ECE 3710 Final Report

Rich Baird, Miguel Gomez, Tyler Liddell, Hyrum Saunders

December 17, 2021

Abstract

Many systems exist that require some level of control over a variable that can be set to a particular value and left to stabilize. Systems like 3D printers, for example, need control over the temperature of the nozzle and a print bed to ensure the conditions are favorable to the printed parts. Others may follow similar steps to control the amount of humidity in a greenhouse or soil saturation levels to keep conditions favorable to plants. The ability to have this control enables quick changes to a device's parameters without the need to recompile or flash new firmware–speeding up prototyping, final part manufacturing, and finally grow yields for these examples. Our goal with this project was to implement a system such as this that could control an object's position depending on the feedback received by the sensors. The system's overall behavior is pointed and somewhat niche, but the control system implemented could be dropped into any system that needs this type of control. Once the module is integrated into the system, the rest is tuning the parameters to match the desired conditions.

The idea is a simple one–capture data incoming that carries some information about the current state of that system, use those values to adjust the behavior to approach a steady-state, feed those values back to the system so it may react accordingly, wait a specified time to allow the system to change, capture data incoming from the system, and repeat. This report will detail the steps taken to accomplish this using our FPGA CPU implementation as the heart of the system.

Introduction

Our final project was a PID control system that balanced a ball on a wooden plate. A camera watched the ball and transmitted the x and y coordinates of the ball to our FPGA using UART. Our PID was implemented on the FPGA. It computed an x and y offset and stored it in registers that our assembly translated into motor movements, storing those movements in other registers. The values from those registers were used by our motor driver module, again implemented on our FPGA, to send signals to the physical motor drivers driving the motors.
Additionally, we connected a nunchuk to our project using an Arduino, so that we could also control the platform using the nunchuk. The nunchuk communicated with the Arduino over I2C, which forwarded the signals from the nunchuk on to our FPGA over UART. The signals were translated into motor movements and used to drive the motors in the same way as the signals from the camera.

1 The Overall Design

Below is a figure showing our final bdf file which we synthesized as our top level module. We abstracted blocks into their own modules to help reduce clutter and increase readability. The top third of the image includes our CPU and the main datapath. The middle section includes the uart interfaces with both the Arduino and Pixycam. The bottom portion includes our x and y PID, the pieces necessary for the PIDs to work, as well as the motor driver. Many blocks, such as muxes, baud_clock, uart_rec, PID, my_dff, SignExtend, absoluter, as well as others are used multiple times throughout the diagram. The idea with each module was to allow it to perform a simple function that can be abstracted away into a single block, and use that block in any place it makes sense to use.

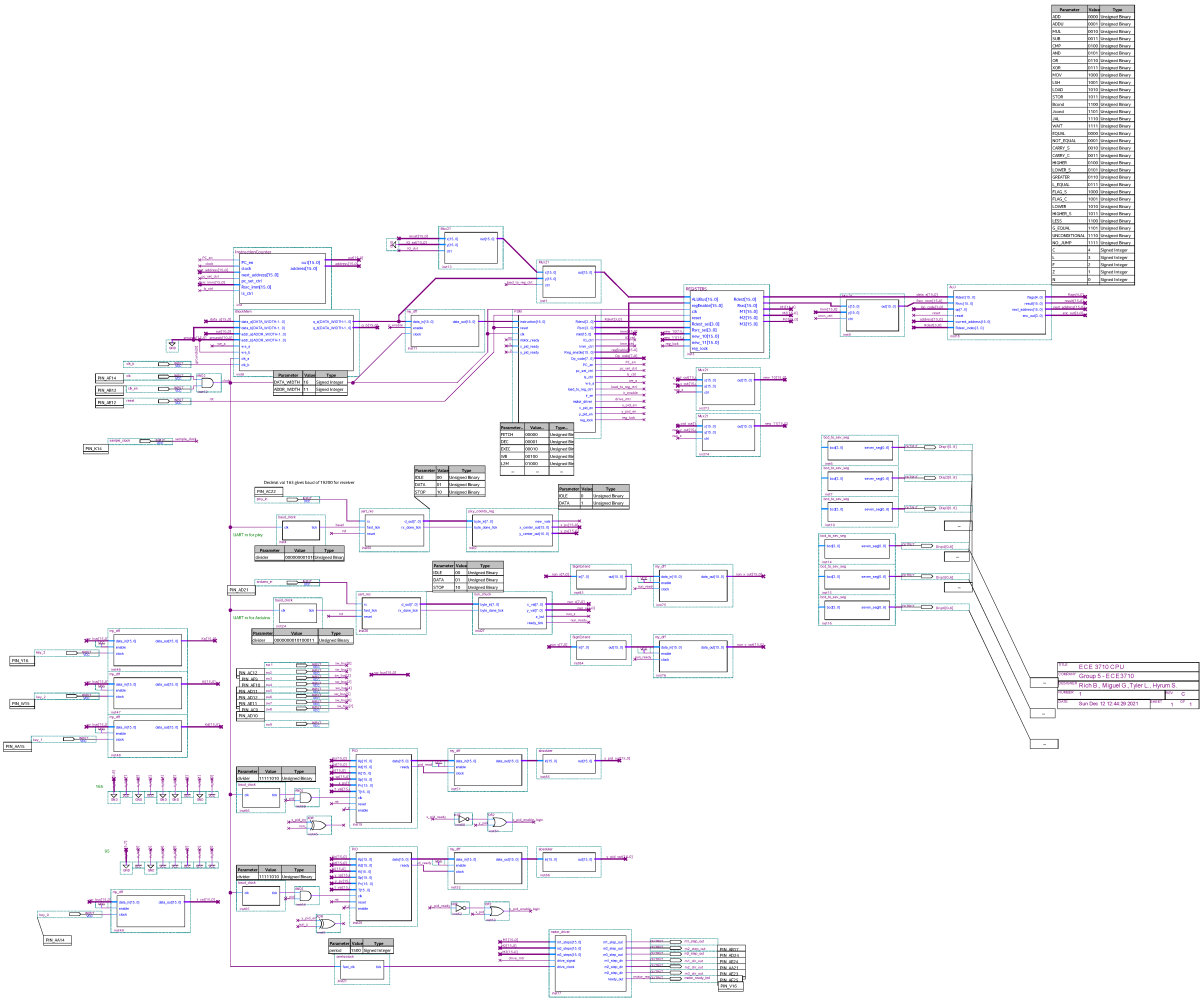

Figure 1: Overall design of the CPU

2 CPU

In our last report we went into great depth about our CPUs datapath, FSM, ALU, register files, and memory. And gave an overview of how the baseline ISA was implemented on our CPU. In this report we will focus more on the augmentations we made to the CPU for our final project.
Our CPU did not require a large amount of augmentation. The datapath had to be modified to include a PID module, motor driver modules, and UART transmit and receive modules. We added control signals for those new modules, as well as additional states to the FSM to update control signals for those modules. We also updated previous parts of the CPU as needed (like adding the ability to lock registers to prevent race conditions). The augmentations that we made to the CPU are described in more detail in the following sections.

3 Communications/Interfaces

3.1 Pixycam Interface

The Pixycam is able to interface in many different ways, however we chose UART as it was the most simple and a UART module could be used elsewhere in our CPU. This required building a UART receiver module, the code for which can be found in Appendix A titled uart_rec.v. The uart_rec module follows the UART protocol, expecting 1 start bit, 1 stop bit, no parity bits, and 8 data bits. The module samples at 8 times the baud rate of the actual transmission to allow it to sample near the middle of each bit. This clock division is done in the baud_clock.v module, which can also be found in Appendix A. The receiver waits in an idle state until the incoming data goes low, aligns itself to the middle of that low wave, then starts counting 8 fast ticks, samples on that tick, and repeats 8 times for the data, until it finally receives the stop bit. This data is outputted on a bus and a ready tick is sent so the data can be stored elsewhere. We are able to receive one byte at a time, however the ordering of those bytes is different for each application. For the Pixycam, the exact documentation can be found at https://docs.pixycam.com/wiki/doku.php?id=wiki:v1:porting_guide, but the table showing the packet ordering can be found below.

Bytes	16-bit word	Description
0, 1	y	sync: 0xaa55=normal object, 0xaa56=color code object
2, 3	y	checksum (sum of all 16-bit words 2-6, that is, bytes 4-13)
4, 5	y	signature number
6, 7	y	x center of object
8, 9	y	y center of object
10, 11	y	width of object
12, 13	y	height of object

Figure 2: Block format of a Pixycam packet to be received over UART

Using this table, we construct our entire block to receive data from the Pixycam and store the last x and y position of the ball (bytes 6-9 in the packet). These positions are then routed to the PID as the current process values so new offsets can be computed.

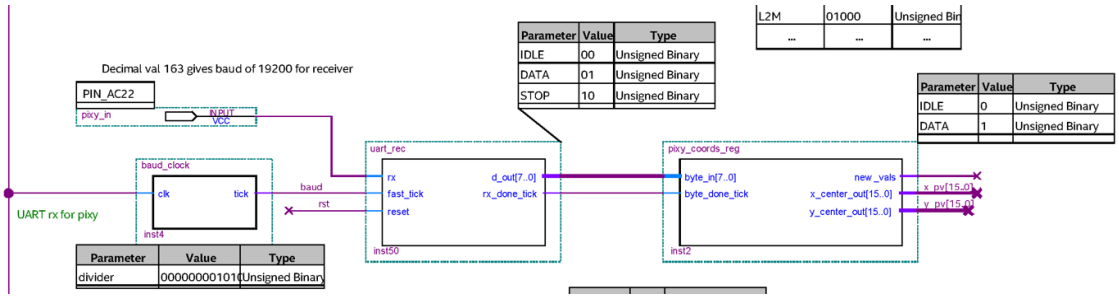

Figure 3: Design used to capture Pixycam data

3.2 Nunchuk Interface

The communication between the Nunchuk control and the CPU is handled in two stages. One stage involves passing the I2C data to the Arduino Nano to capture the inputs from the button as well as the joystick. The second stage involves using the UART receiver module created for communication with the Pixycam to capture data sent from the Arduino to the FPGA. This was done because the I2C protocol for capturing the inputs is already completed and a library for this exists on the Arduino, and the UART interface created for the Pixycam already functioned as intended. This minimized the amount of work necessary for implementing the controls interface as we would not have to implement another communications protocol specifically for the inputs. Since the values obtained from the remote could be changing in the middle of a calculation, we prevent any issues by locking the values at the output of the UART capture shown below by first extending the sign bit of the value and storing it into a DFF.

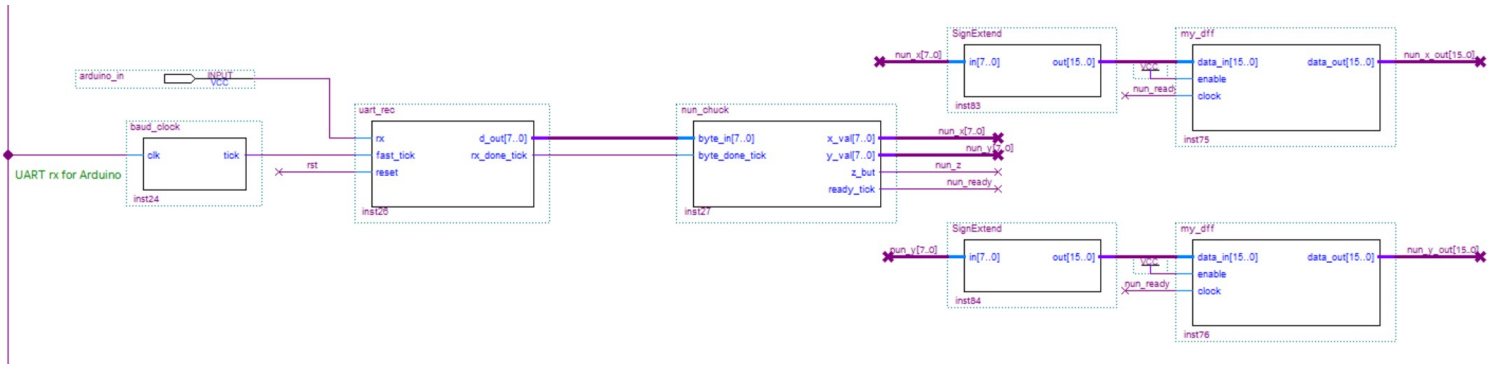

Figure 4: Nunchuk interface for data capture and storage

3.2.1 Arduino

The code necessary for implementing the connection on the Arduino side is fairly simple and can be referenced in Appendix A titled Nunchuk_interface.ino. We instantiate a Serial communication and

choose a baud rate. After that, we initialize the connection between the Nunchuk using the Nunchuk.h interface created by Robert Eisele [1]. The data from the remote is captured by the Arduino and sent using the Serial connection instantiated at the start. The data is transmitted with the UART protocol meaning we capture a single byte at a time from the remote. This is perfect as the data sent for all the inputs will be contained within a single byte for each one.

Input	Size	Order
Joystick-X	8-bit 2's Complement	0
Joystick-Y	8-bit 2's Complement	1
Z-Button	1-bit Unsigned	2

3.2.2 FPGA UART Capture - Nunchuk

Once the data has been captured on the Arduino, we check to see whether or not the remote is currently idle by checking whether or not the values coming in are still the same as they were the last time we transmitted data to the FPGA. This minimizes the number of times the code is blasting the FPGA with more data and gives enough time to pass for the order of the captures to be retained. The code for this capture can be found in appendix A titled nun_chuck.v.

Though the value of the Z-button is a single bit, the information is still passed onto us in the form of a full byte of data. Because of this, we need to make sure to capture only the least significant bit in the transmission, in the others, the bits map directly to the captured data. Since the two values X and Y need to be captured only when the data is ready and sign extended to the size of the internal registers, the DFF modules used to store the data are always enabled and clocked on the ready tick signal from the capture module. This ensures the values are read when the final value, the single bit from the Z-button, has been locked and is ready to be read. That time allows for the other two values to be sign extended and ready to store.

3.2.3 Known bugs

There were a few occasions when something would cause the Nunchuk from being able to register the correct values and seemed like the values had garbage data. This led to some moments when the controls seemed to stutter. I believe this was due to the simple nature of the capture logic. In the future, this could be changed to be a bit more robust by adding a start and stop bit to partition the incoming data. Similar to the Pixycam interface, the use of a start bit can act as a synchronization point for the incoming data. In this case, we are only capturing the data sent as it is received from the remote by the Arduino.
Implementing a more robust version of the ino file in which the data is transmitted between known sync values would help to ensure the remote will always capture the data in the specified order.

4 Hardware

4.1 The Plan and Assembly

The schematic for the build can be seen below. It consists of wood platforms connected by metal rods and 3-D printed parts. The platform in the middle is driven by 3 stepper motors attached to belts and slide rods. The platform will be able to tilt in any way as a function of the 3 platform points that are driven by the motors. Most of the electronics will be mounded underneath with a hole in the platform to allow for wires to pass through.

Figure 5: CAD model of the overall design

The structure was cut, printed and built. The electronics necessary for most of the system were mounted underneath, with the FGPA up top. Below is a picture showing all of the wiring underneath the platform. The electronics will be explained in further detail in the next section. The platform moves as we expected, although we had to add magnets to the balls that hold the platform up and metal rods in the platform that were magnetic in order to keep the platform from bouncing when moving up and down.

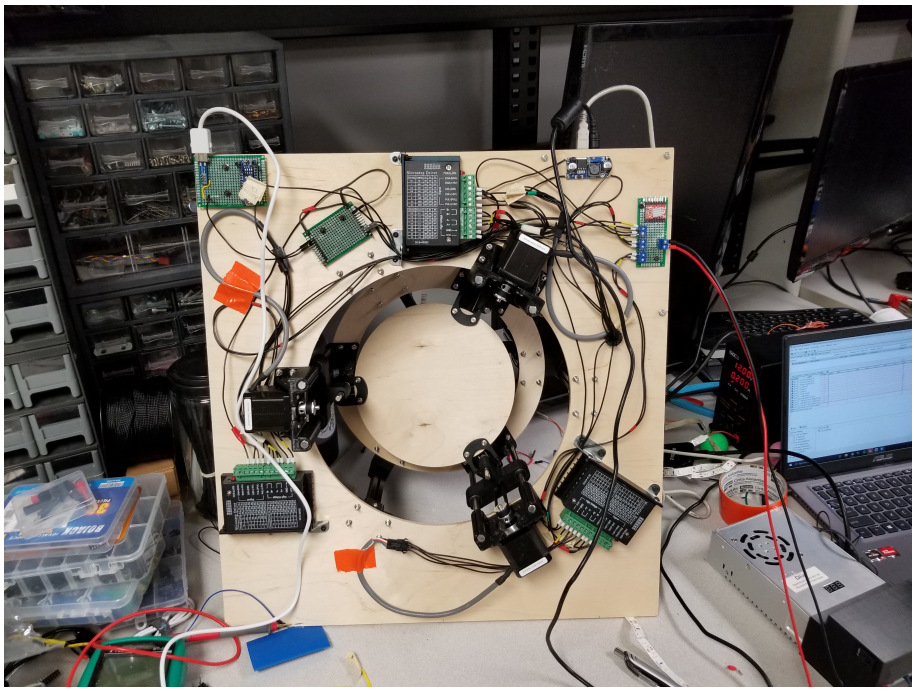

Figure 6: Electronics underneath the platform

4.2 The electronics

Accounting for the devices shown in figure 6, the motors each have a driver that provides a way to control the steps and ultimately movement of the platform. Each of these requires power delivery and some way to split that from the main supply. In the top left corner of Fig. 6, you can see the board that connects the Arduino to the Nunchuk controller. In order to illuminate the platform area to provide a constant source of light for the Pixycam, We placed an LED strip along the edge by the camera. This device also needed power delivery to ensure it functioned when the system powered on. A buck converter was used for this purpose that could step down the necessary voltage from $36V$ down to the $12V$ required by the LED's.

Although it turned out to be unnecessary in the end, a set of fans were obtained to keep the motors cool during operation in the case that they began to heat up. These fans would run at $24V$ provided by a second buck converter and provide enough airflow to ensure the parts stayed within their proper operating temperatures. Knowing the build was going to need a supply of 36 volts at a minimum to drive the steppers efficiently, the DROK AC $110V - 220V$ to DC $0V - 48V$ was a perfect fit. This allowed for a bit of headroom in the case we needed to drive the voltage up to achieve higher torque output. This turned out to be a non issue in the end and the voltage was kept at a steady $36.5V$. A full list of electronics in the build can be found in appendix B.

5 PID Controller

5.1 What is a PID controller

PID stands for **P**roportional **I**ntegral **D**erivative. It is a closed loop control algorithm that takes each of the operations for which it is named into account, in order to direct a given signal to a provided set point. Each operation has an impact on the signal, accounting for noise, drift, and other variables to provide a much more accurate direction to the signal than any one operation alone.

5.2 Implementation

The equation for a PID controller in the continuous time domain is given in (1) where K_p, K_i, and K_d represent tuning constants for the proportional, integral, and derivative operations respectively, and $e(t)$ is the difference between the set point and the current process value. Taking the discrete transformation yields us with an equation (2) that can be implemented with logical arithmetic operations. A final tuning parameter t is added to prevent integral windup. Integral windup is a natural side effect of the PID equation. The error is continuously summed so long as the controller is active which can lead to a situation where the physical limitations of the system being controlled make it impossible to accurately respond to the controller's output. In this scenario the controller continues to read an error value from the system, as the system is unable to correct itself given the controller's response, and the integral value continues to grow. This creates a feedback loop that, if not rectified, will lead to an uncontrolled system. Thus a switch is added using a transformation heaviside function, to reset the integral value after a number of cycles given by t. This results in equation (3). This equation is Designed and synthesized in verilog, without the use of a pre-designed PID library, the controller features two sixteen bit adders, and one sixteen bit multiplier. The adders and multipliers are synthesized in parallel to the ALU, and are implemented using Intel's LLP IP cores. The final block diagram is given in figure 8.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de}{dt} \tag{1}$$

$$u[n] = K_p e[n] + K_i \sum_{j=0}^{t} e[j] + K_d * e[n-1] \tag{2}$$

$$u[n] = K_p e[n] + (H[n+t] - H[n-t])\, K_i \sum_{j=0}^{t} e[j] + K_d * e[n-1] \tag{3}$$

5.3 Usage

The PID controller is at the heart of the design. It receives continuous inputs from the PixyCam, operates on those values and outputs offsets to the CPU of how to get the ball back to the center of the platform. This is happening in a continuous loop in parallel to the CPU. The outputs are stored in a register that is locked during read and during write to prevent race conditions.

5.4 Tuning

From (1) and (2) it can be seen that the outcome of the PID controller depends on the values of the tuning parameters K_p, K_i, and K_d. These parameters control how much each operation influences the final outcome. The controller may be tuned in different ways, and it is ultimately the proportion of the three values relative to one another that determines their impact on the final outcome, so a hard and fast rule about what each parameter does is difficult to define. Thus instead it shall suffice to describe the process followed by the authors.
The first adjustment made is to the proportional value K_p. In this implementation, it is a proportional gain. The value is increased until the system is oscillating across the set point. In this configuration, the K_p value represents the immediate impulse of the controller. The derivative gain K_d is incremented next until the system reaches a critically damped response. The derivative in this configuration provides an offset to the proportional gain, diminishing the controller's overall response. Finally the integral gain K_i is incremented until the system is responsive once more. The t value is adjusted until the system is correcting the position of the ball correctly within an acceptable number of oscillations. Thus the integral gain adds sensitivity back to the system and allows the system to account for steady state errors.

6 Software

6.1 Planning and Analysis

The first thing we had to do with the software was figure out exactly what we needed it to do. Since our projects was so heavily hardware oriented, we weren't sure what role the software would play until we were well into the project. Once we had the platform built out, and the motor drivers working correctly, we were able to determine our software requirements. We decided that the software would receive two values from the PID, an x and y offset, and convert those to a number of steps, as well as a direction(indicated by the sign of the number of steps), for the motor drivers to properly drive the motors.

6.2 Software Design

Now that we knew what the software had to do, we just needed to figure how to write software that would do it. This at first seemed a daunting task. How do you convert a two variable system to a three variable system? After watching a lot of YouTube videos with complex math, we realized that it was even more complex. We weren't trying to convert into a traditional three variable system with x, y, and z axes, because all the motors sat in a plane. We didn't have one motor controlling the x-axis, one controlling the y-axis, and one controlling the z-axis. At this point we didn't even know what kind of math to look at to figure out how to make the conversion. Probably some extremely complicated blend of geometry and trigonometry.
We decided to just sit down and think about it generally, instead of worrying about implementation details. After drawing a few pictures, and thinking about where the motors would be. We realized we could constrain the system to make our lives much easier. Since all the motors were spaced evenly apart, we could think of the platform as an equilateral triangle with a motor attached at each point(M1, M2, and M3), then overlay an x y axis on top of the triangle with the y-axis splitting the triangle cleanly in half, as shown in the figure below.

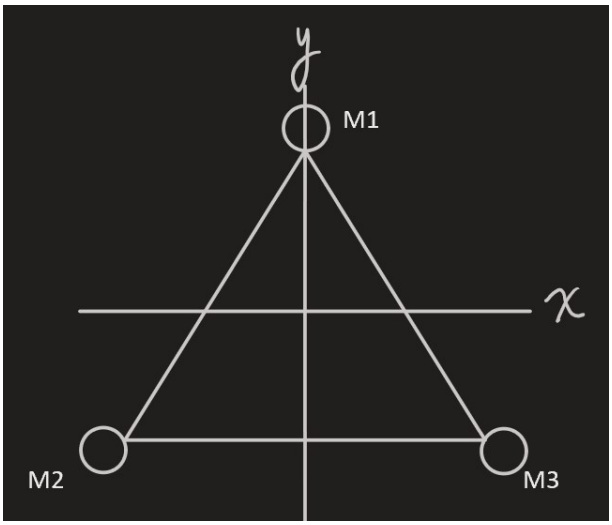

Figure 7: Abstraction of Balancing Platform

Once we abstracted the platform like this. We saw that M1 only ever had to move for changes along the y axis, and M2 and M3 would always move equal distances no matter what. If we were tilting the platform along the y-axis, M2 and M3 would move the same distance in the same direction, if we were tilting the platform along the x-axis, M2 and M3 would move the same distance in opposite directions.

At this point, very simple math could be used to control the platform. Any y offsets received from the PID could be scaled to be the number of steps M1 needed to travel up or down, M2 and M3 would travel the same number of steps in the opposite direction. For x offsets, we only had to worry about M2 and M3. Again, the x offset received would be scaled and tell M2 how far to move up or down, M3 would move the same number of steps in the opposite direction. Once the steps were all added together it would produced a mixed tilt in the y and x direction we intended.

$$M1+ = \text{y\_offset} * \text{scaling\_factor} \tag{4}$$
$$M2- = \text{y\_offset} * \text{scaling\_factor} \tag{5}$$
$$M3- = \text{y\_offset} * \text{scaling\_factor} \tag{6}$$

6.3 Implementation

All of the hard work planning out and designing our software was done. Implementing it was easy due to our planning and the constraints we had set. While writing the software we realized we also needed a way to keep track of the current number of steps each motor had taken in either direction, so that we could keep the platform in the middle of the rails, and also to keep the math correct when responding to the offsets from the PID. For example, say the PID gave us offsets of 0 for x and y because the ball had made it back to the middle. We need to know if the platform is still tilted at this point so we can move it back to a flat position. Beyond that change, our software stayed mostly the same as we had

planned it out.

6.4 Debugging

Miguel learned early on about the Quartus Signal Tap Logic Analyzer, which allows you to look at values in hardware during runtime. Using this, simulation, and external logic analyzers, we were able to debug appropriately. Trying to debug the complex systems we had built required breaking down the problem, which also allowed us to understand the systems more precisely.

Our software had one major bug. It tried to use the registers where the PID would store offsets at multiple times in the software, and the PID sometimes updated the registers before all the calculations were complete. We essentially had a race condition in our code. To fix this we implemented a locking command in our Assembly. We included a special state in our FSM that would enable then disable a flip-flop(which we will call our locking flip-flop) with an inverted feedback loop. The value stored in the locking flip-flop was used as the enable signal for flip-flops in front of our dedicated PID registers, so that when the signal was low those registers would not update, 'locking' the values in the registers. When everything first powers up, the value stored in our locking flip-flop is high, when we start using the values from the PID registers, we first run the command to invert the signal in our locking flip-flop. Once the value is low and our PID registers are locked, we used the values in them until we no longer needed them, then run the assembly command to again invert the signal in our locking flip-flop, thus unlocking the PID registers.

7 Assembler

7.1 An Overview

The assembler, built in python, reads in a file of assembly and writes a machine code file. It can handle flags and has a built in jump pseudo command. In our case we designed a custom version of assembly. It still follows all the same rules, and the commands are mostly the same, it just treats literals and register names differently. Instead of converting the assembly into 1's and 0's, it converts it into hex since we used the readmemh command when setting up our memory in Verilog.

7.2 Pseudo Commands and Tags

Our assembler has a jump pseudo command(JMP ¡tag¿) that will jump to the line with the tag in it. As the assembler loops through the lines of assembly, it keeps a record of what part of memory the current line will be stored in, so that when it encounters a tag, it can store the value specifying the current memory location in a register to jump to later. Then when it encounters the JMP command, it figures out which register holds the value associated with the tag to jump to and inserts the jal command with the proper register.

7.3 Assembler

The assembler just loops through the lines in the input file. When it encounters comments it removes them, if a tag or pseudo command is found it handles it as described in the previous section, and otherwise just converts the assembly commands into hex machine code for our FPGA to run. Not

all commands are currently built into the assembler, just the ones we needed for our project, but the assembler is built in such a way that it is easy to add additional commands if needed.

7.4 New Commands

We added two new commands to our ISA, one that locked certain registers, and one that told the motor module when to drive the motors. Both of these commands were created by adding new states in the FSM that updated control signals either in our register file or motor driver module.

8 Lessons Learned

8.1 Rich Baird

I being the only individual on the team having exposure to a PID, and that only a theoretical one, we thought it prudent to find a freely available core instead of designing our own. We found only one such core on http://www.opencores.org. and settled on using this core early on in our project so that we could focus on other issues. The core was functional, but was also tightly coupled to a hardware interface, Wishbone, which none of us were familiar with. We did not discover this until the integration phase of the project, with under a month until the deadline. After several weeks of trying to understand how this interface was meant to function, and trying to write a module to interface with it, we determined instead to design our own. Having the most experience, I took the lead here and was able to design in just two days, what took us weeks to understand with the pre-designed code. In retrospect, we would be better served to look into options and alternatives early, and not assume that a single path is the right one. A few days of examining options for the PID, and understanding the underlying math and design, could have saved us weeks in the end, which we could have used to better tune the system for the final presentation.

8.2 Miguel Gomez

Although it seemed like a great idea at the time, the decision to implement our entire electrical design from scratch ended up being a difficult time. The necessary circuitry to control stepper motors, and keep them from melting/burning/exploding/failing in some manner was a difficult one. We made a set of current limiting circuits to handle the possible 10A current that could propagate from the power supply at startup. This was limited to a max of 3.3A to each driver, and the circuit keeps the voltage at 36 while limiting its current output to 1.28A. These worked well but ultimately could not be utilized since the drivers at hand were damaged in the testing phases.

Between simple mistakes and a few incorrect calculations, the testing and implementation phase of the motor driving circuits was an exercise in patience and persistence for us all. Due to the time constraints and inability to source more of the drivers in time, we decided to go for a set that we could control and had all the necessary precautionary limits available at the outset.

8.3 Tyler Liddell

I have had little hands-on experience with electronics so this project was very fun to learn more about and be able to apply the theory I know into practical applications. I spent a considerable amount of

time kinking out silly bugs in Verilog whether that was in our CPU system or the UART receiver. Because of this, I feel I have a much better understanding of software and hardware systems, and how they interface with each other. We had a running joke in our group that things were always broken because of "just Quartus/Verilog things" but I enjoy the level of complexity and control you can create over a system. I was also impressed by the breadth of different technologies we brought to the table as a group, as I learned about many of these from the others. I'll also echo what Hyrum says below about the culmination of ideas and concepts that came with this class. The CE program has felt fairly jumpy from topics and ideas, but this class and 3700 really put it all together into a coherent picture that was very satisfying.

8.4 Hyrum Saunders

I learned a significant amount about hardware from everyone else on the team. Up until now I've focused mostly on software, and the only hardware experience I've had came from extremely short, somewhat ineffective labs in early electronics classes. Being able to work on a complex hardware system with lots of moving parts(literally and metaphorically) was a really great learning experience. I got to see how important it is to break a project into parts as small as you can, and then implement and test those tiny little parts of the overall design extensively before bringing it all together.

I also learned a great deal about CPU's this semester(designing one and then using the one you designed to implement something tends to teach you quite a bit about them :) In all seriousness though, I felt like this class was a culmination of everything I've learned in previous semesters. I learned in early electronics classes about transistors, in 3810 and 3700 I learned how to use CMOS pull up and pull down networks to build logic gates, and how to use logic gates to build simple digital circuits like adders, subtractors, and even ALU's. But it seemed at times that everything I was learning was disjointed. This class remedied that and brought together everything I learned in previous semesters so it all made a lot more sense. It was a very cool experience when I realized I knew how a CPU worked from the transistor level up.

Finally, I learned a great deal about problem solving in this class. I cannot begin to count the number of problems we ran into while designing our CPU and building our final project. Every class I take improves my problem solving skills, but I've never had such an exceptional team as the one I had this semester and I learned a great deal about solving problems as a team, and how to divide tasks and problems so that we can solve problems faster, and better then we would alone.

9 Group Member's Responsibilities and Accomplishments

9.1 Rich Baird

- Designed and synthesized the PID
- Contributed much of the core logic behind the FSM
- Assisted as able with the electronic and hardware design and assembly.

9.2 Miguel Gomez

- Took on software we were not exposed to that is part of the Quartus suite (Signal Tap Logic Analyzer). Learned to use it so we could employ it during the hardware debugging process.
- Spent time understanding the circuits needed to complete the build and was in charge of putting these circuits together during testing.
- Brought the knowledge of creating symbol files and drop in components in Quartus for creating BDF files of the overall FPGA design–minimizing time working on coding to only handling the logic of the implementation and not connections.
- Prototyping the brackets used to hold the platform and 3D printed the models to arrive at a solution for attaching belts.
- ~~Was~~ Still is the ~~Latex~~ LaTeXmaster for the team

9.3 Tyler Liddell

- Wrote the UART receiver and transmitter module used to interface with the Pixy and Arduino.
- Installed many of the electronics and wires used in the final design of the system.
- Explored options such as OpenCV running in a Docker container early on in the project before settling on Pixy.
- Helped with overall design and structure of the hardware and software systems.

9.4 Hyrum Saunders

- Reduced a daunting, complex problem of converting x y offsets to motor movements into a simple idea easily implemented in software
- Designed and built the teams Assembler, including the ability to jump to tags
- Helped implement new states in the FSM for motor control and register locking

10 Conclusions and Future Work

Overall, the project was a very challenging, yet very satisfying one. We spent a considerable amount of time to achieve what we did, and we are happy with it. The overall design for the system seems very sound, however in order to tune the system further would require more time and hardware tweaking. As a way to keep the project alive and worked on, we can use it as a model to show off at I.E.E.E. events and recruiting session at the University. The simplicity of the parts and cad designs allows for more iterative changes to be made over time and easily put them to the test.

The amount we have learned over the course of our academic journey has proven difficult at times for us all, but nothing can come close to the feeling of elation that washes over you when you get something to work. Especially when that something is a system as complex as the one we decided to tackle.

Seeing this project come up from an idea on paper to a completed, functioning(almost) implementation of that idea was a fantastic experience and a privilege. I am excited to see how the project turns out after it's 5th, or nth, revision; and I believe I speak for us all when I say we look forward to starting on the next one.

References

[1] E. Robert, "Using a wii nunchuk with arduino • computer science and machine learning." [Online]. Available: https://www.xarg.org/2016/12/using-a-wii-nunchuk-with-arduino/

11 Appendix A

Module 1: uart_rec.v

```
module uart_rec(input rx, input fast_tick, input reset, output reg [7:0] d_out, output
    reg rx_done_tick);
    parameter IDLE = 2'b00,
                DATA = 2'b01,
                STOP = 2'b10;

    reg [2:0] counter = 3'd0;
    reg [1:0] state = IDLE;
    reg [3:0] bit_count = 4'd0;
    reg [7:0] d_to_go_out;

    always @(posedge fast_tick, negedge reset) begin
        if(!reset) begin
            state <= IDLE;
            counter <= 4'd0;
            rx_done_tick <= 1;
        end
        else begin
            case(state)
                IDLE:
                begin
                    //We are in the middle of the beginning
                    if(rx == 1'b0 && counter == 3'd4) begin
                        rx_done_tick <= 1'b0;
                        state <= DATA;
                        d_to_go_out <= 8'd0;
                        counter <= 3'd0;
                        bit_count <= 4'd0;
                    end
                    else if(rx == 1'b0) begin
                        counter <= counter + 1'b1;
                    end
                end
                DATA:
                    if(counter == 3'd7) begin
                        state <= DATA;
                        //Add bit as most significant bit
                        d_to_go_out <= {rx, d_to_go_out[7:1]};
                        bit_count <= bit_count + 1'b1;
                        counter <= 4'd0;
                        if(bit_count == 4'd7) begin
                            state <= STOP;
                            bit_count <= 4'd0;
                            rx_done_tick <= 1'b0;
                            d_out <= { rx, d_to_go_out[7:1] };
                        end
                    end
                    else begin
                        counter <= counter + 1'b1;
                    end
                STOP:
                    if(counter == 3'd7) begin
                        rx_done_tick <= 1'b1;
                        state <= IDLE;
                        counter <= 3'd0;
                    end
                    else begin
                        counter <= counter + 1'b1;
                    end
            endcase
        end
    end
endmodule
```

Module 2: baud_clock.v

```
module baud_clock
#(parameter [15:0] divider = 16'd163)
(input clk, output reg tick);
    reg [15:0] counter = 16'd0;

    always @(posedge clk) begin
        counter <= counter + 1;
        if (counter == divider) begin
            tick <= ~tick;
            counter <= 8'd0;
        end
    end


endmodule
```

Module 3: Nunchuk_interface.ino

```
#include <Wire.h>
#include "Nunchuk.h"

bool sent_0;
void setup() {
    Serial.begin(19200);
    Wire.begin();
    sent_0 = false;
     // Change TWI speed for nuchuk, which uses Fast-TWI (400kHz)
    Wire.setClock(400000);
    nunchuk_init();
}

void loop() {

    if (nunchuk_read()) {
        int8_t numx = nunchuk_joystickX();
        int8_t numy = nunchuk_joystickY();
        int8_t butz = nunchuk_buttonZ();

      if (numx == -1 && numy == 0 && butz == 0 && !sent_0)
         {
             Serial.write(numx);
             Serial.write(numy);
             Serial.write(butz);
             sent_0 = true;

          }
          else if(numx == -1 && numy == 0 && butz == 0 && sent_0)
          {}
          else
          {
             Serial.write(numx);
             Serial.write(numy);
             Serial.write(butz);
             sent_0 = false;
          }
    }
}
```

Module 4: nun_chuck.v

```
module nun_chuck(input [7:0] byte_in, input byte_done_tick, output reg [7:0] x_val,
    output reg [7:0] y_val, output reg z_but, output reg ready_tick);

    reg [1:0] counter = 2'd0;

    always @(posedge byte_done_tick) begin
            ready_tick <= 1'd0;
            counter <= counter + 1'd1;
            if(counter == 2'd0)
                x_val <= byte_in;
                else if (counter == 2'd1)
                y_val <= byte_in;
                else if (counter == 2'd2) begin
                z_but <= byte_in[0];
                ready_tick <= 1'b1;
                counter <= 2'd0;
                end

    end

endmodule
```

Module 5: PID.v

```
module PID(input wire signed [15:0] Kp, Kd, Ki, Sp, Pv, T, input wire clk, reset, enable
    , output reg signed [15:0] data, output reg ready);
reg signed [15:0] e_n, e_nPrev, kpd, derivative, sigma, sigma_n, kpde, diffA, diffB,
    prodA, prodB, sumA, sumB, prodA1, prodB1;
reg [3:0] state, stall;
wire uf, of;
reg [15:0] counter;
wire signed [15:0] difference, sum;
wire signed [31:0]product, product1;
reg add_sub, add_sub1;
wire borrow_out, carry_out;
reg borrow_in, carry_in, sigma_cin;
PIDAdder subtractor (
          .add_sub(1'b0),
          .cin(1'b1),
          .dataa(diffA),
          .datab(diffB),
          .cout(borrow_out),
          .overflow(uf),
          .result(difference)
        );
PIDAdder adder (
          .add_sub(1'b1),
          .cin(carry_in),
          .dataa(sumA),
          .datab(sumB),
          .cout(carry_out),
          .overflow(of),
          .result(sum)
        );
PIDMultiplier multiplier(.dataa(prodA), .datab(prodB), .result(product));
PIDMultiplier multiplier1(.dataa(prodA1), .datab(prodB1), .result(product1));
```

```
always@(posedge clk or negedge reset)
  begin
    if(!reset)
      begin
        e_n <= 0;
        e_nPrev <= 0;
        kpd <= 0;
        derivative <= 0;
        sigma <= 0;
        diffA <= 0;
        diffB <= 0;
        state <= 0;
        ready <= 0;
        data <= 0;
        borrow_in <= 0;
        add_sub <= 1'b0;
        kpde <= 0;
        sigma_n <= 0;
        stall <= 4'd0;
        add_sub1 <= 1'b1;
        carry_in <= 1'b0;
        sumA <= 0;
        sumB <= 0;
        prodA1 <= 0;
        prodB1 <= 0;
        sigma_cin <= 1'b0;
        counter <= 0;
      end
    else if(enable)
      begin
        // time loop reseet
        if(T > 0 && counter == T) begin
          counter <= 0;
          e_nPrev <= 0;
          sigma <= 0;
        end
        else if(T > 0) begin
          counter <= counter + 1;
        end
        else begin
          counter <= 0;
        end
        case(state)
          4'd0:
            begin
              // store e(n - 1)
              e_nPrev <= e_n;
              state <= 4'd1;
              ready <= 1'b0;
              // Begin calculating Sp - Pv
              diffA <= Sp;
              diffB <= Pv;
              // Begin calculating Kp + Kd
              sumA <= Kp;
              sumB <= Kd;
              carry_in <= 0;
            end
```

```
          4'd1:
            begin
              // store e(n)
              e_n <= difference;
              // store kp + kd
              kpd <= sum;
              // begin calculating Kd * e(n - 1)
              prodA <= e_nPrev;
              prodB <= Kd;
              // begin calculating e(n) * (Kp + Kd)
              prodA1 <= sum;
              prodB1 <= difference;
              state <= 4'd2;
              carry_in <= 0;
            end
          4'd2:
            begin
              kpde <= product1[15:0];
              derivative <= product[15:0];
              // begin calculating e(n) - (Kd * e(n - 1))
              sumA <= sigma;
              sumB <= e_n;
              state <= 4'd3;
              carry_in <= sigma_cin;
            end
          4'd3:
            begin
              sigma <= sum;
              prodA <= Ki;
              prodB <= sum;
              state <= 4'd4;
              sigma_cin <= carry_out;
            end
          4'd4:
            begin
              diffA <= product[15:0];
              diffB <= derivative;
              state <= 4'd5;
              carry_in <= 0;
            end
          4'd5:
            begin
              // Begin calculating final value
              sumA <= kpde;
              sumB <= difference;
              state <= 4'd6;
              carry_in <= 0;
            end
          4'd6:
            begin
              // store result and go back to the top
              data <= (sum >>> 3);
              ready <= 1'b1;
              state <= 4'd0;
              carry_in <= 0;
            end
        endcase
      end
    else begin
      ready <= 1'b0;
      state <= 4'd0;
        end
  end
endmodule
```

Appendix B - Supplies Used

12 Electronics

Supplies	Quantity
DROK AC 110V-220V to DC 0-48V	1
STP-MTR 17048 stepper motors	3
Stepper Motor Driver TB6600 4A 9-42V Nema 17	3
PixyCam Smart Vision Sensor	1
Solder breadboad PCB's (Verious sizes)	4
Arduino Nano	1
Buck Converter	2
Wii Nunchuk	1
Cooling Fans	3-4

13 Hardware

Supplies	Quantity
Plywood sheet 20in.x20in.	1
20mmx20mm Vslot	3m
Various 3m screw lengths, washers, and nuts	≈80+ sets
Various 3m Nylon screw lengths, washers, and nuts	≈10+ sets
3D printing filament, PETG or PLA	≈ 1 spool

Tables and Figures

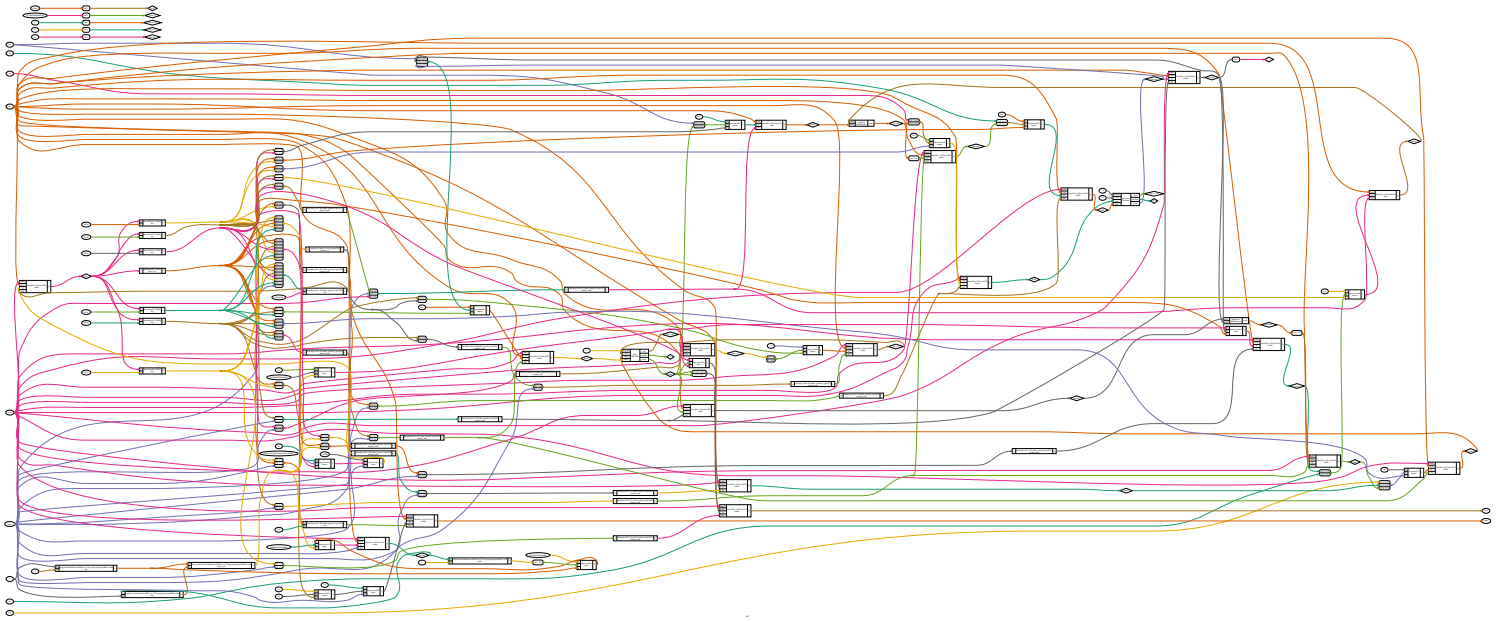

Figure 8: PID Block Diagram