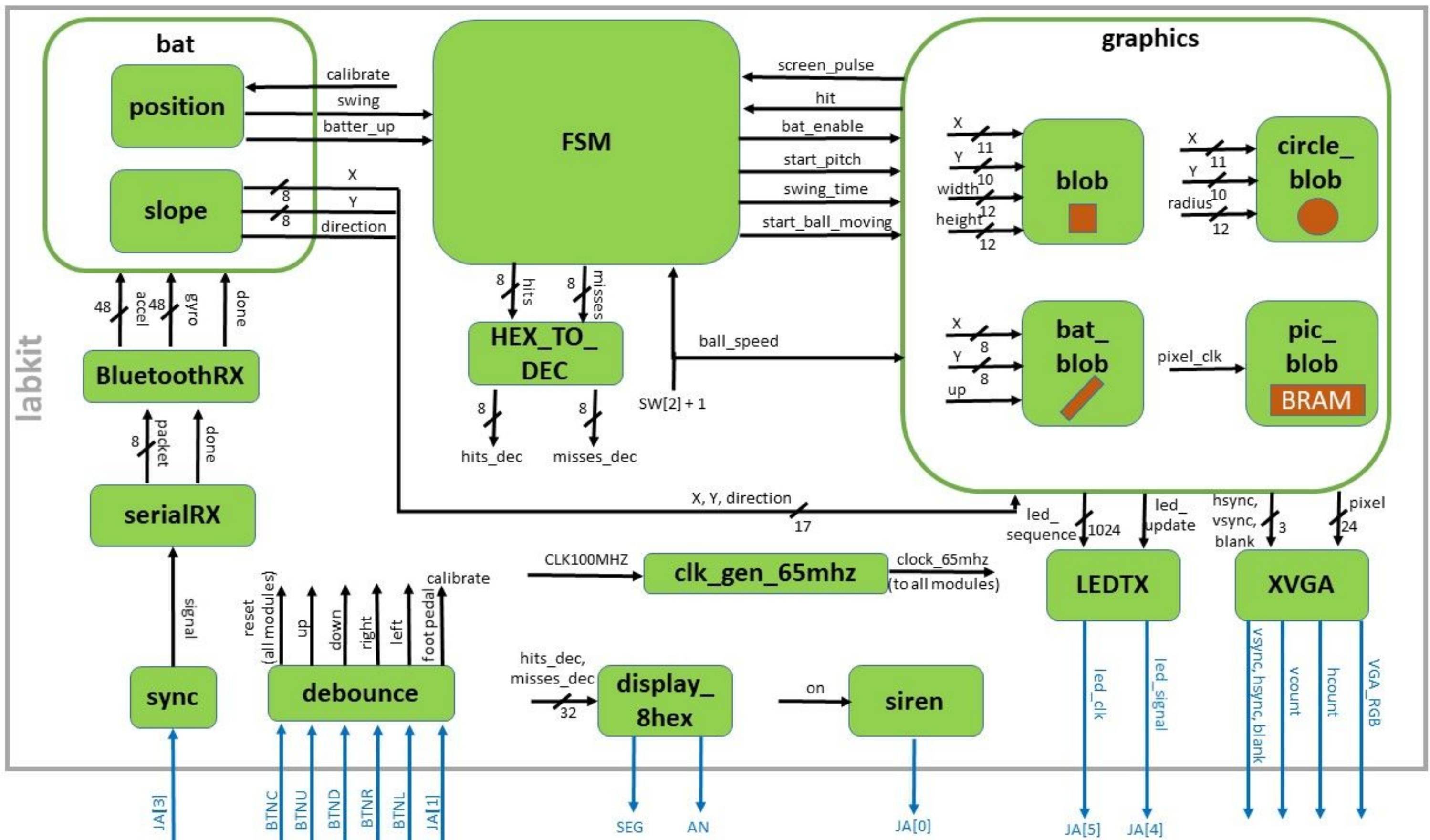

Figure 1. Full system block diagram including all modules and signals, with indicated wire widths. Blue wires represent inputs/outputs through the labkit that interface with physical world, black wires are internal

4 Subsystems

4.1 Finite State Machine (FSM)

The FSM is in control of understanding the current state of gameplay and updating the necessary signals. At a high level, this module takes inputs from both the user and the graphics module. Figure 2 shows the state transitions dependent on external signals. The user is able to signal when they want the ball to be pitched by raising the bat, which changes the state from IDLE to BATTER_UP_WAIT resulting in a one second pause before the ball is pitched. This allows the batter to get ready to hit, and is especially useful when playing in difficult mode.

The FSM then knows when the ball is approaching the batter and changes state to BALL_IN_ZONE, denoting the only state in which the batter can swing and have correct timing for a hit. To determine when the ball switches from just approaching the batter to being in the strike zone and hittable, the FSM receives signals from the graphics module. The graphics module outputs a pulse each time all pixels are refreshed (hcount = SCREEN_WIDTH and vcount = SCREEN_HEIGHT), so by counting the number of pulses, the FSM can time when the ball is a certain size and in the hittable zone. This ensures that what the user is viewing accurately reflects the game state.

If the user swings (external signal from bat module) when the ball is in a hittable state, the state changes to SYNC where the FSM waits for the pulse from graphics that the screen has been refreshed. After this indication from graphics, the FSM enters the SWING state where it waits for the graphics module to analyze whether the ball was hit. In short, the graphics module tells the FSM if there is overlap between the ball and bat for each pixel. If there is, then the graphics module sends a hit signal. If any hit signals are sent for one screen cycle, then the swing was considered a hit and the FSM moves to the HIT state. While in the SWING state, if the FSM receives another screen cycle pulse from graphics, then it will determine that graphics had no overlap between the bat and ball, so the user missed. This puts the FSM in the SWING_MISS state. After about 5 seconds from either the HIT or SWING_MISS state, the FSM returns to the IDLE state, waiting for the batter to signal another pitch.

If the batter swings from the wrong state at any time, the FSM changes to the TIME_MISS state. In this state, the FSM waits for another batter up signal from the user, or it times out after 5 seconds back to the IDLE state.

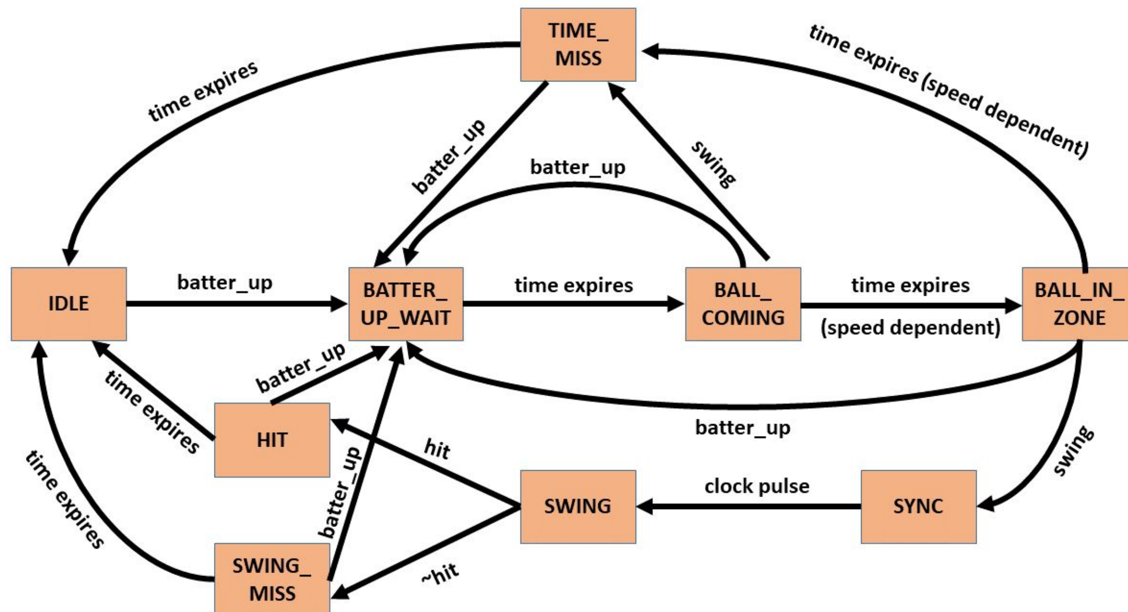

Figure 2. Finite State Machine Transition Diagram

In addition to keeping track of gameplay, the FSM also sends important signals to other modules. It tells graphics when to display the angle of the bat on the screen, based on the current state. It also keeps track of hits and misses so that the score can be displayed to the user. It also determines the exact timing of the swing. In the BALL_COMING state, the FSM determines which side of the ball the user hit based on the how deep the ball is in the strike zone. Again, the FSM knows this information by counting screen refresh pulses from the graphics module. The FSM then sends this timing information to the graphics module.

4.2 Bat Physics

To bring the real bat into a virtual environment, it was necessary to compute an accurate bat angle and position during a full swing from the sensors attached to the bat. The system relies only on data from a 9-axis IMU, streamed through an FPGA input pin, which provides gyroscopic and accelerometer data that allows a calculation of the bat's orientation. The axes on the bat are defined in Figure 3. These definitions follow the right hand rule, with the exception of gyroz, which measures rotation around the negative z axis, not the positive z axis. The IMU is configured to output a signed 16-bit number, with a maximum output of 2000dps for the gyroscope and 2g for the acceleration. The increase in the gyro threshold was necessary for the fast swings present in softball. Once the data is decoded by the serial receiver, acceleration and gyroscopic values are passed into the bat module for processing. The bat module consists of a small FSM to track the different stages of a swing and two submodules that constantly update position and angle calculations based on new data from the IMU.

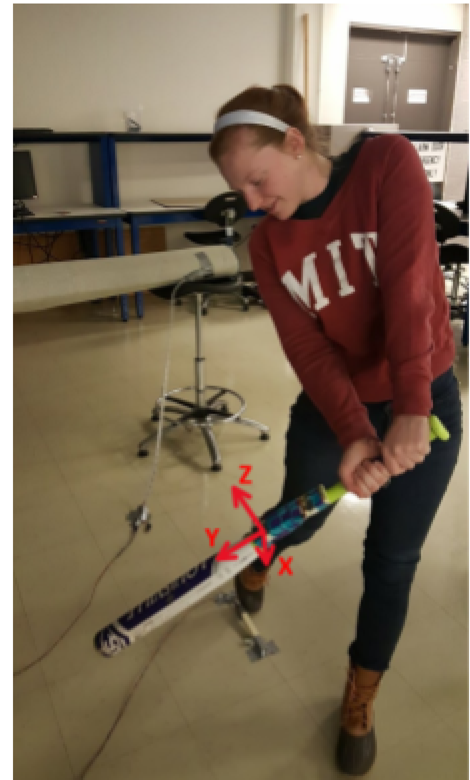

Figure 3. Definitions of IMU axes with respect to bat

4.2.1 Position

The position module needs to calibrate to a starting position, wait until the bat has rotated away, and then indicate once the bat returns to the initial position. This allows the system to fully track a swing from calibration to the single clock pulse of the bat crossing the hitting zone. Position takes in gyroz readings, which is the angular velocity about the vertical axis, to track how the bat rotates around the user and when it crosses the hitting plane, ie returns to the calibrated point. The gyro input is scaled and integrated to estimate the angle from its original position, using the following update equation: $\theta_z[n] = \theta_z[n-1] + g_z[n-1] >> T$. Note that to avoid computationally intensive divisions, right shifts are used, equivalent to dividing by powers of two. Upon calibrating,

the angle is reset to zero and then a swing signal is sent after the angle has changed and returned to zero. Additionally, a range of values (defined as being under some threshold) are considered at the 'zero' position, ensuring that only real motions trigger these signals. An optimum value for the threshold and T were determined through experimentation (see section 5.5 for more detail on this process). Although relying purely on gyroscope values to calculate the angle introduces error due to drift, the effects are negligible since calibration occurs before each hit and the time between calibrating and swinging is only a few seconds. We also found that for some users, a raised bat would also trigger a swing, since the calculated angle becomes zero due to additional rotations. For example lifting the bat directly upwards to the user's shoulder does not result in any rotation about one's torso (z-axis), however the bat will be upside down. To exclude these cases, the zero position is only signalled if the angle with respect to horizontal is within about 60 degrees of the calibration point (ie in front of the user), requiring a similar calculation with gyrox.

4.2.2 Slope

Accurately reflecting the angle of the bat proved to be a major challenge requiring many iterations and tinkering to reach its functional state. On an FPGA, complex trigonometric functions take too many clock cycles, so we chose to represent the bat angle with slope, as a ratio between X and Y values with a positive or negative direction (detailed in section 6.2). When the bat is stationary, all acceleration is due to gravity, which can be used to determine the orientation of the bat. From the IMU, accely values are used to describe Y, since it is zero when the bat is held horizontally and negative when pointing up, positive otherwise, indicating direction of the slope as well. On the other axis, X comes from accelz, which is at its maximum magnitude when the bat is horizontal, since it is perpendicular to the bat, therefore along the same direction as gravitational acceleration. Using a basic low pass equation suffices to calculate the slope for a stationary bat, however to factor in faster movements, an additional high pass term from the integration of gyrox needs to be included to get the combined equation:

$$\begin{aligned}\theta_x[n] &= \left(1 - \frac{\beta}{2^{DIV}}\right)(gyro) + \frac{\beta}{2^{DIV}}(accel) \\ &= (((1 << DIV) - \beta)(\theta_x[n-1] + g_x[n-1] >> T) + \beta(a_y[n-1])) >> DIV\end{aligned}$$

The ratio $\frac{\beta}{2^{DIV}}$ determines the sensitivity to the gyro readings, experimentally found to be around 0.01. The exact order of operations and size of registers turned out to be quite important because if the division occurs first, almost all results will be zero, but with the multiplication occurring first, it is ensured that the buffers are both large enough to not clip the higher order bits. Also note that all registers are signed, so verilog syntax must be used cautiously to guarantee that positive values have a leading zero appended.

With the minor bugs fixed and optimal constants set, the bat could be tracked quite accurately with the merged gyro and accel equation for both X and Y values. However,

as soon as we stepped up to the plate for a few swings, the measurements were completely thrown off due to the centripetal acceleration that the IMU experiences. Since this acceleration points inwards in the negative y direction, it appeared as if there was additional gravity pulling the bat down, resulting in extreme negative unrealistic angles. After discovering this major issue, we attempted many different approaches to compensate for it and did some more research on what solutions may already exist, however none of them seemed practical enough to implement on an FPGA. A similar research project was done in the University of Michigan on tracking a bat swing with an IMU (King, K., Hough, J., McGinnis, R., & Perkins, N. C. 2012. A new technology for resolving the dynamics of a swinging bat. *Sports Engineering*, *15*, 41-52. Web.), but relied on post-processing of the data and complex trigonometric and matrix calculations, which was not ideal for real-time tracking with an FPGA. One of our first ideas was to ignore acceleration readings by lowering the beta ratio while the bat is swinging fast (as determined from the gyro), but this was not reliable enough for the wide variety of possible swings and the complex motion through the air. We also considered relying only on gyro readings, as for position, since they are not affected by centripetal acceleration. We found that these gyro readings were also thrown off during an actual swing due to rotation about other axes, which we were not able to account for using implementable formulas for the FPGA. Thresholding acceleration readings, at say 1g to eliminate any extreme values, was also not enough. We almost decided to switch to a bunting game where the user only determines a stationary angle, but not while swinging.

After weeks of frustration, we came across a couple changes that, combined, could give us accurate results even during a real softball swing. The first fix was setting a constant X value. This eliminated the most extreme angles and did not seriously hinder true reflections of the bat's angle, since we generally expect shallow angles. The next small change that made a huge difference was bringing the IMU closer to user's hands, the approximate point of rotation, thereby reducing centripetal acceleration proportional to the change in radius. Finally, a modification on the equation to reduce the acceleration readings by a factored magnitude of gyroz, presumably proportional to the centripetal acceleration, worked best. It still required a lot of testing to determine all constants, so that spinning around while holding the bat level kept the angle stable. Although not a perfectly accurate equation, our approximations are reasonable enough that the game is playable and nearly reflects the real bat's angle.

4.3 Bat Signals

In the softball game, the ball is pitched once the batter is ready and has the bat raised above their shoulder. Since a calibration signal when the bat is held horizontally in the hitting zone is necessary, we wanted to avoid an additional button press to signal the pitch. Instead, the bat module should use IMU data to determine when a batter is ready for the pitch. The position module uses gyroscopic values to detect when the bat has rotated away from its calibration

point, so the raised bat sensing can be incorporated as an additional state during this movement phase. In this state, position waits until gyro readings fall below a threshold then signals batter_up, indicating the bat is no longer moving, so the user is stationary and ready for the pitch. After a one second delay, the ball appears on the screen. We experimented with a variety of motion detection methods that more heavily constrain orientation and position of the bat, but found that this simpler technique functioned accurately and for the widest range of users.

4.4 Serial Communication

To enable all of the physical bat calculations, 6 degrees of freedom measurements are retrieved from the IMU, an MPU9250 on a breakout board. Instead of wasting weeks defining an I2C protocol and setup sequence on the FPGA, we chose to interface the IMU through a microcontroller, the Teensy V3.2, and define our own serial protocol to communicate with the FPGA (more on this decision in section 7.3). This required defining a protocol to use, enforcing it on the Teensy, and decoding it on the FPGA (see figure 4).

Initially, we implemented the simplest serial stream of all bits the Teensy received from the IMU whenever new data was available. The microcontroller library was based on a public library available for this specific device, requiring only minor adjustments of the sensitivity. The stream code, written in C++ for Arduino, is available in the Appendix (section 11.2). Although communication runs at 115200 baud, the IMU only outputs new values at approximately 400 Hz, so the signal is expected to appear as twelve 8-bit packets (6DOF of 16-bits each) with a start and stop bit, then a few microsecond delay before the next cluster of packets.

To separate the packet reading and cluster organizing functions, thereby allowing for more modularity in the future, two new verilog modules are defined. The serialRX module gets an (synchronized) input signal from the FPGA's IO ports and translates the stream into 8-bit packets from start to end bit, asserting a done signal when ready. Structured as an FSM to handle the IDLE, START, READ, and STOP states, the serialRX behaves similarly to the RS232 module implemented in lab 2b while reading. It shifts in new incoming bits toward the right (LSB first) at a 115 kHz frequency until it has read all eight bits and outputs it as a complete packet. This allows the bluetoothRX module (originally known as imuRX, with slight modifications explained in the next section) to read the inputs packet by packet, and sort them into their proper assignment, accel and gyro x,y,z. This module uses an array to store the six desired readings, assigning them to their respective values. The FSM primarily alternates between reading the least significant bytes (which actually arrives first) and the most significant, then concatenating before updating the array. Once a whole cluster of packets has been read, indicated by a long delay of no new packets, a done signal is asserted and it returns to an IDLE state. Although initially not the most robust system, this setup sufficed to consistently transfer data on a wired connection of over a meter of unshielded wire.

4.5 Wireless Communication

A long wire connecting the bat to the FPGA was not ideal, as it would whip around during the swing and eventually come loose or snap. We had always planned to transfer our serial communication to a wireless module, but weren't sure about the difficulties it may bring. Ideally the transition from an already existing UART communication protocol would just require passing the data over the air instead of a wire, but it introduced some underlying issues with the original protocol. The first step was identifying the components needed to enable wireless communication. Two BlueSMiRF devices from SparkFun are necessary, which run at 115200 baud off of the same 5V supply as the Teensy. Additionally, a 3.7V lithium ion polymer battery with a charger and DC step up converter to 5V is needed to supply power. With the hardware wired up and secured to the bat, the sparkfun guide can be used to setup the bluetooth connection and pair the modules to stream anything directly.

Unfortunately, the reading seemed all jumbled since packets arrive as clusters of clusters, meaning that three full IMU readings would be sent immediately after each other, with a long pause in between. This confused the imuRX packet decoder, which was anticipating a delay between each cluster, not after three sequential clusters, which may also be broken mid-cluster. This unpredictable nature of bluetooth transmission required redefining the protocol to be more robust to such uncertainties. Learning from how UART sends each 8-bit packet, we decided to include a start packet, before sending the cluster of twelve IMU packets. Instead of relying on a long delay to indicate a new cluster, the bluetoothRX waits for the start byte to reset its byte counter, which is then used to index into the array of IMU values. For this approach to work, the Teensy must ensure that the specific start sequence is not sent coincidentally as an IMU reading, so a check-and-replace code selecting for the start byte is included on the Teensy before transmitting the bytes to the bluetooth device. Only the bottom byte needs to be replaced, making an insignificant change from 0x__11 to 0x__10, which does not impact the angle detection. The bluetoothRX module needed to be modified to detect the start byte and then stay in the reading lower bits state while resetting the counter.

Figure 4. Communication system high level diagram. The bluetooth modules were originally replaced with a long wire between Teensy TX pin and FPGA JA[3].

4.6 Gameplay Graphics

This section will discuss the overall visual cues provided by the graphics module to allow the user to play the game.

4.6.1 Shapes

The three main shapes responsible for gameplay include blob (rectangle), circle_blob (a circle), and bat_blob (an angled rectangle). Each of these shapes is overlaid on top of a real picture of a softball pitcher, as shown in figure 5 below.

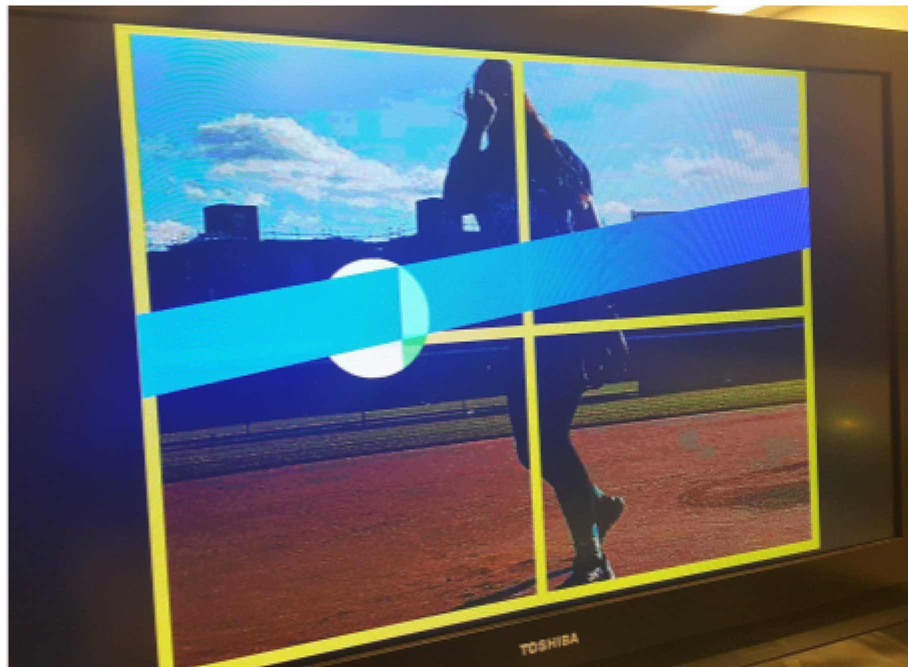

Figure 5. Graphical display with picture in background, strikezone outlined in yellow, blue angled bat, and ball indicating point of contact and hit timing

The rectangular blob was used to make 6 yellow lines that represent the outline of the strike zone. This allows the user to have an idea of where the ball is coming into the zone, and the bat angle required to hit it. This module was mainly created from Pong (Lab 3) and adjusted for the purposes of this game.

The circle_blob was used as the softball. It requires the use of extra registers to handle the multiplication involved in defining a circle. No further pipelining is necessary, as a few clock cycle inconsistencies will not be noticed by the user. The module takes the center of the circle and radius as input, allowing the softball to move and change size. The ball begins to appear when the fsm sends the start_pitch signal to the graphics module.

The bat_blob represents the bat as it comes through the strike zone. The bat is anchored to the center of the left edge of the screen, and the module takes as input an X component, Y component, and up or down indication. These inputs represent the slope of the bat's angle as it comes through the zone. The bat_blob module utilizes multiple equations that use multiplication instead of division to determine the bat outline lines. The module takes into account whether the bat angle was up or down so that the bat_blob module does not have to use signed numbers, and instead uses separate, but related, equations for the different directions. The equations also center the bat vertically and maintains a relatively constant bat width despite different angles. The bat appears according to the bat_enable signal sent by the fsm module when it is in particular states.

Here is an example of the equations that defined the angle for an upward swing:

```
((x_accel * ((SCREEN_HEIGHT>>1) - (vcount + (BAT_WIDTH >>1 )))) <= (y_accel * hcount))
&& ((y_accel * hcount) <= (x_accel * ((SCREEN_HEIGHT>>1) - (vcount - (BAT_WIDTH>>1)))))
```

4.6.2 Ball Movement

The movement of the softball is designed to indicate to the user that the ball is traveling closer to them and when it hittable. The graphics module does this by both increasing the radius of the ball as it flies towards the user as well as changing the color of the ball. The ball's width increases relative to the speed set by the user. With a faster speed, the width of the ball increases more quickly because this represents the ball traversing a distance more quickly. The radius of the ball increases by the speed on each screen pulse (refresh of all pixels). The color of the ball is determined by a combination of red and green components. The ball starts initially all red and then gradually changes to all green. The speed of the color change is also determined by the speed of the pitch and corresponds to the ball width. A greener ball tells the user that the ball is closer to them. The ball is also larger at this time, so it is easier to record a hit. Once a swing is registered by the bat and signaled to the graphics module by the FSM module, the ball stops growing.

To further enhance the user experience, the ball for each pitch is in a random location within the strikezone. In order to determine the random location of the ball, there is a continuous 8 bit counter that increments on each 65 MHz clock edge. Using the bits of this clock to simulate "random" numbers, the system performs a calculation with the numbers to calculate an x and y value for the center of the ball.

4.6.3 Swing Timing Representation

One of the main use cases for this project is to help softball players improve their batting. To do this, we must give the user feedback on the timing of their swing. The timing of the hit is denoted on the ball by shading the side (left, middle, right) either red or green depending on if the swing as a miss or hit. The timing is determined by the swing and where the ball was located in the zone at the time of the swing. For example, if the user swings late, then they would hit the left side of the ball, and that would be shaded accordingly, as shown in figure 5.

4.7 Post Swing Graphics

After the batter has swung, it is important that they receive feedback on their swing so that they can improve their hitting technique. To give appropriate feedback, the system must determine if the swing hit the ball and if so, where the bat hit the ball.

4.7.1 Hit Calculation

The graphics module and finite state machine work in tandem to determine if a swing hit the ball. Once the bat is shown on the screen with the correct angle, graphics iterates through every pixel to determine if any pixel exists that is both part of the bat and the ball. If one exists, there is an intersection, so the bat is considered to have hit the ball. This implementation of hit determination allows the system to use the bat angle and ball size (determined by how far into the strikezone the ball is) to appropriately determine a hit. It also ensure that the graphics shown to the user properly correspond to a hit or miss.

The graphics module sends the finite state machine a high signal for a pixel that has both bat and ball components and a low signal otherwise. The finite state machine waits for every pixel to be checked once and if none of these bits are high, then it considers the swing a miss. Otherwise, the swing is a hit.

4.7.2 Location of Bat Hitting Ball

To more precisely identify where the bat would strike the ball for a particular swing, both the swing timing and angle must be taken into account. As explained in Section 4.6.3, the timing of the swing is represented by the shading of the appropriate side of the ball. When there is a hit, this shading is green.

The bat is represented as a blue slanted rectangle, corresponding to the angle of the swing. When there is a hit, the location of the hit on the ball is identified by the intersection of the shaded area and angled bat. This intersection turns a light blue color to distinguish it from the timing indication and bat representation.

4.8 Trajectory

The trajectory of the ball shows a possible outcome of a hit, so it is only shown if a hit is detected after a swing. The trajectory is determined by where the bat hit the ball. For example, if the bat contacts the ball on the left and in the upper half, then the ball would travel down and to the right, providing visual feedback.

To specify the trajectory of the ball, the ball must look as though it is moving farther away from the batter, and it must be given both a horizontal and vertical component to its movement. To make the ball look like it is travelling away from the batter, the radius of the ball decreases. To specify the trajectory, the X and Y position of the center of the ball changes. To determine the X change, the timing of the swing is used. If the bat contacts the ball on its left side, then the ball should move to the right and should have an increasing X value of its center point. If the bat hits the middle of the ball, then the ball should fly directly into the screen and not vary horizontally. Lastly, if the bat hits the right side of the ball, then the ball should fly to the left and the X value of its center should decrease. To determine vertical ball movement, the graphics module looks for the bat intersecting the ball in both the top and bottom half of the ball. If the bat only connects with the top half, then the ball moves downward. If the bat only contacts the bottom half, then the ball flies upward. And, if the bat contacts both the upper and lower halves, then the ball does ot move vertically. With the combination of both X and Y movements, the ball trajectory takes into account the angle and timing of the swing.

Figure 6. Trajectory of ball shown after the bat swing displayed

4.9 Background Picture

To enhance the user experience we wanted to use a softball-related background for the game. We took a picture of one of the pitchers on the MIT Varsity Softball team and cropped it so that it looked as though she had just finished her pitching motion from the point of view of the batter. The purpose for this image was to have it present in the background with all of the gameplay features on top of it.

To use the picture, it had to be loaded in to multiple BROMs using COE files. Utilizing a MATLAB script, the images RGB values for each pixel were specified in three color maps: red, green, and blue. Each of these color maps were 8 bits by 256 combinations and were each stored in their own BROMs. In addition, there was an 8-bit by 196608-entry index map used to look up for each pixel which row of the color tables should be used. This allowed for an image that used 256 colors specified by 24 bits. This was then later converted to the 12 bit color that the NEXYS requires.

In order to fit into the BROM, the image had to be scaled down to a quarter of the size needed to cover the screen. When creating the image on the screen, it then had to be enlarged by using the same pixel color four times (quadrupling the size of the image). This meant that a change in the lookup value should only occur after every other vertical and horizontal change in pixel value. To simplify this process, the last bit of both hcount and vcount were dropped for the purposes of finding the correct index value. This allowed for the image to be quadrupled and fill the entire screen.

4.10 Adjustable Ball Speed

An additional project feature is the ability to adjust the difficulty of the game. By adjusting the difficulty, the user changes the speed of the pitched ball. The more difficult mode pitches the ball twice as fast as the easier mode, and this can be controlled by a switch on the FPGA. To integrate this with the rest of the system, timing was controlled with the speed as a parameter. This allowed for the ball's width to increase, ball's color change, LED light up sequence, and the determination of hit timing to remain consistent across speeds.

4.11 LED strip

The LED strip consists of a clock-based addressable 30-LED strip, communicating through the additional module, ledTX, which serially transmits a full sequence of color and brightness settings for each LED at the proper frequency, and provide it the correct signals and the right times to sync with the rest of the game.

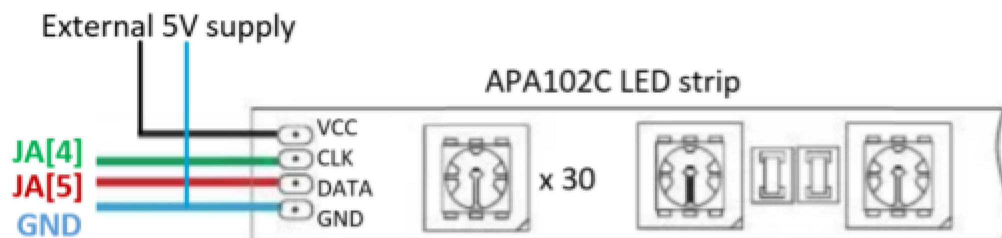

Figure 7. Wiring diagram for LED strip

The datasheet for the APA102C addressable LED strip from Polulu required an approximately 1 MHz clock and a specific start and end signal. Each LED's setting is defined by a 32-bit packet, so we needed 1024 bits to send our (30+2)*32 bits of data to update the LED. The clock is generated from the MSB of a 6 bit counter running on a 65 MHz clock, to divide by 64 and get a 50% duty cycle. The ledTX module detects a change in the input packet and then sets another counter to the size of the full packet. The module counts at the 1 MHz clock rate and always sends the next bit within the packet until it reaches zero and then waits on the next new packet to send.

The full LED sequence is defined by the graphics module and each LED lights up in sequence according to the changing width of the ball. To modularize the code, we did not want to check for a certain ball width and then light up a particular LED. Instead, a sequence is sent to light the first LED and then the rest are defined by concatenating a blank LED signal with the commands for the first 29 LEDs. This had the effect of moving the lit LED down the strip without the graphics module having to control which LED was specifically lit. This also helped with adjusting to different speeds because we simply had to change when the led instructions were changed instead of having to recalculate which LED should be lit for certain ball widths. When these instruction packets are prepared, graphics sends a signal to the serial transmitter to send the packet.

After a swing, the LED corresponding to the timing of the swing remains lit. For example, a hitting a ball on its left side corresponds to a late swing, so the LED that remains lit is farther down on the LED strip, corresponding to later in the zone. This LED remains lit while the bat is on the screen to give the user additional feedback on their swing.

After the bat disappears from the swing, an additional fun feature of the LEDs is that they turn either solid green or red depending on if there was a hit or a miss. This caused us to think about how often we should send the LED update packet if the packet was not changing. We end up sending it a couple of times and then the LEDs remain constantly lit. The resulting behavior is shown in figure 8 below.

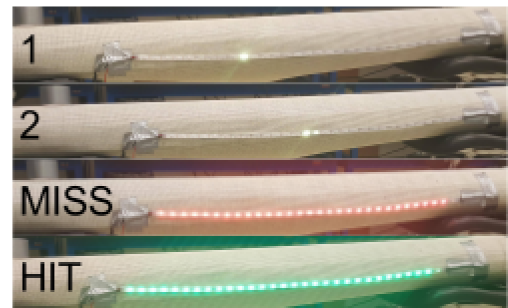

Figure 8. LED sequence during swing. Ball moves from 1 to 2, then hit or miss

4.12 Scoring

As an add-on, we implemented a scoring feature that tracks the number of hits and misses made since resetting. Since the fsm module already tracks different states of hitting or missing, we incorporate the hit/miss counter with the FSM. To display the values, the system utilizes the 7-segment LED display built into the Nexys4 board, with the accompanying display_8hex module provided in previous labs. However, as a typical user, we don't expect hits to be counted in hexadecimal format, so we wrote a hex_to_dec module that converts any hex number into a decimal equivalent for displaying. To simplify the operations, the system restricts

the input to 14 bits, allowing up to four decimal digits, which are each defined as individual decimal counters. When a new hex input is present, the main counter is initialized, and counts down until it reaches zero. As it counts down, the decimal counters are incremented, carrying one to the next digit and resetting to zero whenever they reach 10. Although this technique requires as many clock cycles as the size of the hex number, this delay is not noticeable to the user, since it does not last more than $2^{14} clk/65 \times 10^{6} Hz \approx 2.5 \times 10^{-4} s = 1/4 ms$. For other applications, we may consider expanding this module to take in a variable number of bits and output the corresponding number of decimal digits by implementing an array and synthesizable for loops, or with a LUT; however given our low expected number of hits, this was not necessary.

4.13 Externals

For a more enjoyable user experience, we added various hardware externals, including a sound signalling calibration and a foot pedal as an alternative to a button press to allow a single player to signal their own pitches. The schematics for the speaker and footpedal are shown in figure 9 and figure 10 respectively, each requiring a single IO pin from the FPGA.

The siren module generates an 880 Hz signal from the 65 MHz input clk using a counter and outputs it at 50% duty cycle when enable is high. Using the same hardware setup as in lab 4, we connect this output to drive a piezo speaker to audibly indicate system calibration.

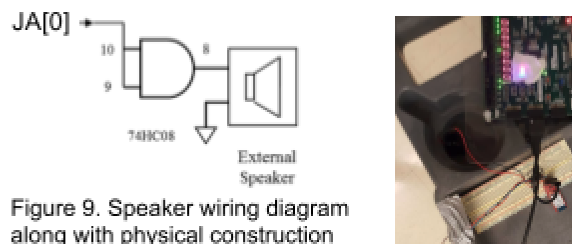

Figure 9. Speaker wiring diagram along with physical construction

The foot pedal is nothing more than a button connected to the FPGA over a long wire. We used a limit switch and attached a long platform to make it more user friendly. As shown on the schematic, a pull down resistor was included to make sure an ‘off’ signal is sent when the pedal is not pressed, and 3.3V are only connected across it when the switch is connected.

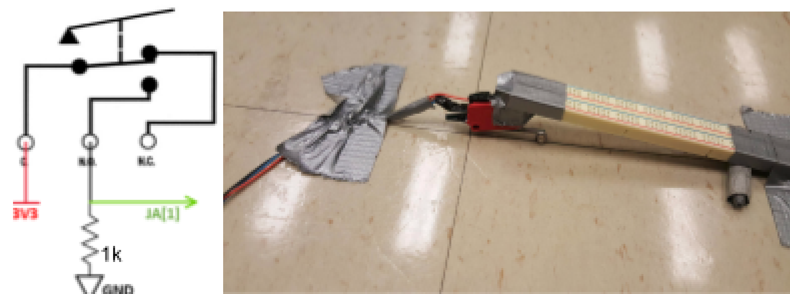

Figure 10. Schematic for active high limit switch used for the foot pedal along with physical construction

4.14 Physical Setup

To set up the complete and playable project, a few square meters of open space is needed to allow for a full bat swing. A TV screen with a VGA port is used instead of a computer monitor to enhance the user experience with fuller immersion in the softball environment. The foot pedal is placed near the batter, at least two meters from the screen to avoid unfortunate collisions with the bat. The strip of LEDs is located in front of the batter, perpendicular to the display, and mounted approximately a meter from the ground, to be clearly in the user's view. The FPGA display must be visible to indicate the score, as well as having the buttons and switches accessible to reset the game or adjust the difficulty. Placing the speaker in a cup amplifies the sound to make it noticable from the batter's position. The complete setup is shown in figure 11 below.

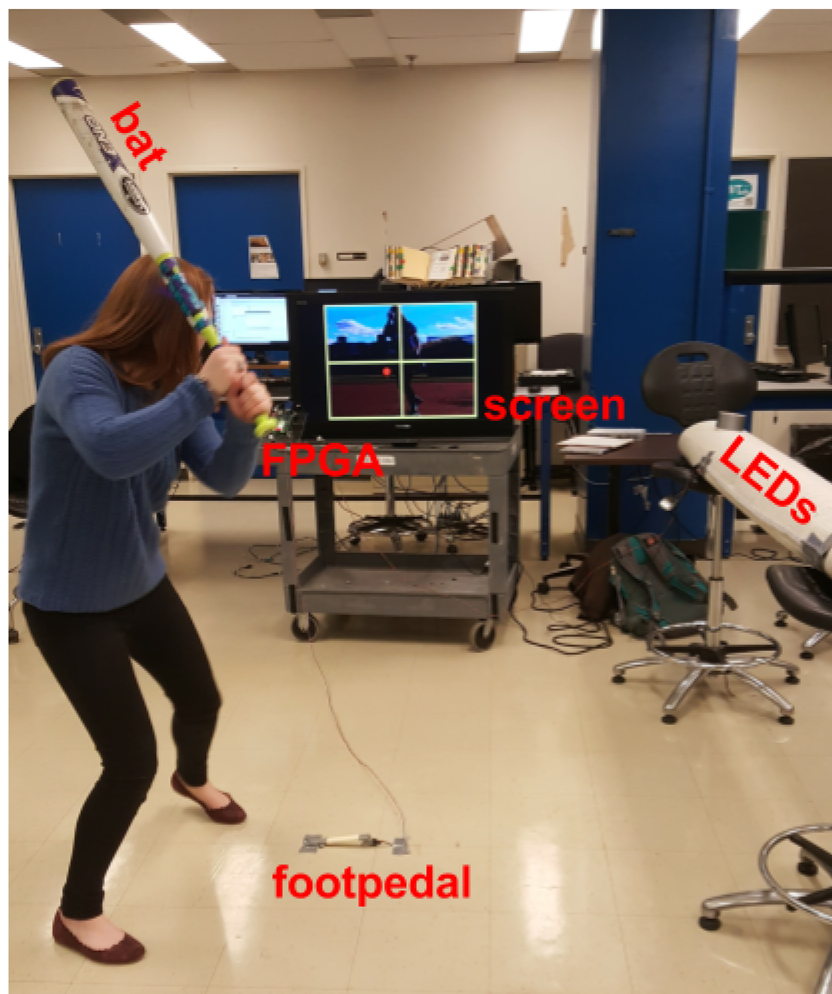

Figure 11. Katie demoing full system

5 Testing and Debugging

We used a wide variety of methods to verify the functionality of each module before combining them all and then trying to find the bugs. We had originally designed the system to be highly modular, allowing each subsystem to be written and tested independently.

5.1 Simulations

Writing test benches and running simulations is very useful for specific modules that require very precise timing and for which we know the exact inputs. Both the fsm and serialRX modules are perfect examples for simulating. For the fsm, we thoroughly planned the whole sequence of possible incoming signals and manually worked out the desired response. Then we ran a simulation with those signals to compare the results and check for any glitches. Similarly, the exact packet-receiving subsystem needed to be precisely timed to align the incoming signal into properly ordered data. We could also verify the correct assertion of the done signal.

5.2 ILA and Oscilloscope

The internal logic analyzer and oscilloscope were useful in displaying streams of data that were occasionally hard to simulate or when we were not sure if a bug was due to hardware or software issues. This was particularly helpful in testing our communication with the IMU, verifying that the correct bits were being sent in the first place, and then observing the difference in how the packets are sent over bluetooth, and also for data sent to the addressable LEDs, to identify where the issue was coming from.

5.3 Manual Mode

To test the graphics we needed only to connect a VGA cable to a monitor, but to test the gameplay functionality with the graphics, it was necessary to provide fake input signals, which normally would come from the physical bat. We called this ‘manual mode’, which could be turned on by the flip of a switch, and allowed button presses to signal calibration and batter_up messages. We also introduced a switch-dependent bat X and Y, to test the various angles possible for the bat blob without necessarily having the IMU calculations ready.

5.4 Indicator Displays

On the FPGA, we used a wide variety of signals to debug our system with instant visual feedback. During testing, we extensively used the hex display to communicate the current state of the module under testing without needing to set up and trigger an ILA. We also relied on the colored LEDs for simple conditions, such as whether the bat is determined to be pointing up or down. Although this binary feedback was enough for initial testing, we found it much more

useful to graphically display the virtual bat on a monitor in real time. Although for the game we are only interested in the snapshot moment when the bat crosses the hitting zone, this method of testing allowed us to really understand the effects of centripetal acceleration during a full swing and we could adjust the settings until finding an ideal set of parameters.

5.5 Calibrating Parameters

Determining the exact set of parameters that work best for the IMU equations was a somewhat tedious process but found it to be much easier than going through endless calculations which might not even be implementable on the FPGA. To avoid waiting several minutes between each test code compilation, we connected the various parameters to a switch-based input, and could test a wide range of possible values with the same code uploaded. To visually test the bat system, we enable the bat to always display the measured angle in real-time. A combination of seeing the effects our switch changes had on the resulting bat angle and knowing the expected impact based on the equation, we were able to fairly quickly narrow down an ideal set of values. The greatest difficulties came from the slope calculator with three varying parameters along with needing to account for different styles of swinging a bat.

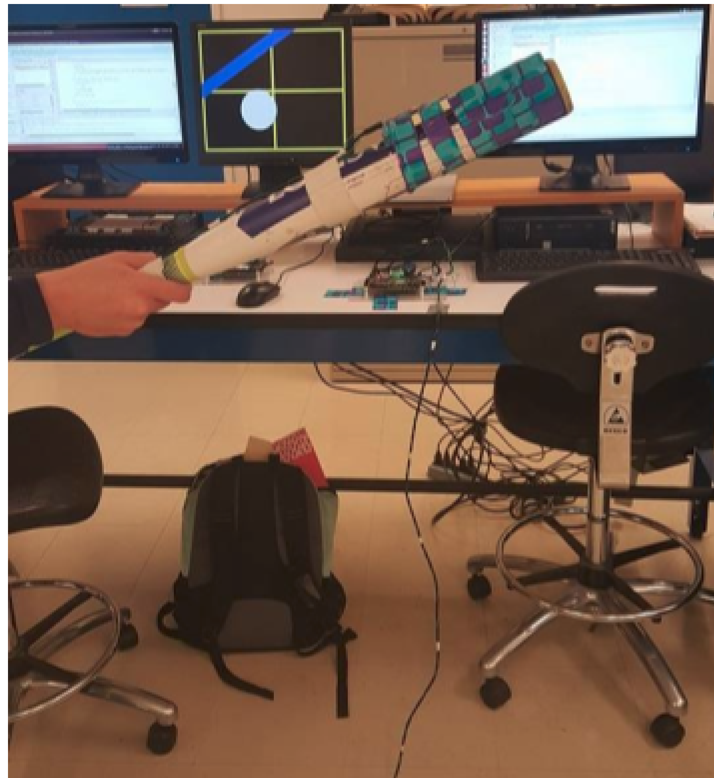

Figure 12. Bat angle displayed in real-time for testing. Also notice the earlier prototype version, with a long connected wire and IMU sensors attached at the end of the bat

6 Challenges

6.1 Bat Angle Calculation

One of the most surprisingly difficult challenges was using the IMU data to track a bat swing. There are many different axes of rotation and variation from batter to batter in their speed and form of their swings. We tried to find the simplest approximation that could reasonably reflect the user's bat angle, but discovered that we would need to make several different adjustments and iterations to be fully operational for the game. The general approach of low passing acceleration and merging with integrated reading from the gyroscope was too sensitive to effects from centripetal acceleration. After attempting many different angle tracking and acceleration compensation techniques (described in more detail in Section 4.2), we settled on a pseudo-compensating formula while also adjusting other factors to eliminate extreme values. By subtracting gyro readings from the acceleration, we were able to approximately scale the measured acceleration by a factor proportional to the centripetal acceleration; however, optimizing for the best constant scaling factors required a lot of testing on a variety of users. This technique was also not very successful until we moved the IMU closer to the point of rotation, which reduced the magnitude of the centripetal acceleration enough to be compensable. Then finally, to eliminate any extreme readings, we fixed a constant X value, since we decided a small angle approximation was close enough and we could avoid trying to compensate for the centripetal acceleration in multiple dimensions.

6.2 Bat Angle Representation

One of the biggest challenges early on in the project was figuring out how to best represent an angled bat on the screen. We wanted the bat to look accurate, so the ability to represent a slanted rectangle was necessary. There were multiple challenges associated with this including how to best represent a bat angle, how to appropriately perform the calculation required to form a slanted line, and how to handle negative numbers.

The first aspect to solving any of these challenges was to develop the equations of the lines (originating from the left of the screen and vertically centered). Using a slope, these equations were formed. Then, we discussed how to best represent this slope, which determined the angle of the bat. Having to perform a division calculation when the divisor is not a power of 2 is not ideal, so instead of inputting a slope to the graphics module, the decision was made to give both an X and Y value. This led to more equation manipulation such that the only calculations required were multiplications instead of divisions.

This then left the question of negative slopes (corresponding to downward angles). We made the design decision to have a separate signal given as input to graphics to indicate whether the

angle of the swing was up or down. This meant that graphics did not have to be aware of any negative values, making the code more modular. The line equations had to be altered slightly for downward angles, but this input is checked before any calculations are completed.

Maintaining a consistent bat width at varying angles also turned out to be a much greater challenge than expected, since our original approach defined a constant bat height within each vertical pixel line. This height value needed to be scaled by an experimentally determined factor dependent on the bat angle. Although fully functional in manual mode, we found this feature unnecessary for smaller angles present in an actual swing.

6.3 Background Picture

The background picture caused a great deal of difficulty and ultimately learning through the process of getting it to appear on screen. Neither of us had any experience with images on the FPGA or handling memory (we did infrared Lab 5, not the audio Lab 5), so we learned about that through the process of putting a real image on the screen. Unfortunately, the NEXYS did not have a large enough memory to hold pixel information for the entire screen, so the image had to first be cropped and sized to be a quarter of the screen.

With no experience understanding what a color map provides, the first attempt to load the image simply stored 9 bit color (3 bits for each red, green, and blue) which could be indexed directly from the location of the pixel. After using MATLAB code to convert this information to a COE file, the image appeared on the screen, but the color was not what we wanted. Due to space restrictions, so few colors could be used and the image did not look right.

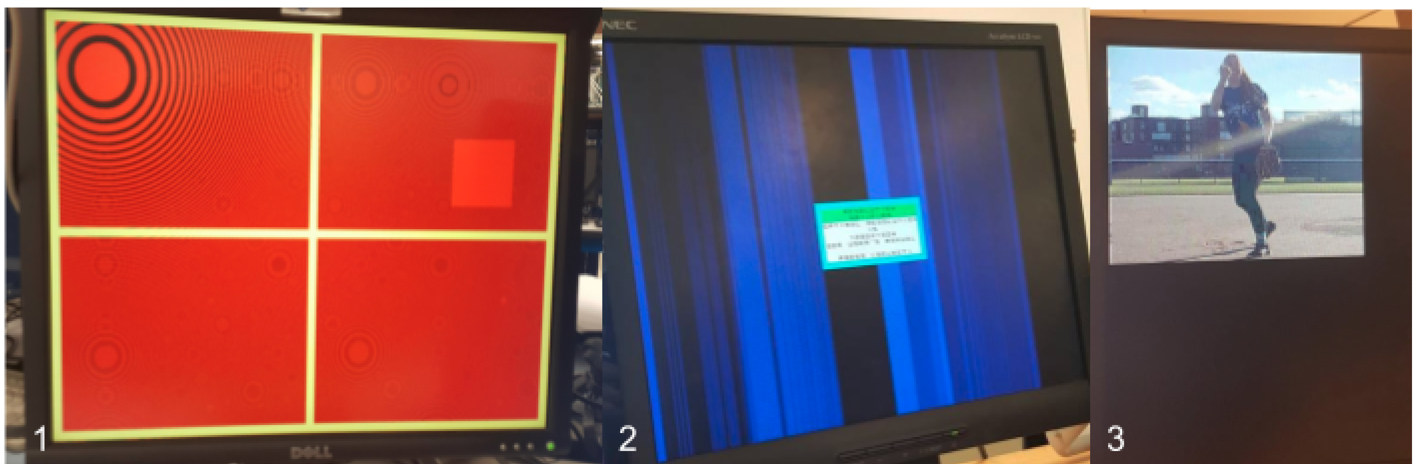

Figure 13. Outputs from earlier iterations of first implementing circle blob without pipelining (#1 in red), then the background picture with too few bits available on the pixel output (#2 in blue), and finally the small picture taking up a quarter of the full screen (#3 to the right).

We then decided to try using color maps to improve the image. This involved using four BROMs instead of just one. Three of these held the 8 bits for each color component (red, green, and blue) and the other held the index of these colors corresponding to each pixel. More MATLAB code was necessary to generate these four COE files, but in the end, the color on the screen looked much better and more accurate.

The last remaining challenge for the image was that it was only specified to be a quarter of the screen due to memory constraints. The image had to be pixelated and quadrupled in size. To do this, the last bit of both hcount and vcount were left out of the calculations for looking up the color index. This meant that every pixel index was used four times instead of once, performing the quadrupling of the photo.

6.4 LEDs

The addressable LEDs provided both a motivation and a challenge for this project. We were very excited to get to this stage of our project where we could implement them, but also had to figure out how to communicate with another piece of external hardware. To learn about the general process for how the LEDs work, we meticulously read the data sheet and developed a plan to form the right instructions and time the data right.

One of the biggest challenges we had was in figuring out how to light the correct LED to correspond with the size of the softball as determined by the graphics module. At first, we planned to do this using equations that take in the size of the ball and then determined the LED that should be lit and then construct a stream of bits that corresponded to those instructions. This approach seemed more complicated than necessary. After thinking more about it, we realized that the system just needed to know when to move the lit LED over by one. We also simplified this process by forming the new instructions by concatenating a blank LED signal to the front of the first 29 LEDs of the previous instruction. This performs the same operation as moving the LED light over 1, but it does not require graphics to know exactly which LED is lit at all times, it only knows when the lit LED should be shifted.

To improve the overall user experience, we had to extend the cable that connects the LEDs to the FPGA. Since we used a long unshielded wire, this led to a glitchy signal arriving at the data input for the lights, causing them to flicker when repeatedly sending the same packet to hold a solid color. To fix this, we reworked the ledTX module to detect and ignore repeat packets. Until a new packet is sent, the LED strip holds the previous value, so the desired effect is produced. Since the original packet is not sent as frequently, the probability of a corrupted packet arriving is much lower, so we no longer noticed any glitchy behavior.

7 Design Decisions

Throughout this project, we made strategic design decisions regarding the implementation of the project and overall user experience.

7.1 Nexys 4 vs Labkit

Our first major decision was whether to use the Nexys 4 or the Labkit. There were several advantages of sticking with the more familiar labkit with more available IO ports and existing code from previous labs, but we felt that the Nexys offered a nicer interface and may be more applicable in future career or personal projects. The first challenge we faced with the Nexys was porting the pong VGA module to the different FPGA setup, since that was the basis for our graphics module. Another concern we had with the Nexys is that it would not provide 5V to power the Teensy, which was needed to communicate with the IMU. We knew that eventually we wanted the bat to be battery powered anyway, so we did not get too concerned about this, and were able to work around the restriction during testing stages by utilizing the NEXYS 4 board's USB port, which is able to supply 5V power.

7.2 IMU vs Camera

A key decision in our approach to this project was what sensors we should use to detect the bat angle. For example, should we calculate the bat angle from a camera image or using only IMU readings? Should we use a special distance sensor to know when the bat is in the plane and what angle it is at? These questions were very difficult to answer without having much understanding on the difficulty of any specific approach. Important factors we considered were the components, setup, speed, and accuracy of the devices. The IMU was available in lab and we knew that theoretically it could provide orientation information. Our biggest concern with the camera was the frame rate to capture the exact moment of the swing and processing time required post-swing. Although the bat angle does not need to be shown in real time, we wanted nearly instantaneous feedback for the user of how they swung. It would have also been much more difficult to determine the exact timing of the swing using a camera, so we might have ended up needing an additional sensor anyway. It seemed that a full camera system would have missed some key readings and overall would have required much more setup for a relatively simple angle calculation. Looking back, we realize that a camera could have been able to provide more information on the height the bat is held at and would obviously not be influenced by centripetal acceleration, but we feel that a single IMU sensor was enough to make the game fun and playable once we figured out the right calculations and communication protocol.

7.3 Communication Protocol

The IMU device we worked with communicates using an I2C protocol, which we learned enough about in lecture that we could have spent weeks on setting up that communication channel and interfacing the IMU directly with the FPGA. We decided that we were more interested in focusing on the gameplay component and IMU calculations than on the communication protocol alone, so we chose to buffer the data through a microcontroller, the Teensy, which handled the I2C protocol and streamed the data serially to the FPGA. Although we still had to create our own serial protocol and have both the Teensy and FPGA agree upon it, we were able to reach a functional prototype state within a week, instead of spending a month debugging the exact timing signals required for I2C. It was also more interesting to see our own protocol in action than trying to implement an existing one using IP cores or researching the specs on our own. We also knew that eventually, we were aiming to make the device bluetooth enabled, so by beginning with a serial communication channel, we would theoretically be able to simply insert two paired bluetooth modules that passed through all the data to act as a virtual wire. Of course this came with more difficulties than anticipated, but building off of an already working serial protocol made the transition much smoother.

7.4 User Interface

Many graphical design decisions were made with the user in mind. First, we decided to not only have the ball grow larger as it approached the batter, but it also changes color. The reason for this was to better help the user understand when they are able to hit the ball with correct timing. We also chose to only show the angle of the bat at the moment it would contact the ball. This was a strategic decision made because the system does have the capability to show the changing bat angle in real time. We chose to only show the angle at one point so that the user would have to focus on the specific part of the swing when the ball would contact the bat. This is consistent with only looking at the timing of the swing at this point as well.

We also thought specifically about how to give the user feedback on their swing. We chose to show them the side of the ball that they hit, because this is important information to a softball player. In softball, batters ideally want to hit the center of the ball. This would be the best timing and hit location for a line drive hit. By illuminating the side of the ball that was hit, the batter can adjust their timing for the next swing. For players who do not care as much about where the bat hit the ball but who want to know where the ball would travel as a result of their hit, we also added in the trajectory of the ball. After pausing for 5 seconds to show the user where their bat intersected with the ball, the ball’s trajectory is shown as another visual cue to the user.

Another user experience decision made was to have the batter calibrate their swing for every pitch. This decision was made for a few reasons. First, it allows the system to know when to judge that the batter has broken the plane of the ball. There are thresholds around this calibrated value to correctly determine this timing. Second, it allows the system to be used by

everyone and all types of swings. This calibration allows the system to recognise swings of many different types so that the system is not designed to only work for one person. Lastly, the calibration allows the system to take into account any gyroscopic drift that may have occurred from the last swing. When the bat, and IMU, whip around on a swing, the sensors can be thrown off. The recalibration process allows the system to take those changes into account. To simplify the user interface, we attached the calibration signal to a foot pedal and avoided needing a second button by signalling the pitch autonomously once the bat is raised.

7.5 LEDs

When we were given the strip of LEDs, we felt like we had a world of possibilities as to how we could utilize them. To determine what to do, we thought about what would be most helpful to the user. We decided that it would be most helpful to light up LEDs in sequence to physically show the batter where the ball was in relation to them. We feel that this was the best decision because as we performed user testing, we noticed that some of the users looked at the screen for timing cues while others used the LEDs.

8 Reflections

8.1 Katie

As we were trying to figure out what to do for our project, I said to Melinda that I had a crazy idea involving swinging a softball bat around the lab. She looked at me like I was crazy at first, but then we sat back and talked about the idea and how it would teach us about interfacing the real world and the virtual world. We decided that this project would allow us to achieve our goals for what we wanted to learn from this project - and I was going to get to swing a bat in lab and talk about softball!

As for the project itself, I learned about making fast iterations to improve the system and user experience as a whole. I focussed mostly on the graphics and gameplay aspects of the project, which were working at a baseline level pretty early on in the project. This meant that I gained a lot of experience in adding new features to existing code. This definitely presented challenges at times in determining the best implementation of new features and how to fit them into the existing code without changing the already working functionality. Through this, I learned about making clear and modular code from the beginning and how to utilize signals that were already being sent for ulterior purposes. Most of the graphics were built in this way where additional features were added that were related to existing graphical features such as the ball size and whether the bat was on the screen.

In addition to all of the technical knowledge I gained from this class (it was my first full EE lab that I have taken), I also learned more product sense skills. I will be starting as a full time

product manager after I graduate, and this project allowed me to think about our system as a product. Most of the design decisions were made from a balance of technical considerations as well as user experience considerations, and these are trade offs that I know I will have to make in the future. I also gained experience working closely with a partner to not only make a technical project but also to present and write deliverables.

8.2 Melinda

When first approaching this project, I felt lost and overwhelmed by the vast number of tasks we had to accomplish, most of which were completely new and unfamiliar to me. How could I possibly interface with an IMU to calculate orientation, let alone play softball with it?!? I also remember writing out our proposed schedule and doubting that we could ever keep up with it; there was so much to do and so little time. After getting over this feeling of hopelessness and beginning to work on the project one step at a time, I started being more confident with what I was able to do and would take on the challenges. Sure, I sometimes had no idea what I was doing, but I felt like I had enough basic knowledge and support from the staff to start working on something and eventually mold it into a working module. By the end of it, I couldn't believe we were able to stay on schedule and complete our goals - even reaching the LEDs that I had been really looking forward to. I also felt that I gained an incredible amount of knowledge and confidence when working with FPGAs and felt much more comfortable with needing to implement whatever behavior was necessary. In fact, I even ended up using an FPGA instead of a microcontroller for my 6.301 project, since I felt more familiar with Verilog than C coding by that point. I really enjoyed this class and know that I will be able to take a lot from it, if not only being able to program digital logic, but also in how to think about and use these devices for practical purposes.

9 Conclusion

As we conclude this project and a wonderful semester, we took some time to reflect on our project experience. There are a few key takeaways that we think would be helpful for future students to know.

9.1 Start Early and Ask for Help

We truly enjoyed working on this project! Besides having the opportunity to make a fun and useful game, we found our time in lab to not be very stressful. We believe that one of the main reasons for this was because we started working on our project very early. We stayed to our original schedule from our project presentation and worked each week to make sure that we met our goals every week. By staying on this schedule, we were able to achieve our baseline and expected goals before Thanksgiving Break and began to work on our stretch goals. This ultimately allowed us to finish the project about a week early and work on adding a few additional fun features and hardware.

We also recommend asking for help early on in the process. After struggling to figure out challenges, we asked the staff for their help in solving problems. Not only did this help us fix our problems, but we also learned a lot from talking with the staff. Clear examples of this were when learning about the physics behind the IMU and color maps when utilizing real images.

9.2 Modular Design

We had a very clear idea of the different modules we were each going to work on at the start of the project. We found a way to modularize our project such that we could both work in parallel by being in lab together at the same time but working on separate parts. This allowed us to be more efficient in our work.

Although we were working on different modules in parallel, we were communicating frequently about the signals sent between the modules. This meant that it was much easier to integrate the system in the end because the modules were designed to work with each other. An example of this is the bat data interfacing with the bat graphic. These components were designed by two different people, but the bat data must report X and Y components of the angles and the graphics module knew that the inputs would be X and Y components. During integration, we just had to change the sources of input for the graphics module to display the real bat data instead of the simulated bat data.

9.3 Iterative Process

We also approached this project in a very iterative way. Throughout the semester, we continued to add features onto the existing project. Although it took time to figure out how to add new features seamlessly into the existing code, by doing this, we ensured the overall timing and gameplay of the system remained reliable. We built a general state machine to begin the project, and as the project grew, we never really removed states, we added them. This way everything fit into the same general structure with more features added in between the original states. Overall, this allowed us to take more risks in what we built because we knew that we could revert the project to a working version.

10 Acknowledgements

We sent many hours in the lab for this project, and through this process, we received help from multiple people along the way. By working with each of these people, we learned much more than we ever expected. Thank you to Mitchell our mentor for helping us scope the project appropriately and giving us advice along the way. Thank you to Gim for giving us ideas on how to push ourselves to make a better and better project. We enjoyed implementing the LEDs and trajectory of the ball! Thank you to Joe for helping us with everything IMU related. We appreciate all of the time you spent with us and for helping us to acquire the electronics we

needed for this project. Thank you to the rest of the TAs and staff for encouraging us to do our best and helping us learn so much this semester!

Also, thank you to Amber who allowed us to take a picture of her pitching and use it as a background in our game. We hope you enjoyed, as you said, "hitting off of yourself for the first time"!

11 Appendix

11.1 User Experience

Here is a list of instructions on the general overall user experience of the game:

1. The batter performs a partial swing to the point where they would hit the ball. At this point, ensure that the breadboard and IMU are facing the ceiling.
2. From this position, step on the foot pedal to calibrate the bat. When the beeping stops, the calibration is complete.
3. Pull the bat back and prepare for the ball to come. In softball, this is called a load.
4. The pitch will be automatically triggered from this position, and a growing ball will appear on the screen.
5. When the ball becomes green, swing the bat at the correct angle to hit the ball.
6. Look at the screen and string of LED lights to see how well you hit the ball. After 5 seconds, the ball's trajectory will show on the screen.
7. Repeat steps 1-6 for another pitch each time.
8. When done, look at the FPGA to see the number of hits and misses.

11.2 PC Code

11.2.1 Picture - generateCOE.m

```
%6.111 Image Color Table MATLAB deme
%Edgar Twigg bwayr@mit.edu
%4/1/2008 (But I swear this file isn't a joke)

%% How to use this file
%Notice how %% divides up sections?  If you hit ctrl+enter, then MATLAB
%will execute all the lines within that section, but nothing else.  You can
%also navigate quickly through the file using ctrl+arrow_key
```

```
%% Getting 24 bit data
%So when you look at a 24 bit bitmap file, the file specifies three 8 bit
%values for each color, 8 each for red, green, and blue.
[picture] = imread('Pitching_Image.bmp');

%% View the image
%This command image will draw the picture you just loaded
figure              %opens a new window
image(picture)            %draws your picture
title('24 bit bitmap')  %gives it a title so you don't forget what it is

%% Manipulate the data in the image
%So now you have a matrix of values that represent the image.  You can
%access them in the following way:
%
%picture(row,column,color)
%
%Remember that MATLAB uses 1-based indexes, and Verilog uses 0!
%
%Also, you can use MATLAB's slice operator to do nifty things.
%picture(:,:,1) would return a 2D matrix with the red value for every row and
%column.
%
%picture(:,1,2) would return a 1D matrix with the green value for every row
%in the first column.

%This is how MATLAB indexes the colors
RED = 1;
GREEN = 2;
BLUE = 3;

%So if we wanted to see the red values of the image only, we could say
figure
image(picture(:,:,RED))
title('Red values in 24 bit bitmap')

%Because the image we gave matlab above specifies only one value per pixel
%rather than usual three (red,blue,green), MATLAB colors each pixel from
%blue to red based on the value at that pixel.

%% Getting 8 bit data
%When you store an 8 bit bitmap, things get a little more complicated.  Now
%each pixel in the image only gets one 8 bit value.  But, you need to send
```

```
%the monitor an r,g, and b!  How can this work?
%
%8 bit bitmaps include a table which specifies the rgb values for each of
%the 8 bits in the image.
%
%So each pixel is represented by one byte, and that byte is an index into a
%table where each index specifies an r, g, and b value separately.
%
%Because of this, now we need to load both the image and it's colormap.
[picture color_table] = imread('Pitching_Image.bmp');

%% Displaying without the color table
%If we try to display the picture without the colormap, the image does not
%make sense
figure
image(picture)
title('Per pixel values in 8 bit bitmap')

%% Displaying WITH the color table
%So to display the picture with the proper color table, we need to tell
%MATLAB to set its colormap to be in line with our colorbar.  The image
%quality is somewhat reduced compared to the 24 bit image, but not too bad.
figure
image(picture)
colormap(color_table)   %This command tells MATLAB to use the image's color table
colorbar                %This command tells MATLAB to draw the color table it is using
title('8 bit bitmap displayed using color table')
% green = color_table(:,2);
% scaled_data = green * 255;

%% More about the color table
%The color table is in the format:
%
%color_table(color_index,1=r 2=g 3=b)
%
%So to get the r g b values for color index 3, we only need to say:
disp('          r       g             b       for color 3 is:')
disp(color_table(3,:))  %disp = print to console

%Although in the bitmap file the colors are indexed as 0-255 and each rgb
%value is an integer between 0-255, MATLAB images don't work like that, so
%MATLAB has automatically scaled them to be indexed 1-256 and to have a
%floating point value between 0 and 1.  To turn the floats into integer
```

```
%values between 0 and 256:

color_table_8bit = uint8(round(256*color_table));

disp('   r         g         b         for color 3 in integers is:')
disp(color_table_8bit(3,:))

%Note that this doesn't fix the indexing (and it can't, since MATLAB won't
%let you have indexes below 1)

%another way to look at the color table is like this (don't worry about how
%to make this graph)
figure
stem3(color_table_8bit)
set(gca,'XTick',1:3);
set(gca,'YTick',[1,65,129,193,256]);
set(gca,'YTickLabel',['  0';' 64';'128';'192';'255']);
set(gca,'ZTick',[0,64,128,192,255]);

xlabel('red = 1, green = 2, blue = 3')
ylabel('color index')
zlabel('value')
title('Another way to see the color table')

%% Even smaller bitmaps
%You can extend what we did for 8-bit bitmaps to even more compressed
%forms, such as this 4-bit bitmap.  Now we only have 16 colors to work with
%though, and our image quality is significantly reduced:
% [picture color_table] = imread('Pitching_Image.bmp');
%
% figure
% image(picture)
% colormap(color_table)
% colorbar
% title('4 bit bitmap displayed using color table')

%% Writing data to coe files for putting them on the fpga
%You can instantiate BRAMs to take their values from a file you feed them
%when you flash the FPGA.  You can use this technique to send them
%colortables, image data, anything.  Here's how to send the red component
%of the color table of the last example

%red = color_table(:,1);          %grabs the red part of the colortable
```

```
%scaled_data = red*255;                %scales the floats back to 0-255
% green = color_table(:,2);
% scaled_data = green * 255;
% blue = color_table(:,3);
% scaled_data = blue*255;
% rounded_data = round(scaled_data);  %rounds them down
% data = dec2bin(rounded_data,8);   %convert the binary data to 8 bit binary #s
%
% %open a file
% output_name = 'color_table_pitch_blue.coe';
% file = fopen(output_name,'w');
%
% %write the header info
% fprintf(file,'memory_initialization_radix=2;\n');
% fprintf(file,'memory_initialization_vector=\n');
% fclose(file);
%
% %put commas in the data
% rowxcolumn = size(data);
% rows = rowxcolumn(1);
% columns = rowxcolumn(2);
% output = data;
% for i = 1:(rows-1)
%       output(i,(columns+1)) = ',';
% end
% output(rows,(columns+1)) = ';';
%
% %append the numeric values to the file
% dlmwrite(output_name,output,'-append','delimiter','', 'newline', 'pc');
%
% %You're done!

%% Turning a 2D image into a 1D memory array
%The code above is all well and good for the color table, since it's 1-D
%(well, at least you can break it into 3 1-D arrays).  But what about a 2D
%array?  We need to turn it into a 1-D array:

picture_size = size(picture);   %figure out how big the image is
num_rows = picture_size(1);
num_columns = picture_size(2);

pixel_columns = zeros(picture_size(1)*picture_size(2),1,'uint8');  %pre-allocate a space for a
new column vector
```

```
for r = 1:num_rows
        for c = 1:num_columns
        pixel_columns((r-1)*num_columns+c) = picture(r,c); %pixel# = (y*numColumns)+x
        end
end

%so now pixel_columns is a column vector of the pixel values in the image

% blue = color_table(:,3);
% scaled_data = blue*255;
% rounded_data = round(scaled_data);  %rounds them down
data = dec2bin(pixel_columns,8);      %convert the binary data to 8 bit binary #s

%open a file
output_name = 'pixel_values.coe';
file = fopen(output_name,'w');

%write the header info
fprintf(file,'memory_initialization_radix=2;\n');
fprintf(file,'memory_initialization_vector=\n');
fclose(file);

%put commas in the data
rowxcolumn = size(data);
rows = rowxcolumn(1);
columns = rowxcolumn(2);
output = data;
for i = 1:(rows-1)
        output(i,(columns+1)) = ',';
end
output(rows,(columns+1)) = ';';

%append the numeric values to the file
dlmwrite(output_name,output,'-append','delimiter','', 'newline', 'pc');

%You're done!

%just to make sure that we're doing things correctly
regen_picture = zeros(num_rows,num_columns,'uint8');
for r = 1:num_rows
        for c = 1:num_columns
        regen_picture(r,c) = pixel_columns((r-1)*num_columns+c,1);
```

```
        end
end

figure
subplot(121)
image(picture)
axis square
colormap(color_table)
colorbar
title('Original Picture')

subplot(122)
image(regen_picture)
axis square
colormap(color_table)
colorbar
title('Regenerated Picture')
```

11.2.2 Teensy 3.2 - imu_stream_bluetooth.ino

```
#include <MPU9250.h>
#include <math.h>

#define FPGA Serial1

MPU9250 imu;

int led = 13;
float accel_data[3] = {0};   // initialize array holding accel data
float gyro_data[3] = {0};   // initialize array holding gyro data


void setup()
{
  pinMode(led, OUTPUT);
  digitalWrite(led, HIGH);

  delay(1000);
  // Serial monitors
  Serial.begin(115200);
  FPGA.begin(115200); //500000
```

```
  digitalWrite(led, LOW);
  //setup bluetooth to connect with pair
  FPGA.print("$$$");  // Enter command mode
  delay(1000);  // Short delay, wait for the BlueSmirf to send back CMD
  FPGA.println("C,000666DACBBE");  // address of FPGA module


  // setup IMU
  byte c = imu.readByte(MPU9250_ADDRESS, WHO_AM_I_MPU9250);
  Serial.print("MPU9250 "); Serial.print("I AM "); Serial.print(c, HEX);
  Serial.println("MPU9250 is online...");
  digitalWrite(led, HIGH);

  // Calibrate gyro and accelerometers, load biases in bias registers
  imu.initMPU9250();
  //imu.MPU9250SelfTest(imu.selfTest);
  imu.calibrateMPU9250(imu.gyroBias, imu.accelBias);
  imu.initMPU9250();// need to reinitialize to make sure gyro configs are saved!
  imu.initAK8963(imu.factoryMagCalibration);
  imu.getAres();
  imu.getGres();
  imu.getMres();
}


void loop()
{
        // read accelerometer data
  imu.readAccelData(imu.accelCount);
  imu.readGyroData(imu.gyroCount);

  // testing!!!
  // print to serial output- pin 1 (3rd down on left)
  int16_t fake[3] = {0x1111, 0x1234, 0xFFFF};   // output as 3355_3355_8800

  // send start byte
  byte start_byte = 0x11;
  byte replace_byte = 0x10;
  FPGA.write(start_byte);
  //make sure to never send 0x00

  // load accel info into array
  byte accel[6];
```

```
  byte *a = accel;
  for(int i=0; i < 3; i++){
    //LSB (cannot be same as start byte)
    *a = imu.accelCount[i];//fake[i];//
    if (*a == start_byte) *a = replace_byte;
    a++;
    //MSB
    *a = (imu.accelCount[i]>>8);//(fake[i]>>8);//
    a++;
  }
  FPGA.write(accel,6);

  // load gyro info into array
  byte gyro[6];
  byte *g = gyro;
  for(int i=0; i < 3; i++){
    *g = imu.gyroCount[i];//fake[i];//
    if (*g == start_byte) *g = replace_byte;
    g++;
    *g = (imu.gyroCount[i]>>8);//(fake[i]>>8);//
    g++;
  }
  FPGA.write(gyro,6);

  //FPGA.write(imu.accelCount[0]>>8);

  delay(1.5); // 400Hz
}
```

11.2.3 IMU - MPU9250.h and .cpp

```
/*
 Note: The MPU9250 is an I2C sensor and uses the Arduino Wire library.
 Because the sensor is not 5V tolerant, we are using a 3.3 V 8 MHz Pro Mini or
 a 3.3 V Teensy 3.1. We have disabled the internal pull-ups used by the Wire
 library in the Wire.h/twi.c utility file. We are also using the 400 kHz fast
 I2C mode by setting the TWI_FREQ  to 400000L /twi.h utility file.
 */
```

```
#ifndef _MPU9250_H_
#define _MPU9250_H_

#include <SPI.h>
#include <Wire.h>

#define SERIAL_DEBUG true

// See also MPU-9250 Register Map and Descriptions, Revision 4.0,
// RM-MPU-9250A-00, Rev. 1.4, 9/9/2013 for registers not listed in above
// document; the MPU9250 and MPU9150 are virtually identical but the latter has
// a different register map

//Magnetometer Registers
#define AK8963_ADDRESS   0x0C
#define WHO_AM_I_AK8963  0x49 // (AKA WIA) should return 0x48
#define INFO             0x01
#define AK8963_ST1       0x02  // data ready status bit 0
#define AK8963_XOUT_L    0x03  // data
#define AK8963_XOUT_H    0x04
#define AK8963_YOUT_L    0x05
#define AK8963_YOUT_H    0x06
#define AK8963_ZOUT_L    0x07
#define AK8963_ZOUT_H    0x08
#define AK8963_ST2       0x09  // Data overflow bit 3 and data read error status bit 2
#define AK8963_CNTL      0x0A  // Power down (0000), single-measurement (0001), self-test
(1000) and Fuse ROM (1111) modes on bits 3:0
#define AK8963_ASTC      0x0C  // Self test control
#define AK8963_I2CDIS    0x0F  // I2C disable
#define AK8963_ASAX      0x10  // Fuse ROM x-axis sensitivity adjustment value
#define AK8963_ASAY      0x11  // Fuse ROM y-axis sensitivity adjustment value
#define AK8963_ASAZ      0x12  // Fuse ROM z-axis sensitivity adjustment value

#define SELF_TEST_X_GYRO 0x00
#define SELF_TEST_Y_GYRO 0x01
#define SELF_TEST_Z_GYRO 0x02

/*#define X_FINE_GAIN      0x03 // [7:0] fine gain
#define Y_FINE_GAIN      0x04
#define Z_FINE_GAIN      0x05
#define XA_OFFSET_H      0x06 // User-defined trim values for accelerometer
#define XA_OFFSET_L_TC   0x07
#define YA_OFFSET_H      0x08
```

```
#define YA_OFFSET_L_TC   0x09
#define ZA_OFFSET_H      0x0A
#define ZA_OFFSET_L_TC   0x0B */

#define SELF_TEST_X_ACCEL 0x0D
#define SELF_TEST_Y_ACCEL 0x0E
#define SELF_TEST_Z_ACCEL 0x0F

#define SELF_TEST_A      0x10

#define XG_OFFSET_H      0x13  // User-defined trim values for gyroscope
#define XG_OFFSET_L      0x14
#define YG_OFFSET_H      0x15
#define YG_OFFSET_L      0x16
#define ZG_OFFSET_H      0x17
#define ZG_OFFSET_L      0x18
#define SMPLRT_DIV       0x19
#define CONFIG           0x1A
#define GYRO_CONFIG      0x1B
#define ACCEL_CONFIG     0x1C
#define ACCEL_CONFIG2    0x1D
#define LP_ACCEL_ODR     0x1E
#define WOM_THR          0x1F

// Duration counter threshold for motion interrupt generation, 1 kHz rate,
// LSB = 1 ms
#define MOT_DUR          0x20
// Zero-motion detection threshold bits [7:0]
#define ZMOT_THR         0x21
// Duration counter threshold for zero motion interrupt generation, 16 Hz rate,
// LSB = 64 ms
#define ZRMOT_DUR        0x22

#define FIFO_EN          0x23
#define I2C_MST_CTRL     0x24
#define I2C_SLV0_ADDR    0x25
#define I2C_SLV0_REG     0x26
#define I2C_SLV0_CTRL    0x27
#define I2C_SLV1_ADDR    0x28
#define I2C_SLV1_REG     0x29
#define I2C_SLV1_CTRL    0x2A
#define I2C_SLV2_ADDR    0x2B
#define I2C_SLV2_REG     0x2C
```

```
#define I2C_SLV2_CTRL      0x2D
#define I2C_SLV3_ADDR      0x2E
#define I2C_SLV3_REG       0x2F
#define I2C_SLV3_CTRL      0x30
#define I2C_SLV4_ADDR      0x31
#define I2C_SLV4_REG       0x32
#define I2C_SLV4_DO        0x33
#define I2C_SLV4_CTRL      0x34
#define I2C_SLV4_DI        0x35
#define I2C_MST_STATUS     0x36
#define INT_PIN_CFG        0x37
#define INT_ENABLE         0x38
#define DMP_INT_STATUS     0x39  // Check DMP interrupt
#define INT_STATUS         0x3A
#define ACCEL_XOUT_H       0x3B
#define ACCEL_XOUT_L       0x3C
#define ACCEL_YOUT_H       0x3D
#define ACCEL_YOUT_L       0x3E
#define ACCEL_ZOUT_H       0x3F
#define ACCEL_ZOUT_L       0x40
#define TEMP_OUT_H         0x41
#define TEMP_OUT_L         0x42
#define GYRO_XOUT_H        0x43
#define GYRO_XOUT_L        0x44
#define GYRO_YOUT_H        0x45
#define GYRO_YOUT_L        0x46
#define GYRO_ZOUT_H        0x47
#define GYRO_ZOUT_L        0x48
#define EXT_SENS_DATA_00   0x49
#define EXT_SENS_DATA_01   0x4A
#define EXT_SENS_DATA_02   0x4B
#define EXT_SENS_DATA_03   0x4C
#define EXT_SENS_DATA_04   0x4D
#define EXT_SENS_DATA_05   0x4E
#define EXT_SENS_DATA_06   0x4F
#define EXT_SENS_DATA_07   0x50
#define EXT_SENS_DATA_08   0x51
#define EXT_SENS_DATA_09   0x52
#define EXT_SENS_DATA_10   0x53
#define EXT_SENS_DATA_11   0x54
#define EXT_SENS_DATA_12   0x55
#define EXT_SENS_DATA_13   0x56
#define EXT_SENS_DATA_14   0x57
```

```
#define EXT_SENS_DATA_15   0x58
#define EXT_SENS_DATA_16   0x59
#define EXT_SENS_DATA_17   0x5A
#define EXT_SENS_DATA_18   0x5B
#define EXT_SENS_DATA_19   0x5C
#define EXT_SENS_DATA_20   0x5D
#define EXT_SENS_DATA_21   0x5E
#define EXT_SENS_DATA_22   0x5F
#define EXT_SENS_DATA_23   0x60
#define MOT_DETECT_STATUS  0x61
#define I2C_SLV0_DO       0x63
#define I2C_SLV1_DO       0x64
#define I2C_SLV2_DO       0x65
#define I2C_SLV3_DO       0x66
#define I2C_MST_DELAY_CTRL 0x67
#define SIGNAL_PATH_RESET  0x68
#define MOT_DETECT_CTRL    0x69
#define USER_CTRL         0x6A  // Bit 7 enable DMP, bit 3 reset DMP
#define PWR_MGMT_1         0x6B // Device defaults to the SLEEP mode
#define PWR_MGMT_2         0x6C
#define DMP_BANK          0x6D  // Activates a specific bank in the DMP
#define DMP_RW_PNT         0x6E  // Set read/write pointer to a specific start address in
specified DMP bank
#define DMP_REG           0x6F  // Register in DMP from which to read or to which to write
#define DMP_REG_1         0x70
#define DMP_REG_2         0x71
#define FIFO_COUNTH       0x72
#define FIFO_COUNTL       0x73
#define FIFO_R_W         0x74
#define WHO_AM_I_MPU9250   0x75 // Should return 0x71
#define XA_OFFSET_H       0x77
#define XA_OFFSET_L       0x78
#define YA_OFFSET_H       0x7A
#define YA_OFFSET_L       0x7B
#define ZA_OFFSET_H       0x7D
#define ZA_OFFSET_L       0x7E

// Using the MPU-9250 breakout board, ADO is set to 0
// Seven-bit device address is 110100 for ADO = 0 and 110101 for ADO = 1
#define ADO 0
#if ADO
#define MPU9250_ADDRESS 0x69  // Device address when ADO = 1
#else
```

```
#define MPU9250_ADDRESS 0x68  // Device address when ADO = 0
#define AK8963_ADDRESS  0x0C   // Address of magnetometer
#endif // AD0

#define READ_FLAG 0x80
#define NOT_SPI -1
#define SPI_DATA_RATE 1000000 // 1MHz is the max speed of the MPU-9250
//#define SPI_DATA_RATE 1000000 // 1MHz is the max speed of the MPU-9250
#define SPI_MODE SPI_MODE3

class MPU9250
{
protected:
  // Set initial input parameters
  enum Ascale
  {
    AFS_2G = 0,
    AFS_4G,
    AFS_8G,
    AFS_16G
  };

  enum Gscale {
    GFS_250DPS = 0,
    GFS_500DPS,
    GFS_1000DPS,
    GFS_2000DPS
  };

  enum Mscale {
    MFS_14BITS = 0, // 0.6 mG per LSB
    MFS_16BITS      // 0.15 mG per LSB
  };

  enum M_MODE {
    M_8HZ = 0x02,  // 8 Hz update
    M_100HZ = 0x06 // 100 Hz continuous magnetometer
  };

  // TODO: Add setter methods for this hard coded stuff
  // Specify sensor full scale
  uint8_t Gscale = GFS_2000DPS;
  uint8_t Ascale = AFS_2G;
```

```
    // Choose either 14-bit or 16-bit magnetometer resolution
    uint8_t Mscale = MFS_16BITS;

    // 2 for 8 Hz, 6 for 100 Hz continuous magnetometer data read
    uint8_t Mmode = M_8HZ;

    // SPI chip select pin
    int8_t _csPin;

    uint8_t writeByteWire(uint8_t, uint8_t, uint8_t);
    uint8_t writeByteSPI(uint8_t, uint8_t);
    uint8_t readByteSPI(uint8_t subAddress);
    uint8_t readByteWire(uint8_t address, uint8_t subAddress);
    bool magInit();
    void kickHardware();
    void select();
    void deselect();
// TODO: Remove this next line
public:
    uint8_t ak8963WhoAmI_SPI();

  public:
    float pitch, yaw, roll;
    float temperature;   // Stores the real internal chip temperature in Celsius
    int16_t tempCount;   // Temperature raw count output
    uint32_t delt_t = 0; // Used to control display output rate

    uint32_t count = 0, sumCount = 0; // used to control display output rate
    float deltat = 0.0f, sum = 0.0f;  // integration interval for both filter schemes
    uint32_t lastUpdate = 0, firstUpdate = 0; // used to calculate integration interval
    uint32_t Now = 0;        // used to calculate integration interval

    int16_t gyroCount[3];   // Stores the 16-bit signed gyro sensor output
    int16_t magCount[3];    // Stores the 16-bit signed magnetometer sensor output
    // Scale resolutions per LSB for the sensors
    float aRes, gRes, mRes;
    // Variables to hold latest sensor data values
    float ax, ay, az, gx, gy, gz, mx, my, mz;
    // Factory mag calibration and mag bias
    float factoryMagCalibration[3] = {0, 0, 0}, factoryMagBias[3] = {0, 0, 0};
    // Bias corrections for gyro, accelerometer, and magnetometer
    float gyroBias[3]  = {0, 0, 0},
          accelBias[3] = {0, 0, 0},
```

```
        magBias[3]   = {0, 0, 0},
        magScale[3]  = {0, 0, 0};
    float selfTest[6];
    // Stores the 16-bit signed accelerometer sensor output
    int16_t accelCount[3];

    // Public method declarations
    MPU9250(int8_t csPin=NOT_SPI);
    void getMres();
    void getGres();
    void getAres();
    void readAccelData(int16_t *);
    void readGyroData(int16_t *);
    void readMagData(int16_t *);
    int16_t readTempData();
    void updateTime();
    void initAK8963(float *);
    void initMPU9250();
    void calibrateMPU9250(float * gyroBias, float * accelBias);
    void MPU9250SelfTest(float * destination);
    void magCalMPU9250(float * dest1, float * dest2);
    uint8_t writeByte(uint8_t, uint8_t, uint8_t);
    uint8_t readByte(uint8_t, uint8_t);
    uint8_t readBytes(uint8_t, uint8_t, uint8_t, uint8_t *);
    // TODO: make SPI/Wire private
    uint8_t readBytesSPI(uint8_t, uint8_t, uint8_t *);
    uint8_t readBytesWire(uint8_t, uint8_t, uint8_t, uint8_t *);
    bool isInI2cMode() { return _csPin == -1; }
    bool begin();
};  // class MPU9250

#endif // _MPU9250_H_


#include "MPU9250.h"

//==================================================================================
=======
//====== Set of useful function to access acceleration. gyroscope, magnetometer,
//====== and temperature data
//==================================================================================
=======
```

```
MPU9250::MPU9250(int8_t cspin /*=NOT_SPI*/) // Uses I2C communication by default
{
  // Use hardware SPI communication
  // If used with sparkfun breakout board
  // https://www.sparkfun.com/products/13762 , change the pre-soldered JP2 to
  // enable SPI (solder middle and left instead of middle and right) pads are
  // very small and re-soldering can be very tricky. I2C highly recommended.
  if ((cspin > NOT_SPI) && (cspin < NUM_DIGITAL_PINS))
  {
    _csPin = cspin;
    SPI.begin();
    pinMode(_csPin, OUTPUT);
    deselect();
  }
  else
  {
    _csPin = NOT_SPI;
    Wire.begin();
  }
}

void MPU9250::getMres()
{
  switch (Mscale)
  {
    // Possible magnetometer scales (and their register bit settings) are:
    // 14 bit resolution (0) and 16 bit resolution (1)
    case MFS_14BITS:
      mRes = 10.0f * 4912.0f / 8190.0f; // Proper scale to return milliGauss
      break;
    case MFS_16BITS:
      mRes = 10.0f * 4912.0f / 32760.0f; // Proper scale to return milliGauss
      break;
  }
}

void MPU9250::getGres()
{
  switch (Gscale)
  {
    // Possible gyro scales (and their register bit settings) are:
    // 250 DPS (00), 500 DPS (01), 1000 DPS (10), and 2000 DPS (11).
    // Here's a bit of an algorith to calculate DPS/(ADC tick) based on that
```

```
    // 2-bit value:
    case GFS_250DPS:
      gRes = 250.0f / 32768.0f;
      break;
    case GFS_500DPS:
      gRes = 500.0f / 32768.0f;
      break;
    case GFS_1000DPS:
      gRes = 1000.0f / 32768.0f;
      break;
    case GFS_2000DPS:
      gRes = 2000.0f / 32768.0f;
      break;
  }
}

void MPU9250::getAres()
{
  switch (Ascale)
  {
    // Possible accelerometer scales (and their register bit settings) are:
    // 2 Gs (00), 4 Gs (01), 8 Gs (10), and 16 Gs  (11).
    // Here's a bit of an algorith to calculate DPS/(ADC tick) based on that
    // 2-bit value:
    case AFS_2G:
      aRes = 2.0f / 32768.0f;
      break;
    case AFS_4G:
      aRes = 4.0f / 32768.0f;
      break;
    case AFS_8G:
      aRes = 8.0f / 32768.0f;
      break;
    case AFS_16G:
      aRes = 16.0f / 32768.0f;
      break;
  }
}


void MPU9250::readAccelData(int16_t * destination)
{
  uint8_t rawData[6];  // x/y/z accel register data stored here
```

```
  // Read the six raw data registers into data array
  readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);

  // Turn the MSB and LSB into a signed 16-bit value
  destination[0] = ((int16_t)rawData[0] << 8) | rawData[1] ;
  destination[1] = ((int16_t)rawData[2] << 8) | rawData[3] ;
  destination[2] = ((int16_t)rawData[4] << 8) | rawData[5] ;
}


void MPU9250::readGyroData(int16_t * destination)
{
  uint8_t rawData[6];  // x/y/z gyro register data stored here
  // Read the six raw data registers sequentially into data array
  readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);

  // Turn the MSB and LSB into a signed 16-bit value
  destination[0] = ((int16_t)rawData[0] << 8) | rawData[1] ;
  destination[1] = ((int16_t)rawData[2] << 8) | rawData[3] ;
  destination[2] = ((int16_t)rawData[4] << 8) | rawData[5] ;
}

void MPU9250::readMagData(int16_t * destination)
{
  // x/y/z gyro register data, ST2 register stored here, must read ST2 at end
  // of data acquisition
  uint8_t rawData[7];
  // Wait for magnetometer data ready bit to be set
  if (readByte(AK8963_ADDRESS, AK8963_ST1) & 0x01)
  {
    // Read the six raw data and ST2 registers sequentially into data array
    readBytes(AK8963_ADDRESS, AK8963_XOUT_L, 7, &rawData[0]);
    uint8_t c = rawData[6]; // End data read by reading ST2 register
    // Check if magnetic sensor overflow set, if not then report data
    if (!(c & 0x08))
    {
      // Turn the MSB and LSB into a signed 16-bit value
      destination[0] = ((int16_t)rawData[1] << 8) | rawData[0];
      // Data stored as little Endian
      destination[1] = ((int16_t)rawData[3] << 8) | rawData[2];
      destination[2] = ((int16_t)rawData[5] << 8) | rawData[4];
    }
  }
```

```
}

int16_t MPU9250::readTempData()
{
  uint8_t rawData[2]; // x/y/z gyro register data stored here
  // Read the two raw data registers sequentially into data array
  readBytes(MPU9250_ADDRESS, TEMP_OUT_H, 2, &rawData[0]);
  // Turn the MSB and LSB into a 16-bit value
  return ((int16_t)rawData[0] << 8) | rawData[1];
}

// Calculate the time the last update took for use in the quaternion filters
// TODO: This doesn't really belong in this class.
void MPU9250::updateTime()
{
  Now = micros();

  // Set integration time by time elapsed since last filter update
  deltat = ((Now - lastUpdate) / 1000000.0f);
  lastUpdate = Now;

  sum += deltat; // sum for averaging filter update rate
  sumCount++;
}

void MPU9250::initAK8963(float * destination)
{
  // First extract the factory calibration for each magnetometer axis
  uint8_t rawData[3];  // x/y/z gyro calibration data stored here
  // TODO: Test this!! Likely doesn't work
  writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down magnetometer
  delay(10);
  writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x0F); // Enter Fuse ROM access mode
  delay(10);

  // Read the x-, y-, and z-axis calibration values
  readBytes(AK8963_ADDRESS, AK8963_ASAX, 3, &rawData[0]);

  // Return x-axis sensitivity adjustment values, etc.
  destination[0] =  (float)(rawData[0] - 128)/256. + 1.;
  destination[1] =  (float)(rawData[1] - 128)/256. + 1.;
  destination[2] =  (float)(rawData[2] - 128)/256. + 1.;
  writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down magnetometer
```

```
  delay(10);

  // Configure the magnetometer for continuous read and highest resolution.
  // Set Mscale bit 4 to 1 (0) to enable 16 (14) bit resolution in CNTL
  // register, and enable continuous mode data acquisition Mmode (bits [3:0]),
  // 0010 for 8 Hz and 0110 for 100 Hz sample rates.

  // Set magnetometer data resolution and sample ODR
  writeByte(AK8963_ADDRESS, AK8963_CNTL, Mscale << 4 | Mmode);
  delay(10);
}

void MPU9250::initMPU9250()
{
  // wake up device
  // Clear sleep mode bit (6), enable all sensors
  writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00);
  delay(100); // Wait for all registers to reset

  // Get stable time source
  // Auto select clock source to be PLL gyroscope reference if ready else
  writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01);
  delay(200);

  // Configure Gyro and Thermometer
  // Disable FSYNC and set thermometer and gyro bandwidth to 41 and 42 Hz,
  // respectively;
  // minimum delay time for this setting is 5.9 ms, which means sensor fusion
  // update rates cannot be higher than 1 / 0.0059 = 170 Hz
  // DLPF_CFG = bits 2:0 = 011; this limits the sample rate to 1000 Hz for both
  // With the MPU9250, it is possible to get gyro sample rates of 32 kHz (!),
  // 8 kHz, or 1 kHz
  writeByte(MPU9250_ADDRESS, CONFIG, 0x03);

  // Set sample rate = gyroscope output rate/(1 + SMPLRT_DIV)
  // Use a 200 Hz rate; a rate consistent with the filter update rate
  // determined inset in CONFIG above.
  writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x04);

  // Set gyroscope full scale range
  // Range selects FS_SEL and AFS_SEL are 0 - 3, so 2-bit values are
  // left-shifted into positions 4:3
```

```
// get current GYRO_CONFIG register value
uint8_t c = readByte(MPU9250_ADDRESS, GYRO_CONFIG);
// c = c & ~0xE0; // Clear self-test bits [7:5]
c = c & ~0x02; // Clear Fchoice bits [1:0]
c = c & ~0x18; // Clear AFS bits [4:3]
c = c | Gscale << 3; // Set full scale range for the gyro
// Set Fchoice for the gyro to 11 by writing its inverse to bits 1:0 of
// GYRO_CONFIG
// c =| 0x00;
// Write new GYRO_CONFIG value to register
writeByte(MPU9250_ADDRESS, GYRO_CONFIG, c );

// Set accelerometer full-scale range configuration
// Get current ACCEL_CONFIG register value
c = readByte(MPU9250_ADDRESS, ACCEL_CONFIG);
// c = c & ~0xE0; // Clear self-test bits [7:5]
c = c & ~0x18;  // Clear AFS bits [4:3]
c = c | Ascale << 3; // Set full scale range for the accelerometer
// Write new ACCEL_CONFIG register value
writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, c);

// Set accelerometer sample rate configuration
// It is possible to get a 4 kHz sample rate from the accelerometer by
// choosing 1 for accel_fchoice_b bit [3]; in this case the bandwidth is
// 1.13 kHz
// Get current ACCEL_CONFIG2 register value
c = readByte(MPU9250_ADDRESS, ACCEL_CONFIG2);
c = c & ~0x0F; // Clear accel_fchoice_b (bit 3) and A_DLPFG (bits [2:0])
c = c | 0x03;  // Set accelerometer rate to 1 kHz and bandwidth to 41 Hz
// Write new ACCEL_CONFIG2 register value
writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, c);
// The accelerometer, gyro, and thermometer are set to 1 kHz sample rates,
// but all these rates are further reduced by a factor of 5 to 200 Hz because
// of the SMPLRT_DIV setting

// Configure Interrupts and Bypass Enable
// Set interrupt pin active high, push-pull, hold interrupt pin level HIGH
// until interrupt cleared, clear on read of INT_STATUS, and enable
// I2C_BYPASS_EN so additional chips can join the I2C bus and all can be
// controlled by the Arduino as master.
writeByte(MPU9250_ADDRESS, INT_PIN_CFG, 0x22);
// Enable data ready (bit 0) interrupt
writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x01);
```

```
  delay(100);
}


// Function which accumulates gyro and accelerometer data after device
// initialization. It calculates the average of the at-rest readings and then
// loads the resulting offsets into accelerometer and gyro bias registers.
void MPU9250::calibrateMPU9250(float * gyroBias, float * accelBias)
{
  uint8_t data[12]; // data array to hold accelerometer and gyro x, y, z, data
  uint16_t ii, packet_count, fifo_count;
  int32_t gyro_bias[3]  = {0, 0, 0}, accel_bias[3] = {0, 0, 0};

  // reset device
  // Write a one to bit 7 reset bit; toggle reset device
  writeByte(MPU9250_ADDRESS, PWR_MGMT_1, READ_FLAG);
  delay(100);

  // get stable time source; Auto select clock source to be PLL gyroscope
  // reference if ready else use the internal oscillator, bits 2:0 = 001
  writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01);
  writeByte(MPU9250_ADDRESS, PWR_MGMT_2, 0x00);
  delay(200);

  // Configure device for bias calculation
  // Disable all interrupts
  writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x00);
  // Disable FIFO
  writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00);
  // Turn on internal clock source
  writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00);
  // Disable I2C master
  writeByte(MPU9250_ADDRESS, I2C_MST_CTRL, 0x00);
  // Disable FIFO and I2C master modes
  writeByte(MPU9250_ADDRESS, USER_CTRL, 0x00);
  // Reset FIFO and DMP
  writeByte(MPU9250_ADDRESS, USER_CTRL, 0x0C);
  delay(15);

  // Configure MPU6050 gyro and accelerometer for bias calculation
  // Set low-pass filter to 188 Hz
  writeByte(MPU9250_ADDRESS, CONFIG, 0x01);
  // Set sample rate to 1 kHz
```

```
writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00);
// Set gyro full-scale to 250 degrees per second, maximum sensitivity
writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 0x00);
// Set accelerometer full-scale to 2 g, maximum sensitivity
writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0x00);

uint16_t  gyrosensitivity  = 131;   // = 131 LSB/degrees/sec
uint16_t  accelsensitivity = 16384; // = 16384 LSB/g

// Configure FIFO to capture accelerometer and gyro data for bias calculation
writeByte(MPU9250_ADDRESS, USER_CTRL, 0x40);  // Enable FIFO
// Enable gyro and accelerometer sensors for FIFO  (max size 512 bytes in
// MPU-9150)
writeByte(MPU9250_ADDRESS, FIFO_EN, 0x78);
delay(40);  // accumulate 40 samples in 40 milliseconds = 480 bytes

// At end of sample accumulation, turn off FIFO sensor read
// Disable gyro and accelerometer sensors for FIFO
writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00);
// Read FIFO sample count
readBytes(MPU9250_ADDRESS, FIFO_COUNTH, 2, &data[0]);
fifo_count = ((uint16_t)data[0] << 8) | data[1];
// How many sets of full gyro and accelerometer data for averaging
packet_count = fifo_count/12;

for (ii = 0; ii < packet_count; ii++)
{
  int16_t accel_temp[3] = {0, 0, 0}, gyro_temp[3] = {0, 0, 0};
  // Read data for averaging
  readBytes(MPU9250_ADDRESS, FIFO_R_W, 12, &data[0]);
  // Form signed 16-bit integer for each sample in FIFO
  accel_temp[0] = (int16_t) (((int16_t)data[0] << 8) | data[1]  );
  accel_temp[1] = (int16_t) (((int16_t)data[2] << 8) | data[3]  );
  accel_temp[2] = (int16_t) (((int16_t)data[4] << 8) | data[5]  );
  gyro_temp[0]  = (int16_t) (((int16_t)data[6] << 8) | data[7]  );
  gyro_temp[1]  = (int16_t) (((int16_t)data[8] << 8) | data[9]  );
  gyro_temp[2]  = (int16_t) (((int16_t)data[10] << 8) | data[11]);

  // Sum individual signed 16-bit biases to get accumulated signed 32-bit
  // biases.
  accel_bias[0] += (int32_t) accel_temp[0];
  accel_bias[1] += (int32_t) accel_temp[1];
  accel_bias[2] += (int32_t) accel_temp[2];
```

```
  gyro_bias[0]  += (int32_t) gyro_temp[0];
  gyro_bias[1]  += (int32_t) gyro_temp[1];
  gyro_bias[2]  += (int32_t) gyro_temp[2];
}
// Sum individual signed 16-bit biases to get accumulated signed 32-bit biases
accel_bias[0] /= (int32_t) packet_count;
accel_bias[1] /= (int32_t) packet_count;
accel_bias[2] /= (int32_t) packet_count;
gyro_bias[0]  /= (int32_t) packet_count;
gyro_bias[1]  /= (int32_t) packet_count;
gyro_bias[2]  /= (int32_t) packet_count;

// Sum individual signed 16-bit biases to get accumulated signed 32-bit biases
if (accel_bias[2] > 0L)
{
  accel_bias[2] -= (int32_t) accelsensitivity;
}
else
{
  accel_bias[2] += (int32_t) accelsensitivity;
}

// Construct the gyro biases for push to the hardware gyro bias registers,
// which are reset to zero upon device startup.
// Divide by 4 to get 32.9 LSB per deg/s to conform to expected bias input
// format.
data[0] = (-gyro_bias[0]/4  >> 8) & 0xFF;
// Biases are additive, so change sign on calculated average gyro biases
data[1] = (-gyro_bias[0]/4)       & 0xFF;
data[2] = (-gyro_bias[1]/4  >> 8) & 0xFF;
data[3] = (-gyro_bias[1]/4)       & 0xFF;
data[4] = (-gyro_bias[2]/4  >> 8) & 0xFF;
data[5] = (-gyro_bias[2]/4)       & 0xFF;

// Push gyro biases to hardware registers
writeByte(MPU9250_ADDRESS, XG_OFFSET_H, data[0]);
writeByte(MPU9250_ADDRESS, XG_OFFSET_L, data[1]);
writeByte(MPU9250_ADDRESS, YG_OFFSET_H, data[2]);
writeByte(MPU9250_ADDRESS, YG_OFFSET_L, data[3]);
writeByte(MPU9250_ADDRESS, ZG_OFFSET_H, data[4]);
writeByte(MPU9250_ADDRESS, ZG_OFFSET_L, data[5]);

// Output scaled gyro biases for display in the main program
```

```
gyroBias[0] = (float) gyro_bias[0]/(float) gyrosensitivity;
gyroBias[1] = (float) gyro_bias[1]/(float) gyrosensitivity;
gyroBias[2] = (float) gyro_bias[2]/(float) gyrosensitivity;

// Construct the accelerometer biases for push to the hardware accelerometer
// bias registers. These registers contain factory trim values which must be
// added to the calculated accelerometer biases; on boot up these registers
// will hold non-zero values. In addition, bit 0 of the lower byte must be
// preserved since it is used for temperature compensation calculations.
// Accelerometer bias registers expect bias input as 2048 LSB per g, so that
// the accelerometer biases calculated above must be divided by 8.

// A place to hold the factory accelerometer trim biases
int32_t accel_bias_reg[3] = {0, 0, 0};
// Read factory accelerometer trim values
readBytes(MPU9250_ADDRESS, XA_OFFSET_H, 2, &data[0]);
accel_bias_reg[0] = (int32_t) (((int16_t)data[0] << 8) | data[1]);
readBytes(MPU9250_ADDRESS, YA_OFFSET_H, 2, &data[0]);
accel_bias_reg[1] = (int32_t) (((int16_t)data[0] << 8) | data[1]);
readBytes(MPU9250_ADDRESS, ZA_OFFSET_H, 2, &data[0]);
accel_bias_reg[2] = (int32_t) (((int16_t)data[0] << 8) | data[1]);

// Define mask for temperature compensation bit 0 of lower byte of
// accelerometer bias registers
uint32_t mask = 1uL;
// Define array to hold mask bit for each accelerometer bias axis
uint8_t mask_bit[3] = {0, 0, 0};

for (ii = 0; ii < 3; ii++)
{
  // If temperature compensation bit is set, record that fact in mask_bit
  if ((accel_bias_reg[ii] & mask))
  {
    mask_bit[ii] = 0x01;
  }
}

// Construct total accelerometer bias, including calculated average
// accelerometer bias from above
// Subtract calculated averaged accelerometer bias scaled to 2048 LSB/g
// (16 g full scale)
accel_bias_reg[0] -= (accel_bias[0]/8);
accel_bias_reg[1] -= (accel_bias[1]/8);
```

```
  accel_bias_reg[2] -= (accel_bias[2]/8);

  data[0] = (accel_bias_reg[0] >> 8) & 0xFF;
  data[1] = (accel_bias_reg[0])      & 0xFF;
  // preserve temperature compensation bit when writing back to accelerometer
  // bias registers
  data[1] = data[1] | mask_bit[0];
  data[2] = (accel_bias_reg[1] >> 8) & 0xFF;
  data[3] = (accel_bias_reg[1])      & 0xFF;
  // Preserve temperature compensation bit when writing back to accelerometer
  // bias registers
  data[3] = data[3] | mask_bit[1];
  data[4] = (accel_bias_reg[2] >> 8) & 0xFF;
  data[5] = (accel_bias_reg[2])      & 0xFF;
  // Preserve temperature compensation bit when writing back to accelerometer
  // bias registers
  data[5] = data[5] | mask_bit[2];

  // Apparently this is not working for the acceleration biases in the MPU-9250
  // Are we handling the temperature correction bit properly?
  // Push accelerometer biases to hardware registers
  writeByte(MPU9250_ADDRESS, XA_OFFSET_H, data[0]);
  writeByte(MPU9250_ADDRESS, XA_OFFSET_L, data[1]);
  writeByte(MPU9250_ADDRESS, YA_OFFSET_H, data[2]);
  writeByte(MPU9250_ADDRESS, YA_OFFSET_L, data[3]);
  writeByte(MPU9250_ADDRESS, ZA_OFFSET_H, data[4]);
  writeByte(MPU9250_ADDRESS, ZA_OFFSET_L, data[5]);

  // Output scaled accelerometer biases for display in the main program
  accelBias[0] = (float)accel_bias[0]/(float)accelsensitivity;
  accelBias[1] = (float)accel_bias[1]/(float)accelsensitivity;
  accelBias[2] = (float)accel_bias[2]/(float)accelsensitivity;
}


// Accelerometer and gyroscope self test; check calibration wrt factory settings
// Should return percent deviation from factory trim values, +/- 14 or less
// deviation is a pass.
void MPU9250::MPU9250SelfTest(float * destination)
{
  uint8_t rawData[6] = {0, 0, 0, 0, 0, 0};
  uint8_t selfTest[6];
  int32_t gAvg[3] = {0}, aAvg[3] = {0}, aSTAvg[3] = {0}, gSTAvg[3] = {0};
```

```
  float factoryTrim[6];
  uint8_t FS = 0;

  // Set gyro sample rate to 1 kHz
  writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00);
  // Set gyro sample rate to 1 kHz and DLPF to 92 Hz
  writeByte(MPU9250_ADDRESS, CONFIG, 0x02);
  // Set full scale range for the gyro to 250 dps
  writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 1<<FS);
  // Set accelerometer rate to 1 kHz and bandwidth to 92 Hz
  writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, 0x02);
  // Set full scale range for the accelerometer to 2 g
  writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 1<<FS);

  // Get average current values of gyro and acclerometer
  for (int ii = 0; ii < 200; ii++)
  {
Serial.print("BHW::ii = ");
Serial.println(ii);
    // Read the six raw data registers into data array
    readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);
    // Turn the MSB and LSB into a signed 16-bit value
    aAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;
    aAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
    aAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;

    // Read the six raw data registers sequentially into data array
    readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);
    // Turn the MSB and LSB into a signed 16-bit value
    gAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;
    gAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
    gAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
  }

  // Get average of 200 values and store as average current readings
  for (int ii =0; ii < 3; ii++)
  {
    aAvg[ii] /= 200;
    gAvg[ii] /= 200;
  }

  // Configure the accelerometer for self-test
  // Enable self test on all three axes and set accelerometer range to +/- 2 g
```

```
writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0xE0);
// Enable self test on all three axes and set gyro range to +/- 250 degrees/s
writeByte(MPU9250_ADDRESS, GYRO_CONFIG,  0xE0);
delay(25);  // Delay a while to let the device stabilize

// Get average self-test values of gyro and acclerometer
for (int ii = 0; ii < 200; ii++)
{
  // Read the six raw data registers into data array
  readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);
  // Turn the MSB and LSB into a signed 16-bit value
  aSTAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;
  aSTAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
  aSTAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;

  // Read the six raw data registers sequentially into data array
  readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);
  // Turn the MSB and LSB into a signed 16-bit value
  gSTAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;
  gSTAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
  gSTAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
}

// Get average of 200 values and store as average self-test readings
for (int ii =0; ii < 3; ii++)
{
  aSTAvg[ii] /= 200;
  gSTAvg[ii] /= 200;
}

// Configure the gyro and accelerometer for normal operation
writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0x00);
writeByte(MPU9250_ADDRESS, GYRO_CONFIG,  0x00);
delay(25);  // Delay a while to let the device stabilize

// Retrieve accelerometer and gyro factory Self-Test Code from USR_Reg
// X-axis accel self-test results
selfTest[0] = readByte(MPU9250_ADDRESS, SELF_TEST_X_ACCEL);
// Y-axis accel self-test results
selfTest[1] = readByte(MPU9250_ADDRESS, SELF_TEST_Y_ACCEL);
// Z-axis accel self-test results
selfTest[2] = readByte(MPU9250_ADDRESS, SELF_TEST_Z_ACCEL);
// X-axis gyro self-test results
```

```
  selfTest[3] = readByte(MPU9250_ADDRESS, SELF_TEST_X_GYRO);
  // Y-axis gyro self-test results
  selfTest[4] = readByte(MPU9250_ADDRESS, SELF_TEST_Y_GYRO);
  // Z-axis gyro self-test results
  selfTest[5] = readByte(MPU9250_ADDRESS, SELF_TEST_Z_GYRO);

  // Retrieve factory self-test value from self-test code reads
  // FT[Xa] factory trim calculation
  factoryTrim[0] = (float)(2620/1<<FS)*(pow(1.01 ,((float)selfTest[0] - 1.0) ));
  // FT[Ya] factory trim calculation
  factoryTrim[1] = (float)(2620/1<<FS)*(pow(1.01 ,((float)selfTest[1] - 1.0) ));
  // FT[Za] factory trim calculation
  factoryTrim[2] = (float)(2620/1<<FS)*(pow(1.01 ,((float)selfTest[2] - 1.0) ));
  // FT[Xg] factory trim calculation
  factoryTrim[3] = (float)(2620/1<<FS)*(pow(1.01 ,((float)selfTest[3] - 1.0) ));
  // FT[Yg] factory trim calculation
  factoryTrim[4] = (float)(2620/1<<FS)*(pow(1.01 ,((float)selfTest[4] - 1.0) ));
  // FT[Zg] factory trim calculation
  factoryTrim[5] = (float)(2620/1<<FS)*(pow(1.01 ,((float)selfTest[5] - 1.0) ));

  // Report results as a ratio of (STR - FT)/FT; the change from Factory Trim
  // of the Self-Test Response
  // To get percent, must multiply by 100
  for (int i = 0; i < 3; i++)
  {
    // Report percent differences
    destination[i] = 100.0 * ((float)(aSTAvg[i] - aAvg[i])) / factoryTrim[i]
      - 100.;
    // Report percent differences
    destination[i+3] = 100.0*((float)(gSTAvg[i] - gAvg[i]))/factoryTrim[i+3]
      - 100.;
  }
}

// Function which accumulates magnetometer data after device initialization.
// It calculates the bias and scale in the x, y, and z axes.
void MPU9250::magCalMPU9250(float * bias_dest, float * scale_dest)
{
  uint16_t ii = 0, sample_count = 0;
  int32_t mag_bias[3]  = {0, 0, 0},
          mag_scale[3] = {0, 0, 0};
  int16_t mag_max[3]  = {0x8000, 0x8000, 0x8000},
          mag_min[3]  = {0x7FFF, 0x7FFF, 0x7FFF},
```

```
      mag_temp[3] = {0, 0, 0};

  // Make sure resolution has been calculated
  getMres();

  Serial.println(F("Mag Calibration: Wave device in a figure 8 until done!"));
  Serial.println(
    F("  4 seconds to get ready followed by 15 seconds of sampling)"));
  delay(4000);

  // shoot for ~fifteen seconds of mag data
  // at 8 Hz ODR, new mag data is available every 125 ms
  if (Mmode == M_8HZ)
  {
    sample_count = 128;
  }
  // at 100 Hz ODR, new mag data is available every 10 ms
  if (Mmode == M_100HZ)
  {
    sample_count = 1500;
  }

  for (ii = 0; ii < sample_count; ii++)
  {
    readMagData(mag_temp);  // Read the mag data

    for (int jj = 0; jj < 3; jj++)
    {
      if (mag_temp[jj] > mag_max[jj])
      {
        mag_max[jj] = mag_temp[jj];
      }
      if (mag_temp[jj] < mag_min[jj])
      {
        mag_min[jj] = mag_temp[jj];
      }
    }

    if (Mmode == M_8HZ)
    {
      delay(135); // At 8 Hz ODR, new mag data is available every 125 ms
    }
    if (Mmode == M_100HZ)
```

```
    {
      delay(12);  // At 100 Hz ODR, new mag data is available every 10 ms
    }
  }

  // Serial.println("mag x min/max:"); Serial.println(mag_max[0]); Serial.println(mag_min[0]);
  // Serial.println("mag y min/max:"); Serial.println(mag_max[1]); Serial.println(mag_min[1]);
  // Serial.println("mag z min/max:"); Serial.println(mag_max[2]); Serial.println(mag_min[2]);

  // Get hard iron correction
  // Get 'average' x mag bias in counts
  mag_bias[0]  = (mag_max[0] + mag_min[0]) / 2;
  // Get 'average' y mag bias in counts
  mag_bias[1]  = (mag_max[1] + mag_min[1]) / 2;
  // Get 'average' z mag bias in counts
  mag_bias[2]  = (mag_max[2] + mag_min[2]) / 2;

  // Save mag biases in G for main program
  bias_dest[0] = (float)mag_bias[0] * mRes * factoryMagCalibration[0];
  bias_dest[1] = (float)mag_bias[1] * mRes * factoryMagCalibration[1];
  bias_dest[2] = (float)mag_bias[2] * mRes * factoryMagCalibration[2];

  // Get soft iron correction estimate
  // Get average x axis max chord length in counts
  mag_scale[0]  = (mag_max[0] - mag_min[0]) / 2;
  // Get average y axis max chord length in counts
  mag_scale[1]  = (mag_max[1] - mag_min[1]) / 2;
  // Get average z axis max chord length in counts
  mag_scale[2]  = (mag_max[2] - mag_min[2]) / 2;

  float avg_rad = mag_scale[0] + mag_scale[1] + mag_scale[2];
  avg_rad /= 3.0;

  scale_dest[0] = avg_rad / ((float)mag_scale[0]);
  scale_dest[1] = avg_rad / ((float)mag_scale[1]);
  scale_dest[2] = avg_rad / ((float)mag_scale[2]);

  Serial.println(F("Mag Calibration done!"));
}

// Wire.h read and write protocols
uint8_t MPU9250::writeByte(uint8_t deviceAddress, uint8_t registerAddress,
                        uint8_t data)
```

```
{
  if (_csPin != NOT_SPI)
  {
    return writeByteSPI(registerAddress, data);
  }
  else
  {
    return writeByteWire(deviceAddress,registerAddress, data);
  }
}

uint8_t MPU9250::writeByteSPI(uint8_t registerAddress, uint8_t writeData)
{
  uint8_t returnVal;

  SPI.beginTransaction(SPISettings(SPI_DATA_RATE, MSBFIRST, SPI_MODE));
  select();

  SPI.transfer(registerAddress);
  returnVal = SPI.transfer(writeData);

  deselect();
  SPI.endTransaction();
#ifdef SERIAL_DEBUG
  Serial.print("MPU9250::writeByteSPI slave returned: 0x");
  Serial.println(returnVal, HEX);
#endif
  return returnVal;
}

uint8_t MPU9250::writeByteWire(uint8_t deviceAddress, uint8_t registerAddress,
                            uint8_t data)
{
  Wire.beginTransmission(deviceAddress);  // Initialize the Tx buffer
  Wire.write(registerAddress);      // Put slave register address in Tx buffer
  Wire.write(data);                 // Put data in Tx buffer
  Wire.endTransmission();           // Send the Tx buffer
  // TODO: Fix this to return something meaningful
  return NULL;
}

// Read a byte from given register on device. Calls necessary SPI or I2C
// implementation. This was configured in the constructor.
```

```
uint8_t MPU9250::readByte(uint8_t deviceAddress, uint8_t registerAddress)
{
  if (_csPin != NOT_SPI)
  {
    return readByteSPI(registerAddress);
  }
  else
  {
    return readByteWire(deviceAddress, registerAddress);
  }
}

// Read a byte from the given register address from device using I2C
uint8_t MPU9250::readByteWire(uint8_t deviceAddress, uint8_t registerAddress)
{
  uint8_t data; // `data` will store the register data

  // Initialize the Tx buffer
  Wire.beginTransmission(deviceAddress);
  // Put slave register address in Tx buffer
  Wire.write(registerAddress);
  // Send the Tx buffer, but send a restart to keep connection alive
  Wire.endTransmission(false);
  // Read one byte from slave register address
  Wire.requestFrom(deviceAddress, (uint8_t) 1);
  // Fill Rx buffer with result
  data = Wire.read();
  // Return data read from slave register
  return data;
}

// Read a byte from the given register address using SPI
uint8_t MPU9250::readByteSPI(uint8_t registerAddress)
{
  return writeByteSPI(registerAddress | READ_FLAG, 0xFF /*0xFF is arbitrary*/);
}

// Read 1 or more bytes from given register and device using I2C
uint8_t MPU9250::readBytesWire(uint8_t deviceAddress, uint8_t registerAddress,
                        uint8_t count, uint8_t * dest)
{
  // Initialize the Tx buffer
  Wire.beginTransmission(deviceAddress);
```

```
  // Put slave register address in Tx buffer
  Wire.write(registerAddress);
  // Send the Tx buffer, but send a restart to keep connection alive
  Wire.endTransmission(false);

  uint8_t i = 0;
  // Read bytes from slave register address
  Wire.requestFrom(deviceAddress, count);
  while (Wire.available())
  {
    // Put read results in the Rx buffer
    dest[i++] = Wire.read();
  }

  return i; // Return number of bytes written
}

// Select slave IC by asserting CS pin
void MPU9250::select()
{
  digitalWrite(_csPin, LOW);
}

// Select slave IC by deasserting CS pin
void MPU9250::deselect()
{
  digitalWrite(_csPin, HIGH);
}

uint8_t MPU9250::readBytesSPI(uint8_t registerAddress, uint8_t count,
                              uint8_t * dest)
{
  SPI.beginTransaction(SPISettings(SPI_DATA_RATE, MSBFIRST, SPI_MODE));
  select();

  SPI.transfer(registerAddress | READ_FLAG);

  uint8_t i;

  for (i = 0; i < count; i++)
  {
    dest[i] = SPI.transfer(0x00);
#ifdef SERIAL_DEBUG
```

```
    Serial.print("readBytesSPI::Read byte: 0x");
    Serial.println(dest[i], HEX);
#endif
  }

  SPI.endTransaction();
  deselect();

  delayMicroseconds(50);

  return i; // Return number of bytes written

  /*
#ifdef SERIAL_DEBUG
  Serial.print("MPU9250::writeByteSPI slave returned: 0x");
  Serial.println(returnVal, HEX);
#endif
  return returnVal;
  */

  /*
  // Set slave address of AK8963 and set AK8963 for read
  writeByteSPI(I2C_SLV0_ADDR, AK8963_ADDRESS | READ_FLAG);

Serial.print("\nBHW::I2C_SLV0_ADDR set to: 0x");
Serial.println(readByte(MPU9250_ADDRESS, I2C_SLV0_ADDR), HEX);

  // Set address to start read from
  writeByteSPI(I2C_SLV0_REG, registerAddress);
  // Read bytes from magnetometer
  //
Serial.print("\nBHW::I2C_SLV0_CTRL gets 0x");
Serial.println(READ_FLAG | count, HEX);

  // Read count bytes from registerAddress via I2C_SLV0
  Serial.print("BHW::readBytesSPI: return value test: ");
  Serial.println(writeByteSPI(I2C_SLV0_CTRL, READ_FLAG | count));
  */
}

uint8_t MPU9250::readBytes(uint8_t deviceAddress, uint8_t registerAddress,
                        uint8_t count, uint8_t * dest)
{
```

```
  if (_csPin == NOT_SPI)  // Read via I2C
  {
    return readBytesWire(deviceAddress, registerAddress, count, dest);
  }
  else  // Read using SPI
  {
    return readBytesSPI(registerAddress, count, dest);
  }
}

bool MPU9250::magInit()
{
  // Reset registers to defaults, bit auto clears
  writeByteSPI(0x6B, 0x80);
  // Auto select the best available clock source
  writeByteSPI(0x6B, 0x01);
  // Enable X,Y, & Z axes of accel and gyro
  writeByteSPI(0x6C, 0x00);
  // Config disable FSYNC pin, set gyro/temp bandwidth to 184/188 Hz
  writeByteSPI(0x1A, 0x01);
  // Self tests off, gyro set to +/-2000 dps FS
  writeByteSPI(0x1B, 0x18);
  // Self test off, accel set to +/- 8g FS
  writeByteSPI(0x1C, 0x08);
  // Bypass DLPF and set accel bandwidth to 184 Hz
  writeByteSPI(0x1D, 0x09);
  // Configure INT pin (active high / push-pull / latch until read)
  writeByteSPI(0x37, 0x30);
  // Enable I2C master mode
  // TODO Why not do this 11-100 ms after power up?
  writeByteSPI(0x6A, 0x20);
  // Disable multi-master and set I2C master clock to 400 kHz
  //https://developer.mbed.org/users/kylongmu/code/MPU9250_SPI/ calls says
  // enabled multi-master... TODO Find out why
  writeByteSPI(0x24, 0x0D);
  // Set to write to slave address 0x0C
  writeByteSPI(0x25, 0x0C);
  // Point save 0 register at AK8963's control 2 (soft reset) register
  writeByteSPI(0x26, 0x0B);
  // Send 0x01 to AK8963 via slave 0 to trigger a soft restart
  writeByteSPI(0x63, 0x01);
  // Enable simple 1-byte I2C reads from slave 0
  writeByteSPI(0x27, 0x81);
```

```
  // Point save 0 register at AK8963's control 1 (mode) register
  writeByteSPI(0x26, 0x0A);
  // 16-bit continuous measurement mode 1
  writeByteSPI(0x63, 0x12);
  // Enable simple 1-byte I2C reads from slave 0
  writeByteSPI(0x27, 0x81);

  // TODO: Remove this code
  uint8_t ret = ak8963WhoAmI_SPI();
#ifdef SERIAL_DEBUG
  Serial.print("MPU9250::magInit to return ");
  Serial.println((ret == 0x48) ? "true" : "false");
#endif
  return ret == 0x48;
}

// Write a null byte w/o CS assertion to get SPI hardware to idle high (mode 3)
void MPU9250::kickHardware()
{
  SPI.beginTransaction(SPISettings(SPI_DATA_RATE, MSBFIRST, SPI_MODE));
  SPI.transfer(0x00); // Send null byte
  SPI.endTransaction();
}

bool MPU9250::begin()
{
  kickHardware();
  return magInit();
}

// Read the WHOAMI (WIA) register of the AK8963
// TODO: This method has side effects
uint8_t MPU9250::ak8963WhoAmI_SPI()
{
  uint8_t response, oldSlaveAddress, oldSlaveRegister, oldSlaveConfig;
  // Save state
  oldSlaveAddress  = readByteSPI(I2C_SLV0_ADDR);
  oldSlaveRegister = readByteSPI(I2C_SLV0_REG);
  oldSlaveConfig   = readByteSPI(I2C_SLV0_CTRL);
#ifdef SERIAL_DEBUG
  Serial.print("Old slave address: 0x");
  Serial.println(oldSlaveAddress, HEX);
  Serial.print("Old slave register: 0x");
```

```
  Serial.println(oldSlaveRegister, HEX);
  Serial.print("Old slave config: 0x");
  Serial.println(oldSlaveConfig, HEX);
#endif

  // Set the I2C slave addres of AK8963 and set for read
  response = writeByteSPI(I2C_SLV0_ADDR, AK8963_ADDRESS|READ_FLAG);
  // I2C slave 0 register address from where to begin data transfer
  response = writeByteSPI(I2C_SLV0_REG, 0x00);
  // Enable 1-byte reads on slave 0
  response = writeByteSPI(I2C_SLV0_CTRL, 0x81);
  delayMicroseconds(1);
  // Read WIA register
  response = writeByteSPI(WHO_AM_I_AK8963|READ_FLAG, 0x00);

  // Restore state
  writeByteSPI(I2C_SLV0_ADDR, oldSlaveAddress);
  writeByteSPI(I2C_SLV0_REG, oldSlaveRegister);
  writeByteSPI(I2C_SLV0_CTRL, oldSlaveConfig);

  return response;
}
```

11.2.4 Bluetooth - bluetooth_passthrough.ino

```
/*
  Based on example Bluetooth Serial Passthrough Sketch
 by: Jim Lindblom
 SparkFun Electronics
 date: February 26, 2013
 license: Public domain
 */

#include <SoftwareSerial.h>

int bluetoothTx = 0;  // TX-O pin of bluetooth mate, Teensy 0 (RX)
int bluetoothRx = 1;  // RX-I pin of bluetooth mate, Teensy 1 (TX)

SoftwareSerial bluetooth(bluetoothTx, bluetoothRx);
```

```
void setup()
{
  Serial.begin(115200);  // Begin the serial monitor at 9600bps

  pinMode(13, OUTPUT);
  digitalWrite(13, HIGH);

  bluetooth.begin(115200);  // The Bluetooth Mate defaults to 115200bps

  // autoconnect command
  bluetooth.print("$$$");  // Enter command mode
  delay(1000);  // Short delay, wait for the Mate to send back CMD
  bluetooth.println("C,000666DACBBE");
}

void loop()
{
  if(bluetooth.available())  // If the bluetooth sent any characters
  {
    // Send any characters the bluetooth prints to the serial monitor
    Serial.print((char)bluetooth.read());
  }
  if(Serial.available())  // If stuff was typed in the serial monitor
  {
    // Send any characters the Serial monitor prints to the bluetooth
    bluetooth.print((char)Serial.read());
  }
  // and loop forever and ever!
}

// setup instructions: https://learn.sparkfun.com/tutorials/using-the-bluesmirf
```

11.3 Verilog Code

11.3.1 labkit.v

```
`timescale 1ns / 1ps

module labkit(
   input CLK100MHZ,
```

```
   input[15:0] SW,
   input BTNC, BTNU, BTNL, BTNR, BTND,
   output[3:0] VGA_R,
   output[3:0] VGA_B,
   output[3:0] VGA_G,
   inout[7:0] JA,
   output VGA_HS,
   output VGA_VS,
   output LED16_B, LED16_G, LED16_R,
   output LED17_B, LED17_G, LED17_R,
   output[15:0] LED,
   output[7:0] SEG,  // segments A-G (0-6), DP (7)
   output[7:0] AN        // Display 0-7
   );

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
//
// SYSTEM
//
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
        // create 65mhz system clock
        wire clock_65mhz;
        clk_gen_65mhz clk_65(.clk_in1(CLK100MHZ), .reset(0), .clk_out1(clock_65mhz),
.locked());

        // ENTER button is user reset
        wire reset,user_reset;
        debounce
db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(BTNC),.clean(user_reset));
        assign reset = user_reset;

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
//
// IOs
//
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
        // debounce all buttons
        wire up,down, left, right;
        debounce db2(.reset(reset),.clock(clock_65mhz),.noisy(BTNU),.clean(up));
```

```
	debounce db3(.reset(reset),.clock(clock_65mhz),.noisy(BTND),.clean(down));
	debounce db4(.reset(reset),.clock(clock_65mhz),.noisy(BTNL),.clean(left));
    debounce db5(.reset(reset),.clock(clock_65mhz),.noisy(BTNR),.clean(right));

	// define inputs/outputs
	assign LED = SW;
	assign JA[7:6] = 0;
	assign JA[2] = 0;

	// set up footpedal to signal calibrate
	wire footpedal;
	debounce
#(.DELAY(500000))db(.reset(reset),.clock(clock_65mhz),.noisy(JA[1]),.clean(footpedal));
	assign calibrate = up|footpedal;

	// set up siren output on calibrate signal
	assign siren = JA[0];
	siren siren1 (.clk(clock_65mhz), .reset(reset), .on(calibrate), .siren(siren));

	// for testing LED strip
	parameter [31:0] START_BIT = 32'b0;
	parameter [31:0] STOP_BIT = 32'hFFFF_FFFF;
	parameter [31:0] LED_BIT = {3'b111,29'b0};
	reg BTNR_pulse;
	reg BTNR_prev;
	always @(posedge clock_65mhz) begin
	BTNR_prev <= right;
	BTNR_pulse <= right !== BTNR_prev;
	end
	wire [31:0] led_bit = right ? LED_BIT : STOP_BIT;
	// set up LED strip
	ledTX #(.SIZE(1024)) leds(.clk(clock_65mhz), .reset(reset), .packet(led_full_sequence),
.update(led_update), .signal(JA[4]), .led_clk(JA[5]));

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
//
// BAT
//
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
	// set up serial packet reader
	wire signal;
```

```
	synchronize sync(.clk(clock_65mhz), .in(JA[3]), .out(signal));
	wire [7:0] packet;
	wire done_serial;
	serialRX serialRX1(.clk(clock_65mhz), .reset(reset), .serial_in(signal), .data(packet),
.done(done_serial));

	// set up imu reader
	wire [15:0] accelx, accely, accelz, gyrox, gyroy, gyroz;
	wire done_imu;
	bluetoothRX bluetoothRX1(.clk(clock_65mhz), .reset(reset), .data(packet),
.write(done_serial),
		.accelx(accelx), .accely(accely), .accelz(accelz), .gyrox(gyrox), .gyroy(gyroy),
.gyroz(gyroz), .done(done_imu));

	// set up bat calculator
	wire [7:0] x,y;
	wire direction, swing;
	wire [7:0] test_x,test_y;		// for debugging
	wire [1:0] bat_state;
	wire swing,batter_up;
	bat bat1(.clk(clock_65mhz), .reset(reset), .start(calibrate), .ready(done_imu),
			.gyrox(gyrox), .gyroy(gyroy), .gyroz(gyroz), .accelx(accelx),
.accely(accely), .accelz(accelz),
			.X(x), .Y(y), .direction(direction), .done(swing), .batter_ready(batter_up),
			.state(bat_state), .pos0(LED16_B), .test_x(test_x), .test_y(test_y),
.test_dir(LED17_G), .SW(SW[15:3]));
	assign LED17_R = ~LED17_G;	// visually display up/down direction of bat in real time

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
//
// FSM
//
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
	// set up FSM
	wire [3:0] fsm_state;
	fsm gameplay(.clock(clock_65mhz), .reset(reset), .batter_up(batter_up|left),
.swing(swing|down), .ball_clock(xvga_screen_pulse), .hit(hit),
	.out_state(fsm_state), .bat_enable(bat_enable), .ball_speed(ball_speed),
.swing_time(swing_time), .start_pitch(start_pitch), .hits(hits),
	.misses(misses), .start_ball_moving(start_ball_moving));
```

```
	// instantiate 7-segment display
	wire [31:0] data;
	wire [6:0] segments;
	display_8hex display(.clk(clock_65mhz),.data(data), .seg(segments), .strobe(AN));
	assign SEG[6:0] = segments;
	assign SEG[7] = 1'b1;

	// display hits and misses in dec on hex_disp
	wire [13:0] hits,misses;
	wire [7:0] hits_dec, misses_dec;
	hex_to_dec hit_dec(.clk(clock_65mhz), .reset(reset), .hex(hits), .dec(hits_dec));
	hex_to_dec miss_dec(.clk(clock_65mhz), .reset(reset), .hex(misses), .dec(misses_dec));
	assign data = {8'b0, hits_dec, 8'b0, misses_dec};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
//
// GRAPHICS
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
	// generate basic XVGA video signals
	wire [10:0] hcount;
	wire [9:0]  vcount;
	wire hsync,vsync,blank;
	xvga xvga(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
	.hsync(hsync),.vsync(vsync),.blank(blank));

	// feed XVGA signals to graphics module
	wire [23:0] pixel;
	wire phsync,pvsync,pblank;
	wire xvga_screen_pulse;
	wire hit;
	wire bat_enable;
	wire [1:0] swing_time;
	wire start_pitch;
	wire [1023:0] led_full_sequence;
	wire led_update;
	wire [1:0] ball_speed = SW[2] + 1;
	wire manual = SW[8];
	wire start_ball_moving;
	graphics pg(.vclock(clock_65mhz),.reset(reset), .hcount(hcount),.vcount(vcount),
.hsync(hsync),.vsync(vsync),.blank(blank),
```

```
	.bat_up(manual?SW[15]:direction), .x_accel(manual?SW[14:12]:x),
.y_accel(manual?SW[11:9]:y), .start_pitch(start_pitch), .bat_enable(bat_enable),
	.softball_side(swing_time), .ball_speed(ball_speed),
.start_ball_move(start_ball_moving), .phsync(phsync), .pvsync(pvsync), .pblank(pblank),
	.pixel(pixel), .screen_pulse(xvga_screen_pulse), .hit(hit),
.led_full_sequence(led_full_sequence), .led_update(led_update));
	// import pong basics into vivado
	reg [23:0] rgb;
	wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767);
	reg b,hs,vs;
	always @(posedge clock_65mhz) begin
	if (SW[1:0] == 2'b01) begin
		// 1 pixel outline of visible area (white)
		  hs <= hsync;
	vs <= vsync;
	b <= blank;
	rgb <= {24{border}};
	end else if (SW[1:0] == 2'b10) begin
	// color bars
	hs <= hsync;
	vs <= vsync;
	b <= blank;
	rgb <= {{8{hcount[8]}}, {8{hcount[7]}}, {8{hcount[6]}}} ;
	end else begin
	// default: virtual softball game
	hs <= phsync;
	vs <= pvsync;
	b <= pblank;
	rgb <= pixel & {24{~blank}};
	end
	end
	// provide outputs
	assign VGA_R = rgb[23:20];
	assign VGA_G = rgb[15:12];
	assign VGA_B = rgb[7:4];
	assign VGA_HS = ~hsync;
	assign VGA_VS = ~vsync;

endmodule

// Credit to g.p.hom
// pulse synchronizer
module synchronize #(parameter NSYNC = 2)  // number of sync flops.  must be >= 2
```

```
                (input clk,in,
                output reg out);
        reg [NSYNC-2:0] sync;
        always @ (posedge clk) begin
                {out,sync} <= {sync[NSYNC-2:0],in};
        end
ndmodule

// Credit to g.p.hom
// use system clock for the clock input
// to produce a synchronous, debounced output
module debounce #(parameter DELAY=1000000)   // .01 sec with a 100Mhz clock
        (input reset, clock, noisy,
        output reg clean);

  reg [19:0] 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
```

11.3.2 fsm.v

```
module fsm #(parameter MAX_BALL_WIDTH = 128) (input clock, reset, batter_up, swing,
ball_clock, hit, [1:0] ball_speed, output [3:0] out_state, output bat_enable, start_pitch, output
[1:0] swing_time, output reg [7:0] hits,misses, output start_ball_moving);
```

```
//state encodings
parameter IDLE = 4'b0000;
parameter BALL_COMING = 4'b0001;
parameter BALL_IN_ZONE = 4'b0010;
parameter SWING = 4'b0011;
parameter TIME_MISS = 4'b0100;
parameter SYNC = 4'b0101;
parameter HIT = 4'b0110;
parameter SWING_MISS = 4'b0111;
parameter BATTER_UP_WAIT = 4'b1000;

parameter LEFT_SIDE_BALL = 2'b00;
parameter MIDDLE_BALL = 2'b01;
parameter RIGHT_SIDE_BALL = 2'b10;

parameter COUNT_5_SEC = 32'd325000000;

//state transition diagram
reg [3:0] state = IDLE;
reg [29:0] counter_hit = 0;
reg [29:0] counter_miss = 0;
reg [29:0] batter_wait_counter = 0;
reg [7:0] ball_counter = 0;
reg [1:0] rswing_time = 3; //either left, middle, or right

reg rbat_enable; //when graphics should display the bat
reg rstart_pitch; //tells graphics when to pitch the ball
reg rstart_ball_moving; //tells graphics when to start showing ball trajectory

always @ (posedge clock) begin

if (reset) begin;
counter_hit <= 0;
counter_miss <= 0;
ball_counter <= 0;
state <= IDLE;
rbat_enable <= 0;
hits <= 0;
misses <= 0;
rstart_ball_moving <= 0;
end

//transitions to next state and updates signal
```

```
else begin

case (state)

 //everything is at rest waiting for the signal to pitch the ball
        IDLE: begin

        rbat_enable <= 0;
        ball_counter <= 0;
        counter_hit <= 0;
        counter_miss <= 0;
        batter_wait_counter <= 0;
        if (batter_up) state <= BATTER_UP_WAIT;
        else state <= IDLE;
        rstart_ball_moving <= 0;

        end

 //gives time for thebatter to get set before the ball is pitched
        BATTER_UP_WAIT: begin

        rbat_enable <= 0;
        ball_counter <= 0;
        rstart_pitch <= 0;

        batter_wait_counter <= batter_wait_counter + 1;

        if (batter_wait_counter >= 65000000) begin //wait 1 second before pitching ball
        state <= BALL_COMING;
        batter_wait_counter <= 0;
        rstart_pitch <= 1;
        end
        else state <= BATTER_UP_WAIT;
        end

 //ball is moving towards teh batter in the first half of the ball's path
        BALL_COMING: begin

        rbat_enable <= 0;
        batter_wait_counter <= 0;
        rstart_pitch <= 0;
```

```
			//handles timing for how far the ball has traveled based on graphical
representation
			if (ball_clock) ball_counter <= ball_counter + 1;

			//ball is in the zone if it has travelled halfway, this is independent of speed
			if (ball_counter >= ((MAX_BALL_WIDTH/ball_speed)/2)) state <=
BALL_IN_ZONE; //corresponds to certain ball width and color
			else if (swing) begin
			state <= TIME_MISS;
			ball_counter <= 0;
			end

			else if (batter_up) begin
			state <= BATTER_UP_WAIT;
			ball_counter <= 0;
			end

			else state <= BALL_COMING;

			end

		//ball is moving towards the batter in the zone (the batter would have correct timing here
if they swung)
		//this state also tells graphics which side of the ball the batter hit
			BALL_IN_ZONE: begin

			rbat_enable <= 0;

			if (ball_clock) ball_counter <= ball_counter + 1; //continues counting ball width
from BALL_COMING state

			if (swing) begin
			state <= SYNC;
			ball_counter <= 0;
			if (ball_counter <= (((MAX_BALL_WIDTH / ball_speed)/8) * 5)) rswing_time <=
RIGHT_SIDE_BALL; //swung early => hit right side
			else if ((ball_counter > (((MAX_BALL_WIDTH / ball_speed)/8) * 5)) &&
				(ball_counter <= (((MAX_BALL_WIDTH / ball_speed)/8) * 7)))
rswing_time <= MIDDLE_BALL; //hit middle part of ball
			else rswing_time <= LEFT_SIDE_BALL; //swung late => hit left side
			end

			else if (ball_counter >= (MAX_BALL_WIDTH / ball_speed)) begin
```

```
                state <= TIME_MISS; //ball is now white and full sized
                ball_counter <= 0;
                misses <= misses + 1;
                end

                else if (batter_up) begin
                state <= BATTER_UP_WAIT;
                ball_counter <= 0;
                end

                else state <= BALL_IN_ZONE;

                end

        //used to synchronize ball clock for hit detection in later states
                SYNC: begin

                rbat_enable <= 0;

                if (ball_clock) state <= SWING;

                else state <= SYNC;

                end

        //the bat has just been swung at the right time, and graphics will determine if the bat hit
the ball
                SWING: begin
                rbat_enable <= 1;
                //if any pixels have both a bat and ball component, then the ball is considered to
be hit
                if (hit) begin
                state <= HIT;
                hits <= hits + 1;
                end
                //no pixels on the screen are detecting an intersection of the bat and the ball, so
this is then a miss
                else if (ball_clock) begin
                state <= SWING_MISS;
                misses <= misses + 1;
                end
                else state <= SWING;
                end
```

```
//the batter made contact with the ball (based on timing and bat angle)
        HIT: begin
        rbat_enable <= 1;
        counter_hit <= counter_hit + 1;
        ball_counter <= 0;

        //has viewed the screen for 5 seconds
        if (counter_hit >= COUNT_5_SEC) begin
        state <= IDLE;
        counter_hit <= 0;
        rbat_enable <= 0;
        rstart_ball_moving <= 1;
        end

        else if (batter_up) begin
        state <= BATTER_UP_WAIT;
        counter_hit <= 0;
        rbat_enable <= 0;
        end

        else state <= HIT;
        end

//the batter missed the ball due to either swinging too early or too late
        TIME_MISS: begin
        counter_miss <= counter_miss + 1;
        ball_counter <= 0;

        if (counter_miss >= COUNT_5_SEC) begin
        state <= IDLE;
        counter_miss <= 0;
        end

        else if (batter_up) begin
        state <= BATTER_UP_WAIT;
        counter_miss <= 0;
        end

        else state <= TIME_MISS;

        end
```

```
        //the batter missed the ball due to an incorrect angle (they had correct timing)
            SWING_MISS: begin
            rbat_enable <= 1; //major difference from TIME_MISS
            counter_miss <= counter_miss + 1;
            ball_counter <= 0;

            if (counter_miss >= COUNT_5_SEC) begin
            state <= IDLE;
            counter_miss <= 0;
            rbat_enable <= 0;
            end

            else if (batter_up) begin
            state <= BATTER_UP_WAIT;
            rbat_enable <= 0;
             counter_miss <= 0;
            end

            else begin
            state <= SWING_MISS;
            rbat_enable <= 1;
            end

            end

        default: state <= IDLE;

        endcase

        end
        end

        assign bat_enable = rbat_enable;
        assign out_state = state;
        assign swing_time = rswing_time;
        assign start_pitch = rstart_pitch;
        assign start_ball_moving = rstart_ball_moving;
endmodule
```

11.3.3 bat.v

```
module bat(
    input clk,
    input reset,
    input start,              // start is the calibration signal for pos
    input ready,              // ready indicates new data is ready from IMU
    input signed [15:0] accelx,
    input signed [15:0] accely,
    input signed [15:0] accelz,
    input signed [15:0] gyrox,
    input signed [15:0] gyroy,
    input signed [15:0] gyroz,

    output reg [7:0] X,       // unsigned!!!
    output reg [7:0] Y,
    output reg direction,
    output reg done,
    output batter_ready,

    output reg [1:0] state,   // for debugging
    output pos0,
    output [7:0] test_x,
    output [7:0] test_y,
    output test_dir,
    input [12:0] SW           // for finding thresholds
    );

    // states
    parameter IDLE = 2'b00;
    parameter CALIBRATE = 2'b01;
    parameter READ = 2'b11;
    parameter SWING = 2'b10;
    reg [1:0] next_state;

    // helper variables
    wire start_pos, done_pos;
    wire [7:0] slope_x, slope_y;
    wire dir;

    // helper modules
    pos posz(.clk(clk), .reset(reset), .update(ready), .start(start_pos), .gyroz(gyroz), .gyrox(gyrox),
```

```
.done(done_pos), .raised(batter_ready), .pos0(pos0));
   slope #(.DIV(11)) slope1(.clk(clk), .reset(start_pos), .update(ready), .T(6), .T2(1),
.beta({11'hF}),
                  .gyrox(gyrox), .gyroy(gyroy), .gyroz(gyroz), .accely(accely), .accelz(accelz),
.z(slope_x), .y(slope_y), .dir(dir));

   // for debugging
   assign test_x = slope_x;
   assign test_y = slope_y;
   assign test_dir = dir;

   // calibrate when in calibration state
   assign start_pos = state==CALIBRATE;


   always @(posedge clk) begin
      if (reset) begin            // system reset
         done <= 0;
         state <= IDLE;
      end
      else begin
         case (state)
            IDLE: begin
               done <= 0;
               state <= start ? CALIBRATE : IDLE;   // start calibration (centered)
            end
            CALIBRATE: begin
               state <= READ;                // signal start_pos
            end
            READ: begin                      // read gyro/accel values until bat returned to
calibration point
               state <= done_pos ? SWING : (start ? CALIBRATE : READ);
            end
            SWING: begin                     // output x and y values from slope
               X <= slope_x;
               Y <= slope_y;
               direction <= dir;
               state <= IDLE;
               done <= 1;
            end
         endcase
      end
   end
```

```
endmodule


module pos (
   input clk,
   input reset,
   input update,
   input start,
   input signed [15:0] gyroz,
   input signed [15:0] gyrox,
   output reg done,
   output reg raised,
   output pos0                 // for debugging
   );

   // set up some constant values
   parameter [2:0] T = 3'sd2;         // divisor for gyro (g/2^T)
   parameter signed [23:0] epsilonz = 24'sh8000;    // threshold for being in zone- use 100 for
testing
   parameter signed [23:0] epsilonx = 24'sh300000;  // threshold for being in zone- use 100 for
testing
   parameter signed [15:0] gz_threshold = 16'sh100; // threshold for stopped movement
   parameter IDLE = 2'b00;            // waiting for start signal
   parameter CENTERED = 2'b01;        // calibrate pos, wait until bat moves
   parameter RAISING = 2'b10;         // raise bat
   parameter SWINGING = 2'b11;        // wait for pos to return to 0

   // create registers to store initial pos (calibrating point)
   reg [1:0] state;
   reg signed [23:0] posz;
   reg signed [23:0] posx;

   wire posz0, posx0;                 // detects when pos is within epsilon of calibrated point
   assign posz0 = posz[23] ? posz > -epsilonz : posz < epsilonz;    // if negative: check
pos<-eps,   else if positive: check pos>eps
   assign posx0 = posx[23] ? posx > -epsilonx : posx < epsilonx;
   assign pos0 = posz0 & posx0;


   always @(posedge clk) begin
      if (reset) begin              // system reset
         posz <= 0;
```

```
         posx <= 0;
         raised <= 0;
         done <= 0;
         state <= IDLE;
      end
      else if (start) begin          // start calibration (centered)
         posz <= 0;
         posx <= 0;
         raised <= 0;
         done <= 0;
         state <= CENTERED;
      end
      else if (update) begin
         // update pos
         posz <= posz + (gyroz >>> {1'b0,T});
         posx <= posx + gyrox;

         // update state
         if (state == CENTERED && !posz0) begin
            state <= RAISING;
         end
         else if (state == RAISING && (gyroz[15] ? gyroz>-gz_threshold : gyroz<gz_threshold))
begin
            state <= SWINGING;
            raised <= 1;
         end
         else if (state == SWINGING && pos0) begin
            done <= 1;
            state <= IDLE;
         end
         else begin
            raised <= 0;
            done <= 0;
         end
      end
      else begin
         raised <= 0;
         done <= 0;
      end
   end


endmodule
```

```
module slope #(parameter DIV=5) (
    input clk,
    input reset,
    input update,
    input [2:0] T,
    input [1:0] T2,
    input [DIV-1:0] beta,
    input signed [15:0] gyrox,
    input signed [15:0] gyroy,
    input signed [15:0] gyroz,
    input signed [15:0] accely,
    input signed [15:0] accelz,
    output [7:0] z,
    output [7:0] y,
    output dir
    );

    // create BIG registers to store previous accel/gyro values
    reg signed [15:0] z_int, y_int, gx;
    reg signed [23:0] az, ay, gz, gy;
    reg signed [31:0] zz, yy;
    assign z = z_int[15] ? -z_int[15:8] : z_int[15:8]; // top 8 bits of abs(z_int)
    assign y = y_int[15] ? -y_int[15:8] : y_int[15:8];

    // calculate signed positive beta and 1-beta
    wire signed [DIV:0] beta_int;
    wire signed [DIV+1:0] beta_comp;
    assign beta_int = {1'b0, beta};
    assign beta_comp = (2'b01<<{1'b0,DIV}) - beta_int;

    // direction is sign of y_accel
    assign dir = y_int[15];

    always @(posedge clk) begin
        if (reset) begin
            y_int <= accely;
        end
        else if (update) begin

            // update position in z direction
            /* works pretty well on it's own, but not in combo with
```

```
        z_int <= (zz >>> {1'b0,DIV}) + (az >>> {1'b0,DIV});
        az <= beta_int * accelz;//(accelz + (gyroy[15]?gyroy:-gyroy)>>>{1'b0,T2});
        gz <= gyrox >>> {1'b0,T};
        zz <= beta_comp * (dir ? (z_int + gz) : (z_int - gz));      // add gz if pointing down, else
subtract gx
         */
        /* let z_int be constant */
        z_int <= 16'h3800;


        // update position in y direction
        /* Affected by centripetal acceleration
        y_int <= (yy >>> {1'b0,DIV}) + (ay >>> {1'b0,DIV});
        ay <= beta_int * accely;
        gy <= gyrox >>> {1'b0,T};
        yy <= beta_comp * (y_int + gy);
        */
        /* accounts for centripetal acceleration */
        // B=0xF, T=6, T2=1
        y_int <= (yy >>> {1'b0,DIV}) + (ay >>> {1'b0,DIV});
        ay <= beta_int * (accely - (((gyroz[15]?-gyroz:gyroz))<<<{1'b0,T2}));
        gy <= (gyrox) >>> {1'b0,T};
        yy <= beta_comp * (y_int + gy);                     // add gy because increasing
downward for both

      end
   end

endmodule
```

11.3.4 serialRX.v

```
module serialRX(
        input clk,                  // 65 MHz clk
        input reset,                // active high reset
        input serial_in,
        output reg [7:0] data,   // 8 bit raw data
        output reg done
        );

        // this section sets up the clk;
        parameter DIVISOR =  564; // create 115,200 baud rate clock, not exact, but should
work. use 4 for tb
```

```
reg [10:0] count;

// sets up states
parameter IDLE =  2'b00;
parameter START = 2'b01;
parameter READ =  2'b11;
parameter STOP =  2'b10;
// constants
parameter NUM_BITS = 3'd7;
parameter START_BIT = 1'b0;
parameter STOP_BIT = 1'b1;

// set up variables
reg [2:0] bits_read;
reg [7:0] data_read;
reg [1:0] state;


always @(posedge clk)
begin
if (reset) begin                              // reset to IDLE state
state <= IDLE;
data[7:0] <= 0;
count <= 0;
done <= 0;
end
else begin                        // FSM transitions
case(state)
        IDLE: begin
        count <= 0;
        done <= 0;
        if (serial_in == START_BIT) begin             // start reading once start bit
received
        state <= START;
        end
        end
        START: begin
        if (count == DIVISOR/2 - 1) begin
        bits_read <= 0;                //should happen by default, but just in case
        data_read <= 0;
        count <= 0;
        state <= READ;
        end
```

```
            else count <= count+1;
            end
            READ: begin
            if (count == DIVISOR - 1) begin
            bits_read <= bits_read + 1;
            count <= 0;
            data_read[7:0] <= {serial_in, data_read[7:1]};  // shift in next bit
            if (bits_read == NUM_BITS) begin              // finished reading all bits, output
value into data
                state <= STOP;
            end
            end
            else count <= count+1;
            end
            STOP: begin
            if (count == DIVISOR - 1) begin
            count <= 0;
            state <= IDLE;
            if (serial_in == STOP_BIT) begin
                data[7:0] <= data_read[7:0];
                done <= 1;
            end
            end
            else count <= count+1;
            end
    endcase

    end
    end

endmodule
```

11.3.5 bluetoothRX.v

```
module bluetoothRX(
    input clk,
    input reset,
    input [7:0] data,
    input write,
    output [15:0] accelx,
    output [15:0] accely,
    output [15:0] accelz,
    output [15:0] gyrox,
```

```
        output [15:0] gyroy,
        output [15:0] gyroz,
        output reg done
        );

        // this section sets up the clk;
        reg [12:0] count;

        // states
        parameter IDLE = 2'b00;
        parameter READL = 2'b01;
        parameter READM = 2'b11;
        reg start;
        reg [1:0] state;
        reg [1:0] next_state;

        // constants
        parameter START_BYTE = 8'h11;
        parameter DIVISOR = 13'd6000; //65MHz * 10bit/115200Hz = 5642, but want to
overshoot
        parameter NUM_OUTPUTS = 3'd6;

        // read into array, then assign to accel and gyro
        reg [15:0] outputs[0:NUM_OUTPUTS-1];  // array of outputs
        reg [2:0] outputs_read;                  // reads six 16bit packets as twelve 8bit packets
        reg [7:0] outputL;               // lower 8bits
        assign accelx = outputs[0];
        assign accely = outputs[1];
        assign accelz = outputs[2];
        assign gyrox  = outputs[3];
        assign gyroy  = outputs[4];
        assign gyroz  = outputs[5];

        // state transitions: figure out next state
        always@* begin
        case(state)
        IDLE:           next_state = write ? READL : IDLE;
        READL:          next_state = (count==DIVISOR-1)|start ? IDLE : (write ? READM :
READL);
        READM:          next_state = (outputs_read >= NUM_OUTPUTS) ? IDLE : (write ? READL
: READM);
        default: next_state = IDLE;
        endcase
```

```
	end

	always @(posedge clk)
	begin
	if (reset) begin				// reset to IDLE state
	state <= IDLE;
	count <= 0;
	outputs_read <= 0;
	done <= 0;
	start <= 0;
	end
	else begin
	// count
	if (count >= DIVISOR) begin		// waited too long, reset to idle state
		state <= IDLE;
		count <=0;
	end
	else if (write) begin			// read byte, reset counter
		count <= 0;
		state <= next_state;
	end
	else begin				// waiting for next byte or a timeout
		count <= count+1;
		state <= next_state;
	end

	// transition
	if (next_state != state) begin
		case(next_state)
		IDLE: begin
		start <= 0;
		if (outputs_read == NUM_OUTPUTS) begin		// finished reading whole
packet -> done and reset
			done <= 1;
		end
		end
		READL: begin
		if (data == START_BYTE) begin			// read the start byte
			outputs_read <= 0;
			start <= 1;			// skip reading in the start byte as a low bit
-> return to IDLE
		end
		else outputL <= data;			// read lower order bits
```

```
                done <= 0;
                end
                READM: begin
                outputs[outputs_read] <= {data, outputL};     // read higher order bits, combine
and put into correct output
                outputs_read <= outputs_read+1;
                end
                endcase
        end
        else done <= 0;

        end
        end

endmodule
```

11.3.6 graphics.v

```
////////////////////////////////////////////////////////////////////////
//
// blob: generate rectangle on screen
//
////////////////////////////////////////////////////////////////////////
module blob
  (input [10:0] x,hcount,
        input [9:0] y,vcount,
        input [11:0] width, height,
        input [23:0] color,
        output reg [23:0] pixel);

  always @ * begin
        if ((hcount >= x && hcount < (x+width)) &&
        (vcount >= y && vcount < (y+height)))
  pixel = color;
        else pixel = 0;
  end
endmodule


//////////////////////////////////////////////////////////////////////////////////
//
// circle_blob: generate a circle on the screen with a particular center and radius
```

```
//
//////////////////////////////////////////////////////////////////////////////
module circle_blob (input [10:0] center_x, hcount,
                    input [9:0] center_y, vcount,
                    input [11:0] radius,
                    input [23:0] color, left_color, middle_color, right_color,
                    input clock, show_side,
                    output reg [23:0] circle_pixel);

          reg [10:0] deltax;
          reg [9:0] deltay;
          reg [21:0] deltax_sq;
          reg [19:0] deltay_sq;
          reg [23:0] radiussquare;

          always @ * begin

          deltax <= (hcount > center_x) ? (hcount - center_x) : (center_x - hcount);
          deltay <= (vcount > center_y) ? (vcount - center_y) : (center_y - vcount);

          end

          always @(posedge clock) begin

          //handles multiplication timing
          deltax_sq <= deltax * deltax;
          deltay_sq <= deltay * deltay;
          radiussquare <= radius * radius;

          if (deltax_sq + deltay_sq <= radiussquare) begin
          if (!show_side) circle_pixel <= color;
          else begin
                    if (hcount <= (center_x - (radius/2))) circle_pixel <= left_color;
                    else if (hcount >= (center_x + (radius/2))) circle_pixel <= right_color;
                    else circle_pixel <= middle_color;
          end
          end
          else circle_pixel <= 0;

          end


endmodule
```

```
///////////////////////////////////////////////////////////////////////////////////////
//
// bat_blob: generate a slanted rectangle from vertical center based on a provided angle
//
///////////////////////////////////////////////////////////////////////////////////////
module bat_blob(input [23:0] x_accel, hcount,
                input [23:0] y_accel, vcount,
                input [23:0] color,
                input clock, up,
                output reg [23:0] bat_pixel);


        always @ (posedge clock) begin

   //uses x and y components to determine angled lines that define bat
        if (up) begin
        if (((x_accel * (380 - (vcount + 48))) <= (y_accel * hcount))
        && ((y_accel * hcount) <= (x_accel * (380 - (vcount - 48))))) bat_pixel <= color;

        else if ((vcount >= (380 - 48)) && (vcount <= (380 + 48))
        && ((y_accel * hcount) <= (x_accel * (380 - (vcount - 48))))) bat_pixel <= color;

        else bat_pixel <= 0;
        end

        else begin
        if (((x_accel * (((vcount) - 380 - 48))) <= (y_accel * hcount))
                && ((y_accel * hcount) <= ((x_accel * (((vcount) - 380 + 48)))))) bat_pixel <=
color;

        else if ((vcount >= (380 - 48)) && (vcount <= (380 + 48))
                && ((y_accel * hcount) <= ((x_accel * (((vcount) - 380 + 48)))))) bat_pixel <=
color;

        else bat_pixel <= 0;

        end

        end

endmodule
```

```
//displays a picture
module picture_blob #(parameter WIDTH = 1024, HEIGHT = 768)
	(input pixel_clk, input [10:0] x, hcount, y, vcount, output reg [23:0] pixel);

	wire [7:0] image_bits;
	wire [7:0] color_index; //9 bits used to look into the color maps
	wire [7:0] color_red; //retrieved from red color map
	wire [7:0] color_green; // retrieved from green color map
	wire [7:0] color_blue; //retrieved from blue color map

	reg [17:0] image_addr = 0; //num bits for 512 X 384

	always @ (posedge pixel_clk) begin

	if ((hcount >= 0 && hcount < WIDTH) && (vcount >= 0 && vcount < HEIGHT)) begin
	image_addr <= ((vcount[10:1] * 512) + hcount[10:1]); //get rid of last bit of vcount and
hcount because need to enlarge image by double in each direction
	pixel <= {color_red, color_green, color_blue};
	end

	else pixel <= 0;

	end

	blk_mem_gen_0 mem_color_index(.clka(pixel_clk), .ena(1), .addra(image_addr),
.douta(color_index)); //retrieves the image index that will be used for color maps
	blk_mem_gen_1 mem_color_red(.clka(pixel_clk), .ena(1), .addra(color_index),
.douta(color_red));
	blk_mem_gen_2 mem_color_green(.clka(pixel_clk), .ena(1), .addra(color_index),
.douta(color_green));
	blk_mem_gen_3 mem_color_blue(.clka(pixel_clk), .ena(1), .addra(color_index),
.douta(color_blue));

 endmodule



////////////////////////////////////////////////////////////////////////////////
//
// graphics: the virtual softball game!
//
```

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

module graphics #(parameter MAX_BALL_WIDTH = 128) (
  input vclock,   // 65MHz clock
  input reset,   // 1 to initialize module
  input [10:0] hcount,   // horizontal index of current pixel (0..1023)
  input [9:0]     vcount, // vertical index of current pixel (0..767)
  input hsync,            // XVGA horizontal sync signal (active low)
  input vsync,            // XVGA vertical sync signal (active low)
  input blank,  // XVGA blanking (1 means output black pixel)
  input bat_up,           //is bat at an upward angle
  input [23:0] x_accel, //x slope component of bat
  input [23:0] y_accel, //y slope component of bat
  input start_pitch, // when to pitch the ball
  input bat_enable, //tells when to show bat on screen
  input [1:0] softball_side, //which side of the softball was hit
  input [1:0] ball_speed,
  input start_ball_move,

  output phsync,   // softball game's horizontal sync
  output pvsync,   // softball game's vertical sync
  output pblank,   // softball game's blanking
  output [23:0] pixel,   // softball game's pixel  // r=23:16, g=15:8, b=7:0 //will be converted to 12
bit color
  output screen_pulse, //when screen has been refreshed
  output hit, //when bat and ball share a pixel
  output [1023:0] led_full_sequence, //commands passed to LEDs
  output led_update //when LEDs should be updated
  );

  assign phsync = hsync;
  assign pvsync = vsync;
  assign pblank = blank;

  reg [7:0] softball_width;
  reg [10:0] softball_x; //softball x center
  reg [10:0] softball_y; //softball y center
  reg [23:0] softball_color;
  reg [7:0] softball_color_red; //used for blending of color as ball approaches
  reg [7:0] softball_color_green; //used for blending of color as ball approaches

  //used to show hit timing when appropriate
  reg [23:0] softball_color_left;
```

```
  reg [23:0] softball_color_middle;
  reg [23:0] softball_color_right;

  reg rscreen_pulse;
  reg [7:0] random_counter = 0; //used to randomly determine where ball is pitched
  reg stop_ball_grow = 0; //0 => don't stop, 1 => stop
  reg hit_memory; //remembers if single pixel hit pulse seen on a full swing

  reg [959:0] rled_sequence = {30{3'b111, 29'b0}}; //led sequence values without the start and
end signal, start with blank
  reg rled_update = 0;
  reg last_led_update; //helps with LED timing for different ball speeds

  reg was_bat_enabled = 0; //if the bat was enabled on the last cycle, used to determine if bat
just went away => want ball motion
  reg move_ball = 0; //says whether the ball should display a trajectory

  //controls vertical ball trajectory
  reg up;
  reg down;
  reg up_memory;
  reg down_memory;

  wire [23:0] ball_pixel;
  wire [23:0] square_pixel;
  wire [23:0] blend_pixel;
  wire [23:0] multi_pixel;

  wire [23:0] left_sz_pixel;
  wire [23:0] top_sz_pixel;
  wire [23:0] right_sz_pixel;
  wire [23:0] bottom_sz_pixel;
  wire [23:0] vertical_middle_sz_pixel;
  wire [23:0] horizontal_middle_sz_pixel;
  wire [23:0] softball_pixel;
  wire [23:0] bat_pixel;
  wire [23:0] total_zone_pixel;
  wire [23:0] ball_zone_multipixel;
  wire [23:0] ball_bat_multipixel;
  wire [23:0] pic_pixel;
  wire [23:0] game_pixel;

  wire [1023:0] led_full_sequence; //entire sequence to control LEDs
```

```
  wire led_update; //when LEDs should be updated with new commands



  parameter SCREEN_WIDTH = 1023;
  parameter SCREEN_HEIGHT = 767;
  parameter STRIKEZONE_THICKNESS = 16;

  //used to display swing timing
  parameter LEFT_SIDE_BALL = 2'b00;
  parameter MIDDLE_BALL = 2'b01;
  parameter RIGHT_SIDE_BALL = 2'b10;

  always @(posedge vclock) begin

        random_counter <= random_counter + 1;

        hit_memory <= hit || hit_memory;

        if (rled_update) rled_update <= 0; //stop sending LED signal

        if (start_ball_move) move_ball <= 1; //continue showing trajectory of ball

        //determines vertical trajectory of ball
        if (hit) begin
        if (vcount > softball_y) up <= 1;
        else down <= 1;
        end

        else begin
        up <= 0;
        down <= 0;
        end

        up_memory <= start_pitch ? 0 : up || up_memory;           // remembers if up pulse was
seen, resets on start_pitch
        down_memory <= start_pitch ? 0 : down || down_memory;          // remembers if up
pulse was seen, resets on start_pitch

        //resets the game
        if (reset) begin
        rled_update <= 0;
        was_bat_enabled = 0;
```

```
        move_ball <= 0;
        up_memory <= 0;
        down_memory <= 0;
        end

        if (start_pitch) begin
        softball_width <= 0;
        softball_color <= 24'hFF_00_00; //ball begins red
        softball_color_red <= 8'hFF;
        softball_color_green <= 8'h00;

        //random ball start constricted to a particular region (equations written out for timing
purposes)
        if (((SCREEN_WIDTH >> 5) + (random_counter[3:0] * (SCREEN_WIDTH >> 4))) <= 450
&&
                ((((SCREEN_HEIGHT >> 5) + (random_counter[7:4] * (SCREEN_HEIGHT >>
4))) <= 200) ||
                ((SCREEN_HEIGHT >> 5) + (random_counter[7:4] * (SCREEN_HEIGHT >> 4)))
>= 450)) begin
        softball_x <= ((SCREEN_WIDTH >> 5) + (random_counter[3:0] * (SCREEN_WIDTH >>
4))) + 512;
        softball_y <= ((SCREEN_HEIGHT >> 5) + (random_counter[7:4] * (SCREEN_HEIGHT
>> 4)));
        end

        else begin
                softball_x <= ((SCREEN_WIDTH >> 5) + (random_counter[3:0] *
(SCREEN_WIDTH >> 4)));
                softball_y <= ((SCREEN_HEIGHT >> 5) + (random_counter[7:4] *
(SCREEN_HEIGHT >> 4)));
        end

        stop_ball_grow <= 0;
        hit_memory <= 0;
        rled_sequence <= {30{3'b111, 29'b0}};  //clears LEDs for the next pitch
        rled_update <= 1;
        was_bat_enabled <= 0;
        move_ball <= 0;
        end


        //determines when the screen should be refreshed; represents each cycle of going
through all the pixels
```

```
if (hcount == SCREEN_WIDTH && vcount == SCREEN_HEIGHT) begin
rscreen_pulse <= 1;

//show bat on the screen
if (bat_enable) begin
	was_bat_enabled <= 1;
	softball_width <= softball_width;
	stop_ball_grow <= 1;

//left side of ball is green to show hit timing
if (softball_side == LEFT_SIDE_BALL) begin
	if (hit_memory || hit) softball_color_left <= 24'h00_FF_00;
	else softball_color_left <= 24'hFF_00_00;
	softball_color_middle <= 24'hFF_FF_FF;
	softball_color_right <= 24'hFF_FF_FF;
end

//middle part of ball is green to show hit timing
else if (softball_side == MIDDLE_BALL) begin
	softball_color_left <= 24'hFF_FF_FF;
	if (hit_memory || hit) softball_color_middle <= 24'h00_FF_00;
	else softball_color_middle <= 24'hFF_00_00;
	softball_color_right <= 24'hFF_FF_FF;
end

//right side of ball is green to show hit timing
else begin
	softball_color_left <= 24'hFF_FF_FF;
	softball_color_middle <= 24'hFF_FF_FF;
	if (hit_memory || hit) softball_color_right <= 24'h00_FF_00;
	else softball_color_right <= 24'hFF_00_00;
end

end

//ball is showing trajectory
else if (move_ball) begin
 //ball is done showing trajectory
	if (softball_width < 2 ) begin
	move_ball <= 0;
	softball_width <= 0;
	end
```

```
            softball_color <= 24'h00_FF_00;

    //ball trajectory
            softball_y <= softball_y - up_memory + down_memory;

    //timing of the hit determines where the ball flies laterally
            case (softball_side)

            LEFT_SIDE_BALL: begin
            softball_x <= softball_x + 1;
            softball_width <= softball_width - 1;
            end

            MIDDLE_BALL: begin
            softball_width <= softball_width - 1;
            end

            RIGHT_SIDE_BALL: begin
            softball_x <= softball_x - 1;
            softball_width <= softball_width - 1;
            end

            default; //do nothing; don't move the ball

            endcase

    end

    //bat has gone away and if no trajectory is shown (a miss) then the ball turns a solid color
and remains still
    else if (stop_ball_grow) begin
            softball_width <= softball_width;
            stop_ball_grow <= 1;
            if (hit_memory) begin
            softball_color <= 24'h00_FF_00;
            rled_sequence <= {30{32'hFF_00_FF_00}};
            rled_update <= 1;
            end
            else begin
            softball_color <= 24'hFF_00_00;
            rled_sequence <= {30{32'hFF_00_00_FF}};
            rled_update <= 1;
            end
```

```
        end

    //ball continues growing and changing color from red to green, denoting a pitch moving
towards the batter
        else if ((softball_width <= MAX_BALL_WIDTH-2) && !stop_ball_grow) begin
        //ball growing speed depends on speed input given by user
        softball_width <= softball_width + ball_speed;
        softball_color_red <= softball_color_red - ball_speed * 2;
        softball_color_green <= softball_color_green + ball_speed * 2;
        softball_color <= {softball_color_red, softball_color_green, 8'h00};

        //signal LED change here
        if (softball_width < MAX_BALL_WIDTH >> 1) begin
        //instantiates the LED sequence to clear
        rled_sequence <= {30{3'b111, 29'b0}};
        rled_update <= 1;
        end

        if (softball_width == MAX_BALL_WIDTH >> 1) begin
        //first LED lights when ball enters strikezone, rest of LEDs are balnk
        rled_sequence <= {32'hFF_00_FF_FF, {29{3'b111, 29'b0}}};
        rled_update <= 1;
        end


        if (softball_width > MAX_BALL_WIDTH >> 1) begin
        //LED shift timing is speed dependent, changes when to update the position of the lit
LED
        if (ball_speed == 1 && !last_led_update) begin
          //shifts lit LED over 1 by using previous sequence
                rled_sequence <= {3'b111, 29'b0, rled_sequence[959:32]};
                rled_update <= 1;
                last_led_update <= 1;
        end

        //LED shift timing is speed dependent, changes when to update the position of the lit
LED
        else if (ball_speed == 2) begin
          //shifts lit LED over 1 by using previous sequence
                rled_sequence <= {3'b111, 29'b0, rled_sequence[959:32]};
                rled_update <= 1;
        end
```

```
        else last_led_update <= 0;

        end


        if (softball_side == LEFT_SIDE_BALL) begin
        //determines color of ball side based on timing and whether it was a hit or miss
        if (softball_pixel && bat_pixel && bat_enable) softball_color_left <= 24'h00_FF_00;
        else softball_color_left <= 24'hFF_00_00;
        softball_color_left <= 24'hFF_FF_00;
        softball_color_middle <= 24'hFF_FF_FF;
        softball_color_right <= 24'hFF_FF_FF;

        end

        else if (softball_side == MIDDLE_BALL) begin
        //determines color of ball side based on timing and whether it was a hit or miss
        softball_color_left <= 24'hFF_FF_FF;
        if (softball_pixel && bat_pixel && bat_enable) softball_color_middle <= 24'h00_FF_00;
        else softball_color_middle <= 24'hFF_00_00;
        softball_color_right <= 24'hFF_FF_FF;

        end

        else begin
        //determines color of ball side based on timing and whether it was a hit or miss
        softball_color_left <= 24'hFF_FF_FF;
        softball_color_middle <= 24'hFF_FF_FF;
        if (softball_pixel && bat_pixel && bat_enable) softball_color_right <= 24'h00_FF_00;
        else softball_color_right <= 24'hFF_00_00;
        end
        end

    //ball grows to largest size
        else begin
        softball_width <= MAX_BALL_WIDTH;
        softball_color_red <= 8'hFF;
        softball_color_green <= 8'hFF;
        softball_color <= 24'hFF_FF_FF;
        stop_ball_grow <= 1;

        end
```

```
        end

        //the screen pulse should only be high for one 65MHz clock cycle
        else rscreen_pulse <= 0;

  end


  //make softball
  circle_blob softball(.center_x(softball_x), .center_y(softball_y), .radius(softball_width),
.clock(vclock), .hcount(hcount), .vcount(vcount),
   .color(softball_color), .show_side(bat_enable), .left_color(softball_color_left),
.middle_color(softball_color_middle),
   .right_color(softball_color_right), .circle_pixel(softball_pixel));


  //make left strike zone line
  blob left_sz(.x(0), .y(0), .width(STRIKEZONE_THICKNESS), .height(SCREEN_HEIGHT),
.color(24'hFF_FF_00), .hcount(hcount), .vcount(vcount), .pixel(left_sz_pixel));

  //make top strike zone line
  blob top_sz(.x(0), .y(0), .width(SCREEN_WIDTH), .height(STRIKEZONE_THICKNESS),
.color(24'hFF_FF_00), .hcount(hcount), .vcount(vcount), .pixel(top_sz_pixel));

  //make right strike zone line
  blob right_sz(.x(SCREEN_WIDTH - STRIKEZONE_THICKNESS), .y(0),
.width(STRIKEZONE_THICKNESS), .height(SCREEN_HEIGHT), .color(24'hFF_FF_00),
.hcount(hcount),
   .vcount(vcount), .pixel(right_sz_pixel));

  //make bottom strike zone line
  blob bottom_sz(.x(0), .y(SCREEN_HEIGHT - STRIKEZONE_THICKNESS),
.width(SCREEN_WIDTH), .height(STRIKEZONE_THICKNESS), .color(24'hFF_FF_00),
.hcount(hcount),
   .vcount(vcount), .pixel(bottom_sz_pixel));

  //make vertical middle strike zone line
  blob vertical_center_sz(.x((SCREEN_WIDTH / 2) - (STRIKEZONE_THICKNESS / 2)), .y(0),
        .width(STRIKEZONE_THICKNESS), .height(SCREEN_HEIGHT), .color(24'hFF_FF_00),
        .hcount(hcount), .vcount(vcount), .pixel(vertical_middle_sz_pixel));

  //make horizontal middle strike zone line
```

```
  blob horizontal_center_sz(.x(0), .y((SCREEN_HEIGHT / 2) - (STRIKEZONE_THICKNESS /
2)),
        .width(SCREEN_WIDTH), .height(STRIKEZONE_THICKNESS), .color(24'hFF_FF_00),
        .hcount(hcount), .vcount(vcount), .pixel(horizontal_middle_sz_pixel));


  //make bat
  bat_blob bat(.x_accel(x_accel), .y_accel(y_accel), .color(24'h00_00_FF), .up(bat_up),
.hcount(hcount), .vcount(vcount), .clock(vclock), .bat_pixel(bat_pixel));

  //make a picture appear in the background
  picture_blob pic(.pixel_clk(vclock), .hcount(hcount), .vcount(vcount), .pixel(pic_pixel));

  //if bat and ball overlap, then it is considered a hit
  assign hit = (softball_pixel && bat_pixel && bat_enable);

  //represents the entire strikezone
  assign total_zone_pixel = left_sz_pixel + top_sz_pixel + right_sz_pixel + bottom_sz_pixel
                          + vertical_middle_sz_pixel + horizontal_middle_sz_pixel;

  //the ball should overlay the strikezone
  assign ball_zone_multipixel = (softball_pixel != 0) ? (softball_pixel) : total_zone_pixel;

  //ball and bat blend color to show where the bat hit the ball
  assign ball_bat_multipixel = bat_pixel + softball_pixel;

  //determines if bat is shown on screen or just ball and strikezone
  assign game_pixel =  ((bat_pixel==0 || !bat_enable) ? (ball_zone_multipixel) :
ball_bat_multipixel);

  //game should overlay the background picture
  assign pixel = (game_pixel != 0) ? game_pixel : pic_pixel;

  //signals every time the screen is refreshed
  assign screen_pulse = rscreen_pulse;

  //controls the LEDs
  assign led_full_sequence = {32'b0, rled_sequence, {32{1'b1}}};
  assign led_update = rled_update;

endmodule
```

11.3.7 ledTX.v

```
module ledTX #(parameter SIZE = 1024)(  //note: max size = 2^16 = 65563
        input clk,
        input reset,
        input [SIZE-1:0] packet,
        input update,
        output signal,
        output led_clk,
        output reg done
        );

        reg [15:0] count;
        reg [5:0] led_count;
        reg [SIZE-1:0] prev_packet0;
        reg [SIZE-1:0] prev_packet1;

        assign signal = (count >= SIZE) ? 1'b1 : packet[count];       // idle if count = size
        assign led_clk = (count >= SIZE) ? 0 : !led_count[5];         // sets output clk to 1/64 of
input clk

        always @(posedge clk) begin
        led_count <= led_count + 1;

        if (update & (count == SIZE) & prev_packet1 != packet) begin
        count <= SIZE-1;                              // begin sending packet by initializing count
        prev_packet0 <= packet;
        prev_packet1 <= prev_packet0;
        end
        else begin
        if (led_count[5:0] == 6'b0) begin
                if (count == 0) begin                 // end of count, so go back to idle state, also
signal done
                count <= SIZE;
                done <= 1;
                end
                else begin                            // decrement count
                done <= 0;
                if (count < SIZE) begin
                count <= count - 1;
                end
                end
```

```
        end
        end
        end

endmodule
```

11.3.8 display.v

```
// Credit to g.p. hom
module xvga(input vclock,
        output reg [10:0] hcount,      // pixel number on current line
        output reg [9:0] vcount,    // line number
        output reg vsync,hsync,blank);

  // horizontal: 1344 pixels total
  // display 1024 pixels per line
  reg hblank,vblank;
  wire hsyncon,hsyncoff,hreset,hblankon;
  assign hblankon = (hcount == 1023);
  assign hsyncon = (hcount == 1047);
  assign hsyncoff = (hcount == 1183);
  assign hreset = (hcount == 1343);

  // vertical: 806 lines total
  // display 768 lines
  wire vsyncon,vsyncoff,vreset,vblankon;
  assign vblankon = hreset & (vcount == 767);
  assign vsyncon = hreset & (vcount == 776);
  assign vsyncoff = hreset & (vcount == 782);
  assign vreset = hreset & (vcount == 805);

  // sync and blanking
  wire next_hblank,next_vblank;
  assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always @(posedge vclock) begin
        hcount <= hreset ? 0 : hcount + 1;
        hblank <= next_hblank;
        hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

        vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
        vblank <= next_vblank;
        vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low
```

```
        blank <= next_vblank | (next_hblank & ~hreset);
  end
endmodule

// Credit to g.p.hom
// Description:  Display 8 hex numbers on 7 segment display
module display_8hex(
        input clk,                // system clock
        input [31:0] data,        // 8 hex numbers, msb first
        output reg [6:0] seg,     // seven segment display output
        output reg [7:0] strobe// digit strobe
        );

        localparam bits = 13;

        reg [bits:0] counter = 0;  // clear on power up

        wire [6:0] segments[15:0]; // 16 7 bit memorys
        assign segments[0]  = 7'b100_0000;
        assign segments[1]  = 7'b111_1001;
        assign segments[2]  = 7'b010_0100;
        assign segments[3]  = 7'b011_0000;
        assign segments[4]  = 7'b001_1001;
        assign segments[5]  = 7'b001_0010;
        assign segments[6]  = 7'b000_0010;
        assign segments[7]  = 7'b111_1000;
        assign segments[8]  = 7'b000_0000;
        assign segments[9]  = 7'b001_1000;
        assign segments[10] = 7'b000_1000;
        assign segments[11] = 7'b000_0011;
        assign segments[12] = 7'b010_0111;
        assign segments[13] = 7'b010_0001;
        assign segments[14] = 7'b000_0110;
        assign segments[15] = 7'b000_1110;

        always @(posedge clk) begin
        counter <= counter + 1;
        case (counter[bits:bits-2])
        3'b000: begin
                seg <= segments[data[31:28]];
                strobe <= 8'b0111_1111 ;
                end
```

```
        3'b001: begin
                seg <= segments[data[27:24]];
                strobe <= 8'b1011_1111 ;
                end

        3'b010: begin
                seg <= segments[data[23:20]];
                strobe <= 8'b1101_1111 ;
                end
        3'b011: begin
                seg <= segments[data[19:16]];
                strobe <= 8'b1110_1111;
                end
        3'b100: begin
                seg <= segments[data[15:12]];
                strobe <= 8'b1111_0111;
                end

        3'b101: begin
                seg <= segments[data[11:8]];
                strobe <= 8'b1111_1011;
                end

        3'b110: begin
                seg <= segments[data[7:4]];
                strobe <= 8'b1111_1101;
                end
        3'b111: begin
                seg <= segments[data[3:0]];
                strobe <= 8'b1111_1110;
                end
        endcase
        end

endmodule

module hex_to_dec #(parameter BITS = 14, parameter DIGITS = 4, parameter BASE = 10)(
        input clk,
        input reset,
        input [BITS-1:0] hex,
        output reg [(4*DIGITS)-1:0] dec
        );
```

```
// tracks an update in the hex value
reg [BITS-1:0] prev_hex;
reg [BITS-1:0] count_hex;

// hard coded for now -> make into an array! reg [3:0] count_dec [DIGITS-1:0]
reg [3:0] count_dec_0;
reg [3:0] count_dec_1;
reg [3:0] count_dec_2;
reg [3:0] count_dec_3;

always @(posedge clk) begin
if (reset) begin
prev_hex <= hex;
count_dec_0 <= 0;
count_dec_1 <= 0;
count_dec_2 <= 0;
count_dec_3 <= 0;
dec <= 0;
end

else if (count_hex != 0) begin                    // increment count and shift base 10
count_hex <= count_hex - 1;
if (count_dec_0 == BASE-1) begin
        count_dec_0 <= 0;
        if (count_dec_1 == BASE-1) begin
        count_dec_1 <= 0;
        if (count_dec_2 == BASE-1) begin
        count_dec_2 <= 0;
        count_dec_3 <= count_dec_3 + 1;
        end
        else count_dec_2 <= count_dec_2 + 1;
        end
        else count_dec_1 <= count_dec_1 + 1;
end
else count_dec_0 <= count_dec_0 + 1;
end

else begin
dec <= {count_dec_3, count_dec_2, count_dec_1, count_dec_0};  // update output once
counting complete

if (prev_hex != hex) begin              // start counting if new data available
```

```
            prev_hex <= hex;
            count_hex <= hex;

            count_dec_0 <= 0;
            count_dec_1 <= 0;
            count_dec_2 <= 0;
            count_dec_3 <= 0;
        end

    end
    end

endmodule
```

11.3.9 siren.v

```
module siren (
  input clk,  // 65MHz
  input reset,
  input on,
  output reg siren
  );

  // divisor values to get 880Hz output
  parameter DIV_880 = 73_863;

  // set up counters
  reg [17:0] count;          // counts to set output frequency
  reg [17:0] div;

  always @ (posedge clk) begin
    if (reset) begin        // reset counter, default 880Hz
      count <= 0;
      div <= DIV_880;
    end
    else if (on) begin
      count <= count + 1;
      if (count == div) begin                   // reset counter
        siren <= 0;
        count <= 0;
      end
      else if (count == div / 2) siren <= 1;    // get 50% duty cycle output
```

```
        end
    end
endmodule
```