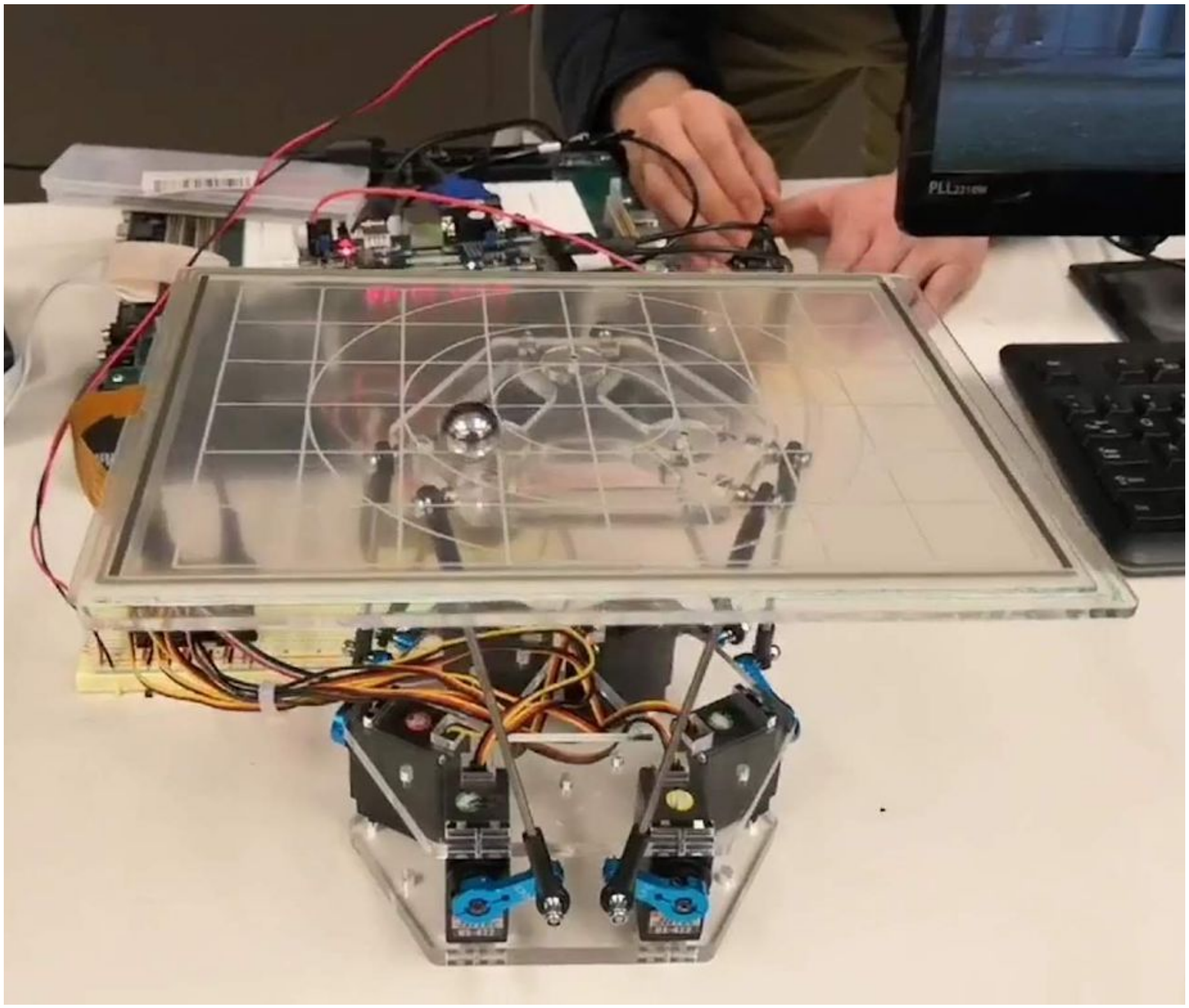
# FPGA Ball and Plate

## 6.111 - Fall 2018 Final Project

Adam Rodriguez, Cameron Ordone, and Kaname Favier



# Table of Contents

## Overview

Our project is the feedback control system of a ball-on-plate balancer.

We used the Nexys 4 DDR board to control a servo-actuated table to keep a steel ball stable at a specific location on its suspended plate. We used a 4-wire resistive touchscreen for position feedback with an external sensing circuit to read and transfer the interpreted ball coordinates to the FPGA.

The system has two modes of operation, both taking user input from an analog joystick. In one mode, the FPGA translates the joystick movements directly into rotations of the platform. In the other, the joystick controls the desired setpoint, while the FPGA uses the difference between the desired and sensed positions to determine the motion of the platform. The movement of the ball can be viewed on the VGA monitor.

## Design Decisions

The table used is a Stewart platform. This device uses six actuators (hobby servos in our case) attached to a rigid base to manipulate its upper plate with six-degrees of freedom (3D translation and rotation).

The platform was designed and built for the project. Since hobby servos do not guarantee repeatability in positioning, we chose to have the platform balance a ball to have some form of feedback determining its orientation. While the platform is capable of moving with several degrees of freedom, balancing a ball requires only two--namely, rotation about the x and y axes.

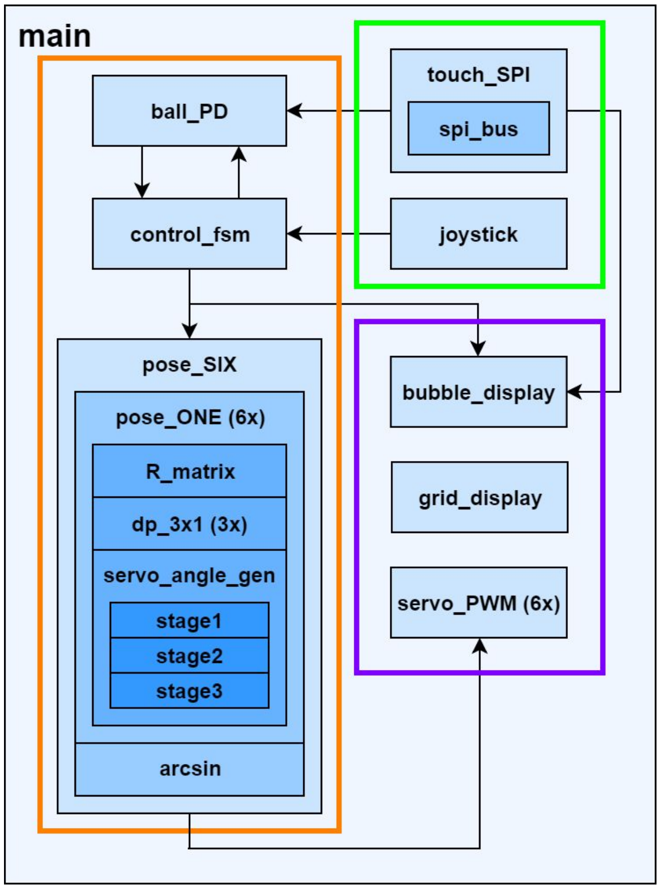
We chose the Nexys 4 DDR over the labkit to use its built-in ADC for reading our input joystick.

To acquire the ball's position, we also had the option of using an NTSC camera. We chose to use the touchscreen over concerns that the NTSC camera would have too much latency and too low a frame rate to stabilize the ball.

Using the ADC for the joystick did not leave enough ports for the touch panel, so to acquire feedback from the touchscreen, we use an external sensing circuit. An Adafruit STMPE610 IC samples and converts the touchscreen voltages into 12-bit coordinates. A Teensy is then used as an intermediate between the IC and the FPGA because the IC requires a nontrivial startup sequence, and normal operation involves register accesses and writes (would require a huge state machine). Likewise, for simplicity, the conditioning (a simple median filter) of the measured output signals was also done on the Teensy.

# Modules

The system is divided into many modules, shown in the figure below, grouped by function into inputs, processing, and outputs.



## Inputs

### touch_SPI (Cameron)

This module detects the X, Y, and TOUCH_STATUS bytes sent from a Teensy SPI master and repackages them into the appropriate 12-bit coordinates and single-bit touch reading. The FPGA was run as a slave because the Arduino library only had built-ins for running as master.

The bytes are first received by a lower-level **spi_bus** module that implements the general SPI communication process.

### joystick (Cameron)

This is an instantiation of Xilinx’s XADC IP used to digitize the joystick potentiometer voltages.

## Processing

### control_fsm (Kaname)

This module is a simple finite state machine that controls the nature of the pitch and roll angles being fed into the inverse kinematics module. Using a switch, the module toggles between two modes of operation: manual and feedback. Under manual control, this module computes the rotation angles from the joystick directly. Under feedback control, it will use the outputs from the ball_PD modules.

### ball_PD (Kaname)

Ball position control was achieved using two modules operating in parallel, one for each axis. The axial errors between the desired and actual position and the derived velocity of the ball were used to generate new pitch and roll targets for the platform. The controllers determined the output plate angles using the PD control methodology. The P and D gains were tunable using the switches on the Nexys board. However, we did not have the time to perfectly tune the gains, so our ball does not remain completely stable.

Each module takes two inputs, the ball's actual position and its desired position along the appropriate axis. These two positions, along with the derived ball velocity are fed into three multipliers. A small FSM controls the output timing of the multipliers. The 1kHz control signal was produced by a clock divider.

## Inverse Kinematics

The core of our project is its inverse kinematics module. Unlike a serial manipulator such as multi-jointed robotic arm, a Stewart platform's inverse kinematics are simpler to solve than its forward kinematics, meaning it's significantly easier to calculate the leg lengths needed to move the platform into a desired state than it is to deduce the state produced by a set of leg lengths. See the references for a detailed derivation of the inverse kinematics.

Modules within this series are driven by propagating **valid** signals. When values are ready to be output these modules, a **validOut** signal is pulsed high, which is accepted as a **validIn** to the next module(s) in the series. Internally, **validIn** gets latched into an **enable** signal used to ensure the module only accepts new inputs after its output is valid.

## Part 1: Calculate leg lengths (Adam)

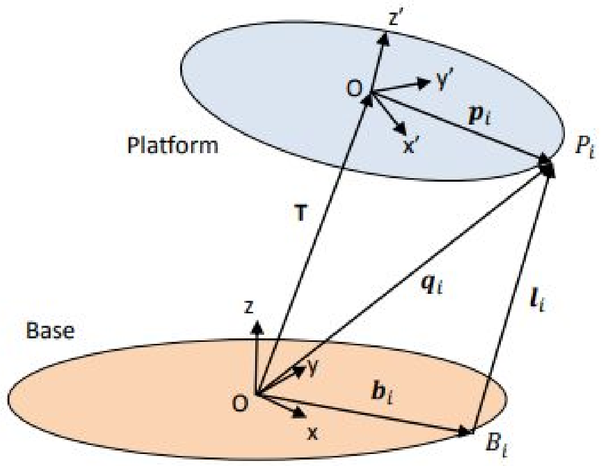
### R_matrix

#### Overview

Given a valid input, computes the nine matrix terms of our simplified rotation matrix.

#### Implementation

It was helpful for us to think of the base and plate as two coordinate planes stitched together by the legs. From its initial orientation parallel to the base, the plate can rotate about any axis.



The plate's rotation with respect to the base is then captured by this Euler matrix:

$$^{P}\mathbf{R}_B = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi)$$

$$= \begin{pmatrix} \cos\psi\cos\theta & -\sin\psi\cos\phi + \cos\psi\sin\theta\sin\phi & \sin\psi\sin\phi + \cos\psi\sin\theta\cos\phi \\ \sin\psi\cos\theta & \cos\psi\cos\phi + \sin\psi\sin\theta\sin\phi & -\cos\psi\sin\phi + \sin\psi\sin\theta\cos\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{pmatrix}$$

The full matrix enables any platform rotation, but, as stated previously, for our goal we only need pitch and roll. We can reduce this matrix by setting yaw to zero. Furthermore, we could get rid of the product terms, which just seemed unnecessarily expensive, by making use of trigonometric identities to write them all as sums, giving us a matrix that can be done with sums and shifts.

If $\mathbf{R}_z = \mathbf{I}$,

$$^{P}\mathbf{R}_B = \begin{pmatrix} \cos\theta & \frac{1}{2}[\cos(\theta-\phi) - \cos(\theta+\phi)] & \frac{1}{2}[\sin(\theta+\phi) + \sin(\theta-\phi)] \\ 0 & \cos\phi & -\sin\phi \\ -\sin\theta & \frac{1}{2}[\sin(\theta+\phi) - \sin(\theta-\phi)] & \frac{1}{2}[\cos(\theta+\phi) + \cos(\theta-\phi)] \end{pmatrix}$$

For the trig operations, we used CORDIC modules from Xilinx's IP catalog. The CORDIC core provides several trigonometric functions, and it uses only addition, shifting, and a small LUT.

To compute the rotation matrix, this module accepts the roll (Rx or $\varphi$) and pitch (Ry or $\Theta$) angles and computes their sum and difference. These four quantities are passed into their own sine-cosine CORDICs, whose outputs are registered before being combined to produce the nine terms in the matrix.

## dp_3x1

**Overview**

Given a valid input, this module computes the dot product of two 3x1 vectors.

**Implementation**

This module instantiates three multipliers and adds their outputs together to compute the dot product.

With valid matrix terms, the next step is to apply the rotation matrix to the six plate attachment points and calculate the leg lengths. Rotating a point requires a matrix multiplication, which works out as three parallel dot products, one for each coordinate.

## pose_ONE

**Overview**

High-level module computes the majority of the inverse kinematics for one servo.

**Implementation**

This module instantiates one **R_matrix** and three **dp_3x1** modules for the first part of the inverse kinematics and one **servo_angle_gen** for the second part. Six copies of this module are instantiated in the higher-level **pose_SIX** module as each servo has a different pair of joints.

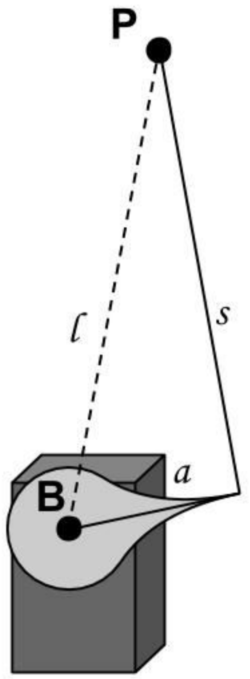
The output of the dp_3x1 modules are the new plate joint coordinates with respect to the base. To compute the leg vector we want, we first add a home position offset for the height.
Then we subtract the corresponding base joint coordinates, giving us the leg (equation is $l_i = {}^{P}R_B \cdot P_i - B_i$). Any translation would have been added as a vector $T$ to this output, but because our project did not require the plate to translate, we left it out.

## Part 2 - Convert leg lengths into servo angles (Kaname and Cameron)

Normally, Stewart platforms are manipulated by linear actuators such as hydraulic pistons or linear motors. In those cases, the above modules would provide everything needed to control the Stewart platform. But since we use rotary servos, we additionally need to identify the set of angles that provide the correct leg lengths.

After calculating the leg lengths, the process of determining the servo angles involves several intermediate steps. First, the relationship between the leg length and servo position are given by:

$$l^2 - (s^2 - a^2) = 2a\, l_z \sin(\alpha) + 2a \cos(\alpha)\, (l_x \cos(\beta) + l_y \sin(\beta))$$



Where $s$, $a$, and $\beta$ are constant build parameters of the platform.

This is an equation of the form:
$L = M \sin(\alpha) + N \cos(\alpha)$
where:
$L = l^2 - (s^2 - a^2)$,
$M = 2a \cdot l_z$,
$N = 2a \cdot [\cos(\beta) + l_y \sin(\beta)]$

And the function we wish to compute is
$\alpha_{servo} = \sin^{-1}\left( \frac{L}{\sqrt{(M^2+N^2)}} \right) + \tan^{-1}\left( \frac{N}{M} \right)$

Since there was potential for a slew of timing issues, we separated the above calculations into separate modules to have an easier time debugging.

### stage1

This modules calculates $L$, $M$, and $N$. We were able to make all multipliers the same latency, so the output registers are updated according to simple counting logic.

### stage2

This module calculates $\sqrt{M^2 + N^2}$ and $\tan^{-1}(N/M)$. First, the $M$ and $N$ are squared using multipliers and their sum is input into a square root CORDIC. The arctan CORDIC module still takes longest to produce a valid output, so it determines the **validOut**.

### stage3

This module uses a high-radix divider IP module to compute $L/\sqrt{M^2 + N^2}$. This value is sent out of the module to index into the **arcsin** LUT.

### servo_angle_gen
This high-level module encapsulates the three stages.

### arcsin
There was no built-in module for computing arcsin, so we generated a lookup table to be accessed by this module. We exploited the fact that this function is odd to cut the necessary LUT size in half to save memory.

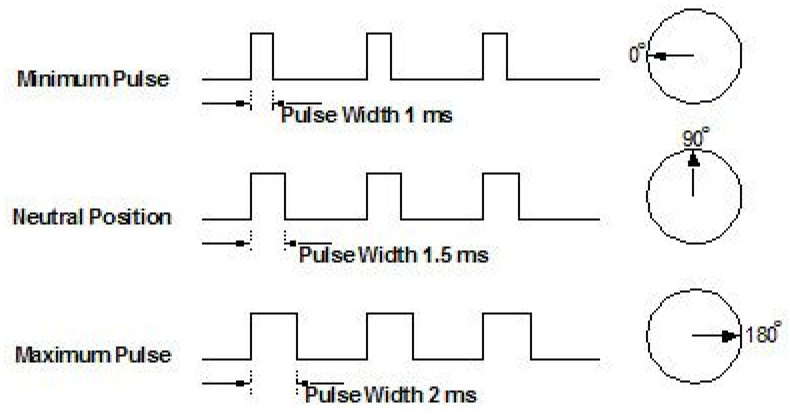
### pose_SIX
This is the highest-level abstraction for the inverse kinematics that gets called in the top-level **main** module. Instantiates six **pose_ONE** modules with their proper joint coordinates.

The **arcsin** LUT was originally located inside the **stage3** module, but we realized there wasn't enough BRAM to contain six copies. Instead we pulled it out to this module, and loop through the six LUT indexes, adding its output to the correct arctangent result to produce the six output angles.

## Outputs

### servo_PWM (Adam)
**Overview**
Given an unsigned int representing a servo angle from 0 to 180, outputs the correct duty ratio PWM signal to position the servo at this angle.

**Implementation**
In order to properly drive the platform servos, the angles calculated from the inverse kinematics need to be converted into the appropriate pulse widths. Most servos typically map their rotation range between pulses from 1-2ms, as seen in the figure below. Most hobby servos hold position properly as long as they receive pulses at a frequency between 40-200Hz.



We ultimately discovered that converting an angle into its appropriate pulse width was simple. First, we measured the minimum and maximum pulse widths for each servo using an oscilloscope. In our implementation, we instantiate our PWM modules with these limits passed in as parameters.

The module first maps the incoming angle to this range to produce the correct pulse width (check the Arduino map() function for reference).

The remainder of this module is just counting. Since we measured our pulse widths in microseconds, the module divides the system clock down to 1Mhz to count 1us increments, up to the PWM period (in our case 16384us or ~61Hz). With this, the PWM signal is then assigned to be the result of a comparison of whether the counter is less than the calculated pulse width.

All of our servos rotate clockwise with increasing pulse width, but half of our servos have arms mounted in a reversed orientation. In order to produce the desired symmetric behavior, there is an additional parameter that changes the final comparison to check against the opposite pulse width.

### bubble_display (Cameron)

This module displays x and y values as small circles. Modified from Lab 5c and used to display touchscreen readings and current ball setpoint.

The dots use the supplied x and y values as the top-left corner of the square that would contain the circle, which means the circle always looked a bit off (total displacement increased with the circle's size). In hindsight, it probably would have been faster and more useful to just draw crosshairs that intersect at the right location.

### grid_display (Cameron)

This module simply draws circles and lines at specific hcount and vcount slices to replicate the grid underneath the touch panel. Mainly used as a background for debugging the external circuit and SPI interface but also useful as a visualization.

## Challenges and Takeaways

We initially underestimated the utility of simulation testbenches and so neglected to iron out the harder modules during the first few weeks when the platform was being built, though none of our modules were particularly complicated by themselves.

By far our biggest time sink was their integration, namely understanding the operation of the IP modules and diagnosing the various issues our numerous IP instantiations would complain about. They're incredibly finicky. As an example, we first tried using a single Multiply-Adder IP to compute the dot product because it was fast and area-efficient. This worked fine in simulation, but caused some funky netlist implementation error that we had no idea how to solve. Two hours later, we just replaced it with a more straightforward sum of multipliers.

It's not always explicitly clear, even in the datasheet, how IP modules accept, process, and output data (i.e., whether its input is registered all at once or shifted in, etc). The arctan CORDIC either did not like its inputs being updated at the same rising edge of its input tvalid signal or the tvalid signal needed to be held high for an extended duration.

Furthermore, the CORDIC modules only accept values in a specific fixed-point format but output their results in another. Fixed-point numbers weren't something we'd gone over in class, so it wasn't immediately obvious how to transform them into integers and back.

We encountered many strange computation results due to underflow and overflow. Figuring out the right number of bits for everything took some time. We were doing weird things with truncating numbers, but really we should have just signed everything that was going into the computation (even for values we were sure would always be positive). Remember to use arithmetic shifts (>>>) over a logical shift (>>).

Just before our final checkoff, we had to re-synthesize our project several times to get the platform to rotate in the right directions. Some axis came out inverted and it took three tries and 20 min to synthesize correctly.

Basically, many small bugs added up and took ages to debug. We encourage future students to avoid this by developing the more difficult modules early and using testbenches to discover errors. If using modules from the IP catalog, make sure you understand the data format they expect, produce, and the number of cycles it takes to compute. Put any numbers that may need tuning on the switches so you only need to compile once.

Lastly, write your modules using an agreed-upon naming convention. They're going to be part of a system, and integration will be difficult and time-consuming with code that uses hastily-written placeholder variables.

## Future Work

Although they was listed in our initial proposal, we ran out of time before we could implement any of our stretch goals. In terms of controls, in the future, one goal we’d like to do is extend manual control to include support for a user-held IMU. Another would be to implement more advanced control algorithms, such as state feedback or ADRC, perhaps user-selectable, to compare their performance during motion profile tracking (ellipse, figure-8, etc).

## References

https://memememememememe.me/post/stewart-platform-math/
http://zipcpu.com/dsp/2017/07/11/simplest-sinewave-generator.html
https://www.fpga4fun.com/SPI2.html

## Appendix

All project files and code are available online at https://github.com/cordone/FPGA-ball-and-plate