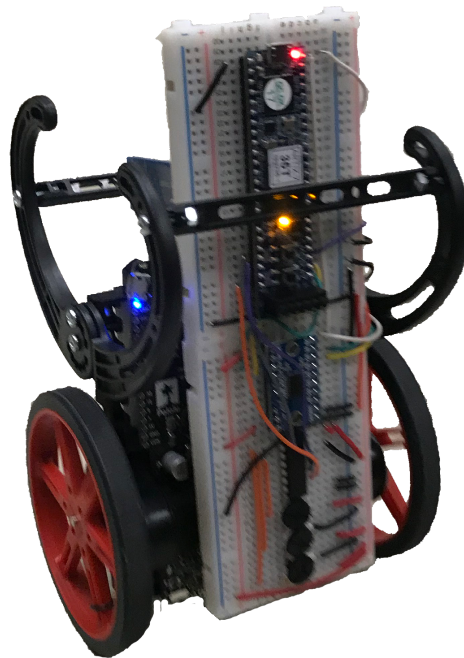
6.111 Final Report

Olek Peraire

12 December 2018



Contents

1 Background **3**

2 Implementation **3**
2.1 UART Communication . . . 3
2.2 PI Controller . . . 3
2.3 PD Controller . . . 3
2.4 SPI Communication . . . 4
2.5 Commands . . . 4

3 Diagrams **4**

4 Modules **7**
4.1 pidcontrol . . . 7
4.2 reader . . . 8
4.3 assigner . . . 8
4.4 send_organize . . . 8
4.5 uart_out . . . 8
4.6 half_clock . . . 8
4.7 main_system . . . 9
4.8 spi_master . . . 9
4.9 labkit . . . 9

5 Challenges and Solutions **9**
5.1 Timing Challenges . . . 9
5.2 Gyro Drift . . . 10
5.3 PD Controller Challenges . . . 10
5.4 Tuning Gains . . . 10
5.5 Assigning Values . . . 10
5.6 Differentiation . . . 11
5.7 ILA Testing . . . 11
5.8 Physical Constraints . . . 11

6 Goals **11**
6.1 Baseline Goals . . . 11
6.2 Expected Goals . . . 12
6.3 Stretch Goals . . . 12
6.4 Conclusion . . . 12

7 Future Work **12**

8 Acknowledgements **12**

9 Appendix **13**
9.1 Final Verilog Code . . . 13
9.2 Final Arduino Code . . . 23

1 Background

My 6.111 final project consists of a two wheeled, self balancing, robot that can receive commands from a remote FPGA via bluetooth. The motivation for this project was to simulate the challenges of real world applications involving FPGAs for navigation/control and communications. Although a two wheeled robot seems simple in nature, this project involved the solution of rather complex problems, including communicating between three different systems as well as non-trivial dynamics to keep the robot upright. These problems are commonplace in more sophisticated robots.

2 Implementation

This project involved using a CMOD A7-35T FPGA which interfaced with a Balboa 32U4 (Atmega 32U4), which had an integrated IMU (LSM6) as well as wheel encoders and motor drivers for the DC motors. There was also an HC-05 bluetooth module mounted on the back of the robot in order to enable bluetooth communication between the FPGAs.

2.1 UART Communication

The communication between the Balboa 32U4 and the CMOD A7 was done through a 3.3V - 5V two-way level shifter. Both devices simultaneously received and transmitted 6 bytes at a baud rate of 9600. Similarly, the CMOD A7 received commands via bluetooth from the Nexys 4 at a baud rate of 9600.

The Balboa and the CMOD were communicating at a relatively slow rate of 18ms. This meant the data we could send back and forth between them was limited. Since each reading (gyro, encoders, accelerometer) was 16 bits long, we could at most send 3-4 readings between updates. This could be a potential problem because if the update time is too slow, the theoretical model for the PD controllers would have to change from a continuous model to a discrete model.

To avoid sending "waste" bytes to sync the Balboa and the CMOD, the two devices were reset at the same time. The CMOD would sync its readings based on the initial readings from the gyro (which had an offset), and the wheel encoders (which always started at 0). The Balboa would then sync its readings based on 2 predetermined bytes sent by the CMOD. This way, the two devices only had to sync up once, and then communicate at full speed without having to waste any bytes while the robot was balancing. This further helped reduce our update times.

2.2 PI Controller

A PI controller based on the angular velocity received from the gyro (angular velocity and angle) was implemented. In order to prevent noise from the gyro, the 5 least significant bits from the reading were discarded, and a relatively high gain was used. This caused a loss of resolution around 0, thus canceling the noise. Once the controller was properly tuned, the robot was able to keep the upright position, but could not track drift. Furthermore, since we were not using accelerometer readings to counter the gyro drift, we used a decaying integral to estimate the angle of the robot. That is, we integrated the gyro to get the angle, assuming the angle was close to 0, so it would decay to 0 over future update cycles. The result of these two compromises was that the robot would stay upright, but did not have a way to track its position, which meant that, once pushed, it would maintain the velocity given to its center of mass while staying upright.

2.3 PD Controller

Proportional feedback based on the wheel encoders was added. This way, once the robot drifted from its origin, it would try to return. The issue was, however, that there was a lot of overshoot on its return path, eventually making the robot unstable. To avoid the overshoot a derivative control was added based on the wheel encoders.

In the end, it took two controllers, a PI based on the gyro, as well as a PD based on the wheel encoders, to keep the robot upright and at a fixed position.

2.4 SPI Communication

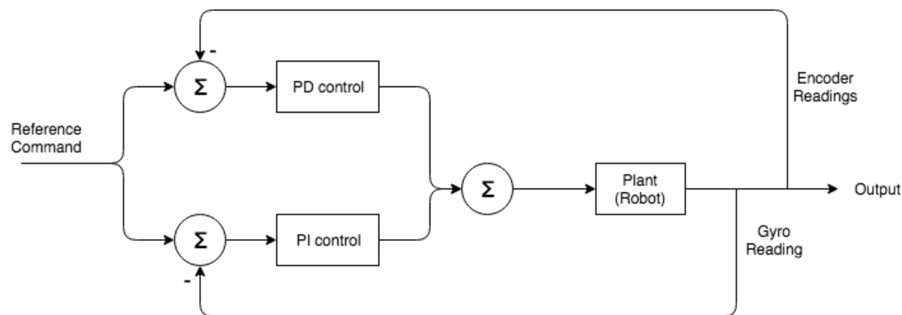
In order to adjust the gains for the controller without having to generate a new bitstream every time, the robot communicated with an MCP3008 ADC (using SPI) that was wired to potentiometers to tune the gains. One practical problem I encountered was that while tuning the gains, the robot would fall over frequently, shifting the sensitive potentiometers. In order to counteract this problem, the SPI module was only triggered, updating the gains, once a button on the FPGA was pressed.

2.5 Commands

By the end of the project, the robot was able to recognize up to 256 different commands, but it was unable to execute most of them. Although the infrastructure was there to execute "forwards" and "backwards" commands, the dynamics of the problem make it rather difficult to execute them. Specifically, there are two relatively simple ways to command the robot to move forward, but both have their challenges. The first way is to command a reference angle that is not 0. This way, the robot will remain tilted and thus, will move forward. However, this can only be done for short periods of time, as this method effectively commands an acceleration, which the robot can only sustain until it reaches its top speed. The second way, and the way we were closer to implementing, is to add an offset to the encoders. The challenge that we found here was that once the offset is added, there is a large "spike" in the commanded speed due to the encoders and their speed. Meanwhile, if the offset is added continually, the derivative part of the control becomes difficult, as the wheels are not truly moving, but the robot thinks they are.

3 Diagrams

The type of control used, as described above, was a PI controller on the gyro readings and a PD controller on the wheel encoder readings. The image below shows how the control system was implemented in a block diagram.

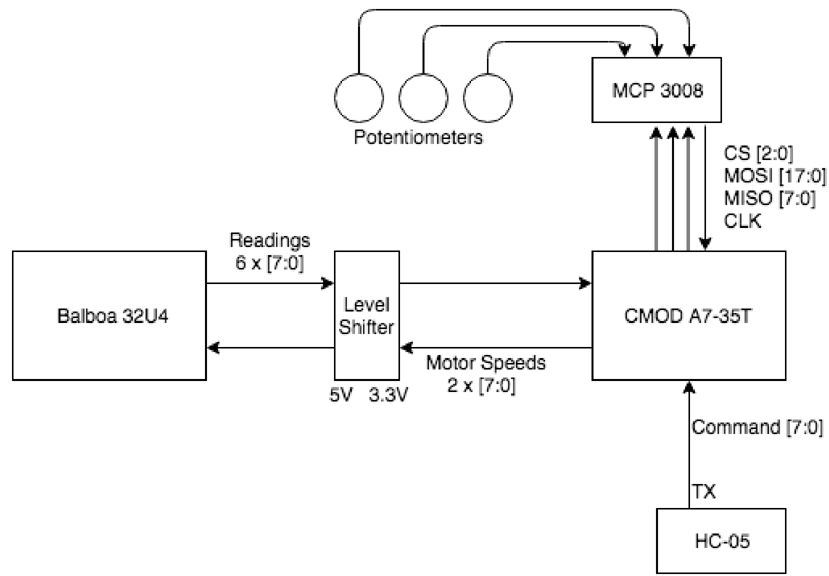


As shown in the diagram, there are two control loops in play. The above loop takes the encoder readings, subtracts them from the desired readings, and inputs the error into a PD controller. Motor speeds are then decided based on the error as well as the derivative of the error, and the motor speeds are passed on.

At the same time, gyro readings are taken, subtracted from from a reference (which included the gyro offset), and fed into the PI controller. The PI controller output motor speeds based on the gyro reading (angular velocity), and its integral (angle).

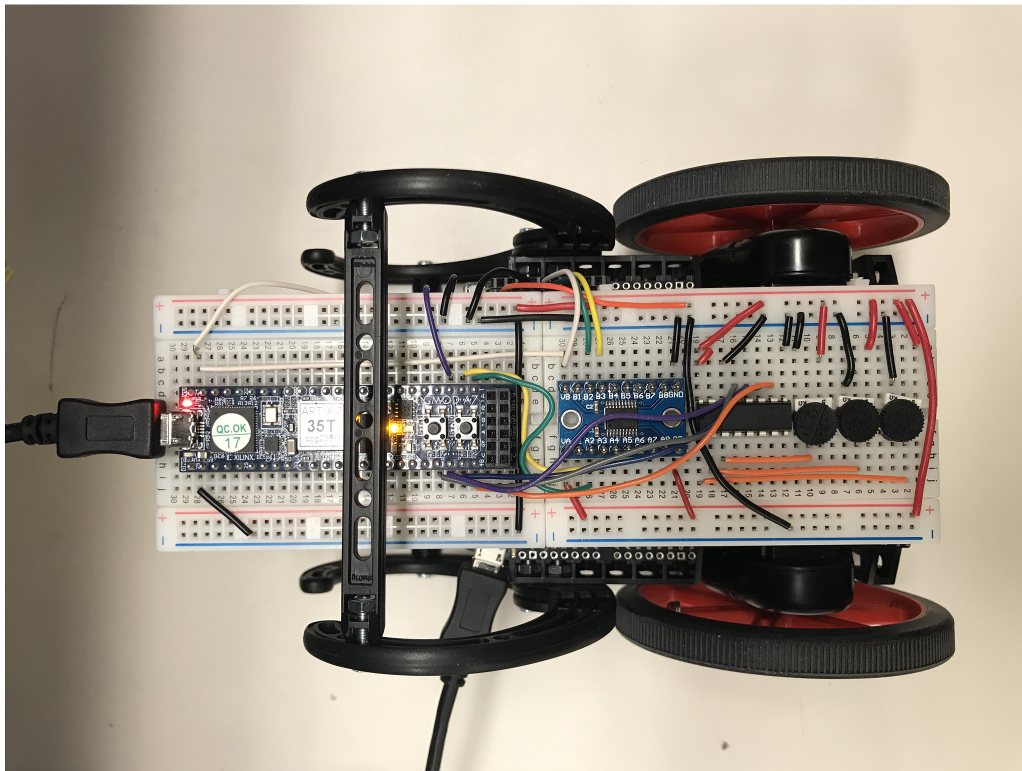
The two outputs from the controllers are then combined and input into the plant, the robot, which commanded the system dynamics. This whole process was executed every 18ms, or 55 Hz.

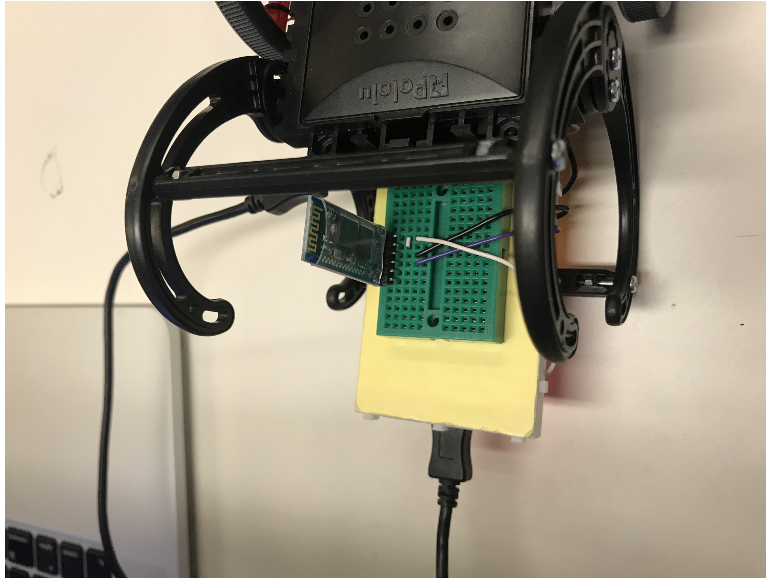
The diagram below shows the circuit schematic and how the robot was set up:



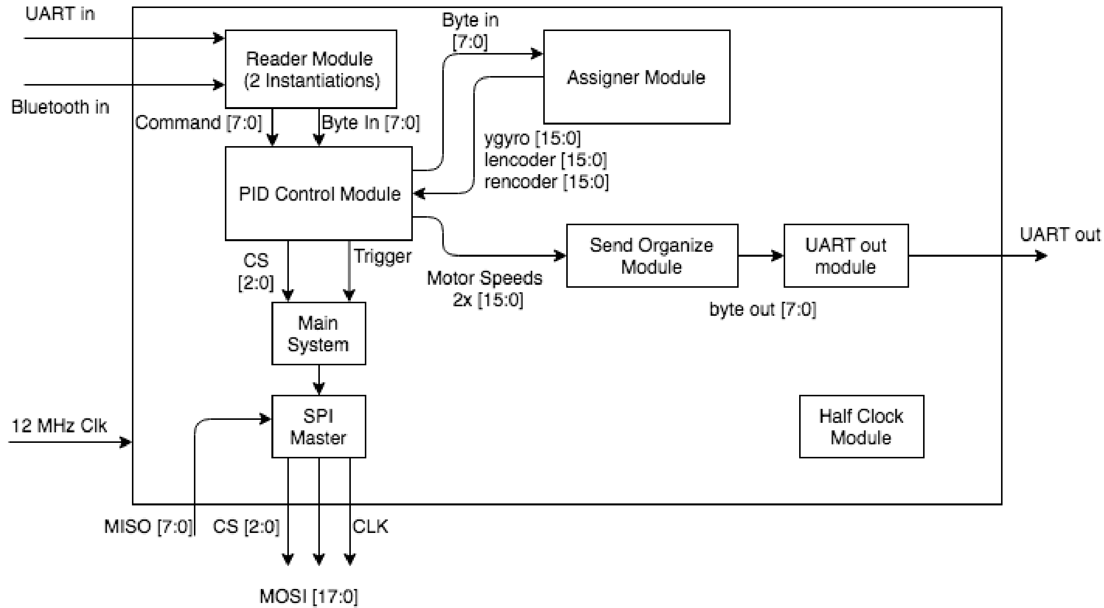
Each signal starts at the Balboa 32U4. It is then sent out, through the level shifter, and read by the CMOD A7. At the same time, the CMOD A7 uses readings from the ADC and potentiometers, along with readings from the Bluetooth module, to generate signals which are sent back. These signals are returned through the level shifter to the Balboa and used to update the motor speeds.

Below are some pictures of the circuit on the robot and how it was mounted. The breadboards were mounted on a piece of polycarbonate, which was cut using a waterjet and mounted onto the Balboa frame using nylon standoffs.



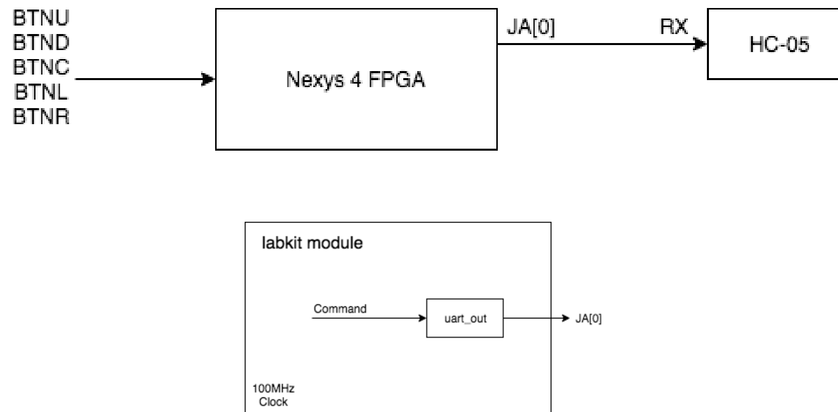

Pololu

Within the CMOD A7, the modules were organized as shown in the block diagram:



The information flow starts in the reader module and bytes are then transferred to the assigner module. Once the values were known, the pid control module, using readings from the main_system module and the read command instantiation, determines new motor speeds. These new speeds are given to the send_organize module, which then provides bytes for the uart_out module to return to the Balboa

The Nexys 4 had a much simpler block diagram and circuit setup, as shown below:



4 Modules

4.1 pidcontrol

The pidcontrol module was the main module that interfaced with all other smaller modules. This module held the values for all of the readings that came in from the reader module, as well as the values that came out of the assigner module. It also implemented the PI and PD controllers using wires and integrated the necessary values using registers. Since it had all of the motor speed values and gains, it also provided an input to the send_organize module as well as the uart_out module. Lastly, this module also handled the commands that were read in by the second reader instantiation.

4.2 reader

The reader module was used to read in UART at a baud rate of 9600. This module was fairly robust, as it checked for start and stop bits, as well as over-sampled the data received, keeping only the sample in the middle of each bit, which helped minimize glitches. This module was implemented using an FSM that was in an idle state until it read the start bit, would then confirm that the low signal was not a glitch, and would begin reading the data in LSB to MSB. Lastly, it would check for the stop bit, once again ensuring that the high signal was not a glitch, and would return to its idle state. Once a full, valid, byte was read, the module would return a valid signal for one clock cycle, indicating to the other modules that a full byte had been read. The parameter ”cpb” (cycles per bit) was used to adjust for different clocks on each FPGA.

4.3 assigner

The assigner module was used to assign the different bytes output by the reader module to the corresponding variables (gyro, left encoder, right encoder). In order to assign the correct bytes to the corresponding variables, this module had to first sync up with the Balboa 32U4. In order to do this, the module would wait for the byte 1111_1111, which was the first byte sent for the gyro reading (the gyro without an offset would return values around -220). In earlier versions, this module used the 0 values due to the encoders starting at 0, but the -1 byte seemed to be more reliable. Once this initial gyro reading was found, the module knew what bytes followed, and was able to assign them to the correct variable. Every 6 bytes received and assigned, each variable had been fully updated, and so the module would return a ”valid” signal for one clock cycle, indicating to the pidcontrol module that these values had been properly updated.

4.4 send_organize

Once the pidcontrol module had motor speeds to return, the send_organize module took the 4 bytes of information and, one byte at a time, updated the byte sent by the uart_out module. In order to prevent glitches, this module only updated the bytes to send once the motor speeds had been properly updated. Also, this module had to be synced with the reader module, as the Balboa expected to read the motor speeds at the same time as it transmitted the readings.

4.5 uart_out

This module was used to communicate the motor speeds to the Balboa. Similar to the reader module, it was based around an FSM which, went told to communicate a byte, would begin the send the start bit. Then using the adjustable parameter, ”cpb” (cycles per bit), it would send each bit of the byte being sent, followed by a stop bit, and would then return to idle. This module was quite delicate in its timing, as the Balboa expected to be reading bytes in at the same time as sending bytes out. So, this module began sending a byte as soon as it was triggered by the reader module, indicating that it had begun reading a byte. Due to the 12MHz clock of the CMOD, one clock cycle delay when communicating at 9600 baud resulted in no noticeable delay, and thus, there were few glitches.

4.6 half_clock

The half clock module simply created a 3MHz clock, one fouth the speed of the system clock (despite the name of the module), in order to run the Vivado ILA. During initial debugging and testing of the modules, the ILA was used to ensure values were read in properly. Because of the difference in speed of the 12Mhz system clock and the 9600 baud communication rate, full bytes were not able to be read in before the ILA stopped. So, a slower, 3MHz clock was created in order to get more ”samples” for the ILA. This way, full bytes could be received before the ILA stopped taking samples.

4.7 main_system

The main_system module was written by Joe Steinmyer but required slight modification. The chip select was made an input, while the byte read was made an output. In order to be able to call it from the pidcontrol module, the inputs and outputs were changed. The inputs, including a trigger, chip select, and the 4 SPI wires were provided by the pidcontrol module, while the main_system module provided the byte read as an output to the pidcontrol module.

4.8 spi_master

The spi_master module was also written by Joe Steinmyer and also required slight modification. The module was changed to read from MSB to LSB.

4.9 labkit

This module was uploaded to a Nexys 4 FPGA and would send one of 4 pre-determined commands when buttons were pressed. This was done using the uart_out module described earlier.

Balboa 32U4 Code

Code was also written for the Balboa in order to communicate with the CMOD A7. This code read the new gyro and encoder readings each cycle, break these readings down to bytes, and send the information via UART at 9600 baud to the CMOD. At the same time, it read back the bytes that were assigned to the motor speeds. Similar to the CMOD, the Balboa also had to know what bytes to assign to what variables. In order to do this, the CMOD sent two pre-determined bytes, followed by the motor speeds. This way, once the Balboa was synced, it knew what byte corresponded to what motor speed. Once this was done, a speed limit recommended by the manufacturer of the robot was imposed on the received speeds, and the speeds were then assigned to the corresponding motors.

5 Challenges and Solutions

5.1 Timing Challenges

Once of the first challenges that came up was the glitches due to updating values while data was sending. In order to prevent glitches, all of the modules that updated byte values needed to update the values at the same time. So, there needed to be some signal that would trigger the different modules to update. The way this was solved was by having the reader module output a "valid" signal, which would trigger a chain of updates. Every six of these valid signals, the assigner module would signal that the gyro and encoders had been properly updated. When updating the gains using the SPI module, we also had to ensure the gains would not be updated while calculating new values for the motor speeds. So, the SPI could only be triggered on a valid set of information. At this point, the feedback control was implemented. As a result of this timing strategy, every module ran on a trigger.

Slow communication rates were another issue that arose during this project. When a digital, or discrete, system over-samples past a certain rate, which is about $0.03 \cdot T$ where T is the characteristic time of the system, the system model and controller can be treated as continuous time. When dealing with an 18ms update time, we were at $0.018 * T$ (assuming T was around 1 second). However, as we added more data, our update times would slow down further. Specifically, had we transmitted the x and z accelerometer readings, we would have been approaching the threshold for continuous modeling. So, we opted to only communicate the gyro and encoder readings, and approached the controls problem in a different way. Luckily, commands received did not need to update nearly as fast as the rest of the system, so speed was not an issue.

5.2 Gyro Drift

The next problem encountered was the gyro drift. Had we used one gyro and two accelerometer readings, a sensor fusion approach (as described in the lecture notes) could have been implemented. This way, we would reduce the low frequency noise of gyros along with the high frequency noise of the accelerometers. Unfortunately, due to timing constraints, we did not use the accelerometer values. So, the way the robot implemented control about the angle offset (θ, the integral part of the PI control), was to assume the angle was close to 0. Specifically, over the course of each clock cycle, the angle calculated would be the combination of the integration of the gyro, as well as a 3% decay from the last angle (for update time t = 18ms)

$$\theta_{new} = t \cdot G_y + 0.97 \cdot \theta_{old}$$

The result of using this approximation was that the robot never knew if it was at 0. It only knew if it deviated from its current state. Luckily, the only angles at which the robot would not deviate from its current state were 0 or when the robot was laying down. So, in the end, the robot was able to stay upright, but would have no way to track its position?, and would be "happy" as long as it was able to maintain a given angle.

5.3 PD Controller Challenges

To fix the issue mentioned above, a PD controller on the wheel encoders was used. At first, we hoped to only need a proportional controller on the wheel encoders. However, this type of control caused a lot of overshoot. So, derivative control on the wheel encoders was necessary. One of the downsides of derivative control is that it has an infinite gain at high frequencies. So, there was an upper bound on the gain prescribed to the speed of the wheel encoders. So, there was a balance between trying to damp the system so that the robot would oscillate over a smaller distance and preventing shaking at high frequencies. Furthermore, some of the overshoot was inevitable, as the type of feedback control implemented included a "zero" in the forward path. Essentially, what this meant was that because we were taking the derivative of the error, and not just the read signal, there was going to be some overshoot.

5.4 Tuning Gains

Another challenge was tuning the gains. At first, it was not too difficult to adjust the I gain for the angle control. That is, the feedback control based on θ. The proportional gain ended up being 90. The next gain to adjust was the P gain, based on ω. The difficult part here was that the ω and θ values were orders of magnitude different. This was due to θ being affected by the update time. So, after considering the order of magnitude difference, and adjusting the proportional gain accordingly, we ended up with a gain of 6500 for ω. Finding the gains for the encoders was a whole new challenge, as they were not related to the angle or angular velocity. However, after much trial and error at different orders of magnitude, the gain for d, the distance covered by the encoders, was 80. Next, a similar order of magnitude analysis was used to find the gain for v, the velocity of the encoders, which turned out to be 30000. The last challenge that came with tuning the gains was that every time the robot would fall over, the potentiometers would shake and the gains would change. The solution for this problem was to make the button a trigger for the SPI communication with the ADC to update the gains. That way, even if the robot fell over, no gains would update until the changes were intentional.

5.5 Assigning Values

There were also difficulties that came with the syncing between the Balboa and the CMOD A7 because we wanted to avoid sending wasted bytes. At first, we tried to use the encoder readings, which always started at 0, as a reference point. However, 4 out of the 6 bytes received started at 0, which made things more complicated. So, the solution was to use the initial gyro readings, which always started at around -220. In this way, the first byte sent for the gyro value was always -1 in twos-complement format. Even

if the value fluctuated away from -220, the first byte would always be -1 which is what the CMOD A7 used as a reference for future values.

5.6 Differentiation

Integrating versus differentiating readings also became a decision that had to be made. Because the update time was 18ms, it was not easy to differentiate values. So, a last minute decision to send v, the velocity of the encoders, along with the original ω, was made. This is because it is a lot easier to integrate the velocity of the encoders over time on the FPGA than to differentiate and find the velocity at a given time step.

5.7 ILA Testing

Initial testing of the modules was also quite tricky, as values were changing faster than one could look at LEDs or other visual forms to debug. So, I used the Vivado ILA to see how the vales changed over time. However, since the 12MHz clock of the CMOD A7 was far faster than the rate at which the data changed at 9600 baud, not much could be seen in the ILA, as the maximum number of samples was around $131,072$. So, a slower clock module was made, at one fourth the speed of the system clock. This way, effectively 4 times as many samples could be taken and studied with the ILA. One recommendation I do have for people who have similar projects in the future is to get a module similar to the uart_out module working, and rigorously tested. This way, the UART communication can be displayed on the serial monitor of the Arduino IDE and values can be seen changing in real time.

5.8 Physical Constraints

The last significant problems that came with this project had to do with the physical constraints set by the robot. Although it was inconvenient to have to use the Polou A* library to interface with the Balboa board, the challenges were more based on the properties of the robot itself. Firstly, the motors were slightly under powered for the original robot, and adding 3 breadboards full of components certainly did not help. This resulted in slightly slow response times. To counteract this, high gains were used. However, a motor speed recommended by the manufacturers was also imposed. So, the robot reached the motor speed rather quickly, giving the controllers little time to help stabilize the robot. At the same time, although the motors were always commanded the same speed, the wheels did not turn the same amount. So, the robot slightly rotated over time. A fix for this would have been to have a feedback controller based on the difference between the encoders, but that would likely have required α-blending of the commands from the different controllers. Lastly, the construction of the robot was not ideal. Most of the mass, carried by the batteries, was close to the floor. As a result of this, the moment of inertia about the axis of rotation of the wheels was quite low, making the robot even more unstable. A future improvement would be to have the batteries and other heavy objects near the top of the robot. That way, the robot would become easier to balance (It is like balancing a yard stick versus a pencil on your hand).

6 Goals

Most of the goals for this project were met, but there were also some unforeseen complications, which made some expected goals more difficult, requiring stretch goals to be implemented.

6.1 Baseline Goals

The baseline goals of UART communication with the Balboa 32U4 and SPI communication with the MCP3008 were both met. Additionally, both the proportional and integral control loops were implemented. Note that the initial design specified this as a PD controller, based on the angle θ and its

derivative, ω. In reality, however, it is more accurate to call this controller a PI controller, as the value read was ω.

6.2 Expected Goals

The expected goal of bluetooth communication was also met, and entire bytes could be sent between FPGAs via bluetooth.

The expected goal of executing commands, however, came with complications. Although the robot was able to correctly recognize commands, as well as execute simple commands as "fall over" or "stay balancing", it was unable to execute commands involving going forwards of backwards. This was originally due to the fact that there was no integral control, so the robot could not track its location. Later on, the lack of ability to execute commands involving movement was due to the unexpected complexity of modifying the read values to trick the system into executing the desired command.

6.3 Stretch Goals

The stretch goal of integral control became critical to implement to be able to keep the robot upright in one place. However, it was not enough. Going past this stretch goal, a derivative controller on the wheel encoders was also necessary, and implemented.

6.4 Conclusion

Overall, almost all baseline and expected goals were met. Despite missing one expected goal, some stretch goals, along with goals not even considered beforehand, became necessary to make the robot work. Additionally, there was quite a bit of complexity in assigning the bytes read, as well as organizing bytes to send, that was not initially considered.

7 Future Work

I think there are multiple routes to be taken to follow up on the work done on this project.

The first addition to the robot should be a controller that prevents the different rotation of the wheels. That is, a proportional feedback controller based on the difference in the encoders readings. This controller would prevent undesired rotation.

Next, the ability to execute more complex commands would be interesting to implement. Specifically, the α-blending of turning commands along with the commanded motor speeds from the balancing controllers.

Executing preset routes would also be a challenge to implement. For example, one could tell the robot to move in a circle, and a series of commands stored in the FPGAs memory could be used to move the robot in a circle.

Another extension would be to expand the robot's communication ability. Using the bluetooth modules, the robot could provide feedback on its relative location, velocity, and angle, back to the Nexys4 FGPA.

Overall, I think that this project provides a solid foundation that could facilitate more sophisticated future projects. For instance one could build a robot that not only self balances, but can also execute commands and communicate back to the Nexys4. This would be a step closer to how real world robots work.

8 Acknowledgements

I would like to thank Joe Steinmyer for his help throughout the project, helping get the components for the robot, as well as his quick replies to the late night emails I sent about the problems I was having.

9 Appendix

9.1 Final Verilog Code

Note that I did not include the verilog for the SPI communication here. However, I have uploaded my modified version of it to the course website.

```
`timescale 1ns / 1ps

module pidcontrol(
    input wire sysclk,
    input [1:0] btn,
    input uartin,
    input bluein,
    inout [1:0] pio,
    inout [3:0] g,
    output [1:0] led,
    output led0_b,
    output led0_r,
    output led0_g,
    output uartout

    );




    //ILA Testing

    wire tclk;

    //ila_0 ilatest( .clk(tclk),
                        .probe0(flip),
                        .probe1(lencoder[7:0]));

    half_clock testclk(.clk(sysclk),
                        .new_clk(tclk));


    //COMMANDS

    wire [7:0] command_byte;
    reg[7:0] command_reg = 8'b11111111;
    wire command_valid;


    assign led0_r = (command_byte != 8'b0000_0001); //STOP
    assign led0_g = (command_byte != 8'b00000010); //FORWARD
    assign led0_b = (command_byte != 8'b00000011); //BACKWARD

    assign kill_switch = (command_reg == 8'b0000_0000) | btn[0];
```

```
reader2 readcommand(.clk(sysclk),
                            .streamin(bluein),
                            .byte_out(command_byte),
                            .clean(command_valid));

//SPI


wire spi_reset;
assign spi_reset = btn[0];

wire spi_driver;
assign spi_driver = btn[1];

wire signed [7:0] spi_byte_out;
wire [2:0] spi_cselect;

assign spi_cselect = 3'b000;

main_system spi(.sysclk(sysclk),
                    .spi_reset(spi_reset),
                    .spi_driver(spi_driver),
                    .cselect(spi_cselect[2:0]),
                    .g(g[3:0]),.pio(pio[1:0]),.byteout(spi_byte_out));



//UART Commuication


wire reset;
wire [7:0] transfer_byte;
wire transfer_valid;
wire sending;

wire [7:0] byte_to_send;

wire flip;

//READ IN VALUES, ASSIGN THEM TO VARIABLES,
    //ORGANIZE BYTES TO RETURN, RETURN BYTES

reader2 readtest(.clk(sysclk),
                    .streamin(uartin),
                    .byte_out(transfer_byte),
                    .clean(transfer_valid));

assigninv2 assignerv2 (.clk(sysclk),
                            .reset(reset),
                            .valid(transfer_valid),
                            .byte_in(transfer_byte),
                            .ygy(ygyro),
                            .lwenc(lencoder_speed),
```

```
                              .rwenc(rencoder_speed),
                              .clean(flip));

send_organize sender(.clk(sysclk),
                              .trigger(transfer_valid),
                              .byte1(motorspeedAvg[15:0]),
                              .byte2(motorspeedAvg[15:0]),
                              .byteout(byte_to_send));


uart_out transmitter(.clk(sysclk),
                              .valid_byte(transfer_valid),
                              .byte_in(byte_to_send),
                              .sent(sending),.streamout(uartout));






//GAINS TO USE ONCE FOUND

parameter signed [15:0] D_RESPONSE = 16'd6500;
parameter signed [15:0] P_RESPONSE = 16'd90;
parameter signed [15:0] P_RESPONSE_a = 16'd0;
parameter signed [15:0] I_RESPONSE = 16'd80;
parameter signed [15:0] I_D_RESPONSE = 16'd30000;
parameter signed [15:0] I_diff_RESPONSE = -16'd60;
parameter signed [15:0] CYCLE_TIME = 16'd10;



    //VALUES WE ARE READING IN

    wire signed [15:0] ygyro,
                       rencoder_speed,
                       lencoder_speed,
                       lencoder_hold,
                       rencoder_hold;


    wire signed [23:0] adj_gain;

    //VALUES WE ARE THEN GETTING


reg signed [15:0] lencoder, rencoder;


reg signed [23:0] angle;

wire signed [23:0] angle_rate,
```

```
                    d_part, p_part,
                    i_p_part,
                    i_d_part,
                    p_part_a,
                    diff_part,
                    dist_diff;



    reg signed[11:0] gain_reg;
assign adj_gain = gain_reg;


    //VALUES WE WILL OUTPUT

wire signed [31:0] motorspeedAvg,
                   motorspeedLeft,
                   motorspeedRight;




    //ASSIGNMENTS


    //this gives us a slight buffer that will help avoid glitches
assign rencoder_hold = rencoder_speed;
assign lencoder_hold = lencoder_speed;


    //get the angle rate, reducing noise,
    //and cancelling the gyro offset
assign angle_rate = (ygyro + $signed(220))>>>5;



assign d_part = (angle_rate) * D_RESPONSE;
    //part based on omega

assign p_part = angle * P_RESPONSE;
    //part based on theta

assign i_d_part = (rencoder_speed + lencoder_speed) * I_D_RESPONSE;
    //part based on encoder speed


assign i_p_part = (rencoder + lencoder) * I_RESPONSE;
    //part based on encoder distance

assign motorspeedAvg = kill_switch ? 0:
                                        ((d_part)
                                        + (p_part)
                                        + (i_p_part)
```

```
                                        + (i_d_part)
                                        + p_part_a) >>> 13;

//assign all the parts to the motor speed average


        //get the difference between the wheels

    assign ddiff = lencoder - rencoder;
    assign diff_part= ddiff* I_diff_RESPONSE;



   //these two lines were never implemented,
   //but they are here to help make the robot not turn

   assign leftmotorwire = motorspeedAvg - diff_part;
   assign rightmotorwire = motorspeedAvg + diff_part;

    always @(posedge sysclk) begin

        if(command_valid) command_reg <= command_byte;


        if(flip)begin

                if (~btn[1]) gain_reg <= spi_byte_out * $signed(4);
                               //get new gain


                               //integrate encoder speeds
                rencoder <= rencoder + rencoder_hold;
                lencoder <= lencoder + lencoder_hold;

                angle <= ($signed(1000)*((angle) + angle_rate * CYCLE_TIME))>>>10;
                               //integrate angular velocity and make angle decay

        end

    end

endmodule




//uart out communication module
module uart_out(input clk, valid_byte, [7:0] byte_in, output sent, streamout);
    reg [2:0] state;
    reg [15:0] cycle_counter;
    reg [3:0] bit_index;
    reg streamreg;
    assign streamout = streamreg;
```

```
wire [15:0] cpb;
assign cpb=  1250;

reg sentreg;
assign sent = sentreg;

reg [7:0] send_byte;
reg send_one;

parameter START_BIT = 3'b000;
parameter DATA_BITS = 3'b001;
parameter STOP_BIT = 3'b010;
parameter IDLE = 3'b011;



    //this module uses an fsm that starts with the start bit,
    //and holds each necessary bit for cpb, the clock cycles per bit.

always @(posedge clk) begin
    if (valid_byte) begin
            state<= START_BIT;
            cycle_counter <= cpb;
            send_byte <= byte_in;
end

if(cycle_counter == cpb) begin
    cycle_counter <= 0;

case(state)
    START_BIT: begin
    sentreg <= 1;
    streamreg <= 0;
    state <= DATA_BITS;
end

DATA_BITS:begin
    sentreg <= 1;
    streamreg <= send_byte[bit_index];
    if (bit_index == 7) begin
            bit_index <= 0;
            state <= STOP_BIT;
    end
    else bit_index <= bit_index + 1;
end

STOP_BIT: begin
    sentreg <= 0;
    streamreg <= 1;
    state <= IDLE;
    send_one <= 0;
end
```

```
    endcase

    end

    else cycle_counter <= cycle_counter + 1;

    end


endmodule



//this module will read in uart
module reader2 (input clk,
                input streamin,
                output [7:0] byte_out,
                output clean);

    parameter [15:0] cpb = 1250;

    parameter IDLE = 3'b000;
    parameter START = 3'b001;
    parameter DATA = 3'b010;
    parameter STOP = 3'b011;

    reg [2:0] state;
    reg [15:0] cycle_counter;

    reg [7:0] byteoutreg;
    reg [3:0] bit_index;
    reg starting;

    assign clean = starting;
    assign byte_out = byteoutreg;


        //this module uses an fsm that gets triggered when it
        //reads a start bit. it then waits cpb, cycles per bit,
        //and samples the input to get the following bits.
        //Once it starts reading, it also sends a high signal
        //for one clock cycle to signal to other modules that it is reading.


    always @(posedge clk) begin
            starting <= 0;
            case(state)

                IDLE: if(streamin == 0) begin
                    starting <= 0;
                    state <= START;
                    cycle_counter <= 0;
                end
```

```
        START: begin
            starting <= 0;
            if (cycle_counter == cpb / 2) begin
                cycle_counter <= 0;
                state <= DATA;
            end

            else if (streamin == 0) cycle_counter <= cycle_counter + 1;

            else begin
                state <= IDLE;
                cycle_counter <= 0;
            end

        end

        DATA: begin
            starting <= 0;
            if (cycle_counter == cpb) begin
                cycle_counter <= 0;

                if(bit_index <= 7) begin
                    byteoutreg[bit_index] <= streamin;
                    bit_index <= bit_index + 1;
                end

                else begin
                    state <= STOP;
                    bit_index <= 0;
                end

            end

            else cycle_counter <= cycle_counter + 1;

        end

        STOP: begin
            starting <= 0;
            if (cycle_counter == cpb/4) begin
                cycle_counter <= 0;
                state <= IDLE;
                starting <= 1;
            end
            else cycle_counter <= cycle_counter + 1;

        end


    endcase
```

```
    end


endmodule


//this module assigns the different bytes
//that are read to their corresponding values
module assigninv2 (input clk,
                        reset,
                        valid,
                        input [7:0] byte_in,
                        output [15:0] ygy,
                        lwenc,
                        rwenc,
                        output clean);

    reg synced;

    reg [15:0] ygyhold, lwenchold, rwenchold;
    reg [15:0] yreg, lreg, rreg;
    reg flip;
    reg [15:0] old_l;
    assign clean = flip;

    assign ygy = ygyhold;
    assign lwenc = lwenchold;
    assign rwenc = rwenchold;


    reg [3:0] byte_counter = 1;

    always @(posedge clk) begin

        if(valid) begin
            if(byte_counter == 0) begin
                ygyhold <= yreg;
                lwenchold <= lreg;
                rwenchold <= rreg;
                flip <= 1;
            end
            else flip <= 0;

            if(synced) begin
                if(byte_counter == 5) byte_counter <= 0;
                else byte_counter <= byte_counter + 1;

                                //assign values based on number
```

```
                        //of bytes that have been read
                case(byte_counter)
                1: yreg[15:8] <= byte_in;
                0: yreg[7:0] <= byte_in;
                3: lreg[15:8] <= byte_in;
                2: lreg[7:0] <= byte_in;
                5: rreg[15:8] <= byte_in;
                4: rreg[7:0] <= byte_in;
                endcase

            end

                    //this is how the module syncs to the input bytes,
                    //using the MS byte of the gyro reading
            else if(byte_in == 8'b11111111) begin
                synced <= 1;
                byte_counter <= 2;
                yreg[15:8] <= byte_in;
            end


        end
        else flip <= 0;
    end

endmodule





//this module organizes the values that need to be sent
//back to the balboa into bytes. It takes in a maximum of 4 bytes.

module send_organize(input clk,
                    trigger,
                    input [15:0] byte1,
                    byte2,
                    output [7:0] byteout);

    reg [3:0] byte_counter;

    reg [7:0] byteoutreg;

    assign byteout = byteoutreg;
    reg enable;

    always @(posedge clk) begin
        if(trigger) enable <= 1;

        if (enable) begin
            enable <= 0;
            if(byte_counter == 5) byte_counter <= 0;
```

```
            else byte_counter <= byte_counter + 1;

            case(byte_counter)
                0: byteoutreg <= byte1[7:0];
                1: byteoutreg <= byte1[15:8];
                2: byteoutreg <= byte2[7:0];
                3: byteoutreg <= byte2[15:8];
                4: byteoutreg <= 8'b1000000;
                5: byteoutreg <= 8'b1000000;
//these last two bytes are sent to allow the balboa to sync
            endcase
        end
    end


endmodule



//this module creats a slower clock that is to be used with the ILA
module half_clock(input clk, output new_clk);
    reg [3:0] flip;
    reg clkreg;
    always @(posedge clk) begin
        if (flip == 4) begin
            flip <= 0;
            clkreg <= ~clkreg;
        end
        else flip <= flip+1;
    end

    assign new_clk = clkreg;

endmodule
```

9.2 Final Arduino Code

```
#include <Balboa32U4.h>
#include <Wire.h>
#include <LSM6.h>
#include<math.h>
#include <elapsedMillis.h>

int primary_timer = 0;
elapsedMicros comm;
uint8_t comm_pin = 0;

int8_t newbyte = -1;

uint8_t thresholds[] = {128, 64, 32, 16, 8, 4, 2, 1};
int32_t ygyro, ygyro0 , lwe, rwe;
```

```
int16_t lmspeed, rmspeed, mspeed, mspeedold;
int16_t lmspeedold, rmspeedold;

int16_t lencspeed, lencoder, lencoder_old;
int16_t rencspeed, rencoder, rencoder_old;


int16_t angle1, anglerate1;


bool synced = 0;
int8_t first1, first2, second1, second2;


LSM6 imu;
Balboa32U4Motors motors;
Balboa32U4Encoders encoders;
Balboa32U4Buzzer buzzer;
Balboa32U4ButtonA buttonA;

void setup()
{
  Wire.begin();


  //Initialize IMU, code from board manufacturer
  if (!imu.init())
  {
    ledRed(1);
    while (1)
    {
      Serial.println(F("Failed to detect the LSM6."));
      delay(100);
    }
  }
  imu.enableDefault();
  delay(1000);



  //Code from manufacturer of board:

  // Set the gyro full scale to 1000 dps because the default
  // value is too low, and leave the other settings the same.
  imu.writeReg(LSM6::CTRL2_G, 0b101011000);

  // Set the accelerometer full scale to 16 g because the default
  // value is too low, and leave the other settings the same.
  imu.writeReg(LSM6::CTRL1_XL, 0b10000100);


  //Declare communication ouputs
```

```
  Serial1.begin(9600);
  pinMode(comm_pin, OUTPUT);
  digitalWrite(comm_pin, 1);
  primary_timer = millis();
}


void loop()
{
  //update values
  imu.read();
  lwe = encoders.getCountsLeft();
  rwe = encoders.getCountsRight();



  //differentiate encoders
  lencoder_old = lencoder;
  lencoder = lwe;

  lencspeed = lencoder-lencoder_old;

  rencoder_old = rencoder;
  rencoder = rwe;
  rencspeed = rencoder - rencoder_old;

  //break values into bytes to send
  int8_t hey[6] = {};
  uint8_t heyback[6] = {};

  hey[0] = imu.g.y;
  hey[1] = (imu.g.y >> 8);
  hey[2] = lencspeed;
  hey[3] = (lencspeed >> 8);
  hey[4] = rencspeed;
  hey[5] = (rencspeed >> 8);

  Serial.println("SET");

  for (uint16_t j = 0; j < 6; j++) {

    Serial1.write(hey[j]);

    heyback[j] = Serial1.read();

    digitalWrite(1, HIGH);
    delay(2);
    //add a delay to minimize glitches

  }


  //sync to the values being sent back
```

```
  if (synced == 0) {
    for (uint8_t j = 0; j < 6; j++) {
      if (heyback[j] == 64 && heyback[(j + 1) % 6] == 64) {
        first1 = (j + 2) % 6;
        first2 = (j + 3) % 6;
        second1 = (j + 4) % 6;
        second2 = (j + 5) % 6;
        synced = 1;
        break;
      }
    }
  }

  //assign speeds

  lmspeed = ((heyback[first2] << 8) + heyback[first1]);
  rmspeed = ((heyback[second2] << 8) + heyback[second1]);

 //speed limits

  if (rmspeed > 300) {
    rmspeed = 300;
  }
  if (lmspeed > 300) {
    lmspeed = 300;
  }
  if (rmspeed < -300) {
    rmspeed = -300;
  }
  if (lmspeed < -300) {
    lmspeed = -300;
  }


  //get average speed to prevent glitches affecting the motors
  mspeed = 0.5*lmspeed + 0.5*rmspeed;

  if (mspeed > 300) {mspeed = 300;}

  if (mspeed < -300) {mspeed = -300;}

  motors.setSpeeds(mspeed,mspeed);

}
```