Audio-Controlled Levitator

David Mejorado and Raul Largaespada

Project Introduction

We wanted to implement a project that combined both audio processing and controls, therefore we will be building an audio-controlled levitator. The proposed system would take in audio information via the on board microphone of the Nexys 4DDR. The main frequency of this audio signal will then be extracted via the Fast Fourier Transform (FFT) and be used as the reference signal to a PSD control system. The levitator will consist of a ping-pong ball like object that will be placed inside an acrylic tower. A fan at the bottom of the tower will act as the actuator moving the ball up and down, while a IR sensor placed at the top of the cylinder will provide measurements on the current position of the ball.

The system will operate under two modes: Continuous and Discrete. In continuous mode the ping pong ball will respond continuously to the pitch being detected, while in discrete mode a new reference is only sent in response to a button press. The VGA display will also be used to provide information about the state of the system, including: FFT histogram, real-time tracking of ping bong ball relative to reference point, peak frequency, and visual representation of incoming audio.

1. Audio Processing (David)

1.1 Microphone Interface

The Nexys 4 DDR's on board microphone produces a PDM on the rising edge of a 2.4 Mhz clock. This posed the challenge of taking a 1-bit signal and converting to a bit length that allows for appropriate processing. Overall, this task was the most difficult to overcome and was the biggest time sink in the frequency detection aspect of the project. My initial implementation used an 8-bit accumulator over 255 samples in order to take advantage of the full 8-bit range and a final sampling rate of 8.7 kHz. However, when I proceeded to verify the functionality of this by outputting the values directly to the audio out via a PWM module, the quality of sound was severely distorted and noisy. I then attempted increasing the bit size and increasing the final sampling rate by interleaving multiple accumulators of varying sizes. After achieving little improvement in audio quality, I changed my initial design to a popular implementation of a moving average with a decimation factor done in hardware commonly referred to as a Cascaded Integrator Comb filter.

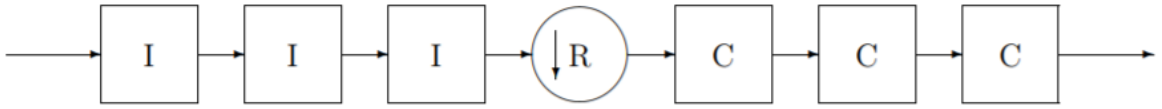

Figure 1. Block diagram of 3 stage CIC filter with decimation ration R.

Furthermore, the literature described that pairing a CIC with a low pass filter gave the best results. The design that was ultimately used in the final version consisted of the following: a 5-stage CIC with a decimation ratio of 15, followed by a 16 cycle accumulator that serves as both a low pass filter, method for increasing the bit depth, and to further decimate the signal.

Audio Interface:
(104 MHz)

In order to maintain a clean interface between the raw PDM signal and the valid PCM data that would be used by the FFT IP core this module was implemented. Furthermore, this separation allowed for a thorough isolated testing of the microphone prior to integration with the rest of the system. Within this module, a 2.4 MHz clock was generated with the use of counter as well as an enable signal at the rising edge of each clock cycle. Before being sent to the first stage of processing within the CIC the PDM data was converted into signed 8-bit data with 0 corresponding to -1 and 1 with +1. This was a simple way to increase the bit depth and facilitate the further calculations that needed to be done. This data was then sent to the *CIC* and finally to *OverSampler16* before being output to the next stage of processing.

CIC:
(104 MHz)

This module was designed with a variable decimation ratio. This variability allowed for testing the efficiency of various decimation ratios without the necessity of resynthesizing. The samples were read using an enable signal at 2.4MHz and a counter was used to generate the output frequency.

OverSampler16:
(104 MHZ)

This module was obtained from the FFT Demo provided as an example, and was used as the accumulator mentioned above. This module accumulated over 16 clock cycle providing an effective decrease in sampling rate by a factor of 16. This was also the final stage of audio processing outputting an unsigned 14-bit audio sample at 10kHz.

1.2 Pitch Detection

The ultimate goal of the project was to control the movement of the ping pong ball by varying the pitch being vocalized. In order to do this, we decided to use the FFT IP core provided by Xilinx. As data was outputted by *AudioInterface*, it was first stored in a circular 4096x16 BRAM. We then used the *Bram2FFT* module provided by the FFT Demo to interface with the FFT IP core. The real outputs of the FFT were stored in a second BRAM of identical size, however this BRAM ran at to clocks. The FFT wrote into the BRAM at a 104MHz clock and the data was read with a 65Mhz clock. Both clocks were generated using the Clock Wizard IP core. This data was then sent to *Frequency Detection,* where the values outputted by the FFT were compared to a threshold and if the value exceeded the threshold the associated address was stored. One thing to note is that only the positive values were sent forward to the frequency detection stage, while any negative value was set to zero. This prevented some of the noise from being propagated since a lot of it was contained to negative values

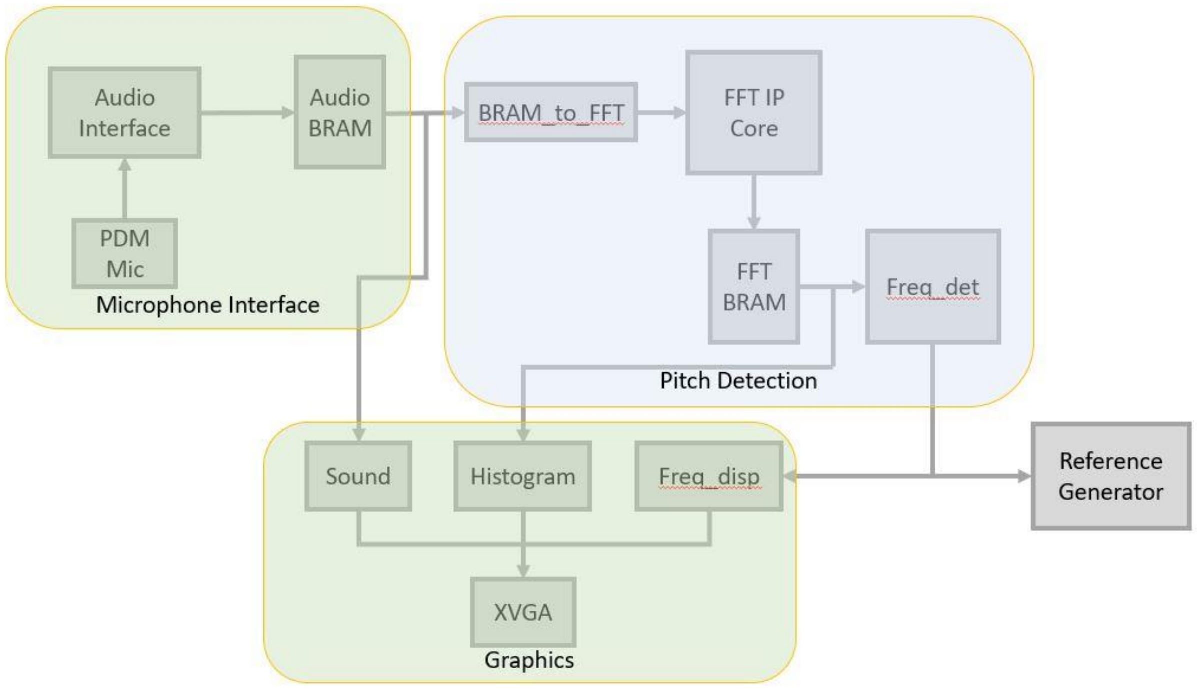

Figure 2. Overview from Microphone to Reference Signal Generator

FFT:
(104MHz)

When picking the parameters for the FFT IP core we took into consideration the frequency resolution, the size of the core, and what our needs would be. In order to obtain high frequency resolution, we chose a 4096 point FFT and used the Pipelined, Streaming I/O option. With these parameters we could avoid the complications of verifying additional enabling signals and have a quicker and continuous response to drive the control. The optional scaling parameter of the FFT was implemented and controlled via the on board switches. This allowed for real time adjustments to the size and sensitivity of the FFT in response to changes in ambient noise levels. The output of this core was a signed 16 bit values at 104 MHz along with its associated address number which corresponds to the bin number of the FFT.

Originally, the method we intended on using the FFT demo block design to perform the FFT. However, the initial attempts did not output a reasonable signal. This was seen in both the Integrated Logic Analyzer (ILA) and the graphic created by the *histogram* module. The FFT outputs were very small and showed no structure even in the presence of a pure tone being played into the mic. Instead of trying to debug the parts of the code that wasn't working it was decided that simply implementing the FFT IP core ourselves would be more time efficient. Therefore, instead of taking the magnitude of the FFT output like the Demo did, we only used the real output of the FFT core. This was determined to be sufficient by playing a pure tone into the microphone and analyzing the output of the FFT with the ILA. By matching the maximum FFT value to the appropriate address number and then performing the arithmetic to retrieve the frequency value in Hz, I was able to verify that the FFT was functioning properly and was sufficient for our application.

Freq_Det:
(65 MHz)

After analyzing the output of the FFT on the ILA it became clear that with some scaling and tuning the most prominent frequency being heard by the microphone could easily be extracted by applying a threshold and storing the address of whichever FFT value last crossed the threshold. One other detail that was also considered when applying the threshold was that the first 50 bins of the FFT were consistently noisy and non-responsive. Therefore, in order to provide stability to the system only bins greater than 50 were compared to the threshold.

The accuracy of this method was further tested by creating a ROM that linked the bin number outputted by the FFT with a frequency in Hz that would then be display on the hex display via the *display_8hex* module. In this manner a variety of pure tones were played with a tone generator app and compared to the value on the display. The output was consistently within 2Hz accuracy for frequency between 0.12 kHz and 1.3kHz.

Bram_to_fft:
(104 MHz)

This module was not altered or repurposed from the original FFT Demo. It contains the logic to communicate with the FFT core to send data at valid moments.

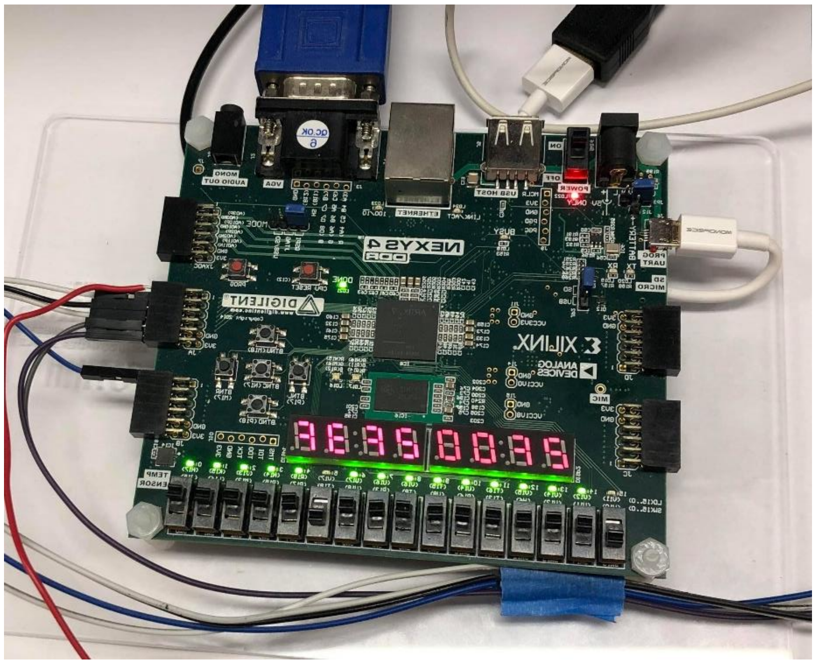

Figure 3. Example Hex display where first 2 digits are the received reference, followed by the sent, and detected reference.

1.3 Thoughts

Although I ultimately arrived at a design that provided usable PCM audio data, I was left unsure how much of it was entirely necessary. Since I did not implement the FFT until after the audio was functional, according to my ears, I do not know if the frequency would have been able to be extracted with some of my earlier iterations. However, on the opposite side of this argument, I felt that there was still much to be done in acquiring a clean audio signal. Resources online pointed towards much more complex techniques that I did not have the time to explore.

2. Reference Signal (David)

During initial design considerations it was brought up that the controller might not respond well to a direct input from the detected frequency due to noise. And furthermore, a frequency in Hz or a bin number would need to be translated to physical description of the position of the ping pong ball. Therefore, the final implementation would output a 6-bit reference signal to the controller in 2 modes: Continuous and Discrete.

In continuous mode a reference signal was continuously sent in response to detected frequencies from 163 Hz to 314 Hz which is approximately one octave. Any frequency below 163Hz output a 0 and any above 314Hz output the max, 63. This range was chosen because it is a range that is easily swept in one breath and would work the best for a demonstration. In discrete mode there are two key differences, the detection range is from 163Hz to 478Hz, and the reference signal corresponding would only be sent on the rising edge of a button press. For ease of use and testing three values were displayed on the hex display. The detected reference signal, the sent signal, and the signal received by the controller.

One last feature is the option for the user to alter between the pitch generated reference signal and a direct switch input. Specifically switch 6 toggled the use of the switches 0 to 5 as the reference signal to the controller.

ReferenceGen:
(65 MHz)
This module contains the arithmetic conversion from frequency to reference. It also houses the two modes of operations through a switch.

3. Graphics (David)

The VGA display can be classified into 3 sections: real-time audio data display, FFT histogram, and system state and dynamics. The top half of the screen is dedicated to displaying the real-time audio signal being generated by microphone interface. This provides a measure to the user of the ambient noise being picked up by the mic and also aided in verifying the functionality of the mic. The bottom of the screen is made up of a red histogram of the FFT output. Depending on the scaling, is how sensitive the FFT histogram will be to sound. When tuned properly only prominent frequencies are visible.

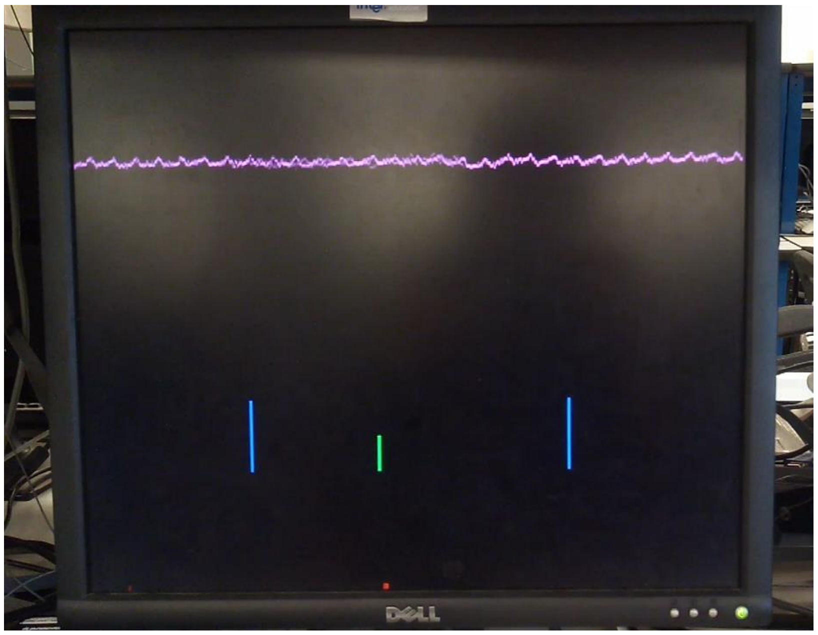

Figure 4. Display in discrete mode.

Each mode displays a different range of frequency range as is evident by the position of the blue bars. The bar to the left corresponds to the lowest height the ping pong ball can go while the right bar corresponds to the highest point. The shorter green bar appears over the last point above the histogram that was greater than the threshold. The relationship of the green bar between the blue bars is indicative of the relative location that that specific frequency is setting. Finally, the purple square above the green bar is indicating the relative position of the ping pong ball. This was done by using the output of the IR sensor as the input to the position of the square. However, this 6-bit output of the IR sensor had to be properly scaled to match the correct location on the screen.

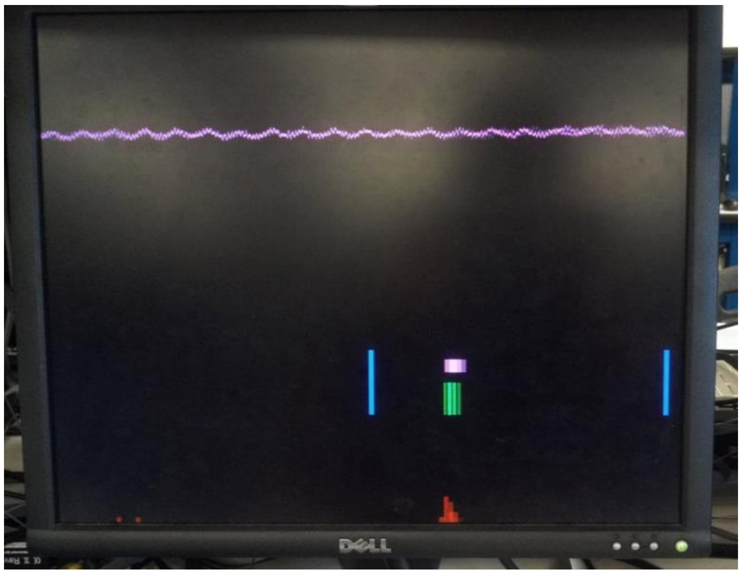

Figure 5. Display in continuous mode.

Sound:
(65 MHz)

This module receives the output of the BRAM storing the incoming audio and scales it to be placed on the upper half of the screen.

Freq_Disp:
(65 Mhz)

This module takes in the detected frequency and performs the appropriate scaling to place the bar in the correct location according to the mode the system is currently in.

Histogram:
(65 Mhz)

This module uses the FFT output to generate a histogram on the VGA display. This was taken from the FFT demo provided and remained unchanged.

3. Implementation of Levitator Hardware and Closed-Loop Discrete PD Control (Raul)

System Overview

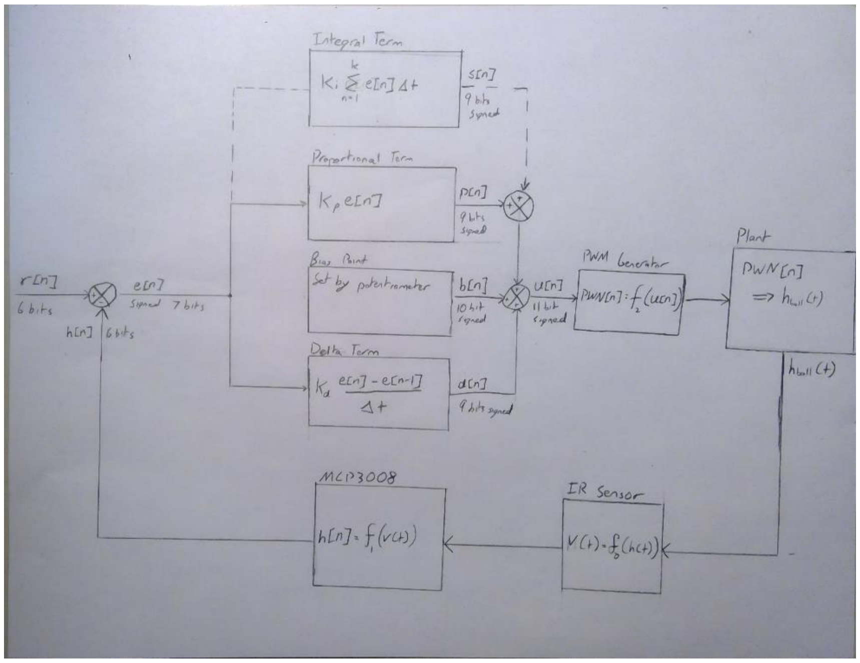

Figure X: block diagram of PD control loop with additional sum term. Reference signal r[n] is generated from audio pitch detection circuitry.

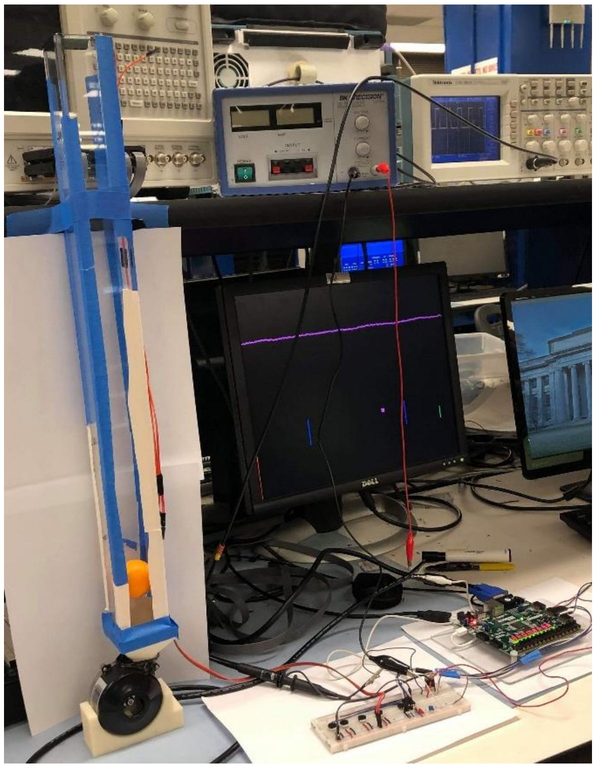

Figure X: The completed system.

This system follows a standard control loop architecture with proportional and delta terms for the discrete PD controller. The height signal h[n] is subtracted from the reference signal r[n] to form the error signal e[n], which serves as the main input to the controller. The proportional and delta terms are generated from the error signal and are summed with an additional constant bias term and a sum term that functioned but was not used in the final implementation. Each term can be tuned using a 10kΩ potentiometer connected to an analog-to-digital converter.

The controller output is then turned into a PWM duty cycle, which is fed into the actuator, a centrifugal fan. The PWM signal controls the fan's connection to ground, rapidly activating and deactivating it such that the fan spins at a particular rate. The fan is powered by a 12V DC power supply. The fan's spinning causes the levitated ping-pong ball to hover or rise. The height of the ball is tracked using an IR analog distance sensor. The output of the sensor is fed into the ADC before being fed back into the controller and subtracted from the reference signal.

Hardware Implementation

Circuitry and Sensor

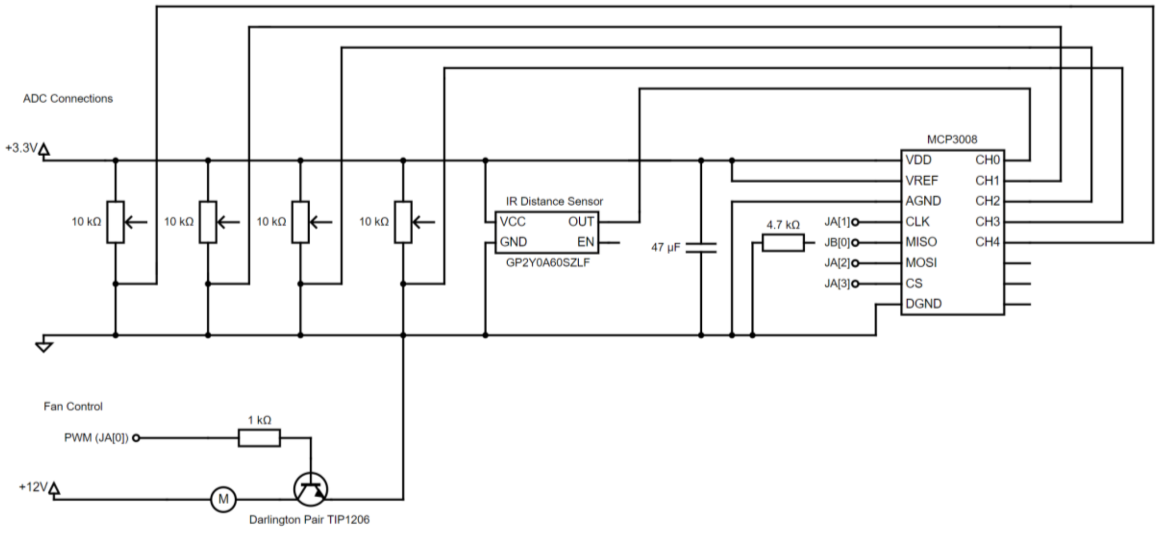

Figure Y: circuit schematic showing the connections to the analog-to-digital converter and the actuator control hardware.

The circuitry was split into two sections connected to a common ground. The upper section lays out the various connections to the MCP3008 analog-to-digital converter. The MCP3008, the Sharp GP2Y0A60SZLF IR analog distance sensor, and the four tuning potentiometers are all powered by a 3.3V rail connected to the Nexys 4 FPGA. The potentiometers follow the standard variable resistor voltage divider architecture with a maximum resistance of 10kΩ, and each is fed to a different input channel on the MCP3008. The IR distance sensor features two pin outputs, the enable pin which is unused, and the OUT pin which outputs the analog voltage reading from the sensor and is connected to channel 0 on the MCP3008.

The MCP3008 V_{DD} and V_{REF} pins are connected to the 3.3 volt rail and the analog ground (AGND) and digital ground (DGND) pins are both connected to the same ground rail. The SPI interface wires are all connected to the FPGA. CLK, MOSI, and CS are connected to FPGA output pins and MISO is connected to an input pin. The MISO pin is connected to ground via a 4.7kΩ pulldown resistor to ensure consistent readings. The 3.3V and ground rails are also connected to a 47µF bypass capacitor.

The fan control part of the circuit is much smaller and consists of a 12V rail connected to the motor which controls the fan. The motor's connection to ground is through a TIP1206 Darlington pair of transistors, and this connection is controlled by a PWM signal from the FPGA running through a 1kΩ resistor.

Plant and Actuators

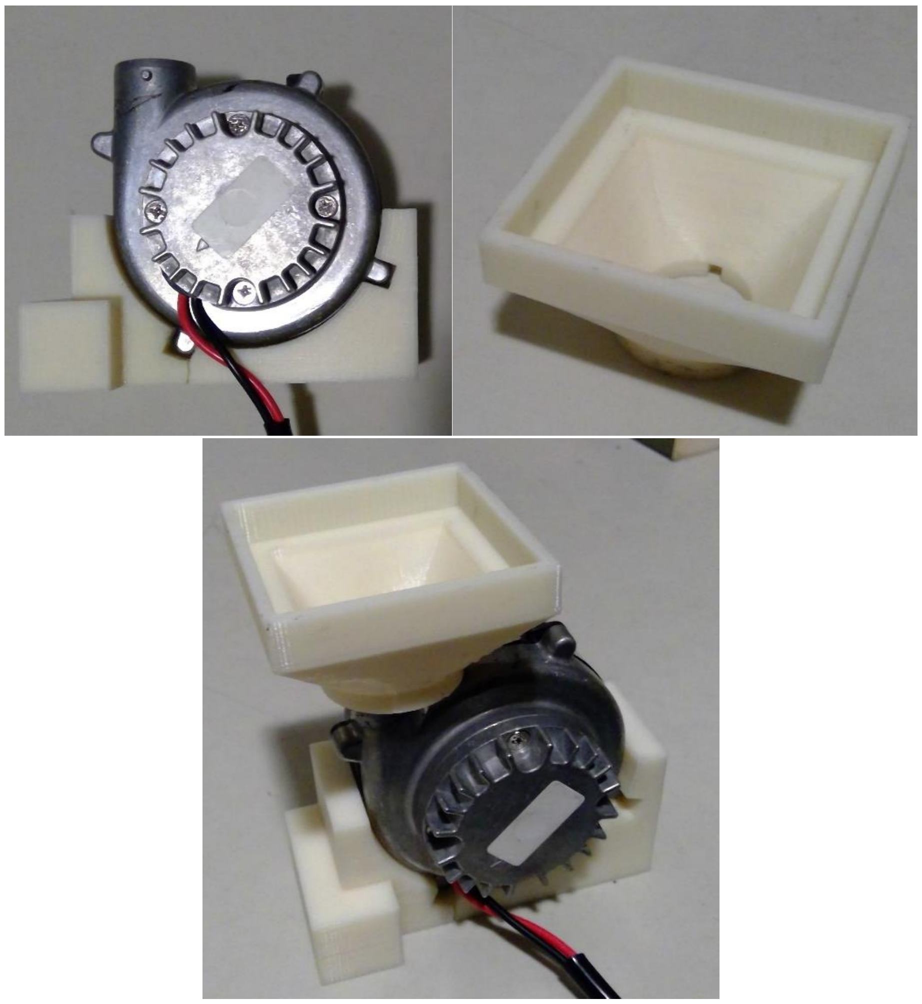
Figure X: The fan slotted into the base part, the fan's connection to the tower, and the fan assembled with both pieces.

The system relied on a 12V centrifugal fan to generate lift for the ping-pong ball. The fan was linked to the levitator tower and fixed such that the lift force was perpendicular to the ground using 3D-printed hardware. The levitator tower itself was constructed from laser-cut acrylic and held together using tape. At the top of the levitator tower the IR analog distance sensor was attached. This was powered using the same 3.3V rail that powered the rest of the circuitry.

The 3D-printed base consisted of a block with a semicircular section cut out to accommodate the fan's chassis, with additional wedges cut out at specific angles. These wedges were meant to anchor the ridges used to hold the fan together and were set at angles of 64° and 26° offset from the vertical such that the fan's exhaust vent would point perpendicular to the ground. The angle of these additional wedges was determined by carefully measuring the fan's diameter and the location and size of the attachment ridges and using a protractor to determine each ridge's offset from the vertical, adding additional size for tolerance.

The exhaust vent was connected to the tower via a 3D-printed adapter that matched the circular diameter of the exhaust vent to the side length of the square tower. An extra slot, visible in the top right image above, was cut out to accommodate an extra outcropping on the fan.

System Modeling

To model the system, we used a simple difference equation approach where the current position is the sum of the previous position plus the previous velocity times the timestep, and so forth. Extending this approach to acceleration we end with the final equations

$$x[n] = x[n-1] + \Delta t \cdot v[n-1] + \Delta \mathrm{t}^2 \cdot a[n-2]$$

$$a[n] = \frac{\gamma \cdot u[n]}{\mathrm{m}}$$

where x[n] represents the ball's current position, v[n] is the ball's velocity, a[n] is the ball's acceleration, Δt is the timestep, u[n] is the controller output signal, m is the ball's mass, and γ is a constant of proportionality between the controller's output and the force on the ball. The timestep is set at 1/60 seconds based on the update rate of the IR sensor. γ was not modeled extensively but served as an unknown as the system was tested and tuned.

Verilog Implementation

Overview

The components of the system completed in Verilog include the PD controller, the PWM signal generation, the ADC SPI interface, and the IR sensor and potentiometer readings. Each module of the system was fulfilled a different purpose and featured its own set of implementation challenges.

Module: labkit

The labkit module contained all the high-level modules required for the control system to function. This included the PSD_controller, pwm_generator, ADC, and IR_sensor modules. Within the labkit module these four modules were connected via different wires, and additional testing wires were instantiated as necessary. Each module was tested separately.

Module: PSD_controller

Inputs:

- clock: system clock, set to 65 Mhz
- i_reset: reset wire, used to restart PSD and set outputs to 0.
- [5:0] i_ref: reference height, set by audio frequency
- [5:0] i_height: height of the ping-pong ball as determined by sensor
- [9:0] i_Kp_tune: 10-bit proportional gain tuning from the ADC
- [9:0] i_Ki_tune: 10-bit sum gain tuning from the ADC
- [9:0] i_Kd_tune: 10-bit delta gain tuning from the ADC
- [9:0] i_bias_tune: 10-bit bias point tuning from the ADC

Outputs:

- o_PSD_done: output flag that pulses high when the PSD finishes a control loop
- signed [10:0] o_control_sig: control signal used for PWM generation
- [9:0] o_test, o_test2, o_test3, o_test4: testing outputs for debugging

This module served as a wrapper module for the various modules that made up the PD controller. The module was implemented as a finite state machine which activated the other modules of the system and moved on to the next step as the previous one was completed. At the start of the module each submodule and its related wires are instantiated before the actual state machine begins its operation.

The PSD FSM starts in the idle state. While in this state the current i_ref and i_height signals are constantly fed into the r_ref and r_height registers. However, once either i_ref or r_ref is different from its previous value, i.e. the register value does not equal the input value, the r_error_start register is activated and the system moves into the calculate error state. Because the IR sensor updates at a frequency of 60 hz, the control loop runs at about the same speed. Because the internal clock runs at 65 Mhz, this means that we have a large number of clock cycles to complete all calculations.

The activation of the r_error_start register causes the *calc_error* module to begin running. The controller remains in the calculate error state until the *calc_error* module outputs a 'done' signal, at which point the controller activates r_PSD_start and moves to the PSD calculation state. r_PSD_start causes the proportional and delta modules to activate, and the controller waits until it receives done signals both modules before moving to the output calculation state. Here r_command_start is activated and the system moves to the done state. While in the done state the system waits until the command calculation is finished, after which the system pulses r_PSD_done and returns to the idle state.

The actual calculations involved in the *PSD_controller* module are carried out in the submodules instantiated before the state machine section of this module. The state machine itself determines which submodules are activated whereas the wires concerning each computation are updated within the submodules themselves. When the reset button is activated, all registers and the output are set to zero and the FSM returns to the IDLE state. After all steps of the computation are completed the output wire o_control_sig is updated and this value is fed into the *pwm_generator* module. To test this module, both the reference height and the ball height were set to FPGA switch inputs and the commanded output was tracked on the FPGA 8-digit hex display.

Module: calc_error

Inputs:
- Clock: system clock, set to 65 Mhz
- i_start_err: start signal for this module
- i_reset: reset signal for this module
- [5:0] i_ref_err: input reference height *r[n]*
- [5:0] i_height_err: input ball height *h[n]*

Outputs:
- o_done_err: done signal
- signed [6:0] o_error: error signal calculated.

This module uses a finite state machine to perform signed subtraction *r[n] - h[n]* to determine e[n], the error signal. r[n] and h[n] range from 0 to 63. Therefore e[n] has a range of -63 to 63, represented by a 7-bit two's complement integer. Because of the input constraints, integer overflow or underflow is not possible and is not checked for. The bit size of error signal is equal to the size of reference/height signals plus one.

The state machine for this module consists of three states. In the idle state, the inputs i_ref_err and i_height_err are each given leading zeros and placed into signed 7-bit registers, converting them from unsigned to signed values. Once i_start_err is activated, the system moves to the calculate error state, which subtracts the signed reference height and ball height registers and pulses the done signal. The done state switches off the done signal and returns the FSM to idle. If the reset input is set, all registers are set to zero and the FSM returns to idle.

Module: proportional_term

Inputs:
- clock: system clock, 65 Mhz
- i_reset: reset signal
- signed [6:0] i_error: error signal generated by *calc_error* module

- i_start: start signal
- [9:0] i_Kp_tune: 10-bit ADC value for tuning proportional gain K_p

Outputs:
- o_p_done: done signal that pulses after calculations are complete
- signed [8:0] o_prop_term: the actual proportional term calculated by module
- [9:0] test, test2: testing wires for debugging

This module uses a state machine to calculate the proportional term of the PD controller. The proportional term is calculated using the following formula:

$$p[n] = K_p \cdot e[n]$$

where e[n] is the error signal and K_p is the proportional gain. Within this module the default value of the gain is hard-coded but can be manually tuned via a potentiometer connected to an ADC. The details of this tuning calculation are explained in the *gain_tuner* module. The module uses parameters to set the default K_p value, the maximum and minimum K_p values, and the range of the gain tuner. Parameters are also used to set the maximum and minimum proportional term output modules.

The state machine consists of four states: idle, calculate proportional term, calculate overflow, and done. During the idle state, the value of K_p used in the calculation is stored in a register r_Kp and is continually set to the default value KP_BASE plus a value Kp_offset. This offset value is generated by the gain tuner module and corresponds to the current angle on the potentiometer. Before setting r_Kp, the module first checks if the current value of Kp_offset, when added to r_Kp, would cause r_Kp to fall outside the ranges set by the KP_MIN and KP_MAX parameters. If this is the case, r_Kp is set to the minimum or maximum value as necessary; otherwise r_Kp is set to KP_BASE plus Kp_offset.

Once i_start is activated, the module moves to the s_CALC state, which simply multiplies i_error by r_Kp and enters it into a register r_prop_raw before moving to the s_CALC_OVERFLOW state. This state compares the sign of i_error with the sign of the bottom nine bits of r_prop_raw. If the signs differ, over/underflow occurred and the value of r_prop_term, a register which stores the final output value, is set to the maximum or minimum value as set by parameters PROP_MAX and PROP_MIN. If no over/underflow occurred then r_prop_term is set to the bottom nine bits of r_prop_raw and the done signal is pulsed in the s_DONE state before the state machine returns to idle.

Module: delta_term

Inputs:
- clock: system clock, set to 65 Mhz
- signed [6:0] i_error: the error signal produced by the *error_calc* module
- i_start : start signal

- [9:0] i_Kd_tune: 10-bit tuning signal from ADC
- i_reset: reset signal

Outputs:

- o_d_done: done signal
- signed [8:0] o_del_term: delta term output
- [9:0] test: testing wire for debugging

This module is used to generate the delta term of the PD controller. The formula for the delta term is as follows:

$$d[n] = \mathrm{K_d}\frac{e[n] - e[n-1]}{\Delta \mathrm{t}}$$

which discretely approximates a derivative. Here K_d is the gain for the delta term and Δt is the timestep. In the final implementation, the calculation was not divided by the timestep Δt, but instead this division is implicitly incorporated into the K_d gain. This prevented the need for a divisor module, which would slow down the computation. Additionally, *e[n-4]* was used instead of *e[n-1]*. This slows down the derivative calculation and makes it less susceptible to errors due to noise.

Within this module the default value of the gain is hard-coded but can be manually tuned via a potentiometer connected to an ADC. The module uses parameters to set the default K_d value, the maximum and minimum K_d values, and the range of the gain tuner. Parameters are also used to set the maximum and minimum delta term output modules, as well as maximum and minimum values for intermediate calculations.

The state machine starts in the idle state. In this state, the value of r_Kd, the delta gain value used in the delta term calculation, is continually set to be the default value KD_BASE plus an offset Kd_offset set by the potentiometer angle. A check to make sure r_Kd falls between its maximum and minimum values is carried out identical to that in the proportional term module. When the start signal is detected the module moves to the s_CALC state.

In this state the value of *e[n] – e[n-4]* is calculated by shifting previous values of *e* into different registers and subtracting the past value of e from the current value (equal to i_error). This difference is placed into the r_differnce register, and in the next state s_CALC_OVERFLOW this register is checked for over/underflow and set to the appropriate value if this is the case.

After the over/underflow check the FSM moves to s_CALC2 wherein the r_difference register is multiplied by r_Kd and stored in a register r_del_raw, which is checked for over/underflow in an identical process to r_prop_raw in the *proportional_term* module. After the second over/underflow check the r_del_term register, connected to the output o_del_term, is updated and the done signal is pulsed.

While the reset wire is high, all internal registers are set to zero and the state machine returns to the idle state.

Module: gain_tuner

Inputs:

- clock: system clock, set to 65 Mhz
- [9:0] i_tune: 10-bit tuning value set by potentiometer angle
- i_reset: reset signal
- signed [5:0] i_offset_min: minimum value for the offset
- signed [5:0] i_offset_max: maximum value for the offset

Outputs:

- signed [7:0] o_offset: calculated offset value ranging from i_offset_min to i_offset_max

This module takes in the 10-bit ADC value from a gain tuning potentiometer and converts it into a tuning offset value. This offset is then added to the hard-coded gain value so that it can be tuned without resynthesizing Verilog. As input the module takes clock and reset wires, the minimum and maximum gain offset values, and the 10-bit ADC output.

The module linearly scales the ADC value such that 0 on the ADC corresponds to the minimum gain offset and 1023 corresponds to the maximum gain offset. Intermediate ADC values are converted to offset values using the following function:

$$offset = \mathrm{i}_{\text{offset min}} + floor\left(\frac{(ADC + b) \times d}{1024}\right)$$

where d is the offset range $d = \mathrm{i}_{\text{offset max}} - \mathrm{i}_{\text{offset min}}$ and b is a buffer value that allows the system to reach the most extreme offset values without requiring an ADC value of exactly 0 or 1023. b is dependent on the offset range, and an optimal b is calculated as follows:

$$b = floor\left(\frac{\frac{\mathrm{d}+1}{\mathrm{d}} \times 1024 \text{ - } 1023}{2}\right)$$

b is not calculated in the program and is instead calculated by the user and entered as a parameter, to avoid the use of division. When changing the minimum and maximum offset values, a different b will have to be calculated each time. By default it is set to 25, which is a healthy value for many offset ranges. The calculated offset is added to the gain in the appropriate proportional, sum, or delta term module. Instead of dividing by 1024 and flooring, the Verilog simply right shifts by 10 bits. A register chain is used so that the calculation can be completed without propagation delay errors. When the reset input is high all internal registers are set to zero.

The controller module uses a potentiometer to determine the bias point and thus this module is used to calculate what the bias should be.

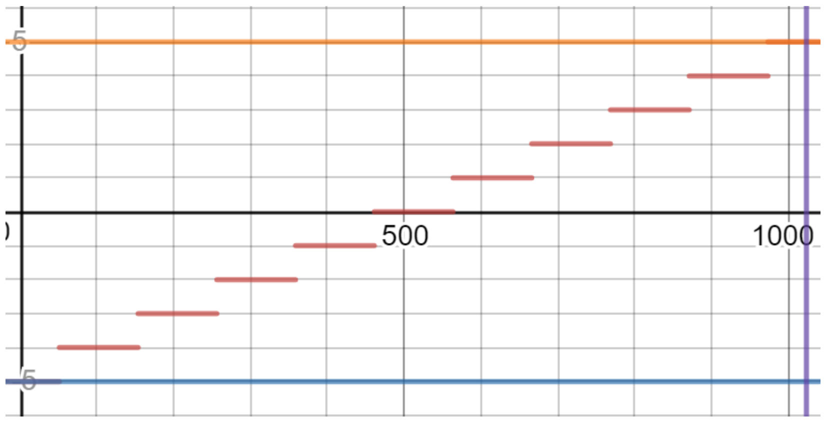

Figure X: This graph shows the output of the tuning module as a function of the ADC output. Here the ADC runs from 0-1023 and the offset can range from -5 to 5. The discontinuous red lines represent the tuner output as the ADC value increases. Other lines represent the boundaries of the ADC and the offset output.

Module: command_calc

Inputs:

- clock: system clock set to 65 Mhz
- i_start: start signal
- i_reset: reset signal
- signed [8:0] i_prop: proportional term
- signed [8:0] i_sum: sum term
- signed [8:0] i_delta: delta term
- signed [9:0] i_bias: bias term, ranges from 0 to 31 set by potentiometer angle

Outputs:

- signed [10:0] o_command: controller command signal output
- o_u_done: done signal

This module takes in the outputs from the proportional, sum, delta, and bias modules and sums them to determine the final controller output. The summation is carried out via a state machine where each state corresponds to a different calculation, to prevent propagation delay errors resulting from too many calculations being attempted in a single clock cycle. The bias term is multiplied by 10 so that it ranges from 0 to 310 instead of 0 to 31. When the reset input is activated, all internal registers are set to 0 and the state is reset to idle. In the final implementation, the *sum_term* module's output was not used.

Module: pwm_generator

Inputs:

- clock
- signed [10:0] i_control
- i_reset

Outputs:

- o_pwm: the PWM output signal

This module creates a PWM signal switching between 0 to 3.3 volts. The frequency of the signal is defined in the PWM_FREQ parameter. The signal implementation is based on a pair of counters. r_counter_PWM increments up to COMMAND_MAX-1. COMMAND_MAX is the control signal input value that corresponds to a 100% duty cycle PWM. By default it is set to 600, because the control signal is unlikely to ever exceed this value. Lowering or increasing this value will change how easily the controller output is able to generate a large duty cycle PWM.

r_counter_PWM increments every $\frac{T_{PWM}}{\text{COMMAND MAX}}$ seconds, where T_PWM is the period of the PWM signal, by default set to 0.001 seconds (1khz PWM signal) .This corresponds to approximately CLK_MAX = 108 clock cycles when the clock is running at 65mhz. The equations used to calculate CLK_MAX are as follows, where T_65 is the period of the 65mhz clock.

$$\text{CLK MAX} = \frac{T_{PWM}/\text{COMMAND MAX}}{T_{65}} = \frac{65\ \text{Mhz}}{f_{PWM} \times\ \text{COMMAND MAX}} \approx 108$$

when COMMAND_MAX = 600. r_counter_PWM is set to increment every CLK_MAX clock cycles, creating a PWM frequency of approximately 1khz. r_counter_clk keeps track of these CLK_MAX clock cycles, resetting at 108. If the value of r_counter_PWM is less than the value of control input, the PWM output signal will be high. Otherwise it will be low. This means that a negative controller output or a controller output of 0 will give a 0% duty cycle PWM. A controller output of COMMAND_MAX/2 is approximately a 50% duty cycle, and a controller output of COMMAND_MAX is a 100% duty cycle. r_counter_PWM resets at COMMAND_MAX-1.

Because the value of CLK_MAX is hard coded (to avoid completing a division operation), changing COMMAND_MAX requires recalculating the value of CLK_MAX.

Within the *labkit* module, the control signal input is liked to the output of the *PSD_controller* module and the o_pwm PWM signal output is linked to the JA[0] output pin on the Nexys 4.

Module: ADC

Inputs:

- sysclk: the system clock, set to 65 Mhz
- ADC_start
- miso: SPI MISO wire, output from the ADC

Outputs:

- mosi, sck, cs: other SPI wires, inputs to ADC
- [9:0] o_channel0, o_channel1, o_channel2, o_channel3, o_channel4: wires that store values from channels 0-4 of the ADC

This module uses a finite state machine to read values from channels 0-4 of the ADC. It is based on code originally written by Joe Steinmeyer and used with his permission. This module is used in conjunction with the *spi_master* module and serves as a wrapper to determine what information is sent out of and sent back to *spi_master*. The module begins by instantiating all the different wires and registers that are used to interface with *spi_master*, as well as the *spi_master* module itself. The rest of the module is dedicated to the state machine. In the labkit module, an additional clock and an ADC_start signal are generated, running at 1 Mhz and 1 khz respectively. The SCK, MOSI, and CS wires are set to the output pins JA[1], JA[2], and JA[3] respectively.

While in the idle state, the trigger register, which activates *spi_master*, is held at zero until the ADC_start input is high. Once the ADC has been told to start, the registers selection, bytes_to_send, and data_to_send are set to the appropriate values to read a single ended reading from channel 0 on the SPI device, the MCP3008. The state machine then triggers *spi_master* in the T1 state and waits until it has finished reading data in the RW1 state. In this state, once the new_data wire from *spi_master* is high, the current bottom nine bits of data_receieved are set to the appropriate channel register. After reading channel 0, the state machine moves to the READ_NEXT_CHANNEL state, which sets the selection, bytes_to_send, and data_to_send registers to the necessary values to read from channels 1-4 on the MCP3008. selection and bytes_to_send do not change their values across channels, but the channel selection bits of data_to_send change each time. After setting each register the state machine moves back through T1 and RW1, and after looping through all five channels the state machine returns to idle.

Module: spi_master

Inputs:

- sysclk: system clock, set to 65 Mhz
- ss: for selecting slaves from input side
- [INOUTWIDTH-1:0] data_to_send: bits sent to the SPI device
- [15:0] how_many_bytes: if we want repeated reading/writing
- miso: SPI MISO wire
- rst: reset wire
- trigger: used to start FSM

Outputs:

- reg sck, mosi
- reg [7:0] cs: for selecting slaves on output side (one hot wiring)
- reg [INOUTWIDTH-1:0] data_in,
- reg busy

- reg new_data
- reg load

This module was originally written by Joe Steinmeyer and used with his permission. It uses a state machine to set a series of shift registers such that the right bits are sent to the SPI device, in this case the MCP3008 ADC, to read data from a single channel. As part of the *ADC* module, this state machine is looped through five times to read data from five different channels.

Module: IR_sensor

Inputs:

- clock: system clock, set to 65 Mhz
- [9:0] i_IR: 10-bit voltage reading from the IR sensor from the ADC

Outputs:

- [5:0] o_height: 6-bit height signal sent to the PD controller

This module takes the 10-bit analog voltage reading from the ADC and converts it to a height in centimeters, and then to the appropriate 6-bit height signal used by the *PSD_controller* module. The IR sensor has a highly nonlinear curve mapping voltage to distance. Therefore to create this module the sensor was calibrated and the curve was linearized. The calibration was completed by carefully recording the sensor's voltage value at different known distances. Due to the nature of the curve, four different linearizations were used depending on the input ADC values. The linearizatons were completed using curve fitting tools in MATLAB.

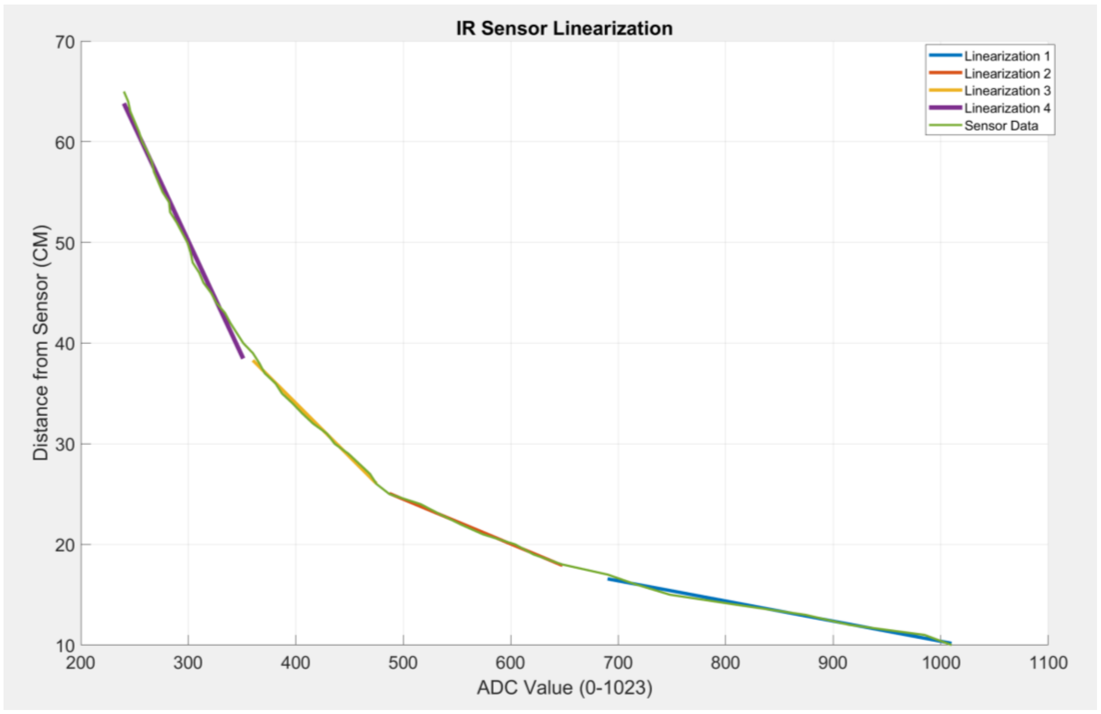

Figure X: Linearization of the IR data in MATLAB

The calibration data from the IR sensor in the plot is the green line, which can be seen to follow a decaying exponential curve. The four straight lines overlaying the green line are the different linearization curves used depending on the value of the ADC. It can be seen that the combination of all four linearizations matches the exponential curve very well. Using the linearizations, the function mapping the 10-bit ADC reading to a height in centimeters is as follows:

$$\mathrm{h_{cm}} = 60 - \frac{\alpha * \mathrm{ADC} - \gamma}{256}$$

Here, α and γ are parameters that define the linearization slope and y-intercept. In Verilog, instead of dividing by 256 the appropriate register is right shifted 8 bits. Different values for α and γ are used depending on the reading of the ADC, corresponding to the four linearizations completed. This module first uses the ADC value to determine the ball's height in centimeters, which based on the height of the tower ranges from 0-50 cm.

After the height in centimeters is computed, the module converts that height into a 6-bit signal to be used as input to the *PSD_controller* module. This conversion follows this formula:

$$\mathrm{h_{6bit}} = \frac{(h_{\mathrm{cm}} - 15) * \beta}{128}$$

Here β is a separate parameter used to map the 0-50 $\mathrm{h_{cm}}$ input to a 0-63 $\mathrm{h_{6bit}}$ output. $\mathrm{h_{cm}}$ is decreased by 15 so that the mapping from $\mathrm{h_{cm}}$ to $\mathrm{h_{6bit}}$ will only span the center section of the levitation tower, rather than the entire tower. This prevents the ball from encountering erroneous readings or dampened dynamics that occur near the top and bottom of the tower respectively.

Module: sum_term

Inputs:
- clock: system clock, set to 65 Mhz
- signed [6:0] i_error: the input error signal from the *error_calc* module
- i_start: start signal
- [9:0] i_Ks_tune: 10-bit tuning value from ADC
- i_reset: reset wire

Outputs:
- o_s_done: done signal
- signed [8:0] o_s_term: sum term output
- [9:0] test: test signal for debugging

This module was used to generate the sum term of the PSD controller according to the following formula:

$$s[n] = \mathrm{K_i} \sum_{\mathrm{i=1}}^{\mathrm{k}} e[n] * \Delta\mathrm{t}$$

where K_i is the gain applied to the summation. The module would move through different states to calculated both the summation value and the final sum term, checking for over/underflow after each calculation. The value of the sum and the final output were both limited to prevent integral windup. In the final implementation this module was not used, as it was found to wind up too quickly, and was replaced with a constant bias point instead.

Challenges and Lessons Learned

Successfully bridging the gap between hardware and software was the most difficult part of the project. Particular difficulty was encountered when calibrating the IR sensor. In total, three different calibration setups were used, with the third being successful in producing consistent readings from the sensor. This set up attached the IR sensor to a hardcover book such that the sensor's line of sight would be perpendicular to the ground. Facing the sensor was another hardcover book covered in white paper, to provide a reflective parallel surface for the sensor to read. An Arduino was used to read values from the sensor, and after 500 values were taken the average reading was recorded. This average reading was used for curve fitting. The carefulness required to set up this calibration suggests that the use of more consistent hardware could have improved the performance of the final system.

Successfully tuning the system was also difficult. A large reason behind this was certain unmodeled dynamics of the fan and sensor system whose consequences did not become apparent until the hardware system was tested. Near the bottom of the tower, significantly more force is required to lift the ping-pong ball in the air. At the very bottom, even running the fan at full speed will not cause the ball to float. However, after the ball is in the air, it requires very little airflow to keep it floating. To prevent the ball from reaching the bottom, a spare piece of acrylic was included in the tower. Additionally, the IR sensor has a minimum distance reading of about 10 cm, at which the voltage from the sensor reaches a peak. If an object is moved closer than this distance, the output voltage will actually decrease. This means that if the ball accidentally moved closer than 10 cm, the control system would interpret this as the ball moving away from the sensor. If the controller was set to place the ball as close to the sensor as possible, this would mean the system would increase the PWM output if the ball moved closer than 10 cm, because the sensor reading was telling the controller the ball was much lower. Therefore the ball would become stuck at the top of the tower.

These problems were alleviated via careful tuning of the mapping from the ball's height in centimeters to the 6-bit output from the *IR_sensor* module so that a height of ~15 cm would produce a 0 output and a distance of ~10 cm from the sensor would provide a 63 output. This constrained the ball to the middle of the tower were dynamics were mostly linear, although excess velocity of could still shift force the ball out of this zone.

Future implementations would benefit from a taller tower so that this linearized region would be extended.

Creating the *gain_tuner* module, which linearly mapped a potentiometer angle to a specified range of outputs, also required careful math to ensure that each value within the range could be easily and consistently reached using the potentiometer. To properly visualize, different graphing tools were used to connect ADC output to the specified range of outputs.

Finally, when the sum term was added to the final implementation, excessive windup occurred almost immediately, causing the value of this term to alternate between the maximum and minimum values respectively. One reason for this was because in ordinary PSD controller implementations, the sum term summation is multiplied by the time step, which in this case is 1/60 seconds, the update rate of the sensor. This significantly slows down the development of the sum term. In our implementation, we did not divide by 60 in order to speed up the system. It is likely that leaving out this division caused the sum term to accumulate too quickly to be useful. Future implementations could use a divisor module to complete this computation, because the 65 Mhz clock frequency would probably allow this computation to complete before the 60 hz sensor updated. Alternatively, the intermediate registers of the sum term calculations could be placed in an oversized register and then scaled back down, so that the summation could still accumulate but would not appear as quickly on the output due to the scale down.

Appendix: Verilog Modules

1. David

Main Module:

```
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
// Engineer: Mitchell Gu
// Project Name: Nexys4 FFT Demo
////////////////////////////////////////////////////////////////////////////////

module nexys4_fft_demo (
   input wire CLK100MHZ,
   input wire [15:0] SW,
   input wire BTNC, BTNU, BTNL, BTNR, BTND,
   input wire M_DATA,
   input wire [7:0] JB,
   //input wire AD3P, AD3N,  // The top pair of ports on JXADC on Nexys 4
   output wire [3:0] VGA_R,
   output wire [3:0] VGA_B,
   output wire [3:0] VGA_G,
   output wire VGA_HS,
   output wire VGA_VS,
   output wire M_CLK,M_LRSEL,
   output wire AUD_PWM, AUD_SD,
   output wire [7:0] JA,
   output wire LED16_B, LED16_G, LED16_R,
   output wire LED17_B, LED17_G, LED17_R,
   output wire [15:0] LED, // LEDs above switches
   output wire [7:0] SEG,  // segments A-G (0-6), DP (7)
   output wire [7:0] AN    // Display 0-7
   );

   // SETUP CLOCKS
   // 104Mhz clock for XADC and primary clock domain
   // It divides by 4 and runs the ADC clock at 26Mhz
   // And the ADC can do one conversion in 26 clock cycles
   // So the sample rate is 1Msps (not posssible w/ 100Mhz)
   // 65Mhz for VGA Video
   wire clk_104mhz, clk_65mhz;
   clk_wiz_0 clockgen(
      .clk_in1(CLK100MHZ),
      .clk_out1(clk_104mhz),
      .clk_out2(clk_65mhz));
```

```
    // INSTANTIATE XVGA SIGNALS (1024x768)
    wire [10:0] hcount;
    wire [9:0] vcount;
    wire hsync, vsync, blank;
    xvga xvga1(
        .vclock(clk_65mhz),
        .hcount(hcount),
        .vcount(vcount),
        .vsync(vsync),
        .hsync(hsync),
        .blank(blank));

// **************** BEGIN BASIC IO SETUP ******************************//

    // INSTANTIATE SEVEN SEGMENT DISPLAY
    display_8hex display(
        .clk(clk_65mhz),
        .data(Display),
        .seg(SEG[6:0]),
        .strobe(AN));
    assign SEG[7] = 1;
    wire [31:0] Display;
    wire [9:0] scaled;
    assign scaled = frequency - 10'd65;

    assign Display = SW_clean[14] ? {2'b0,r_test,8'b0,2'b0,reference,2'b0,want} :
{6'b0,hertz,2'b0,IR_height,2'b0,r_test};



    // Parametrized debounce module to do all 16 switches and 5 buttons
    wire BTNC_clean, BTNU_clean, BTND_clean, BTNL_clean, BTNR_clean;
    wire [15:0] SW_clean;
    debounce #(.COUNT(21)) db0 (
        .clk(clk_65mhz),
        .reset(1'b0),
        .noisy({SW, BTNC, BTNU, BTND, BTNL, BTNR}),
        .clean({SW_clean, BTNC_clean, BTNU_clean, BTND_clean, BTNL_clean,
BTNR_clean}));


// **************** END BASIC IO SETUP ******************************//
```

```
    // INSTANTIATE 16x OVERSAMPLING
    // This outputs 14-bit samples at a 62.5kHz sample rate
    // (2 more bits, 1/16 the sample rate)

    wire [13:0] osample16;
    wire done_osample16;
    wire [13:0] audio_data;

    oversample16 osamp16_1 (
        .clk(clk_104mhz),
        .sample(audio_data[13:2]),
        .eoc(eoc),
        .oversample(osample16),
        .done(done_osample16));


////////////////////////////////////////////////////////////////////////////////

    wire eoc;
    wire low_data;


    Audio_Interface microphone(
        .clk_104mhz(clk_104mhz),
        .M_DATA(M_DATA),
        .M_CLK(M_CLK),
        .data(audio_data),
        .sample_ready(eoc)
        );

    assign M_LRSEL = 0;




//////////////////////////////////////////////////////////////////////////////////////
    // INSTANTIATE SAMPLE FRAME BLOCK RAM
    // This 16x4096 bram stores the frame of samples
    // The write port is written by osample16.
    // The read port is read by the bram_to_fft module and sent to the fft.
    wire fwe;
    reg [11:0] fhead = 0; // Frame head - a pointer to the write point, works as circular buffer
    wire [15:0] fsample;  // The sample data from the XADC, oversampled 15x
    wire [11:0] faddr;    // Frame address - The read address, controlled by bram_to_fft
    wire [15:0] fdata;    // Frame data - The read data, input into bram_to_fft
    bram_frame bram1 (
        .clka(clk_104mhz),
        .wea(fwe),
        .addra(fhead),
        .dina(fsample),
        .clkb(clk_104mhz),
        .addrb(faddr),
        .doutb(fdata));

    // SAMPLE FRAME BRAM WRITE PORT SETUP
    always @(posedge clk_104mhz) if (done_osample16) fhead <= fhead + 1; // Move the pointer
every oversample
    assign fsample = {osample16, 2'b0}; // Pad the oversample with zeros to pretend it's 16 bits
    assign fwe = done_osample16; // Write only when we finish an oversample (every 104*16
clock cycles)

    // SAMPLE FRAME BRAM READ PORT SETUP
    // For this demo, we just need to display the FFT on 60Hz video, so let's only send the frame
of samples
    // once every 60Hz. If you want to though, you can send frames much faster, one right after
each other.
    // For this 4096pt fully pipelined FFT, the limit is 104Mhz/4096cycles_per_frame = 25kHz
(approx)
    // The next two modules just synchronize the 60Hz vsync to the 104Mhz domain and convert
it to a 1 cycle pulse.
    wire vsync_104mhz, vsync_104mhz_pulse;
    synchronize vsync_synchronize(
        .clk(clk_104mhz),
        .in(vsync),
        .out(vsync_104mhz));

    level_to_pulse vsync_ltp(
        .clk(clk_104mhz),
        .level(~vsync_104mhz),
        .pulse(vsync_104mhz_pulse));

    // INSTANTIATE BRAM TO FFT MODULE
    // This module handles the magic of reading sample frames from the BRAM whenever start is
asserted,
    // and sending it to the FFT block design over the AXI-stream interface.
    wire last_missing; // All these are control lines to the FFT block design
    wire [31:0] frame_tdata;
    wire frame_tlast, frame_tready, frame_tvalid;
    bram_to_fft bram_to_fft_0(
        .clk(clk_104mhz),
        .head(fhead),
        .addr(faddr),
        .data(fdata),
        .start(vsync_104mhz_pulse),
        .last_missing(last_missing),
        .frame_tdata(frame_tdata),
        .frame_tlast(frame_tlast),
        .frame_tready(frame_tready),
        .frame_tvalid(frame_tvalid)
    );
///////////////////////////////////////////////////////////////////////
    wire empty,empty2,empty3,empty4,empty5,empty6,empty7,empty8;

        xfft_0 test(
        .aclk(clk_104mhz),
        .s_axis_config_tdata({5'b11100,SW_clean[12:7],2'b0,1'b1}),
        .s_axis_config_tvalid(frame_tvalid),
        .s_axis_config_tready(empty),
        .s_axis_data_tdata(frame_tdata),
        .s_axis_data_tvalid(frame_tvalid),
        .s_axis_data_tready(frame_tready),
        .s_axis_data_tlast(frame_tlast),
        .m_axis_data_tdata(magnitude_tdata),
        .m_axis_data_tvalid(magnitude_tvalid),
        .m_axis_data_tready(1'b1),
        .m_axis_data_tlast(empty2),
        .m_axis_data_tuser(magnitude_tuser),
        .event_frame_started(empty3),
        .event_tlast_unexpected(empty4),
        .event_tlast_missing(empty5),
        .event_status_channel_halt(empty6),
        .event_data_in_channel_halt(empty7),
        .event_data_out_channel_halt(empty8));

    // This is the FFT module, implemented as a block design with a 4096pt, 16bit FFT
    // that outputs in magnitude by doing sqrt(Re^2 + Im^2) on the FFT result.
    // It's fully pipelined, so it streams 4096-wide frames of frequency data as fast as
    // you stream in 4096-wide frames of time-domain samples.
    wire [31:0] magnitude_tdata; // This output bus has the FFT magnitude for the current index
    wire [11:0] magnitude_tuser; // This represents the current index being output, from 0 to 4096
    wire [11:0] scale_factor; // This input adjusts the scaalidling of the FFT, which can be tuned to
the input magnitude.
    wire magnitude_tlast, magnitude_tvalid;




    // Let's only care about the range from index 0 to 1023, which represents frequencies 0 to
omega/2
```

```
   // where omega is the nyquist frequency (sample rate / 2)
   wire in_range = ~|magnitude_tuser[11:10]; // When 13 and 12 are 0, we're on indexes 0 to
1023

   // INSTANTIATE HISTOGRAM BLOCK RAM
   // This 16x1024 bram stores the histogram data.
   // The write port is written by process_fft.
   // The read port is read by the video outputter or the SD care saver
   // Assign histogram bram read address to histogram module unless saving
   wire [9:0] haddr; // The read port address
   wire [15:0] hdata; // The read port data
   bram_fft bram2 (
      .clka(clk_104mhz),
      .wea(in_range & magnitude_tvalid),  // Only save FFT output if in range and output is valid
      .addra(magnitude_tuser[9:0]),       // The FFT output index, 0 to 1023
      .dina(magnitude_tdata[15:0]),       // The actual FFT magnitude
      .clkb(clk_65mhz),  // input wire clkb
      .addrb(haddr),     // input wire [9 : 0] addrb
      .doutb(hdata)      // output wire [15 : 0] doutb
   );

   reg [9:0] addr_in=0;
   wire [9:0] addr_out;
   wire [15:0] soundin;
   wire [15:0] soundout;
   always @(posedge clk_104mhz) begin addr_in<= done_osample16 ? addr_in +1 : addr_in;
end
   assign soundin = {audio_data,2'b0};

   bram_fft bram3 (
         .clka(clk_104mhz),
         .wea(done_osample16),  // Only save FFT output if in range and output is valid
         .addra(addr_in),       // The FFT output index, 0 to 1023
         .dina(soundin),       // The actual FFT magnitude
         .clkb(clk_65mhz),  // input wire clkb
         .addrb(addr_out),     // input wire [9 : 0] addrb
         .doutb(soundout)      // output wire [15 : 0] doutb
      );



   // INSTANTIATE SOUND VISUAL
   wire [2:0] sound_pixel;


   sound aud_disp(
    .clk(clk_65mhz),
    .data(soundout[15:6]),
    .hcount(hcount),
    .vcount(vcount),
    .blank(blank),
    .pixel(sound_pixel),
    .vaddr(addr_out));




// INSTANTIATE HISTOGRAM VIDEO
// A simple module that outputs a VGA histogram based on
// hcount, vcount, and the BRAM read values
reg [14:0] data;
reg [9:0] addr;

always @(posedge clk_65mhz) begin
    data<= hdata[15] ? 0 : hdata[14:0];
    addr<=haddr;
    end


wire [2:0] hist_pixel;
wire [1:0] hist_range;
histogram fft_histogram(
    .clk(clk_65mhz),
    .hcount(hcount),
    .vcount(vcount),
    .blank(blank),
    .range({1'b1,SW_clean[15]}), // How much to zoom on the first part of the spectrum
    .vaddr(haddr),
    .vdata(data),
    .pixel(hist_pixel));


//////////////////////////////////////////////////////////////////////

wire [9:0] frequency;

freq_det pitch(
        .clk(clk_65mhz),
        .addr(addr),
        .data(data),
        .thresh(4'b1110),
        .frequency(frequency));
    ////////////////////////////////////////////////////////////////////
    wire [5:0] reference;
    wire [5:0] want;
    ReferenceGen Ref(
       .clk(clk_65mhz),
       .send(BTND_clean),
       .mode(SW_clean[15]),
       .freq(frequency),
       .Ref(reference),
       .want(want)
       );

    wire [11:0] hertz;

    dist_mem_gen_0 find(
         .a(frequency),      // input wire [9 : 0] a
         .clk(clk_65mhz),  // input wire clk
         .spo(hertz)  // output wire [11 : 0] spo
       );



//    ////////////////////////////////////////////////////////////
    wire [2:0] line;

    freq_disp bar(
          .clk(clk_65mhz),
          .bin(frequency),
          .hcount(hcount),
          .vcount(vcount),
          .blank(blank),
          .range({1'b1,SW_clean[15]}),
          .pixel(line));

    // INSTANTIATE PWM AUDIO OUT MODULE
    // 11 bit PWM audio out is reasonable because otherwise, the PWM frequency would
    // drop close to the audible and unfiltered range. 11bits -> 104Mhz/2^11=51Khz
    wire [10:0] pwm_sample;
    pwm11 pwm_out(
       .clk(clk_104mhz),
       .PWM_in(osample16[13:3]),
       .PWM_out(AUD_PWM),
       .PWM_sd(AUD_SD));
/////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////
```

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

/////////////////////////////////////////////////////////////////////////////
//Final Project Implementation

    //testing wires
    wire [5:0] r_test;
    assign r_test = SW_clean[6] ? reference : SW_clean[5:0];
//    wire [5:0] h_test;
//    assign h_test = SW[15:10];
    wire reset;
    assign reset = BTNC;

    //PSD Controller Instantiation
    wire signed [10:0] control_signal;
    wire [9:0] ADC_ch0, ADC_ch1, ADC_ch2, ADC_ch3, ADC_ch4;
    wire [5:0] IR_height;
    wire PSD_done;
    wire [9:0] PSD_test;
    wire [9:0] PSD_test2;
    wire [9:0] PSD_test3;
    PSD_controller PSD(.clock(clk_65mhz), .i_ref(r_test), .i_height(IR_height), .i_reset(reset),
        .i_Kp_tune(ADC_ch1), .i_Ki_tune(ADC_ch2), .i_Kd_tune(ADC_ch3),
.i_bias_tune(ADC_ch4),
        .o_control_sig(control_signal), .o_PSD_done(PSD_done), .o_test(PSD_test),
.o_test2(PSD_test2), .o_test3(PSD_test3));

    //PWM Generator Instantiation
    //testing wires
    //wire [10:0] PWM_test;
    //assign PWM_test = SW[10:0];
    wire PWM_signal;
    pwm_generator PWM(.clock(clk_65mhz), .i_control(control_signal), .o_pwm(PWM_signal),
.i_reset(reset));

    //ADC Instantiation
    wire ADC_sck, ADC_mosi, ADC_miso, ADC_cs;
    synchronize syn_ADC(.clk(clk_65mhz), .in(JB[0]), .out(ADC_miso));
    reg ADC_sysclk, ADC_start;
    wire w_ADC_sysclk, w_ADC_start;

    ADC MCP3008(.sysclk(ADC_sysclk), .ADC_start(ADC_start), .miso(ADC_miso),
        .mosi(ADC_mosi), .cs(ADC_cs), .sck(ADC_sck),
        .o_channel0(ADC_ch0), .o_channel1(ADC_ch1), .o_channel2(ADC_ch2),
.o_channel3(ADC_ch3), .o_channel4(ADC_ch4));
```

```
    //code for generating proper clock and start signals for ADC. MCP3008 only runs at a max of
3.6 MHhz
    //creates a ~1 mhz clock and a 1khz start pulse
    reg [3:0] r_ADC_clk_counter = 4'd0;
    reg [15:0] r_ADC_start_counter = 16'd0;
    reg [9:0] ADC_test;//ADC testing
    always @(posedge clk_65mhz) begin
        //~1 mhz clock
        if (r_ADC_clk_counter == 4'd15) begin
            ADC_sysclk <= ~ ADC_sysclk;
            r_ADC_clk_counter <= 4'd0;
        end
        else r_ADC_clk_counter <= r_ADC_clk_counter + 1;

        //1 khz ADC start signal
        if (r_ADC_start_counter == 16'd64999) begin
            ADC_start <= 1'b1;
            r_ADC_start_counter <= 16'd0;
        end
        else begin
            r_ADC_start_counter <= r_ADC_start_counter + 1;
            ADC_start <= 1'b0;
        end


        //ADC testing
        case (SW[1:0])
            2'b00: ADC_test <= ADC_ch0;
            2'b01: ADC_test <= ADC_ch1;
            2'b10: ADC_test <= ADC_ch2;
            2'b11: ADC_test <= ADC_ch3;
        endcase

    end
    assign w_ADC_sysclk = ADC_sysclk;
    assign w_ADC_start = ADC_start;

    //IR Sensor Module Instantiation

    IR_Sensor sensor(.clock(clk_65mhz), .i_IR(ADC_ch0), .o_height(IR_height));

    //assign data = {{2'b00, PSD_test}};//, {2'b0, PSD_test3}};

    assign JA[0] = PWM_signal;
    assign JA[1] = ADC_sck;
    assign JA[2] = ADC_mosi;
    assign JA[3] = ADC_cs;









////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
//
    // VGA OUTPUT
    // Histogram has two pipeline stages so we'll pipeline the hs and vs accordingly


    ///////////////////////////////////////////////////////////////////
    ///////////////////Make the range bars
    wire [2:0] tot_pixel;
    wire [9:0] horiz;
    reg [3:0] fence;
    reg [3:0] fence1;
    wire [9:0] x,x1;
    wire [9:0] y;
    parameter HEIGHT=100;
    assign y=10'd500;
    assign horiz = (hcount[9:0] >> {1'b1,SW_clean[15]});
    assign x=10'd65;
    assign x1=SW_clean[15] ? 10'd125 : 10'd190;




    always @(posedge clk_65mhz) begin
        fence<= ((horiz >= x && horiz < (x+1)) &&
            (vcount >= y && vcount < (y+HEIGHT))) ? 3'b100 : 3'b0;

        fence1<= ((horiz >= x1 && horiz < (x1+1)) &&
             (vcount >= y && vcount < (y+HEIGHT))) ? 3'b100 : 3'b0;
        end
    ////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////
    //////////////Track location of Ball////////////////////////////////////
    parameter MID=512;
    parameter QUART=256;
    parameter HALF=1;
    parameter FOURTH=0;

    wire [10:0] xball;
    wire [9:0] yball;
    wire ball_state;
    assign ball_state=SW_clean[15];

    assign yball=10'd515;
    assign xball=bottom;
    reg [10:0] bottom;
    reg [2:0] ball;
    wire [5:0]track;

    assign track =SW_clean[14] ? r_test : IR_height;


    always @ (posedge clk_65mhz)  begin   // generate round puck

        case(ball_state)
          // compute x -xcenter and y-ycenter

            HALF: begin
                    if (SW_clean[13]) begin
                        bottom<=(track>=6'd58) ? MID+(6'd58<<3)+11'd5 : MID+(track<<3)+11'd5;
                        ball <=(vcount >= yball && vcount < (yball+10'd20)) &&
                                (hcount >= xball && hcount < (xball+11'd20)) ? 3'b101 : 3'b0 ;

                    end
                    else
                            ball <=3'b0;
                    end

            FOURTH: begin
                    if (SW_clean[13]) begin
                       bottom<=(track>=6'd61) ? QUART+(6'd61<<3)+11'd5 :
QUART+(track<<3)+11'd5;

                       ball <=(vcount >= yball && vcount < (yball+10'd15)) &&
                               (hcount >= xball && hcount < (xball+11'd15)) ? 3'b101 : 3'b0 ;

                          end
                          else
                                  ball <=3'b0;
```

```
                end
        endcase

        end

  ///////////////////////////////////////////////////////////////////////////////
  ///////////////Instantiate actual pixels bing sent to VGA/////////////////////////


  assign tot_pixel = hist_pixel + sound_pixel + line+fence+fence1+ball;
  reg [1:0] hsync_delay;
  reg [1:0] vsync_delay;
  reg hsync_out, vsync_out;
  always @(posedge clk_65mhz) begin
     {hsync_out,hsync_delay} <= {hsync_delay,hsync};
     {vsync_out,vsync_delay} <= {vsync_delay,vsync};
  end
  assign VGA_R = {4{tot_pixel[0]}};
  assign VGA_G = {4{tot_pixel[1]}};
  assign VGA_B = {4{tot_pixel[2]}};
  assign VGA_HS = hsync_out;
  assign VGA_VS = vsync_out;

  // Assign RGB LEDs
  assign {LED16_R, LED16_G, LED16_B} = 3'b000;
  assign {LED17_R, LED17_G, LED17_B} = 3'b000;

  // Assign switch LEDs to switch states
  assign LED = SW;
//
///////////////////////////////////////////////////////////////////////////////


endmodule
```

Switch Debounce Module

```
// use your system clock for the clock input

// to produce a synchronous, debounced output


module debounce #(parameter DELAY=1000000, parameter COUNT=1) (

   input wire clk,

   input wire reset,

   input wire [COUNT-1:0] noisy,

   output reg [COUNT-1:0] clean);


   genvar i;

   generate

      for (i = 0; i < COUNT; i = i + 1) begin

         reg [19:0] count;

         reg new;


         always @(posedge clk) begin

            if (reset) begin

               count <= 0;

               new <= noisy[i];

               clean[i] <= noisy[i];

            end

            else if (noisy[i] != new) begin

               new <= noisy[i];

               count <= 0;

            end
            else if (count == DELAY)
               clean[i] <= new;
            else
               count <= count+1;
         end
      end
   endgenerate

endmodule
```

```
module level_to_pulse (
   input wire clk,
   input wire level,
   output wire pulse);

   reg last_level;
   always @(posedge clk) begin
      last_level <= level;
   end
   assign pulse = level & ~last_level;

endmodule
```

```
module display_8hex(
   input wire clk,                // system clock
   input wire [31:0] data,        // 8 hex numbers, msb first
   output reg [6:0] seg,      // seven segment display output
   output reg [7:0] strobe    // digit strobe
   );

   localparam bits = 13;

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

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


always @(posedge clk) begin
  counter <= counter + 1;
  case (counter[bits:bits-2])
    3'b000: begin
      seg <= segments[data[31:28]];
      strobe <= 8'b0111_1111 ;
    end
    3'b001: begin
      seg <= segments[data[27:24]];
      strobe <= 8'b1011_1111 ;
    end
    3'b010: begin
      seg <= segments[data[23:20]];
      strobe <= 8'b1101_1111 ;
    end
    3'b011: begin
      seg <= segments[data[19:16]];
      strobe <= 8'b1110_1111;
    end
    3'b100: begin
      seg <= segments[data[15:12]];
      strobe <= 8'b1111_0111;
    end
    3'b101: begin
      seg <= segments[data[11:8]];
      strobe <= 8'b1111_1011;
    end
```

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

```
module pwm11 (
   input wire clk,
   input wire [10:0] PWM_in,
   output reg PWM_out,
   output wire PWM_sd
   );
   reg [10:0] new_pwm=0;
   reg [10:0] PWM_ramp=0;
   always @(posedge clk) begin
      if (PWM_ramp==0) new_pwm <= PWM_in;
      PWM_ramp <= PWM_ramp + 1'b1;
      PWM_out <= (new_pwm>PWM_ramp);
   end
   assign PWM_sd = 1;
endmodule

// pulse synchronizer
module synchronize #(parameter NSYNC = 2) ( // number of sync flops.  must be >= 2
   input wire clk,in,
   output reg out);

  reg [NSYNC-2:0] sync;

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

```
endmodule
```

module xvga(

```
   input wire vclock,
   output reg [10:0] hcount,    // pixel number on current line
   output reg [9:0] vcount,     // line number
   output reg vsync, hsync, blank);

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

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

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
```

```
  always @(posedge vclock) begin
    hcount <= hreset ? 0 : hcount + 1;
    hblank <= next_hblank;
    hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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

```
module sound (
    input wire clk,
    input wire [9:0] data,
    input wire [10:0] hcount,
    input wire [9:0] vcount,
    input wire blank,
    output reg [2:0] pixel,
    output reg [9:0] vaddr);



    reg blank1;
    reg [10:0] hheight;
    reg [9:0] vtop;
    reg [9:0] vbot;
    wire [9:0] sound;



    always @(posedge clk) begin
        vaddr <= hcount[9:0];
        blank1<=blank;
        vtop<=10'd470 - vcount;
        vbot<=10'd475 - vcount;
        pixel <= blank1 ? 3'b0 : (vtop < data && vbot > data) ? 3'b101 : 3'b0;

    end

endmodule
```

module freq_det(

```
    input wire clk,

    input wire [9:0] addr,

    input wire [14:0] data,

    input wire [4:0] thresh,

    output wire [9:0] frequency

    );


    reg [9:0]   temp_freq;

    reg [9:0]   N;

    reg [14:0]  F;


    parameter WAIT    =2'b00;

    parameter MAYBE   =2'b01;

    parameter DISP    =2'b11;



    reg [1:0] state;

    reg [8:0] threshhold;



    always @(posedge clk) begin

       threshhold<= {thresh[4:1],5'd25};


       if ({data[12:0],2'b0}>threshhold && addr>50) begin

          temp_freq<=addr;

          F<=data;
```

```
            end

         else begin
            temp_freq<=temp_freq;
            end
         end


      assign frequency = temp_freq;


endmodule
```

```
module freq_disp(
    input wire clk,
    input wire [9:0] bin,
    input wire [10:0] hcount,
    input wire [9:0] vcount,
    input wire blank,
    input wire [1:0] range,
    output reg [2:0] pixel);

    reg blank1;
    reg [9:0] hheight;
    reg [9:0] vaddr;
    reg [9:0] vheight;
    reg [9:0] vtop;
    reg [9:0] vbot;
    wire [9:0] sound;
    reg [9:0] data;
    wire [9:0] horiz;
    assign horiz = (hcount[9:0] >> range);

    parameter [9:0] MAX = 10'd250;
    parameter [9:0] MIN = 10'd65;
    always @(posedge clk) begin
      vaddr <= hcount[9:0];
      data<=10'd50;
      blank1<=blank;
      vtop<=10'd600 - vcount;
```

```
            vbot<=10'd700 - vcount;

            if (bin== horiz) begin
                pixel <= blank1 ? 3'b0 : (vtop < data && vbot > data) ? 3'b010 : 3'b0;
            end

            else
                pixel<=3'b0;

        end

endmodule
```

```
module ReferenceGen(
   input wire clk,
   input wire send,
   input wire mode,
   input wire [9:0] freq,
   output wire [5:0] Ref,
   output reg [5:0] want
   );

   reg [5:0] curr_loc;
   wire send_pulse;



   level_to_pulse SAMP(.clk(clk),
                  .level(send),
                  .pulse(send_pulse));

   parameter CONT=1;
   parameter DISC=0;
   parameter [9:0] MAX = 10'd190;
   parameter [9:0] MAXL = 10'd120;
   parameter [9:0] MIN = 10'd65;
   always @ (posedge clk) begin

   case(mode)
      CONT: begin
         if (freq>MIN && freq<MAXL) begin
```

```
            curr_loc<= (freq-MIN);
            end
          else if (freq>=MAXL) begin curr_loc<=6'd63; end
          else
            curr_loc<=6'd0;
        end

        DISC: begin
          curr_loc<= send_pulse ? want : curr_loc;

          if (freq>MIN && freq<MAX) begin
                want<= (freq-MIN)>>1;
                end
          else if (freq>=MAX) begin want<=6'd63; end
          else
            want<=6'd0;

        end


      endcase

      end

    assign Ref=curr_loc;


    endmodule
```

module bram_to_fft(

```
input wire clk,
input wire [11:0] head,
output reg [11:0] addr,
input wire [15:0] data,
input wire start,
input wire last_missing,
output reg [31:0] frame_tdata,
output reg frame_tlast,
input wire frame_tready,
output reg frame_tvalid
);

// Get a signed version of the sample by subtracting half the max
wire signed [15:0] data_signed = {1'b0, data} - (1 << 15);

// SENDING LOGIC
// Once our oversampling is done,
// Start at the frame bram head and send all 4096 buckets of bram.
// Hopefully every time this happens, the FFT core is ready
reg sending = 0;
reg [11:0] send_count = 0;

always @(posedge clk) begin
  frame_tvalid <= 0; // Normally do not send
  frame_tlast <= 0; // Normally not the end of a frame
  if (!sending) begin
```

```
        if (start) begin // When a new sample shifts in
          addr <= head; // Start reading at the new head
          send_count <= 0; // Reset send_count
          sending <= 1; // Advance to next state
        end
      end
      else begin
        if (last_missing) begin
          // If core thought the frame ended
          sending <= 0; // reset to state 0
        end
        else begin
          frame_tdata <= {16'b0, data_signed};
          frame_tvalid <= 1; // Signal to fft a sample is ready
          if (frame_tready) begin // If the fft module was ready
            addr <= addr + 1; // Switch to read next sample
            send_count <= send_count + 1; // increment send_count
          end
          if (&send_count) begin
            // We're at last sample
            frame_tlast <= 1; // Tell the core
            if (frame_tready) sending <= 0; // Reset to state 0
          end
        end
      end
    end

endmodule
```

module histogram(

```
input wire clk,

input wire [10:0] hcount,

input wire [9:0] vcount,

input wire blank,

input wire [1:0] range,

output wire [9:0] vaddr,

input wire [14:0] vdata,

output reg [2:0] pixel

);


// 1 bin per pixel, with the selected range

assign vaddr = hcount[9:0] >> range;


reg [9:0] hheight; // Height of histogram bar

reg [9:0] vheight; // The height of pixel above bottom of screen

reg blank1; // blank pipelined 1


always @(posedge clk) begin


   // Pipeline stage 1

   hheight <= vdata[14:5]<<3;

   vheight <= 10'd767 - vcount;

   blank1 <= blank;

   // Pipeline stage 2

   pixel <= blank1 ? 3'b0 : (vheight < hheight) ? 3'b001 : 3'b0;

 end
```

```
endmodule
```

module CIC

```
                                    #(parameter width = 12)
                                     (input wire              clk,
                                     input wire              rst,
                                     input wire        [15:0] decimation_ratio,
                                     input wire signed [7:0]  d_in,
                                     output reg signed [7:0]  d_out,
                                     output reg                                  d_clk);

reg signed [width-1:0] d_tmp, d_d_tmp;



// Integrator stage registers


reg signed [width-1:0] d1=0;
reg signed [width-1:0] d2=0;
reg signed [width-1:0] d3=0;
reg signed [width-1:0] d4=0;
reg signed [width-1:0] d5=0;

// Comb stage registers


reg signed [width-1:0] d6=0, d_d6=0;
reg signed [width-1:0] d7=0, d_d7=0;
reg signed [width-1:0] d8=0, d_d8=0;
reg signed [width-1:0] d9=0, d_d9=0;
reg signed [width-1:0] d10=0;
reg [15:0] count=0;

reg v_comb;  // Valid signal for comb section running at output rate

reg d_clk_tmp;


    always @(posedge clk)
    begin
        if (rst)
        begin
            d1 <= 0;
            d2 <= 0;
            d3 <= 0;
            d4 <= 0;
            d5 <= 0;
            count <= 0;
        end else
        begin
            // Integrator section
            d1 <= d_in + d1;

            d2 <= d1 + d2;

            d3 <= d2 + d3;

            d4 <= d3 + d4;
```

```
                d5 <= d4 + d5;

                // Decimation

                if (count == decimation_ratio - 1)
                begin
                        count <= 16'b0;
                        d_tmp <= d5;
                        d_clk_tmp <= 1'b1;
                        v_comb <= 1'b1;
                end else if (count == decimation_ratio >> 1)
                begin
                        d_clk_tmp <= 1'b0;
                        count <= count + 16'd1;
                        v_comb <= 1'b0;
                end else
                begin
                        count <= count + 16'd1;
                        v_comb <= 1'b0;
                end
        end
end

always @(posedge clk)  // Comb section running at output rate
begin
        d_clk <= d_clk_tmp;
        if (rst)
        begin
                d6 <= 0;
                d7 <= 0;

                d8 <= 0;

                d9 <= 0;

                d10 <= 0;

                d_d6 <= 0;

                d_d7 <= 0;

                d_d8 <= 0;

                d_d9 <= 0;

                d_out <= 8'b0;

        end else

        begin

                if (v_comb)

                begin

                        // Comb section

                        d_d_tmp <= d_tmp;


                        d6 <= d_tmp - d_d_tmp;

                        d_d6 <= d6;


                        d7 <= d6 - d_d6;

                        d_d7 <= d7;


                        d8 <= d7 - d_d7;

                        d_d8 <= d8;


                        d9 <= d8 - d_d8;

                        d_d9 <= d9;


                        d10 <= d9 - d_d9;
```

```
                                        d_out <= d10 >>> (width - 8);
                                end
                        end
                end
        endmodule
```

```
module oversample16(
   input wire clk,
   input wire [11:0] sample,
   input wire eoc,
   output reg [13:0] oversample,
   output reg done
   );

   reg [3:0] counter = 0;
   reg [15:0] accumulator = 0;

   always @(posedge clk) begin
      done <= 0;
      if (eoc) begin
         // Conversion has ended and we can read a new sample
         if (&counter) begin // If counter is full (16 accumulated)
            // Get final total, divide by 4 with (very limited) rounding.
            oversample <= (accumulator + sample + 2'b10) >> 2;
            done <= 1;
            // Reset accumulator
            accumulator <= 0;
         end
         else begin
            // Else add to accumulator as usual
            accumulator <= accumulator + sample;
            done <= 0;
            end
            counter <= counter + 1;
        end
    end
endmodule
```

2. Raul

PSD Controller

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: MIT
// Engineer: Raul Largaespada
//
// Create Date: 11/19/2018 04:31:45 PM
// Design Name:
// Module Name: PSD_controller
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
```

```
module PSD_controller(
    input clock,
    input i_reset,
    input [5:0] i_ref,
    input [5:0] i_height,
    input [9:0] i_Kp_tune,
    input [9:0] i_Ki_tune,
    input [9:0] i_Kd_tune,
    input [9:0] i_bias_tune,
    output o_PSD_done,
    output signed [10:0] o_control_sig,
    output [9:0] o_test,
    output [9:0] o_test2,
    output [9:0] o_test3,
    output [9:0] o_test4
    );

    parameter s_IDLE = 0;
    parameter s_CALC_ERROR = 1;
    parameter s_CALC_PSD = 2;
    parameter s_CALC_OUTPUT = 3;
    parameter s_DONE = 4;
    reg [1:0] state_main = 3'b000;

    reg [5:0] r_ref, r_height;

    //error signal calculation module
    reg r_error_start;
    wire error_done;

    wire signed [6:0] error_signal;

    calc_error error(.clock(clock), .i_start_err(r_error_start), .i_ref_err(r_ref), .i_height_err(r_height),
.i_reset(i_reset),

        .o_done_err(error_done), .o_error(error_signal));


    //PSD term modules

    reg r_PSD_start;


    wire p_done;

    wire signed [8:0] prop_term;

    reg r_p_done;

    wire [9:0] ptest1;

    wire [9:0] ptest2;

    proportional_term prop_calc(.clock(clock), .i_error(error_signal), .i_start(r_PSD_start),
.i_Kp_tune(i_Kp_tune), .i_reset(i_reset),

        .o_p_done(p_done), .o_prop_term(prop_term), .test(ptest1), .test2(ptest2));


    wire s_done;

    wire signed [8:0] s_term;

    reg r_s_done;

    sum_term sum_calc(.clock(clock), .i_error(error_signal), .i_start(r_PSD_start), .i_Ks_tune(i_Ks_tune),
.i_reset(i_reset),

          .o_s_done(s_done), .o_s_term(s_term));


    wire d_done;

    wire signed [8:0] del_term;

    reg r_d_done;

    delta_term delta_calc(.clock(clock), .i_error(error_signal), .i_start(r_PSD_start),
.i_Kd_tune(i_Kd_tune), .i_reset(i_reset),
        .o_d_done(d_done), .o_del_term(del_term));


    parameter BIAS_MIN = 0;

    parameter BIAS_MAX = 31;

    wire signed [9:0] bias;

    gain_tuner #(.OFFSET_BUFFER(17)) bias_point(.clock(clock), .i_reset(i_reset),.i_tune(i_bias_tune),
.i_offset_min(BIAS_MIN), .i_offset_max(BIAS_MAX),

    .o_offset(bias));


    //command signal calculation registers

    wire signed [10:0] control_signal;

    reg signed [10:0] r_control_signal;

    reg r_command_start;

    wire command_done;

    reg r_PSD_done;

    command_calc command(.clock(clock), .i_reset(i_reset), .i_start(r_command_start),

    .i_prop(prop_term), .i_sum(sum_term), .i_delta(del_term), .i_bias(bias),

      .o_u_done(command_done), .o_command(control_signal));


    always @(posedge clock) begin

      if (!i_reset) begin

        case (state_main)

          s_IDLE: begin

            r_error_start <= 0;

            r_PSD_done <= 0;

            //always update registers

            r_ref <= i_ref;

            r_height <= i_height;

            //if anything changed, run control loop
            if (r_ref != i_ref || r_height != i_height) begin
                state_main <= s_CALC_ERROR;
                r_error_start <= 1;
            end
            else state_main <= s_IDLE;
          end

          s_CALC_ERROR: begin
            //wait until error signal is calculated, then move to next state
            r_error_start <= 0;
            if (error_done) begin
                state_main <= s_CALC_PSD;
                r_PSD_start <= 1;
                end
            else state_main <= s_CALC_ERROR;
          end

          s_CALC_PSD: begin
            r_PSD_start <= 0;
            if (p_done) r_p_done <= 1'b1;
            if (s_done) r_s_done <= 1'b1;
            if (d_done) r_d_done <= 1'b1;
            if (r_p_done && r_s_done && r_d_done) begin
                state_main <= s_CALC_OUTPUT;
            end
            else state_main <= s_CALC_PSD;
          end

          s_CALC_OUTPUT: begin
```

```
                r_p_done <= 0;

                r_s_done <= 0;

                r_d_done <= 0;

                r_command_start <= 1;

                //r_control_signal <= prop_term + bias*10;// + prop_term;

                //r_control_signal <= prop_term + s_term + del_term;//prop_term+ del_term +
(bias*10);//prop_term + del_term + (bias*10);//+ s_term + del_term;

                state_main <= s_DONE;


              end


              s_DONE: begin

                r_command_start <= 1'b0;

                if (command_done) begin

                  r_PSD_done <= 1'b1;

                  state_main <= s_IDLE;

                end

                else state_main <= s_DONE;

              end


            endcase

          end

          else begin

            //r_control_signal <= 10'd0;

            r_ref <= 6'b0;

            r_height <= 6'b0;

            r_PSD_start <= 1'b0;

            r_error_start <= 1'b0;

            r_PSD_done <= 1'b0;
            r_p_done <= 1'b0;
            r_s_done <= 1'b0;
            r_d_done <= 1'b0;
            state_main <= 3'd0;
        end
    end

    assign o_control_sig = control_signal;
    assign o_PSD_done = r_PSD_done;
    assign o_test = prop_term;
    assign o_test2 = ptest2;
    assign o_test3 = del_term;


endmodule
```

calc_error

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: MIT
// Engineer: Raul Largaespada
//
// Create Date: 11/19/2018 05:23:51 PM
// Design Name:
// Module Name: calc_error
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
```

```
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module calc_error(
   input clock,
   input i_start_err,
   input i_reset,
   output o_done_err,
   output signed [6:0] o_error,
   input [5:0] i_ref_err,
   input [5:0] i_height_err
   );
   /*
   Uses a finite state machine to perform signed subtraction r[n]-h[n] to determine e[n],
   the error signal. r[n] and h[n] range from 0 to 63. Therefore e[n] has a range of -63 to
   63, represented by a 7-bit two's complement integer. Overflow is not possible and is not
   checked for. Bit size of error signal is equal to sizes of reference/height signals plus one.

   Terminology:
      r[n] = reference command signal
      h[n] = current levitator height
   */
```

```
//FSM

parameter s_IDLE = 0;

parameter s_CALC = 1;

parameter s_DONE = 2;

reg state_err = 2'b0;


//registers for error signal calc

reg signed [6:0] r_ref_err;

reg signed [6:0] r_height_err;

reg signed [6:0] r_error;

reg r_done_err;


always @(posedge clock) begin

    if (!i_reset) begin

        case(state_err)

            s_IDLE: begin

                //at start, load signed registers with positive signed vals for r and h

                if (i_start_err) begin

                    state_err <= s_CALC;

                    r_ref_err <= {1'b0, i_ref_err};

                    r_height_err <= {1'b0, i_height_err};

                end

                else state_err <= s_IDLE;

            end


            s_CALC: begin

                //calculate error signal thru signed subtraction, flash done signal

                r_error <= r_ref_err - r_height_err;
                r_done_err <= 1'b1;
                state_err <= s_DONE;
            end

            s_DONE:begin
                //stop done signal, return to idle
                r_done_err <= 1'b0;
                state_err <= s_IDLE;
            end

        endcase
      end
      else begin //reset everything
        r_error <= 7'b0000000;
        r_ref_err <= 7'b0000000;
        r_height_err <= 7'b0000000;
        state_err <= s_IDLE;
      end
    end

    assign o_done_err = r_done_err;
    assign o_error = r_error;

endmodule
```

proportional_term

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: MIT
// Engineer: Raul Largaespada
```

```
//
// Create Date: 11/19/2018 10:06:51 PM
// Design Name:
// Module Name: proportional_term
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module proportional_term(
   input clock,
   input signed [6:0] i_error,
   input i_start,
   input [9:0] i_Kp_tune,
   input i_reset,
   output o_p_done,
   output signed [8:0] o_prop_term,
   output [9:0] test,
   output [9:0] test2
   );
   /*
```

```
    Uses a FSM to calculate the proportional control term, done by multiplying the error signal by a gain
Kp.

    Input is a 7-bit signed integer, ranging from -63 to 63. Output is limited to a signed 8 bit term, and
ranges from

    -256 to 255. Calculations that result in over or underflow are set to the most extreme values instead.
Bit size of

    proportional term is limited to bit size of error signal plus one.


    Terminology:

        e[n]: error signal = i_error

        Kp: proportional gain = r_Kp

        p[n]: proportional term = r_prop_term


    Calculation:

        p[n] = Kp * e[n]


    */


    parameter s_IDLE = 2'b00;

    parameter s_CALC = 2'b01;

    parameter s_CALC_OVERFLOW = 2'b10;

    parameter s_DONE = 2'b11;

    reg [1:0] state_prop;


    //default value/range for Kp

    parameter signed KP_BASE = 6'd2;

    parameter signed KP_MAX = 6'd31;

    parameter signed KP_MIN = 6'd0;


    //Kp, ranges from -32 to 31, will keep positive
    reg signed [5:0] r_Kp = KP_BASE;


    //tuning offset and range
    //added to r_Kp to adjust gain value without resynthesizing
    parameter signed [5:0] TUNING_MAX = 6'd5;
    parameter signed [5:0] TUNING_MIN = -5;
    wire signed [7:0] Kp_offset;


    gain_tuner #(.OFFSET_BUFFER(51)) p_gain_tuner(.clock(clock), .i_reset(i_reset),.i_tune(i_Kp_tune),
.i_offset_min(TUNING_MIN), .i_offset_max(TUNING_MAX),
        .o_offset(Kp_offset));



    //raw multiplication value from Kp*e[n]. Large number of bits to prevent under/overflow. Will check
Nth bit
    //to determine if under/overflow occured, where N = bit size of output signal. N=8 by default.
    reg signed [20:0] r_prop_raw;
    parameter [10:0] overflow_bit = 11'd8;


    reg signed [8:0] r_prop_term;
    parameter signed [8:0] PROP_MAX = 9'd255;
    parameter signed [8:0] PROP_MIN = -256;
    reg r_p_done;


    always @(posedge clock) begin
        if (!i_reset) begin
            case (state_prop)
                s_IDLE: begin
                    //when idle, offset Kp with current value, wait for start signal
```

```
          //check if tuning will cause over/underflow


          if (KP_BASE  < -Kp_offset ) r_Kp <= KP_MIN; //if true, Kp would be negative, force to be
positive

          else if ((Kp_offset > 0) && KP_BASE + Kp_offset < 0) r_Kp <= KP_MAX; //if both true, Kp
overflowed

          else r_Kp <= KP_BASE + Kp_offset; //no over/underflow, normal addition


          //wait for start signal to calculate values

          if (i_start) state_prop <= s_CALC;

          else state_prop <= s_IDLE;

        end


        s_CALC: begin

          //calculate proportional term

          r_prop_raw <= r_Kp*i_error;

          state_prop <= s_CALC_OVERFLOW;

        end


        s_CALC_OVERFLOW: begin

          //check if propotional term under/over flows 8 bit size, flash done signal when set

          //Kp is always positive, only need to check if e[n] is positive or negative

          if (i_error > 0 && r_prop_raw[overflow_bit] == 1) r_prop_term <= PROP_MAX; //Kp , e[n] > 0,
p[n] < 0, overflow occured, set to max

          else if (i_error < 0 && r_prop_raw[overflow_bit] == 0 && r_prop_raw != 0) r_prop_term <=
PROP_MIN; //Kp, p[n] > 0, e[n] < 0. underflowed, set to min

          else r_prop_term <= r_prop_raw[8:0];

          r_p_done <= 1'b1;

          state_prop <= s_DONE;

        end
```

```
            s_DONE: begin
                r_p_done <= 1'b0;
                state_prop <= s_IDLE;
            end
        endcase
    end

    else begin
        state_prop <= s_IDLE;
        r_Kp <= KP_BASE;
        r_p_done <= 1'b0;
        r_prop_raw <= 0;
        r_prop_term <= 0;
    end

  end

  assign o_prop_term = r_prop_term;
  assign o_p_done = r_p_done;
  assign test = Kp_offset;
  assign test2 = r_Kp;

endmodule
```

delta_term

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: MIT
// Engineer: Raul Largaespada
```

```
//
// Create Date: 11/20/2018 06:11:47 PM
// Design Name:
// Module Name: delta_term
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module delta_term(
   input clock,
   input signed [6:0] i_error,
   input i_start,
   input [9:0] i_Kd_tune,
   input i_reset,
   output o_d_done,
   output signed [8:0] o_del_term,
   output [9:0] test
   );
   /*
   Uses a FSM to generate the delta term of a PSD controller. As input takes a 7 bit signed
```

```
error signal ranging from -63 to 63, and outputs an 8 bit signed delta term ranging from -256
to 255. Calculations that would result in under/overflow are set to the most extreme values instead.

Calculation is done by multiplying the difference between the current and previous
error signals by the gain Kd and the sampling frequency 60Hz. Because multiplying by the sampling
frequency would increase the output by a factor of 60, instead we ignore the sampling frequency and
implicitely integrate it into the gain Kd. To compensate, Kd will need to be larger than in a comparable
PSD that does multiply by the sampling frequency.

Terminology:
    e[n]: error signal at time n = i_error
    Kd: delta gain
    difference[n]: e[n]-e[n-1] = r_difference
    d[n]: delta term = o_del_term

Calculation:
    d[n] = Kd * (e[n]-e[n-1])/(deltaT)
         = Kd * samplingFrequency * (e[n]-e[n-1])
*/

parameter s_IDLE = 3'd0;
parameter s_CALC = 3'd1;
parameter s_CALC_OVERFLOW = 3'd2;
parameter s_CALC2 = 3'd3;
parameter s_CALC_OVERFLOW2 = 3'd4;
parameter s_DONE = 3'd5;
reg [2:0] state_delta = 3'd0;

//Default values/range for delta gain Kd
```

```
  parameter signed [5:0] KD_BASE = 6'd2;

  parameter signed [5:0] KD_MAX = 6'd32;

  parameter signed [5:0] KD_MIN = 6'd0;


  //Kd, should be kept positive

  reg signed [5:0] r_Kd = KD_BASE;


  //tuning offset for Kd

  //added to r_Kp to adjust gain value without resynthesizing

  wire signed [7:0] Kd_offset;

  parameter signed [5:0] TUNING_MAX = 6'd15;

  parameter signed [5:0] TUNING_MIN = -5;


  gain_tuner #(.OFFSET_BUFFER(26)) d_gain_tuner(.clock(clock), .i_reset(i_reset),.i_tune(i_Kd_tune),
.i_offset_min(TUNING_MIN), .i_offset_max(TUNING_MAX),

    .o_offset(Kd_offset));


  //previous error signal, sampling frequency

  reg signed [8:0] r_difference; //current minus previous error signals

  reg signed [8:0] r_error_prev = 9'd0;

  reg signed [8:0] r_error_prev2 = 9'd0;

  reg signed [8:0] r_error_prev3 = 9'd0;

  reg signed [8:0] r_error_prev4 = 9'd0;

  parameter signed sampling_freq = 6'd60;

  parameter signed DIFFERENCE_MAX = 9'd255;

  parameter signed DIFFERENCE_MIN = -256;

  //add additional shift registers here for slower derivatives
```

```
    //raw multiplication value from Ks*sampling_freq*(e[n]-e[n-1]). Large number of bits to prevent
under/overflow.

    //Will check Nth bit to determine if under/overflow occured, where N = bit size of delta term. N=8 by
default.

   reg signed [20:0] r_del_raw;

   parameter [10:0] overflow_bit = 11'd8;


   //actual sum term with min/max range

   reg signed [8:0] r_del_term;

   parameter signed DELTA_MAX = 9'd255;

   parameter signed DELTA_MIN = -256;

   reg r_d_done;


   always @(posedge clock) begin

      if (!i_reset) begin

         case(state_delta)

            s_IDLE: begin

               //when idle, offset Ki with current value, wait for start signal

               //check if tuning will cause over/underflow

               if (KD_BASE  < -Kd_offset ) r_Kd <= KD_MIN; //if true, resulting Kd would be negative, force
to be one instead

               else if ((Kd_offset > 0) && KD_BASE + Kd_offset < 0) r_Kd <= KD_MAX; //if both true, Kd
overflowed

               else r_Kd <= KD_BASE + Kd_offset; //no over/underflow, normal addition


               //wait for start signal to calculate values

               if (i_start) state_delta <= s_CALC;

               else state_delta <= s_IDLE;

            end
```

```
        s_CALC: begin
            //state calculates e[n]-e[n-1]
            r_error_prev <= i_error;
            r_error_prev2 <= r_error_prev;
            r_error_prev3 <= r_error_prev2;
            r_error_prev4 <= r_error_prev3;
            //add additional shift registers here for slower derivatives
            r_difference <= (i_error - r_error_prev4);
            state_delta <= s_CALC_OVERFLOW;
        end

        s_CALC_OVERFLOW: begin
            //detects over/underflow in e[n]-e[n-1] calculation
            //probably not necessary given bit widths and input ranges but good for safety
            if ((i_error > 0 && r_error_prev < 0) && r_difference < 0) r_difference <= DIFFERENCE_MAX;
//overflow
            else if ((i_error < 0 && r_error_prev > 0) && r_difference > 0) r_difference <=
DIFFERENCE_MIN; //underflow
            //else keep difference the same
            state_delta <= s_CALC2;
        end

        s_CALC2: begin
            //multiplies difference by Kd and optionally by sampling_freq to get delta term
            r_del_raw <= r_Kd*r_difference;
            state_delta <= s_CALC_OVERFLOW2;
        end

        s_CALC_OVERFLOW2: begin
```

```
                //check if delta term under/over flows 8 bit size, flash done signal
                //Kd is always positive, only need to check if difference[n] is positive or negative
                if (r_difference > 0 && r_del_raw[overflow_bit] == 1) r_del_term <= DELTA_MAX;
                //difference[n] > 0, d[n] < 0, overflow occured, set to max
                else if (r_difference < 0 && r_del_raw[overflow_bit] == 0 && r_del_raw != 0) r_del_term <=
DELTA_MIN;
                //d[n] > 0, difference[n] < 0. underflowed, set to min
                else r_del_term <= r_del_raw[8:0]; //else keep as is
                r_d_done <= 1'b1;
                state_delta <= s_DONE;
            end

            s_DONE: begin
                r_d_done <= 1'b0;
                state_delta <= s_IDLE;
            end

        endcase

    end

    else begin
        r_difference <= 8'd0;
        r_error_prev <= 7'd0;
        r_del_raw <= 20'd0;
        r_Kd <= KD_BASE;
        r_d_done <= 1'b0;
        r_del_term <= 8'd0;
        state_delta <= s_IDLE;
        end
    end

    assign o_del_term = r_del_term;
    assign o_d_done = r_d_done;

endmodule
```

sum_term

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: MIT
// Engineer: Raul Largaespada
//
// Create Date: 11/20/2018 02:12:20 PM
// Design Name:
// Module Name: sum_term
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
```

```
module sum_term(
    input clock,
    input signed [6:0] i_error,
    input i_start,
    input [9:0] i_Ks_tune,
    input i_reset,
    output o_s_done,
    output signed [8:0] o_s_term,
    output [9:0] test
    );
    /*
    Uses a FSM to calculate the sum term of the PSD controller, by summing all previous error terms and
    multiplying by the sampling time a gain Ki. Because the sampling time 1/60 s is a constant, we have
ignored
    it in these calculations and implicitely integrated it into Ki. This means that, to compensate, the gain Ki
will
    need to be smaller than in a comparable implementation which does multiply by the sampling time.

    Input is a 7-bit signed integer i_error, ranging from -63 to 63. Output is limited to an 8-bit signed
integer
    ranging from -256 to 255. Limiting the output width prevents integral windup.Calculations that would
result in
    under/overflow are set to the most extreme values instead.

    Terminology:
        e[n]: error signal = i_error
        Ki: sum gain = r_Ki
        total[n]: output from summation before being multiplied by Ki = r_total
        s[n]: sum term = o_s_term
    Calculation:
        s[n] = Ki * (e[n]+s[n-1])

    */

    parameter s_IDLE = 0;
    parameter s_CALC = 1;
    parameter s_CALC_OVERFLOW = 2;
    parameter s_CALC2 = 3;
    parameter s_CALC_OVERFLOW2 = 4;
    parameter s_DONE = 5;
    reg [2:0] state_sum;

    //default value for Ks, range for Ks
    parameter signed KI_BASE = 6'd1;
    parameter signed KI_MAX = 6'd31;
    parameter signed KI_MIN = 6'd0;

    //Ks register, must be positive
    reg signed [5:0] r_Ki = KI_BASE;

    //tuing offset, added to r_Ki to adjust gain value without resynthesizing
    //change to a wire when possible
    wire signed [7:0] Ki_offset;
    parameter signed [5:0] TUNING_MAX = 6'd5;
    parameter signed [5:0] TUNING_MIN = -5;

    gain_tuner #(.OFFSET_BUFFER(51)) s_gain_tuner(.clock(clock), .i_reset(i_reset),.i_tune(i_Ks_tune),
.i_offset_min(TUNING_MIN), .i_offset_max(TUNING_MAX),
    .o_offset(Ki_offset));


    //summed total of all error inputs

    reg signed [8:0] r_total = 9'd0;

    reg signed [8:0] r_total_prev = 9'd0;

    parameter signed [8:0] TOTAL_MAX = 9'd255;

    parameter signed [8:0] TOTAL_MIN = -256;


    //raw multiplication value from Ki*total[n]. Large number of bits to prevent under/overflow. Will
check Nth bit

    //to determine if under/overflow occured, where N = bit size of sum term. N=8 by default.

    reg signed [20:0] r_sum_raw;

    parameter [10:0] overflow_bit = 11'd8;


    //actual sum term, total error multiplied by Ki

    reg signed [8:0] r_sum_term;

    parameter signed [8:0] SUM_MAX = 9'd255;

    parameter signed [8:0] SUM_MIN = -256;

    reg r_s_done;



    always @(posedge clock) begin

        if (!i_reset) begin

            case (state_sum)


                s_IDLE: begin

                    //when idle, offset Ki with current value, wait for start signal

                    //check if tuning will cause over/underflow
                    if (KI_BASE  < -Ki_offset ) r_Ki <= KI_MIN; //if true, resulting Ki would be negative, force to be
one instead
                    else if ((Ki_offset > 0) && KI_BASE + Ki_offset < 0) r_Ki <= KI_MAX; //if both true, Ki
overflowed
                    else r_Ki <= KI_BASE + Ki_offset; //no over/underflow, normal addition


                    //wait for start signal to calculate values
                    if (i_start) state_sum <= s_CALC;
                    else state_sum <= s_IDLE;
                end


                s_CALC: begin
                    //state is only meant to calculate new total error value before moving to over/underflow
detection
                    r_total_prev <= r_total;
                    r_total <= r_total + i_error;
                    state_sum <= s_CALC_OVERFLOW;
                end


                s_CALC_OVERFLOW: begin
                    //state calculates if total over/underflowed, sets to max value otherwise
                    //range limitations prevent integer windup
                    if ((i_error > 0 && r_total_prev > 0) && r_total < 0) r_total <= TOTAL_MAX;// two positive
operands with negative sum, overflow occured
                    else if ((i_error < 0 && r_total_prev < 0) && r_total > 0) r_total <= TOTAL_MIN; //2 negative
operands with positive sum, underflow occured
                    //else, keep r_total the same
                    //move to next state
                    state_sum <= s_CALC2;
                end
```

```
            s_CALC2: begin
               //r_total is multiplied by sum gain Ki
               r_sum_raw <= r_Ki * r_total;
               state_sum <= s_CALC_OVERFLOW2;
            end


            s_CALC_OVERFLOW2: begin
               //check if sum term under/over flows 8 bit size, flash done signal
               //Ks is always positive, only need to check if total[n] is positive or negative
               if (r_total > 0 && r_sum_raw[overflow_bit] == 1) r_sum_term <= SUM_MAX; //total[n] > 0,
s[n] < 0, overflow occured, set to max
               else if (r_total < 0 && r_sum_raw[overflow_bit] == 0 && r_sum_raw != 0) r_sum_term <=
SUM_MIN; //s[n] > 0, total[n] < 0. underflowed, set to min
               else r_sum_term <= r_sum_raw[8:0];
               r_s_done <= 1'b1;
               state_sum <= s_DONE;
            end


            s_DONE: begin
               r_s_done <= 0;
               state_sum <= s_IDLE;
            end


         endcase
      end


      else begin
         r_total <= 8'd0;
```

```
            r_total_prev <= 8'd0;
            r_Ki <= KI_BASE;
            r_s_done <= 1'b0;
            r_sum_raw <= 0;
            r_sum_term <= 0;
            state_sum <= s_IDLE;
        end
    end

    assign o_s_done = r_s_done;
    assign o_s_term = r_sum_term;


endmodule
```

command_calc

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11/20/2018 09:28:14 PM
// Design Name:
// Module Name: command_calc
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
```

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module command_calc(
   input clock,
   input i_start,
   input i_reset,
   input signed [8:0] i_prop,
   input signed [8:0] i_sum,
   input signed [8:0] i_delta,
   input signed [9:0] i_bias,
   output signed [10:0] o_command,
   output o_u_done
   );
   /*
   This module sums the proportional, sum, delta, and bias terms to generate the command output u[n].
   */
   //parameter signed [9:0] U_MAX = 10'd511;
   //parameter signed [9:0] U_MIN = -512;


   parameter s_IDLE = 0;
   parameter s_CALC2 = 1;
   parameter s_CALC3 = 2;
   parameter s_DONE = 3;
```

```
reg [3:0] state_command;


reg signed [10:0] calc1, calc2, calc3;

reg signed [10:0] r_bias_calc;

reg r_command_done;



always @(posedge clock) begin

  if (!i_reset) begin

    case(state_command)

      s_IDLE: begin

        r_command_done <= 0;

        if (i_start) begin

          calc1 <= i_prop + i_delta;

          r_bias_calc <= i_bias*10;

          state_command <= s_CALC2;

        end

        else state_command <= s_IDLE;

      end

      s_CALC2: begin

        calc2 <= calc1 + r_bias_calc;


        state_command <= s_IDLE;

      end

      s_CALC3: begin

        calc3 <= calc2 + i_sum;

        r_command_done <= 1;

        state_command <= s_DONE;

      end
```

```
            s_DONE: begin
                r_command_done <= 0;
                state_command <= s_IDLE;
            end
            default: state_command <= s_IDLE;
        endcase
    end
    else begin
        calc1 <= 0;
        calc2 <= 0;
        calc3 <= 0;
        r_command_done <= 0;
        state_command <= s_IDLE;
    end
end

assign o_command = calc2;
assign o_u_done = r_command_done;

endmodule
```

gain_tuner

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: MIT
// Engineer: Raul Largaespada
//
// Create Date: 11/27/2018 03:43:31 PM
// Design Name:
// Module Name: gain_tuner
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module gain_tuner(
    input clock,
    input [9:0] i_tune,
    input i_reset,
    input signed [5:0] i_offset_min,
    input signed [5:0] i_offset_max,
    output signed [7:0] o_offset
    );

    /*
    This module takes in the 10-bit ADC value from a gain tuning potentiometer and converts it into a tuning offset value.
    This offset is then added to the hard-coded gain value so that it can be tuned without resynthesizing Verilog. As input,
    the module takes clock and reset wires, the minimum and maximum gain offset values, and the 10-bit ADC output.
    The module linearly scales the ADC value such that 0 on the ADC corresponds to the minimum gain
offset and 1023 corresponds

    to the maximum gain offset. Intermediate ADC values are converted to offset values using the
following function:


        offset = i_offset_min + floor((ADC+b) * d/1024)


    where d is the offset range d = (i_offset_max - i_offset_min) and b is a buffer value that allows the
system to reach the

    most extreme offset values without requiring an ADC value of exactly 0 or 1023. b is dependent on the
offset range, and an

    optimal b is calculated as follows:


        b = floor((((d+1)/d)*1024 - 1023)/2)


    b is not calculated in the program and is instead calculated by the user and entered as a parameter, to
avoid the use of division.

    When changing the minimum and maximum offset values, a different b will have to be calculated each
time. By default it is set

    to 25, which is a healthy value for many offset ranges.


    The calculated offset is added to the gain in the appropriate proportional, sum,or delta term module.
Instead of dividing by 1024

    and flooring, the Verilog simply right shifts by 10 bits. A register chain is used so that the calculation
can be completed without

    propagation delay errors.

    */


    reg signed [6:0] r_offset_range; //d in equations above

    reg signed [7:0] r_offset;

    reg signed [10:0] r_tune;
```

```
//registers for proper clocking

reg signed [11:0] r1;
reg signed [17:0] r2;
reg signed [7:0] r3;


//provides a buffer so that the maximum offset can be hit without an ADC output of exactly 1023
parameter signed [9:0] OFFSET_BUFFER = 8'd25; //b
//use equation for optimal buffer defined above
//default set to 25, should be greater than 0

always @(posedge clock) begin
    if (!i_reset) begin
        r_offset_range <= i_offset_max - i_offset_min;
        r_tune <= {1'b0, ~i_tune};//because the ADC output is unsigned, converts to positive signed
        //inversion of i_tune means that turning potentiometer CW increases value instead of opposite

        r1 <= r_tune + OFFSET_BUFFER;
        r2 <= r1 * r_offset_range;
        r3 <= r2>>4'd10;
        r_offset <= i_offset_min + r3;
    end
    else begin
        r_offset <= 0;
        r1 <= 0;
        r2 <= 0;
        r3 <= 0;
    end
    end

    assign o_offset = r_offset;

endmodule
```

pwm_generator

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: MIT
// Engineer: Raul Largaespada
//
// Create Date: 11/26/2018 03:50:14 PM
// Design Name:
// Module Name: pwm_generator
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module pwm_generator(
    input clock,

    input signed [10:0] i_control,

    input i_reset,

    output o_pwm

    );

    /*

    This module creates a PWM signal switching betwen 0 to 3.3 volts. The frequency of the signal is
defined in the PWM_FREQ parameter.

    The signal implementation is based on a pair of counters. r_counter_PWM increments up to
COMMAND_MAX-1 = 765-1 = 764. COMMAND_MAX

    is the control signal input value that corresponds to a 100% duty cycle PWM. By default it is set to
765, the maximum possible

    value of the control signal (occurs when the proportional, sum, and delta modules each output their
maximum value of 255. 255*3 = 765).

    Lowering or increasing this value will change how easily the controller output is able to generate a
large duty cycle PWM.


    r_counter_PWM increments every (T_PWM)/(COMMAND_MAX) seconds, where T_PWM is the
period of the PWM signal, by default set to

    1/1000hz.This corresponds to approximately CLK_MAX = 85 clock cycles when the clock is running at
65mhz. The equations used to

    calculate CLK_MAX are as follows, where T_65 is the period of the 65mhz clock.

        CLK_MAX = (T_PWM/COMMAND_MAX)/(T_65) = (65mhz)/(PWM_FREQ*COMMAND_MAX) =
84.967 with default values


    Rounding CLK_MAX to 85 gives the parameter value. r_counter_PWM is set to increment every
CLK_MAX clock cycles, creating a PWM

    frequency of approximately 1khz. r_counter_clk keeps track of these CLK_MAX clock cycles, resetting
at 85. If r_counter_PWM is less than the

    control input, the PWM signal will be high. Otherwise it will be low. This means that a negative
controller output or

    a controller output of 0 will give a 0% duty cycle PWM. A controller output of COMMAND_MAX/2 is
approximately a 50% duty cycle, and a
```

```
    controller output of COMMAND_MAX is a 100% duty cycle. r_counter_PWM resets at
COMMAND_MAX-1.


    Because the value of CLK_MAX is hard coded (to avoid completing a division operation), CHANGING
COMMAND_MAX REQUIRES RECOMPUTING THE

    VALUE OF CLK_MAX.


    */


    //set up PWM paramters/counters

    parameter [10:0] PWM_FREQ = 10'd1000; //frequency of PWM signal

    parameter signed [11:0] COMMAND_MAX = 10'd600; //command signal value that corresponds to
100% duty cycle

    parameter [8:0] CLK_MAX = 9'd108; //(65mhz/(PWM_FREQ*COMMAND_MAX), number of clock
cycles in T_PWM/COMMAND_MAX seconds

        //CLK_MAX is the number of clock cycles of the 65mhz clock for counter PWM to increment once


    reg signed [11:0] r_counter_PWM;

    reg [6:0] r_counter_clk;

    reg r_PWM;


    always @(posedge clock) begin

        if (!i_reset) begin

            //if (i_control > 0) begin

                if (r_counter_clk == (CLK_MAX)) begin

                    r_counter_PWM <= r_counter_PWM + 1;

                    r_counter_clk <= 7'd0;

                end

                else r_counter_clk <= r_counter_clk + 1;
        if (r_counter_PWM == COMMAND_MAX-1) r_counter_PWM <= 10'd0;

        if (r_counter_PWM < i_control && i_control > 0) r_PWM <= 1'b1;
        else r_PWM <= 1'b0;
      //end
      //else r_PWM <= 0;
    end

    else begin
      r_counter_PWM <= 10'd0;
      r_counter_clk <= 7'b0;
      r_PWM <= 1'b0;
    end
  end

  assign o_pwm = r_PWM;

endmodule
```

ADC

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: MIT
// Engineer: Raul Largaespada
//
// Create Date: 11/29/2018 05:01:11 PM
// Design Name:
// Module Name: ADC
// Project Name:
// Target Devices:
```

```
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module ADC(input sysclk,
   input ADC_start,
   /*
   input [1:0]btn,   // Button inputs
   output [1:0]led,  // Led outputs
   output led0_b, //blue led output
   output led0_g, //green led output
   output led0_r, //red led output
   inout [3:0]g, //inouts (see xdc file)
   */

   input miso,
   output mosi, sck, cs,

   output [9:0] o_channel0,
   output [9:0] o_channel1,
   output [9:0] o_channel2,
   output [9:0] o_channel3,
   output [9:0] o_channel4

   //inout [1:0]pio

   );

   /*

   This module uses a FSM to read inputs and outputs from an MCP3008 ADC. It is based on code
originally written by Joe Steinmeyer (jodalyst) and

   used with his permission. The FSM starts by reading channel 0 of the ADC, and reads channels 1, 2,
and 3 before looping back to IDLE. The

   frequency of the input clock is meant to be around 3MHz, a good value for an MCP3008 running at 5V
VDD. Large sections of commented out code

   are unused leftovers from Joe's original code.

   */


   //wire cs, mosi,miso,sck; //chip select, si, so, sck

   parameter commwidth = 17;


   //assign g[3:0] = {sck,miso,mosi,cs}; //link pins to wires with understandable names

   //assign g[3:0] = {cs, mosi, miso,sck};


   /*

   //some random heartbeat for LEDs to know it is uploading

   assign led[1] = sysclk;

   //led flashing "heartbeat" code...


   reg [23:0] r;

   always @(posedge sysclk)

      begin

         if(btn[0] == 1)

            r <= 0;

         else
```

```
        r <= r + 1;
      //if (&r) brightness<=8'd127;
   end
assign led0_r =1'b1; //r[20]?1'b1:1'b0;   //6hz 25% pwm
assign led0_g =1'b1; //r[21]?1'b1:1'b0;   //12hz
assign led0_b =r[22]?1'b1:1'b0;  //3Hz
assign led[0] = r[23];
//assign pio[8] = r[23];
//done
reg [7:0] brightness;
//dimmer dm(.clock(sysclk), .brightness(brightness), .driver(pio[0]));

assign pio[1] = 1'b0;
*/

//values from first four channels
reg [9:0] r_channel0;
reg [9:0] r_channel1;
reg [9:0] r_channel2;
reg [9:0] r_channel3;
reg [9:0] r_channel4;
reg [2:0] r_channel_select = 3'b000;

//wires/regs for interacting with spi_master
wire [commwidth-1:0] data_received;
wire new_data; //new data is present!
wire spi_busy;
reg trigger; //used to trigger spi;
reg [2:0] selection; //which device to pick
```

```
    reg [15:0] bytes_to_send; //number of bytes to send

    reg [commwidth-1:0] data_to_send; //data to send out

    reg rst;

    wire [7:0] chip_selects; ///chip selects

    assign cs = chip_selects[0]; //tft chip select


    spi_master #(.INOUTWIDTH(commwidth))
  spm(.sysclk(sysclk),.ss(selection),.data_to_send(data_to_send),

    .how_many_bytes(bytes_to_send), .new_data(new_data), .cs(chip_selects), .data_in(data_received),

    .mosi(mosi),

    .miso(miso),

    .sck(sck),

    .rst(rst),

    .busy(spi_busy),

    .trigger(trigger));


    reg [3:0] state;

    localparam IDLE = 4'h0;

    localparam T1 = 4'h1;

    localparam RW1 = 4'h2;

    localparam READ_NEXT_CHANNEL = 4'h3;


    /*

    localparam READ1 = 4'h3;

    localparam START2 = 4'h4;

    localparam RUN2 = 4'h5;

    localparam PAUSE2 = 4'h6;

    localparam START3 = 4'h7;

    localparam RUN3 = 4'h8;
```

```
localparam PAUSE3 = 4'h9;
localparam START4 = 4'hA;
localparam RUN4 = 4'hB;
localparam PAUSE4 = 4'hC;
*/

always @(posedge sysclk)begin
   case(state)
      IDLE: begin
         if(ADC_start)begin
            //set up for reading channel 0
            r_channel_select <= 3'b000; //start at channel 0
            rst <= 1'b0;
            selection <= 3'b0; //pick device 0
            bytes_to_send <= 16'd1; //send two bytes and read one byte
            data_to_send <= 17'b11_000_0000_0000_0000; //start, SGL, ch0
            state <= T1;
         end else begin
            trigger <= 1'b0;
            r_channel_select <= 3'b000; //start at channel 0
            state <= IDLE;
         end
      end
      //trigger SPI data read
      T1: begin
         trigger<=1'b1;
         if (~new_data && spi_busy) begin
            state <= RW1;
         end
```

```
 //state <= RW1;
end
RW1:begin
 trigger <=1'b0;
 if (new_data)begin
  case (r_channel_select)
   //load data into current channel, set up to read next channel
   0: begin
    r_channel0 <= data_received[9:0];
    r_channel_select <= 3'b001;
    state <= READ_NEXT_CHANNEL;
   end
   1: begin
    r_channel1 <= data_received[9:0];
    r_channel_select <= 3'b010;
    state <= READ_NEXT_CHANNEL;
   end
   2: begin
    r_channel2 <= data_received[9:0];
    r_channel_select <= 3'b011;
    state <= READ_NEXT_CHANNEL;
   end
   3: begin
    r_channel3 <= data_received[9:0];
    r_channel_select <= 3'b100;
    state <= READ_NEXT_CHANNEL;
   end
   4: begin
    r_channel4 <= data_received[9:0];
```

```
                r_channel_select <= 3'b00;//all five channels read, go back to idle
                state <= IDLE;
            end
            default:
                state <= IDLE; //if there's a weird error just go back to idle
        endcase

    end
end
//set up to read channel 1, 2, or 3
READ_NEXT_CHANNEL: begin
    rst <= 1'b0;
    selection <= 3'b0; //pick device 0
    bytes_to_send <= 16'd1; //send two bytes and read one byte
    case (r_channel_select)
        0:
            data_to_send <= 17'b11_000_0000_0000_0000; //start, SGL, ch0, in this state this case
should never occur since we're reading channel 1/2/3
        1:
            data_to_send <= 17'b11_001_0000_0000_0000; //start, SGL, ch1
        2:
            data_to_send <= 17'b11_010_0000_0000_0000; //start, SGL, ch2
        3:
            data_to_send <= 17'b11_011_0000_0000_0000; //start, SGL, ch3
        4:
            data_to_send <= 17'b11_100_0000_0000_0000; //start, SGL, ch4
        default:
            state <= IDLE;  //if there's a weird error just go back to idle
    endcase
```

```
            state <= T1;
          end

          default:
            state <= IDLE;
        endcase
    end



    assign o_channel0 = r_channel0;
    assign o_channel1 = r_channel1;
    assign o_channel2 = r_channel2;
    assign o_channel3 = r_channel3;
    assign o_channel4 = r_channel4;

    ///logic analyzer below:
    //ila_0 myila(.clk(sysclk),.probe0(sck),
    // .probe1(si), .probe2(so),.probe3(ccs),
    // .probe4(new_data), .probe5(trigger));
endmodule
```

spi_master

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: jodalyst
// Engineer: jodalyst
//
// Create Date: 10/06/2017 07:38:04 PM
// Design Name: SPI_master module for MCP3008 (experimental)
// Module Name: spi_master
// Project Name:  SPI_demo
// Target Devices:  Artix-7 on CMOD A7-35T by Digilent
// Tool Versions:  Vivado
// Description:
//
// Dependencies:  On CMOD by Digilent this thing needs a faster clock than what it comes with
// in order to be able to run in worthwhile SPI clock domains, so you'll need to use a clock
// multiplier from
//
// Revision:
// Revision 0.03 core functionality working
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


//Version 2...you need your sysclk to be 2 times SCLK speed!
module spi_master #(parameter INOUTWIDTH = 24)
   (input sysclk,
   input [2:0] ss, //for selecting slaves from input side
   input [INOUTWIDTH-1:0] data_to_send,
   input [15:0] how_many_bytes, //if we want repeated reading...or writing..how long of a continuous
are we going to deal with
   input miso,
   output reg sck,
   output reg mosi,
   output reg [7:0] cs, //for selecting slaves on output side (one hot wiring)
   output reg [INOUTWIDTH-1:0] data_in,
   output reg busy,
   output reg new_data,
   output reg load,

   input rst,

   input trigger //active high

   );



   reg [INOUTWIDTH-1:0] buffer_in; //buffer for data to be received from slave (and sent "out" to user)


   reg [INOUTWIDTH-1:0] buffer_out; //"buffer for data to be sent to slave

   //output reg [7:0] buffer_in; //buffer for data to be received from slave (and sent "out" to user)

   localparam IDLE = 4'h0,

      PRERUN1 = 4'h1,

      PRERUN2 = 4'h2,

      RUN = 4'h3,

      FINISH = 4'h4;


   reg [15:0] bytes_to_run;

   reg [15:0] byte_count;

   reg [4:0] state;

   reg [8:0] count; //allows supporting INOUTWIDTH of up to 256 bits



   always @(posedge sysclk) begin

      if (rst)begin

         state <= IDLE;//simply return to idle..abandon all hope

      end else begin

           case (state)

                IDLE: begin

                    sck <= 1'b0; //always assume we are this in IDLE!
           if (trigger)begin
               buffer_out <= data_to_send;
               buffer_in<= 8'b0;
               bytes_to_run <= how_many_bytes;
               cs <= ~(8'b1<<(ss)); //pull sel down, leave others up
               data_in <= 0; //empty output register
               count <= 8'b0; //reset count
               state <= PRERUN1; //move onto PRERUN
               busy <= 1'b1;
               new_data <= 1'b0;
               byte_count <= 16'b0;
               load <=1'b0;
           end else begin
               cs <= ~(8'b0);  //all high in rest state
               mosi <= 1'b0;  //all low in rest state
               busy <= 1'b0;//low for busy
               load <= 1;
               new_data <= 1'b0;
               state <= IDLE;
               //and remain in IDLE
           end
       end
       PRERUN1: begin
           sck <= 1'b0; //register
           buffer_out <= {buffer_out[INOUTWIDTH-2:0],1'b0}; //push new data in now that both
   sides have measured
           mosi <= buffer_out[INOUTWIDTH-1]; //new value on mosi
           count <= count +1;
           state <= PRERUN2;
```

```
                end
                PRERUN2: begin
                    sck <= 1'b1;
                    state <= RUN;
                end
                RUN: begin
                    if (sck)begin //about to be on rising edge!
                       buffer_in <= {buffer_in[INOUTWIDTH-1:0],miso};//take a measurement and shove in
                        if (count == INOUTWIDTH)begin //we've read 8 bits in...time to decide are we done or
keep goin!
                           new_data <= 1'b1;  //set the new data flag!
                           data_in <= {buffer_in[INOUTWIDTH-1:0],miso}; //load the data_in with what is in
buffer_in (from the slave)
                           if (byte_count +1'b1 == bytes_to_run)begin  //we done
                             sck <= 1'b0; //clock can shut off.
                             state <= FINISH; //we've run the number of bits we needed!
                           end
                           else begin
                                buffer_out <= {data_to_send[INOUTWIDTH-2:0],1'b0}; //grab fresh set of data
                                mosi <= data_to_send[INOUTWIDTH-1]; //new value on mosi
                                count <= 8'b1;
                                byte_count <= byte_count +1'b1; //one more byte!
                                state <= RUN; //not needed, but for clarity
                                sck <= ~sck; //keep going, child.
                             end
                          end
                         else begin
                             buffer_out <= {buffer_out[INOUTWIDTH-2:0],1'b0}; //push new data in now that
both sides have measured
                             mosi <= buffer_out[INOUTWIDTH-1]; //new value on mosi
```

```
                            sck <= ~sck; //keep clock on
                            count <= count +1;
                            new_data <= 1'b0; //deassert new_data (usually keeps at 0...coming from a
                        end
                    end
                    else begin
                        sck= ~sck;
                    end
                end
                FINISH: begin
                    cs <= ~(8'b0);
                    state <= IDLE;
                end
                default: begin
                    state <= IDLE;
                end
            endcase
        end
    end
endmodule

//minimal only in charge of
/*
module spi_write_1_byte(input clock, input
    output reg dev_trigger,
    output reg done);
    reg [3:0] state;
    localparam IDLE = 4'h0,
    localparam START1 = 4'h1;
    localparam RUN1

    always @(posedge fastclk)begin

        if (clock_25mhz)begin

            case(state)

                IDLE: begin

                  dc <= 1'b1;

                  trigger <= 1'b0;

                  rst <=1 1'b0;

                  if(btn[1] == 1)begin

                    selection <= 3'b0; //pick device 0

                    bytes_to_send <= 16'b1; //send one byte

                    data_to_send <= SWRESET;

                    state <= START1;

                end

                START1: begin

                  trigger<=1'b1;

                  state <= RUN1

                  pause_counter <= 24'b0;

                end

                RUN1:begin

                  trigger <=1'b0;

                  if (data

                end

                PAUSE1:begin

                  if (&pause_counter)begin

                    state <= START2;

                    data_to_send <= SLPOUT;

                  end else begin

                    pause_counter <= pause_counter +1;
```

```
                    end
                end
                START2: begin
                    trigger <=1 1'b1;
                    state <= RUN2;
            endcase
        end
    end
endmodule
*/
```

IR_sensor

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 12/08/2018 12:08:43 PM
// Design Name:
// Module Name: IR_Sensor
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module IR_Sensor(
   input clock,
   input [9:0] i_IR,
   output [5:0] o_height
   );
   /*
   Module takes the 10 bit IR sensor value from the ADC and converts it to a CM hieght, and then to the
appropriate
   6 bit reference signal for the controller.
   */

   parameter signed alpha1 = -6;
   parameter signed alpha2 = -11;
   parameter signed alpha3 = -27;
   parameter signed alpha4 = -57;

   parameter signed gamma1 = 34;
   parameter signed gamma2 = 47;
   parameter signed gamma3 = 77;
   parameter signed gamma4 = 118;

   reg signed [11:0] r_IR_signed;
   reg signed [16:0] r_d_calc1;//times alpha
   reg signed [8:0] r_d_calc2;//divided by 256
   reg signed [8:0] r_d_calc3;//add gamma
   reg signed [8:0] r_height_cm;//subtract from 60

   reg [8:0] r_height_ref1;
   reg [16:0] r_height_ref2;
   reg [5:0] r_height_ref3;

   parameter beta = 234;

   always @(posedge clock) begin
     r_IR_signed <= {1'b0, i_IR};

     if (r_IR_signed >= 10'd669) begin
       r_d_calc1 <= r_IR_signed * alpha1;
       r_d_calc2 <= r_d_calc1>>8;
       r_d_calc3 <= r_d_calc2 + gamma1;//positive after this addition
       r_height_cm <= 60 - r_d_calc3;
     end

     else if (r_IR_signed < 10'd669 && r_IR_signed >= 10'd481) begin
       r_d_calc1 <= r_IR_signed * alpha2;
       r_d_calc2 <= r_d_calc1>>8;
       r_d_calc3 <= r_d_calc2 + gamma2;//positive after this addition
       r_height_cm <= 60 - r_d_calc3;
     end

     else if (r_IR_signed < 10'd481 && r_IR_signed >= 10'd355) begin
       r_d_calc1 <= r_IR_signed * alpha3;
       r_d_calc2 <= r_d_calc1>>8;
       r_d_calc3 <= r_d_calc2 + gamma3;//positive after this addition
```

```
            r_height_cm <= 60 - r_d_calc3;
        end

        else if (r_IR_signed < 10'd355) begin
            r_d_calc1 <= r_IR_signed * alpha4;
            r_d_calc2 <= r_d_calc1>>8;
            r_d_calc3 <= r_d_calc2 + gamma4;//positive after this addition
            r_height_cm <= 60 - r_d_calc3;
        end

        //at this point we have the height in cm at r_height cm
        //converting 10-50 cm height to 6 bit reference signal
        r_height_ref1 <= r_height_cm[7:0] - 15;//
        r_height_ref2 <= r_height_ref1 * beta;
        r_height_ref3 <= r_height_ref2>>7;


    end

    assign o_height = r_height_ref3;

endmodule
```