Metal Detecting Robot

Junior Design Project

Team 11

Michael Fevola (EE)

Milton Pereza (COE)

Ryan Smith (COE)

Viktor Wahlquist (EE)

Submitted May 10th, 2019

II. Abstract

Our team was tasked with designing a single self-contained autonomous robotic system. Our design was required to navigate a 6' x 6' course and navigate around obstacles in the course. Moreover, there were 4 washers on the ground that the rover was required to detect. Our rover was designed to cover as much surface area of the box as possible within the three minutes given.

The main controller for the rover was the papilio duo hardware. More specifically, the Atmega32u4 microcontroller on the board was utilized as the main controller for navigation. The microcontroller was used to control the rovers movements and use ultrasonic sensors to drive its decisions. C language code was utilized on this platform. A Xilinx Spartan 6 FPGA was used to drive the motors and keep track of encoder and metal detector counts. On the FPGA side, shift registers were used to serially transmit values from uC to FPGA. This allowed for select signals to be used for the ultrasonic sensor echo mux and the motor output decoder. Therefore, our functions for setting movement and interfacing sensors were kept relatively straightforward and simple on the microcontroller side. VHDL was the chosen language for the FPGA.

The algorithm consisted of a two stage process. Stages are changed based on the number of encoder ticks counted. The first stage algorithm is line following. Our first stage was a line following algorithm that would alternate between right and left bias approximately every 16 feet. The idea was to try to cover as much perimeter area as possible and reach all four corners. The second stage consisted of randomized turns. At approximately 90 feet, our algorithm begins to cover the center part of the course. "Randomized" turns were implemented and the rover would turn randomly when any of the three sensors detected an obstacle or wall. This allows the rover to take random paths across the middle of the course to detect mines

While the rover is navigating through the course it is also tasked with detecting mines. The metal detector circuit was designed to reliably sense the metal mines from a reasonable distance while also outputting a DC voltage that represents a logical 1 or 0 for the FPGA. The most reliable design for the metal detection circuitry was a Colpitt's Oscillator. The output of the Oscillator was sinusoidal so an active low-pass filter was used to remove all sinusoidal components. The result from the filter was then plugged into an Analog Comparator, where a reference voltage determined when the Op Amp would output a low or high voltage. The end result was that when metal was present a logical 1 was sent to the FPGA and when metal was not present, a logical 0 was sent to the FPGA.

III. Table of Contents

IV. Project Overview

This project required knowledge of electronic circuits, signal processing, AVR-programming, and VHDL to complete. The team tasked with completing the project consisted of two electrical engineering majors and two computer engineering members. The goal was to use the Rover 5 robot to create an autonomous rover capable of counting the number of metal objects it encountered while simultaneously avoiding obstacles. In order to accomplish this goal, the team designed and constructed a metal detection circuit and designed an algorithm that used ultrasonic sensors to navigate an "arena" with obstacles in it.

The metal detection circuit was designed using LTSpice, a power supply, an Oscilloscope and various circuit elements. Through extensive research and testing, a final circuit design was established. It is composed of three main parts; A Colpitts oscillator circuit, an active low-pass filter, and an analog comparator. The Colpitts oscillator uses 9 V voltage source to create a sinusoidal wave with a certain DC RMS voltage. Whenever the metal is close to the inductor, the DC RMS output of the oscillator will decrease. The output of the oscillator is then put through the active low pass filter which filters out the sinusoidal component, leaving only the DC component. The DC component is later fed into the analog comparator that will output a logical 1 to the FPGA if the metal is close and a logical 0 otherwise. The final design was constructed on a breadboard after which it was tested and modified for better results. After it was established that the detection circuit could accurately detect the metal washers, the circuit output was connected to the FPGA, and the circuit itself was attached to the rover using a 3D-printed frame.

The rover could be started and stopped using an IR receiver interfaced with the microcontroller. The rover was programmed to use an interrupt service routine to interrupt the main algorithm and enter an idle state whenever the IR receiver detected a signal from an Infrared remote. If the signal was sent again, the rover would enter its active state. Once tested, the IR receiver circuit was connected to the microcontroller and attached to the top of the rover.

The focus was then shifted onto the autonomous navigation system. The movement of the rover was incorporated using the same methods as in previous laboratory activities (using a PWM signal and the provided motor board). Three ultrasonic sensors attached to the rover were used to detect obstacles. While the rover is moving, the encoder ticks are sent to the FPGA to count the number of feet it has traveled so far. It displayed this value on the seven segment display.

The main algorithm was programmed in C on the microcontroller and uses input from the sensors to determine how the rover will move to avoid crashing into the obstacles and walls. The rover essentially had two modes of operation. It was designed to follow the edge of the arena until it had traveled a total of 90 feet. Once it traveled 90 feet, the rover starts turning for a

predetermined amount of time anytime it detects an object close to one of the sensors. This allowed the rover to leave the edges and move across the middle of the arena instead.

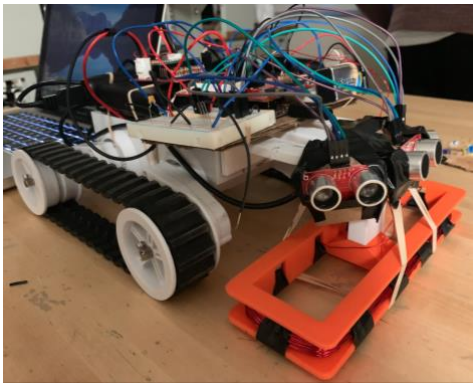


Fig [0] - Finished and assembled rover

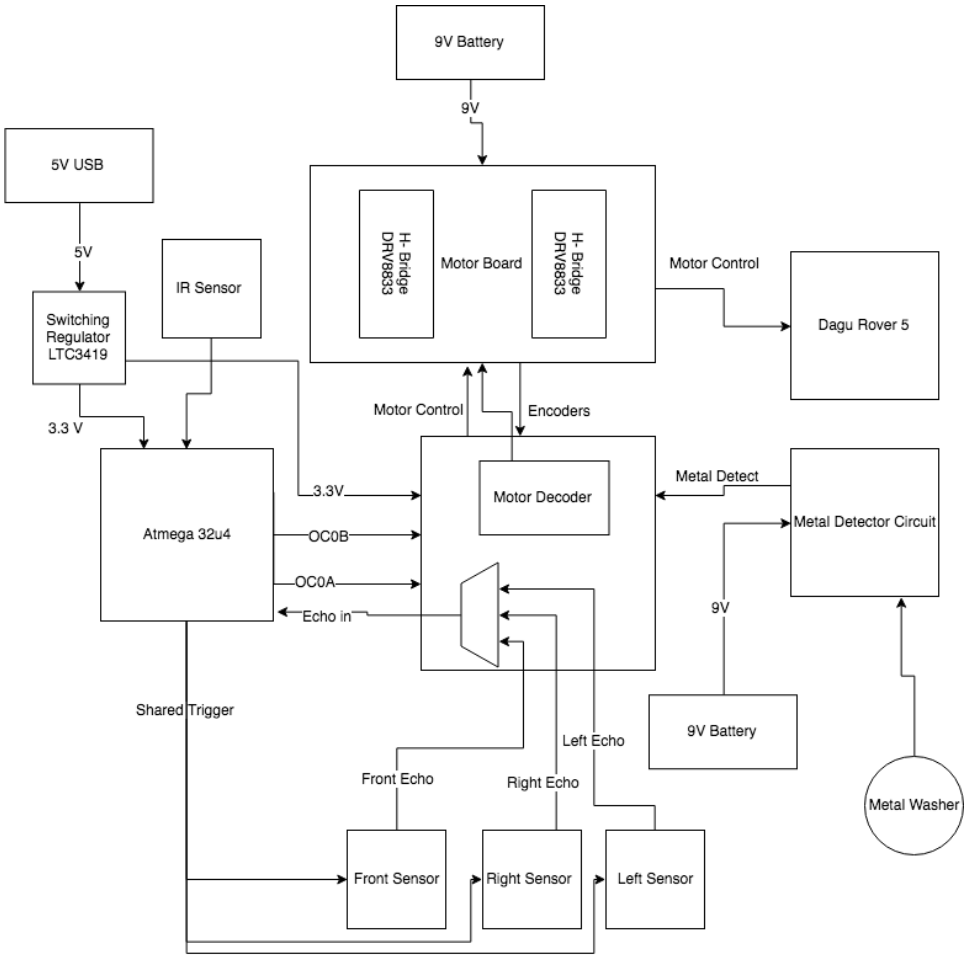


Fig [1] - Hardware Block diagram

V. Requirements

[Rover 1.1] The rover shall be a single self-contained autonomous robotic system

[Rover 1.2] The rover shall be less than 2 feet long and 2 feet wide

[Rover 1.3] The rover shall have the ability to be started or stopped at any point in time

[Rover 1.4] The rover shall use the Papilio Duo as the main controller

[Navigation 1.1] The rover shall navigate a 3’ by 3’ course in 3 or less minutes to find as many mines as possible

[Navigation 1.2] The rover shall not be designed to intentionally interfere with gameplay elements

[Display 1.1] The rover shall display the number of mines it has found

[Display 1.2] The rover shall display distance travelled by the robot in any unit selected

[Safety 1.1] The rover shall be designed for safety

VI. Verification of Requirements

[Rover 1.1]

The rover featured a plastic frame screwed into the top of the rover to hold the metal detector. The plastic frame also served as a mount for all three of the ultrasonic sensors. A cardboard frame was mounted on top of the plastic to hold accompanying circuitry, the Papilio DUO, a 9-Volt battery, and a rechargeable battery pack. The rover undercarriage held the final battery pack that plugged into the motorboard. Everything necessary to help the rover navigate the course and pick up mines was either attached or mounted to the top of the rover. The three ultrasonic sensors were used as interfaces via software to allow for autonomous movement and obstacle avoidance.

[Rover 1.2]

The width of the rover does not exceed the given frame of 9'', but with the addition of plastic inductor frame and the way the sensors were mounted it increased the length to approximately 1'. Even with these modifications the rover is still under the required 2' by 2' dimension constraint.

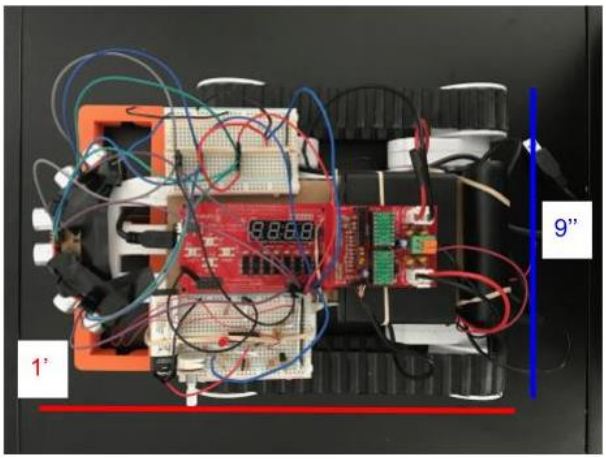


Fig [2] - A top down view of the assembled rover with dimension measurements

[Rover 1.3]

The rover was started and stopped with the use of an IR remote and receiver. The IR receiver was interfaced via the Atmega32u4 microcontroller. An interrupt service routine was utilized to interrupt the main rover algorithm in order to put the rover into an idle state. Once in the idle state, the function polled for another button press for which the rover would start running again. See appendix for source code for ISR and idle_state.

[Rover 1.4]

Navigation logic and I/O is processed entirely by the Papilio DUO. The Spartan 6 FPGA is used for I/O interfacing and connects the ultrasonic sensors, motors, motor encoders, and metal detector. The 32u4 microcontroller is programmed with the navigation logic and uses shared pins with the FPGA to take inputs from the sensors and output to the motors. The IR sensor used for start and stop is connected directly to the microcontroller and interrupts the navigation process when triggered.

[Navigation 1.1]

The rover navigated the demo course and found the maximum of four mines using an autonomous two-stage driving algorithm. The algorithm consisted of a line following routine with alternating turn biases, followed by a randomized turn routine to aid in covering as much surface area of the course as possible. Distance traveled was used to switch between stages. The line following algorithm was used for the first 90 feet the rover traveled where the rover would alternate from left and right turn bias approximately every 16 feet. This would cover the perimeter of the course as the ultrasonic sensors were configured to allow the rover to get as close the the walls as possible. At 90 feet, the rover switches to a randomized routine where sensors keep it towards the center of the course. The rover makes pseudorandom turns in order to constantly change the path traveled.

[Navigation 1.2]

Ultrasonic sensors were utilized and placed in a way to reduce blind spots during autonomous movement to prevent collisions with obstacles. The algorithm allowed the rover to ride near the wall but explicitly prevented it from actually touching the wall unless it hit a blind spot with the sensors. The frame for the inductor was also designed to swivel if it hit an obstacle or wall preventing the rover from moving an obstacle.

[Display 1.1]

With the use of the metal detector, the rover was able to sense mines on its own. The metal detector output a logical 1 for detection of metal and a 0 if no metal was near. This value was then sent to the wing of the papillio where the value was recorded and used to increment the values in Hex on the right center anode.

[Display 1.2]

The count was implemented using left motor's quadrature encoders. The signal used to increment the counter for the ticks was an xor of channels A and B from the encoder to create a 50% duty cycle wave. Whenever the output of the xor is high, the tick counter is incremented. The ratio of ticks per foot is approximately 256:1. By utilizing that ratio the rover was able to display the amount of feet traveled in Hex on the two leftmost anodes.

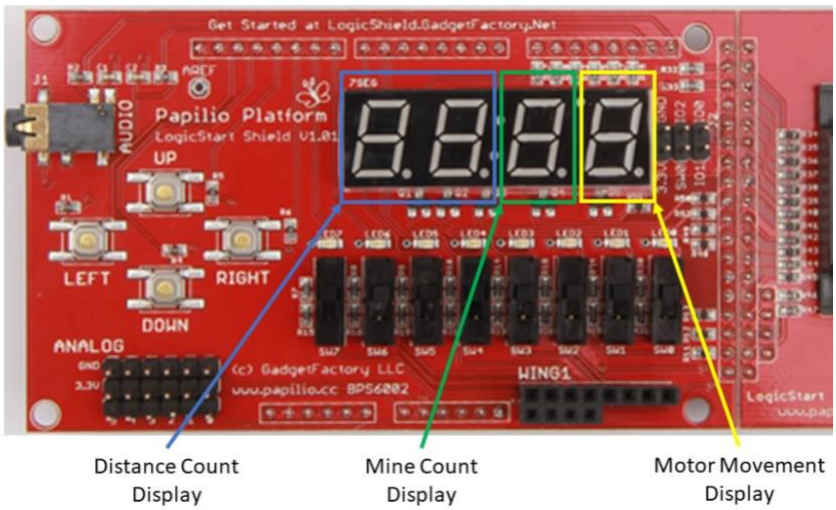


Fig [3] - FPGA seven segment display description

[Safety 1.1]

The rover was equipped with a battery pack with a on/off switch as well as a power connector that had an additional on/off switch to ensure safety and minimize risk if a failure was to occur. The Papilio was powered a rechargeable battery pack rated for a max of 5V so it was impossible to send it too much voltage. The design of the rover also avoided the use of lithium ion batteries as they were deemed unsafe for the project.

VII. Technical Design Description

i. Metal Detector Circuit

a. Overview

The metal detector circuit was designed to sense metal washers at a reasonable distance, with a distinct change in output so that it was effective in detecting metal. The circuit was put together in a fashion to output a logical 1 when detecting metal and a logical 0 when no metal is near. With this in mind the circuit also should not output any false positives but should also be sensitive enough to detect metal when any is near.

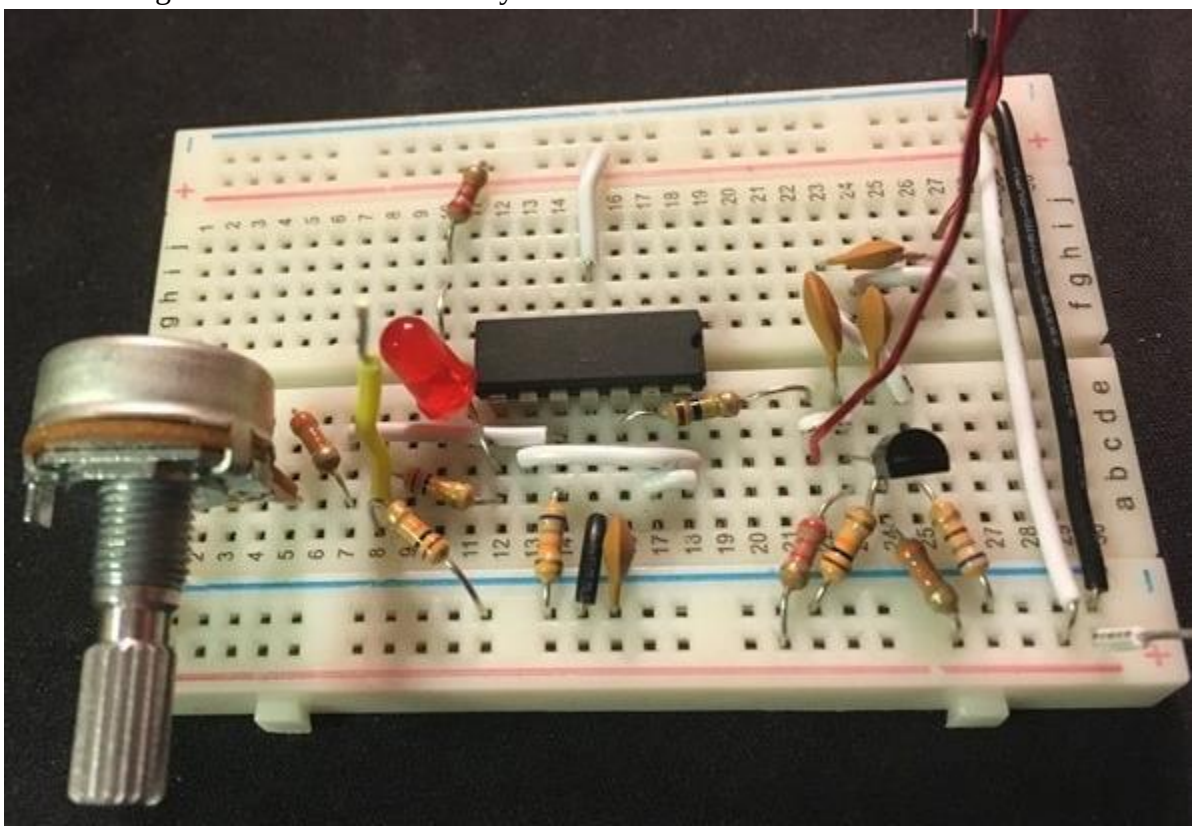


Fig [4] - Metal detector circuit

b. Inductor

The custom-built inductor used in the circuit is made up of magnet wire wrapped around a 3D-printed frame. The inductor is rectangular and is 2 inches long and 10 inches wide. The wire is wrapped approximately 31 times around the frame. The inductance of the inductor was calculated to be 232 μH using the following formula:

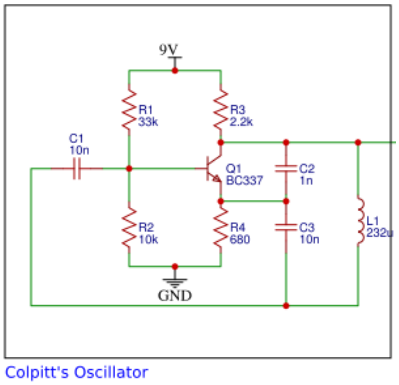
$L = \frac{Rtan(\varphi)}{2\pi f}$, where φ is the phase difference, R is equal to 10 Ω, and f = 1 kHz.

Equation [1] - RL circuit inductance formula

Fig [5] - RL circuit phase difference at 1 kHz

c. Colpitt's Oscillator

The Colpitt's Oscillator is a linear electronic oscillator circuit that uses a LC tank circuit made up of two capacitors in series connected in parallel to an inductor. The inductor in this case is the inductor described in the previous section. The Colpitts oscillator operates using a BC337 transistor



Fig[6] - Colpitt's Oscillator

The circuit operates at a resonant frequency of approximately 330.427 kHz which was calculated using the following formula:

$$f_0 = \frac{1}{2\pi\sqrt{L\frac{C_2C_3}{C_2+C_3}}} \rightarrow \frac{1}{2\pi\sqrt{232\mu H\frac{10^{-9}F * 1^{-9}F}{10^{-9}F + 1^{-9}F}}} = 330.427\,kHz$$

Equation [2] - equation for natural frequency of the oscillator

In the simulation of the circuit the fast fourier transform of the sinusoidal Colpitt's output is roughly the same as the calculated natural frequency.

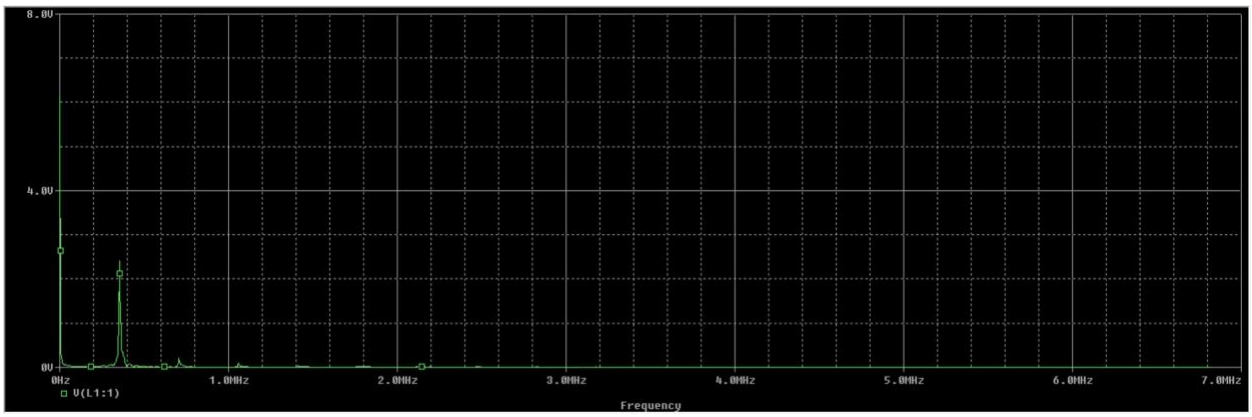


Fig [7] - Fast fourier transform of Colpitt's Oscilltator output

Normally the circuit will operate at 9 V but it oscillates when it hits the natural frequency and no current is running through the emitter. The voltage at the output will be the result of the voltage divider going into the base and the input voltage. In the case of this circuit the voltage will swing from approximately 2.09 V to 9 V. The simulation model of the circuit produces a swing that is approximately the same as the theoretical swing.

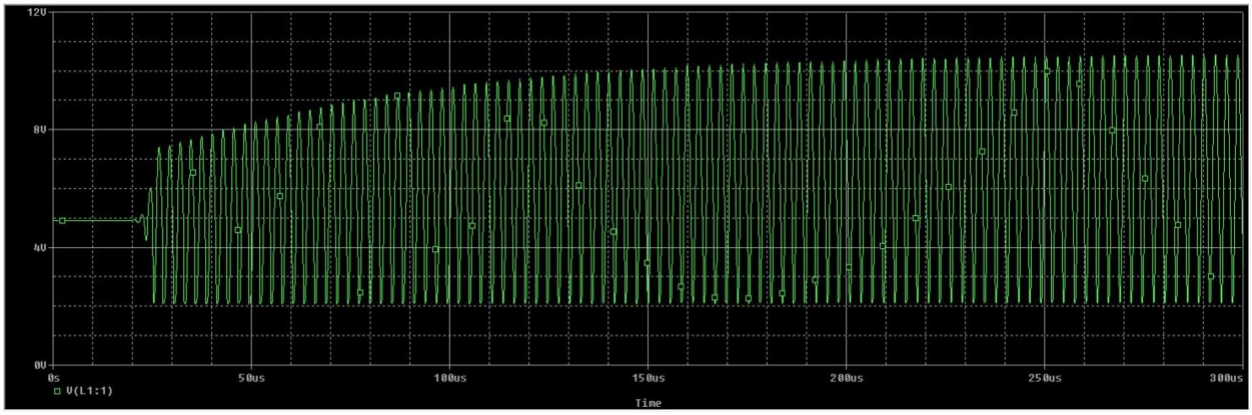


Fig [8] - Sinusoidal output of Colpitt's Oscillator without metal present

While the simulation only represents the circuit without metal present the readings are similar in oscillation not so much minimum and maximum value.

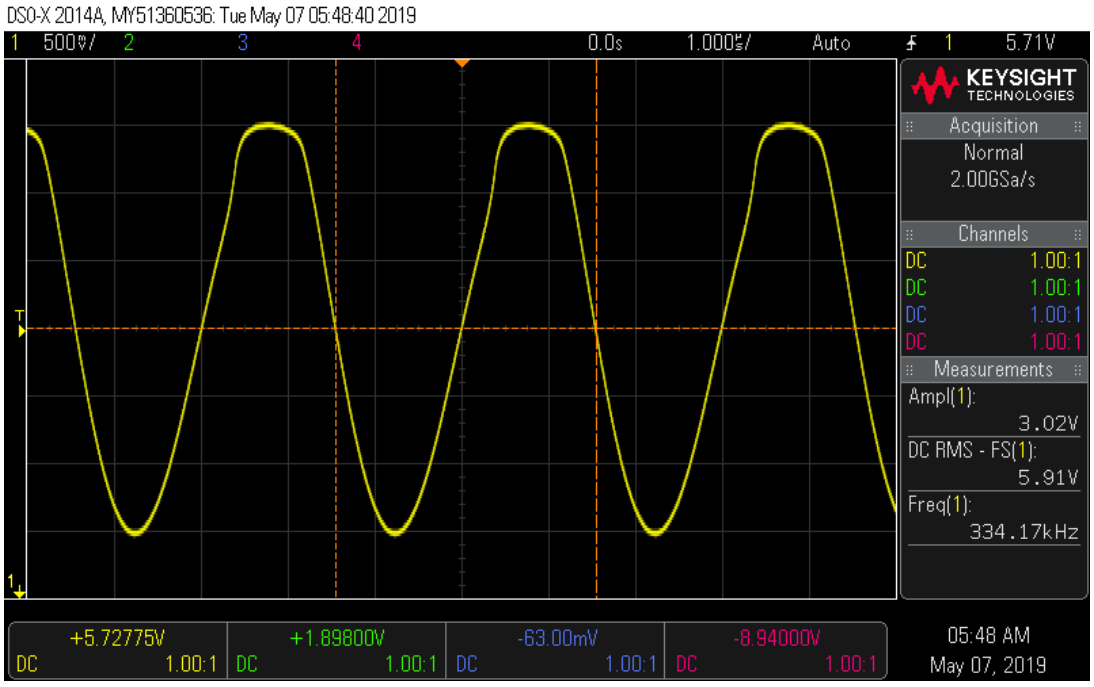


Fig [9] - Output of Colpitt's Oscillator without metal present

The frequency if very close to the approximated natural frequency from Equation [ ]. The DC-RMS also dropped a reasonable amount of voltage when metal was present, enough that it made sense to create the Analog Comparator for the output of the circuitry.

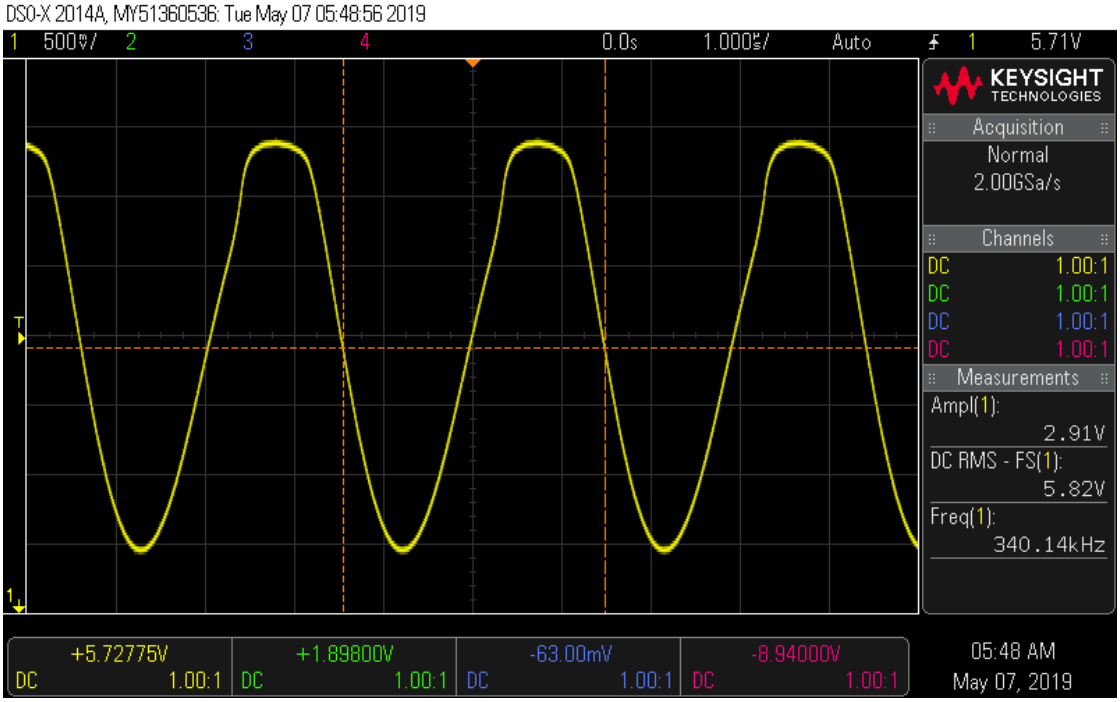


Fig [10] - Output of Colpitt's Oscillator with metal present

d. Active Low Pass Filter:

The Active Low Pass Filter was designed to remove any sinusoidal components from the output of Colpitt's Oscillator. The design is a simple low pass filter used a 100 kΩ resistor in series and a 10 nF capacitor in parallel. The resulting cutoff frequency can be represented by:

$$f = \frac{1}{RC} = \frac{1}{100k * 10^{-9}} = 1kHz$$

Equation[3] -Filter cutoff frequency

Colpitt's Oscillator consistently outputs a frequency of approximately 330 kHz so it is safe to say that the cutoff frequency would remove all sinusoidal parts of the Colpitt's output. Of course a passive low pass does not keep a unity gain the solution would be to introduce an Op Amp with negative feedback to maintain unity gain. By putting the output of the passive low pass into the positive input of the op amp and putting the negative input in feedback with the output we can create a gain of 1.

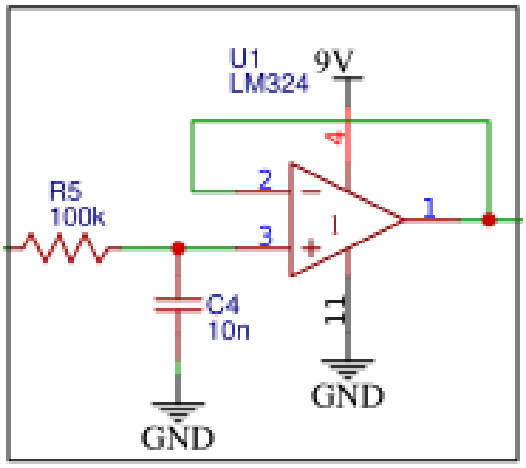


Fig [11] - Active Low Pass Filter

$$Gain(AV) = \frac{V_{out}}{V_{in}} = -\frac{R_F}{R_{in}} = -\frac{1}{1} = -1$$

Equation[4] - Gain of negative feedback Op Amp

In simulation it is impossible to predict the behavior of the circuit while it is detecting metal but the approximation of the low pass filter is quite close to the measured values when metal is not present.

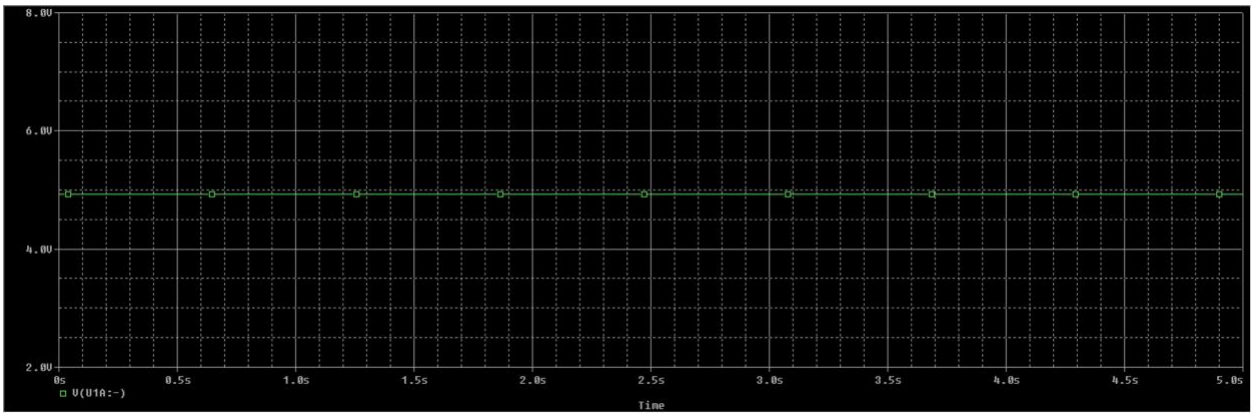


Fig [12] - Simulated output of the Active Low Pass Filter without metal present

Fig [12] depicts the DC-RMS values of the Colpitt's Oscillator output. This figure proves that the Active Low Pass Filter removes the sinusoidal input as well as keeping the DC value approximately the same.

The following figures depict the output of the circuit when there is metal present as well as when there is no metal present.

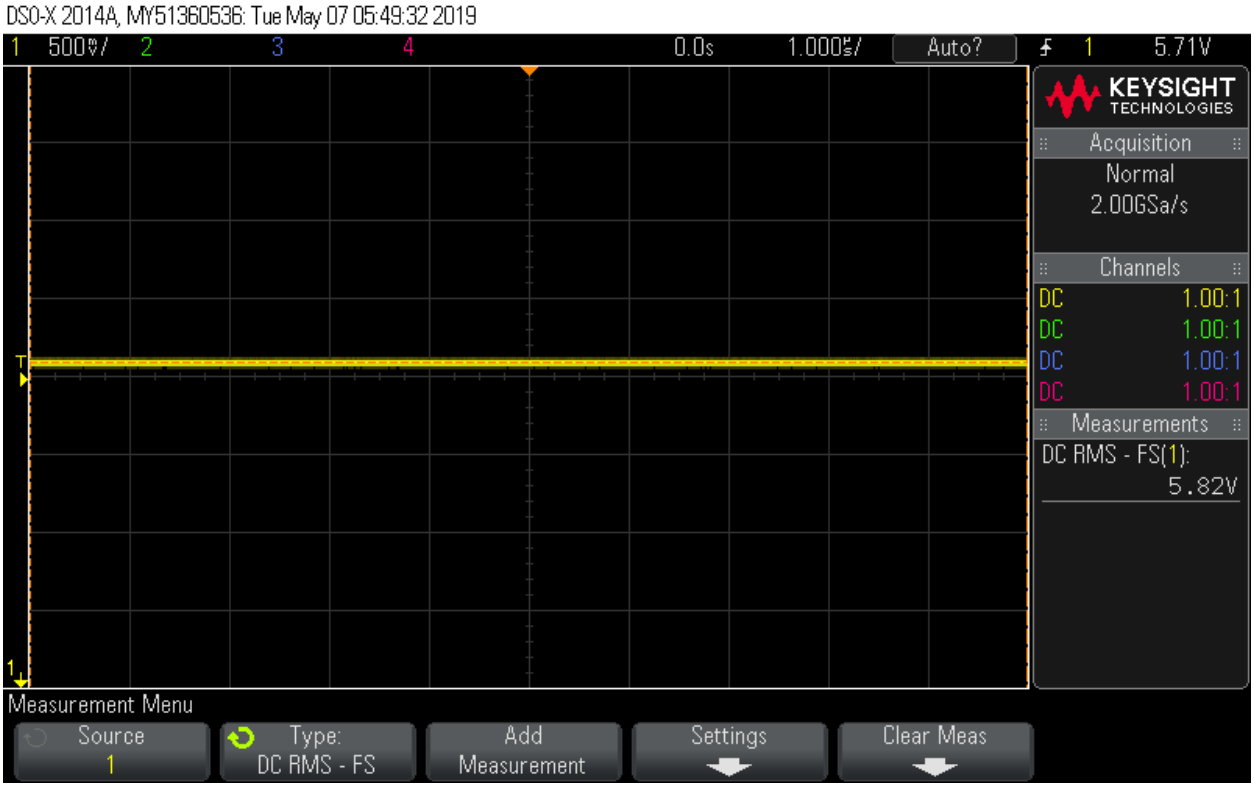
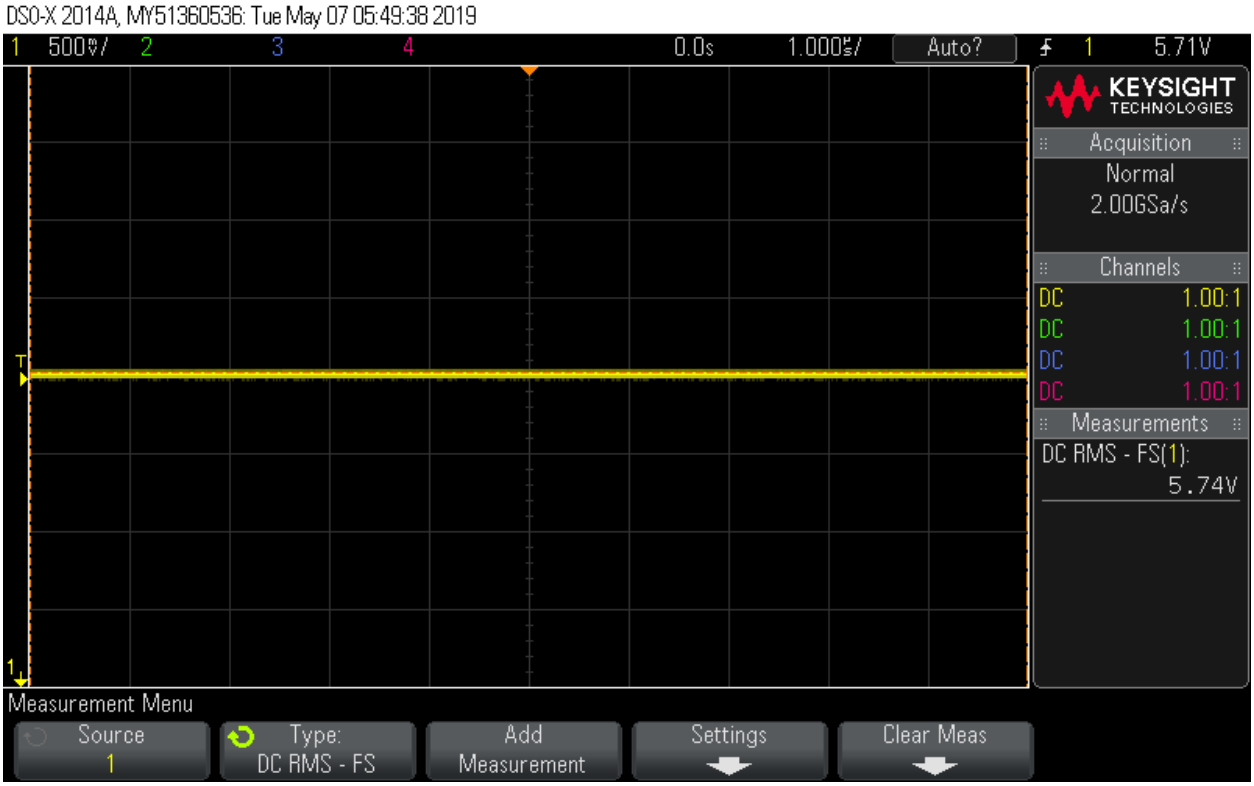


Fig [13] - Output of the Active Low Pass Filter without metal present

According to *Fig [13]* it is evident that unity gain is present as there is only a .1 V difference in the DC-RMS values when metal is not present.

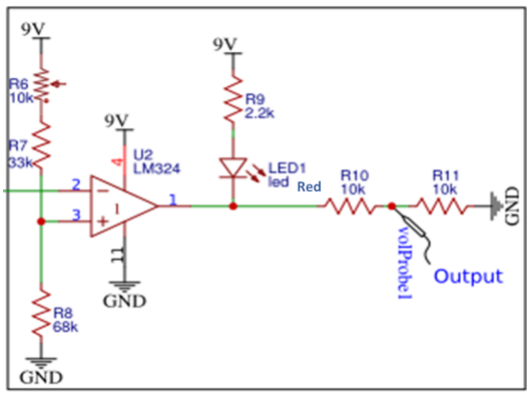


Fig[14] - Output of the Active Low Pass Filter with metal present

According to *Fig [14]* there is a distinguishable drop in voltage with metal present. With such a noticeable difference the next logical step would be to set up a voltage comparator to translate this difference into a logical 1 or 0.

e. Analog Comparator:

The analog comparator was designed to output a logical 1 when metal is present and a 0 otherwise. This signal would then be what needed to be sent to the Papilio wing. The goal of the comparator was to produce a voltage lower than .9 V and between 1.3V and 3.3V with a little leeway on the upper end of the voltage. This comparator features an adjustable reference voltage an active high output, and an indicator LED that turns off when metal is near the coil.



Fig[15] - Analog Comparator

Since the voltage from the filter will be fluctuating from 5.82 to 5.74 the necessary reference voltage should be a value between both of those voltages. Since the reference voltage is about 6 V the necessary voltage divider ratio should be about 3:1, that is why the 68k and 33k with some sort of potentiometer make sense to use in the voltage divider. Using the following equation it is possible to find an acceptable range of values the potentiometer must be in.

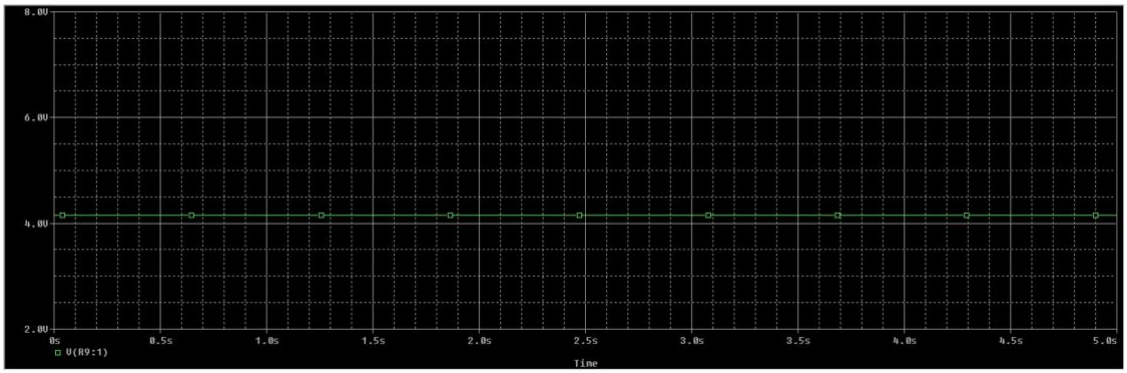
$$V_2 = \frac{R_2}{(R_2 + R_1)} * V_{in} \rightarrow V_2 = \frac{68k}{(68k + 33k + R_{pot})} * 9 \rightarrow 5.82 > \frac{68k}{(68k + 33k + R_{pot})} * 9$$

$$\rightarrow 5.74 < \frac{68k}{(68k + 33k + R_{pot})} * 9$$

$$4.2k \leq R_{pot} \leq 5.5k$$

Equation[5] - Voltage division for comparator reference voltage

In this instance it makes the most sense to use a 10k potentiometer. With about 40 divisions of resistance in the potentiometer (250Ω per division), there is a lot of room to tune the potentiometer to make the circuit as sensitive as possible. Unfortunately this circuit fails in this regard in simulation. Using a 38k resistor to simulate the potentiometer in series with the 33k is actually too much voltage going into the positive end of the Op Amp. The circuit would not be able to accurately detect metal in simulation.



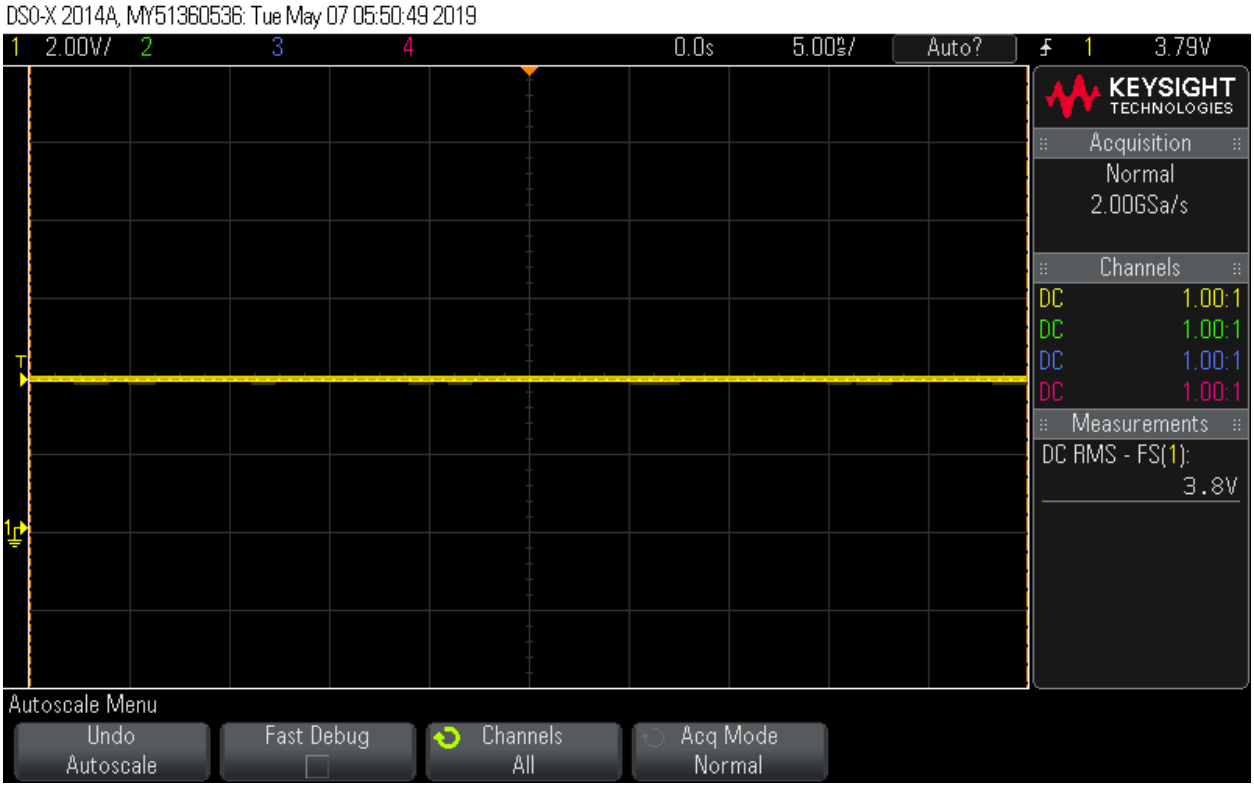
Fig[16] - Simulated output of the Analog Comparator without metal present

The last part of the comparator utilizes an LED as a visual key to detect metal. The LED is normally high, but when metal is present the LED stays low until the metal is removed. Normally the negative input of the Op Amp is high so the voltage at the output is negative which means the diode is on and the voltage after the divider should be extremely low. When the positive rail is higher the voltage is higher and can be approximated by:

$$V = \frac{V_{in} - V_D}{2} \rightarrow V_D \text{ is } 1.8V \text{ for a red LED} \rightarrow \frac{9 - 1.8}{2} \rightarrow V \simeq 3.6V$$

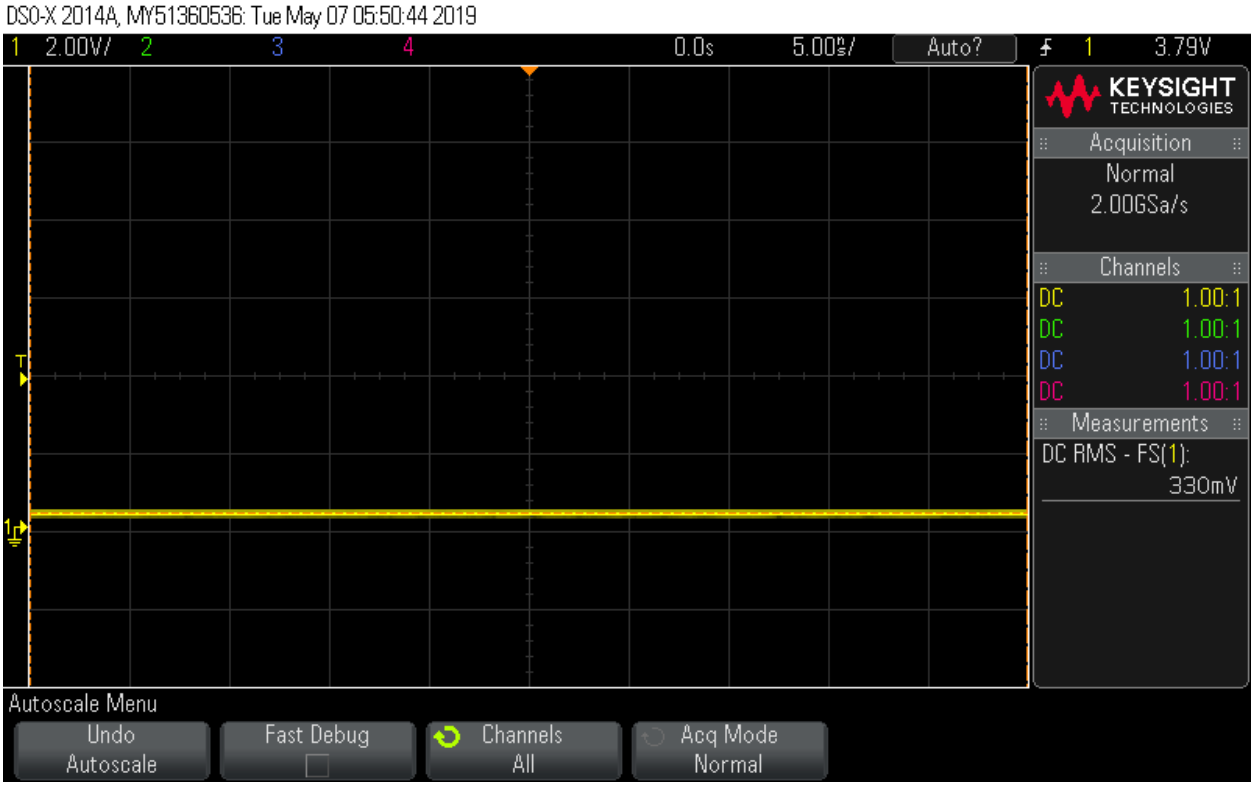
Equation[6] - Equation to find the voltage at the output of the Op Amp

The following figure shows that the approximation closely approximates the real value of the output voltage when metal is near.



Fig[17] - Output of the Analog Comparator when metal is present

Fig[17] shows that the output of the comparator is 3.8V which is very close to the calculated 3.6 V. This 3.8 V when sent to the wing of the Papilio would be read as a logical 1 since it is above 3.3V. Similarly the 330 mV shown in the figure below would read as a logical 0 to the Papilio since it is below 1.3 V.



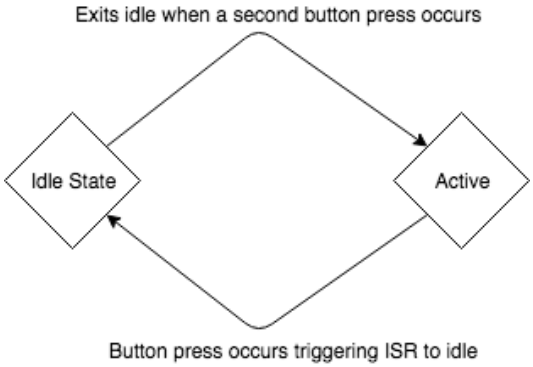
Fig[18] - Output of the Analog Comparator when metal is not present

ii. IR Receiver

a. Overview

The infrared remote and receiver was implemented to start and stop the rover. The IR receiver was interfaced via the Atmega32u4 microcontroller. An interrupt service routine was utilized to interrupt the main rover algorithm in order to put the rover into an idle state when a pin change interrupt occurs on pin B4 of the papilio wing. Once in the idle state, the function polled for another button press for which the rover would start running again.

b. Block Diagram



Fig[19] - Start stop routine

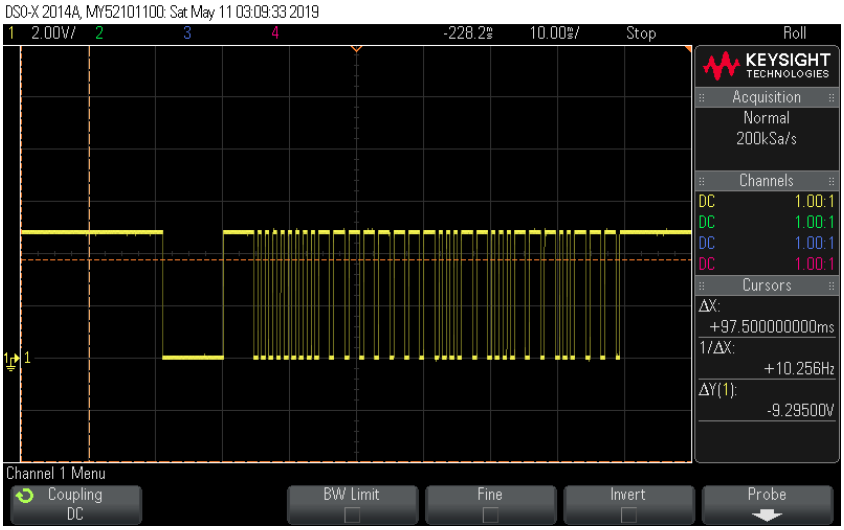
c. Circuit Components

The Sparkfun Infrared Remote outputs a 32 bit pulse code that corresponds to each button. However, because its application was only for stopping and starting, the first low pulse of each signal sent was interpreted and used for our implementation. In the ISR, after the first low pulse is received, the ISR is disabled to ignore the rest of the button code. When using polling in the idle state for the second button press, a delay is used to accomplish the same thing.

Fig [20] - Sparkfun Infrared Remote

The IR Receiver Diode, the TSOP38238 is pulled up to 5v by default and is active low when a button press occurs. The output of the receiver was directly connected to the atmega32u4 microcontroller on pin B4.

Fig[21] - IR Receiver Diode- TSOP38238



Fig[22] - Active low receiver IR signal UP 0x00FF9867

iii. I/O Interface

a. Overview

To simplify the implementation of the navigation algorithm, the FPGA was used as the center of most of the I/O. A shift register on the FPGA is controlled by the microcontroller to enable the serial transmission of control signals. Using this method, the microcontroller navigation algorithm outputs simple control signals to the register. To take in multiple inputs such as the echo signals, the microcontroller sets the select bits according to which sensor it needs to read. On the FPGA, these select bits are used on a mux that output the desired signal.

b. Motor Control

Both motors on the Dagu Rover are connected to the FPGA using the provided motor board. To enable PWM control on both motors, a decoder was used to assign PWM signals from the microcontroller according to the movement desired.

For PWM control on the motors, the desired movement is alternated with a brake command. This results in each motor needing 1 PWM signal at a time on its inputs. As a result of alternating with a brake signal, the PWM of the movement signal is the inverse of the PWM signal provided by the microcontroller. As an example, to achieve a 90% duty cycle for forward movement, the match on the microcontroller needs to be set at a 10% duty cycle. The following chart describes the outputs on the motor inputs needed for each movement.

PWM Movement	**1A**	**1B**	**2A**	**2B**
Forward	1	PWM	1	PWM
Backwards	PWM	1	PWM	1
Counter - clockwise	PWM	1	1	PWM
Clockwise	1	PWM	PWM	1
Brake	1	1	1	1

The following table describes the output of the select signals, and the information used to program movement on the microcontroller.

Select Signals for Movement	d2	d1	d0
Forward	0	0	0
Backwards	0	0	1
Counter Clockwise	0	1	0
Clockwise	0	1	1
Brake	1	0	0

The next illustration shows the implementation of the decoder on the FPGA. This solution allows for simple commands to be followed on the microcontroller to change movement. By using two separate PWM signals, this design has the capability of running the motors at different speeds. However this capability was not used as the algorithm had no need for it.

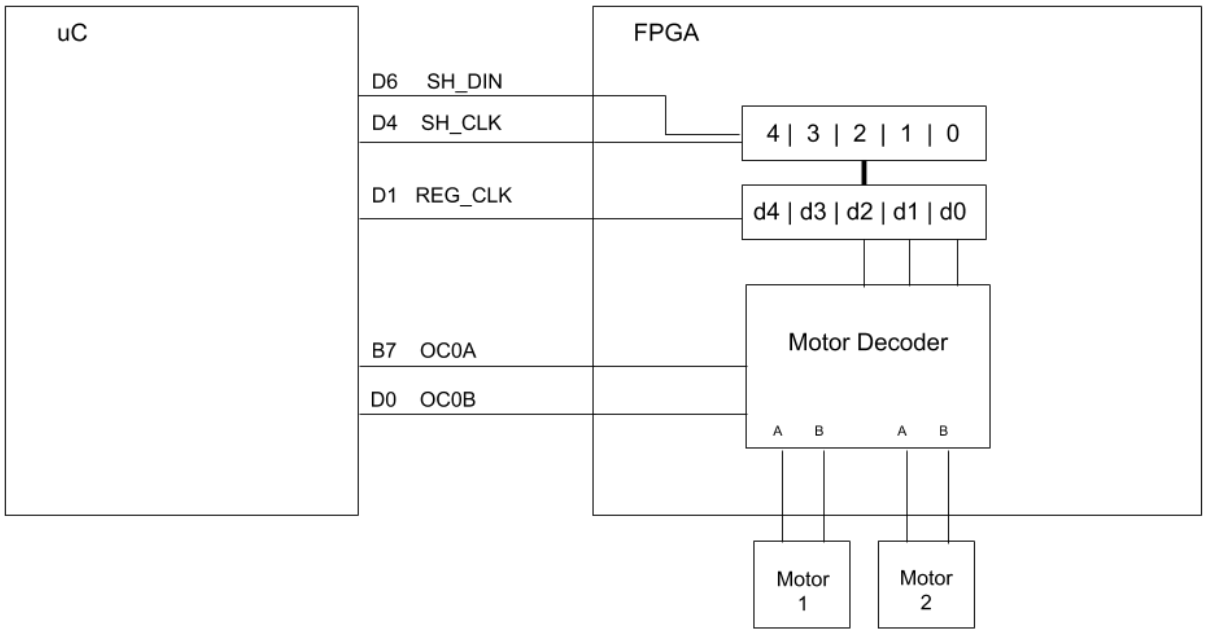


Fig [23] - Motor Decoder diagram

c. Ultrasonic Sensors

To detect obstacles in a wide range around the front of the rover, three ultrasonic sensors are used. In order to read multiple echo signals, the microcontroller uses a similar method as the motor control. The FPGA is used to select the appropriate signal to read and relay to the microcontroller where it can be evaluated and translated into a useful representation of distance.

The following illustration shows the implementation of this method. The microcontroller uses one pin as a common trigger for each ultrasonic sensor. Each echo is then input to the fpga where it is put into a mux. The microcontroller changes the echo being selected by changing the corresponding bits in the shift register.

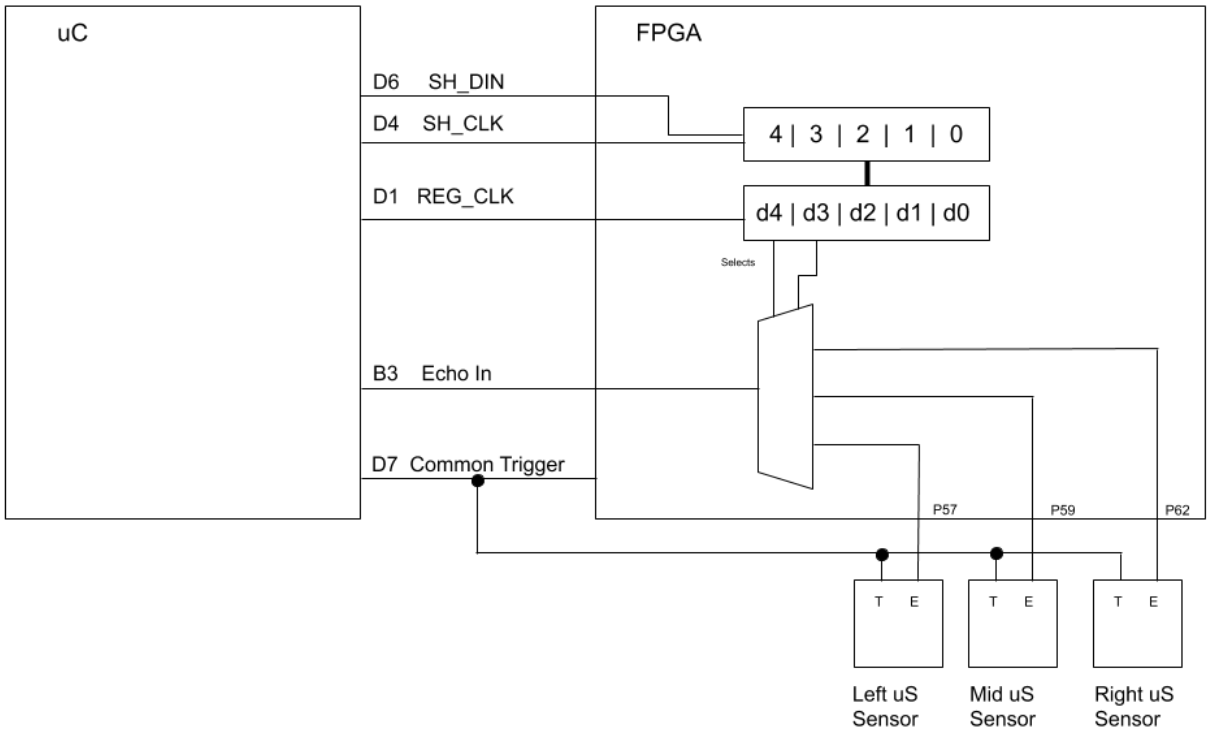


Fig [24] - Ultrasonic Sensor mux

Only two select signals are needed as there are three echos. The following shows the select signals and their corresponding outputs.

Echo Mux	d4	d3
Left	0	0
Middle	0	1
Right	1	0

d. Encoders and Metal Detector Count

To calculate distance travelled, one set of motor encoders are taken by the FPGA design and XOR'd. Since the design moves the motors at an equal speed, one set of encoders is an accurate representation of the total distance travelled. The resulting XOR signal of the encoders is used in a signal divider that reduces the amount of pulses by a factor of 256. This provides a signal to represent a foot travelled by the motors. The signal is used in an up counter module and set to a byte on the seven segment display.

A similar method is used to count the amount of mines detected by the metal detector. Since the metal detector provides a logical 1 when a mine is detected, a simple up counter was used on the FPGA board to keep count. The result of this up counter was also set to one of the seven segment display anodes.

VIII. Design Integration

The first part of integration involved outfitting the rover to make sure everything fit on it at once. This also involved attaching the inductor to the robot while also making the inductor as low to the ground as possible. The simplest solution was a 3D printed frame that could be screwed into the top of the rover. The frame needed to be screwed into the top of the rover and hold, the inductor while also being thin enough to be held at the top by the short screws. The resulting frame was actually quite light and still allowed the rover to be balanced while not dragging on the ground.

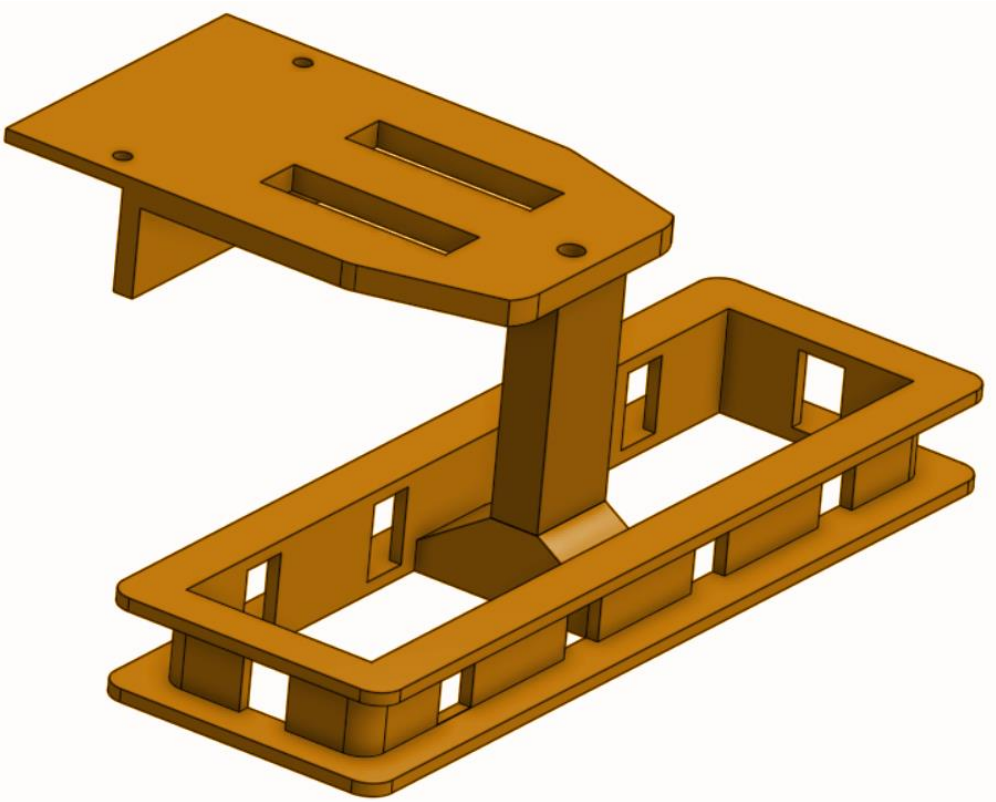


Fig [25] - rover mounted inductor frame

The frame also housed three ultrasonic sensors which were used for autonomous driving. Sensors were placed in the front and angled on the sides of the rover for obstacle avoidance and wall detection. Sensors placed on the right and left sides were angled approximately 45 degrees in order to eliminate blind spots when driving. In the driving algorithm, the sensors were configured to allow for the rover to follow closely to the perimeter of the course as well as for traversing the middle area. These two instances were achieved by changing the distances which the sensors would trigger a code function.

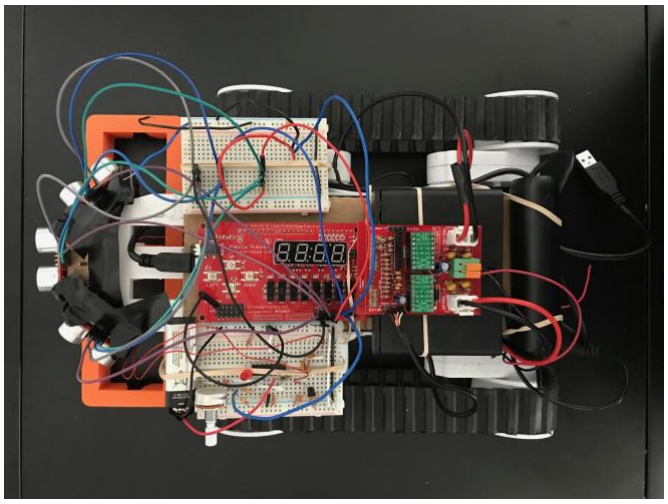


Fig [26] - Top view of rover to display sensor placement

Wire management helped to produce a clean, neat looking design when everything came together. Figure 26 shows the rover wire management as well as the angled sensors.

IX. Power Management

In order to ensure all circuitry and the rover were powered correctly, all circuitry was broken down into subsystems which were given their own battery packs. The metal detector was tested consistently using 9 V from the DC power source in the lab, naturally it was assigned a regular nine volt battery. In order to avoid powering the Papillio, encoders and ultrasonic sensors with too many volts, a 5 V rechargeable battery pack was used. This ensured that the circuit would not overload with voltage and it still provided the necessary current to each component. The battery pack also eliminated the need to constantly plug in and unplug the Papilio as there was on on/off switch on the pack. The motors require 9 V and more current than any other part of the circuit so they were given their own battery pack consisting of 6 1.5 V batteries. The battery

X. Software

i. Overview

Interfacing I/O on the FPGA allows for simple code on the microcontroller side. Functions to change peripheral behavior simply set the corresponding bits on the FPGA and serially transmit the values to the shift register. A global array of bites, shift_reg[4], stores the values transmitted to the FPGA shift register. The utility functions ouptut_byte() and update_IO() are used to shift out an individual byte and update the values on the I/O by enabling the buffer register. These two functions are used any time the microcontroller changes values on the peripherals.

ii. PWM/Motor Control

To output PWM signals on the FPGA, a timer counter peripheral is used in conjunction with two separate match values. The timer counter peripheral is configured with a match of 256. This value was determined to be a suitable solution given the options given by the control register. The resulting frequency of the PWM signal is thus around 1,000 Hz. This provides an adequate response from the motors.

These two signals given by the match are then output to the FPGA where the decoder places them on the proper motor input. To change the signals present on the motor inputs, functions are used for each motor movement. The functions simply set the proper motor decoder bits and transmit them to the FPGA.

iii. Algorithm

Besides displaying the distance travelled, the encoder count on the FPGA is used to trigger different phases on the navigation algorithm. The changes in navigation are designed to reduce the amount of time the rover spends travelling over spots covered previously. A change in navigation is often sufficient to move the rover out of a position where it is stuck or moving in a loop. With these considerations, the movement algorithm has two phases.

For the first phase, the rover is designed to follow the edge of the course. This phase is designed to have the rover find the two corner mines. A signal from the FPGA toggles every 16 feet dictated by the encoder count. The microcontroller uses this signal to switch between turn biases in the edge following algorithm. This results in the rover travelling 16 feet following the edge on one side and then turning around and switching directions. Using this method, the rover travels different paths by escaping loops when it detects obstacles and changing movement if it becomes stuck.

For the second phase, a “panic mode” algorithm is used. This algorithm triggers once a larger distance of 90 feet has been travelled and all washers have not been located. Similar to the previous signal, this signal is generated by the FPGA by comparing the encoder count.

In this algorithm, the rover rotates for a predetermined amount of time whenever it detects an object close to one of its sensors. This allows the rover to dart from edge to edge

covering the middle of the course where it likely missed mines with the edge following algorithms.

To accomplish this behavior, a for loop is used with a decreasing amount of iterations between loops. The body of the for loop is made up of a small delay, thus the iteration bound is directly responsible for the amount of time the rover spins. Once the iteration bound decays to its minimum point, it is reset to its maximum value. The minimum and maximum values are programmed to produce a complete turn that sets the rover away from a wall.

When put together, these three algorithms drive the rover along diverse areas in the course. We were able to find all four mines using this method.

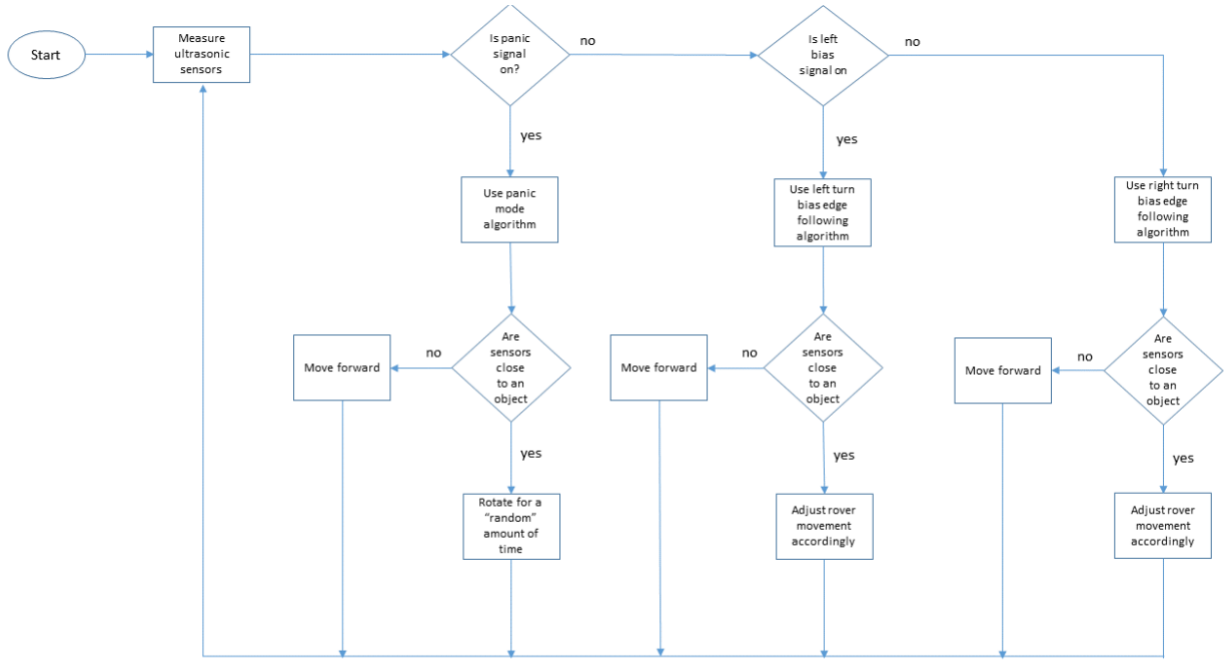


Fig [27] - Algorithm flow chart

XI. Appendix

i. Bill of Materials

Part	Quantity	Part Number	Manufacturer/Supplier
680Ω	1		State of the Art
2.2kΩ	2		State of the Art
10kΩ	3		State of the Art
33kΩ	2		State of the Art
100kΩ	1		State of the Art
10kΩ Potentiometer	1	RV120F-10015F-B10K	Alpha Products
1nF	1		Vishay Semiconductors
10nF	3		Vishay Semiconductors
Red LED	1	D1N5765	KingBright
OP Amp	1	LM324	Texas Instruments
IR Receiver	1	TSOP38238	Vishay Semiconductors
Infrared Remote Control	1		Sparkfun
Half Bread-boards	2		Adafruit Industries
Papillio Logic start Shield			Gadget Factory
Papillio Duo	1		Gadget Factory
Ultrasonic Sensors	3	HC-SR04	Sparkfun
Dagu 5 Rover Chassis	1		Pololu
9V battery	1		Energizer
9V battery pack	1		Pololu
5V Rechargeable battery pack	1		Onn Portable Battery Bank
Coin Cell Battery	1	CR2025	Panasonic
Magnet Wire (20AWG)	1		BNTECHGO

ii. Robot Power Analysis

	Schematic name	Part	Voltage (V)	Current (mA)	Power (mW)
Metal Detector Circuit					
Resistors	R1	2.2kΩ	4.083	1.856	7.578048
	R2	33kΩ	7.058	0.2139	1.5097062
	R3	680Ω	1.276	1.876	2.393776
	R4	10kΩ	1.942	0.1942	0.3771364
	R5	100kΩ	4.921	0.00004457	0.00021932897
	R6	38kΩ	3.226	0.07027	0.22669102
	R7	68kΩ	5.774	0.07035	0.4062009
	R8	10kΩ	4.158	0.4158	1.7288964
	R9	10kΩ	4.158	0.4158	1.7288964
	R11	2.2kΩ	9	3.394	30.546
Transistor	Q1	BC337 | Vc	4.917	1.856	6.76998674
		Vb	1.942	0.01947	
		Ve	1.276	-1.876	
Op Amp	U1A	LM324 | V+	9	1.008	9.072660173
		V out	4.912	0.0001344	
	U1B	LM324 | V+	9	1.423	9.35586
		V out	8.316	-0.415	
LED	D1	D1N5765	0.7	0.0000002204	0.00000015428
Sub-Circuit Power					71.69407772
Ultrasonic Sensors, IR Receiver and Papillio Duo					
IR Remote			3	0.2	0.6
IR Receiver		TSOP38238	5	0.33	1.65
Ultrasonic Sensors			5	0.000000001	0
Papillio			3.3	142	468.6

Sub-Circuit Power					470.85
Robot					
Rover	(both motors)		9	210	3780
Total Power					4322.544078
					P = I * V
	5v Rechargeable battery		12.06Wh		
	IR Remote		.495Wh	Coin Cell Battery 20 mm (CR2025)	
	6 AA batteries		9Wh		
	9 V battery		5.49Wh		
	metal detector circuit			76.57536264	hours
	Ultrasonic Sensors,IR Receiver and Papillio Duo			25.64593301	hours
	IR Remote			825	hours
	Rover			2.380952381	hours

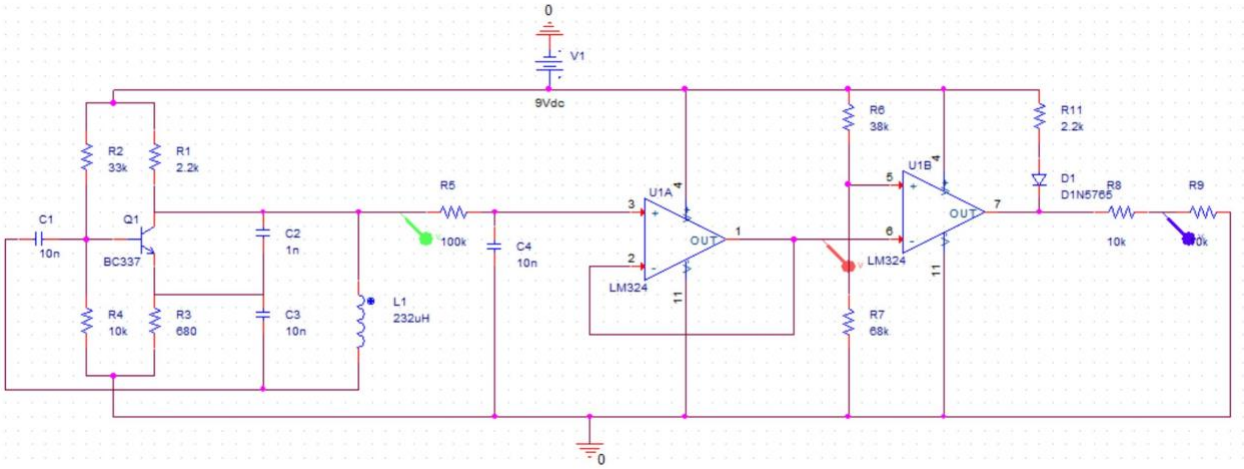


Fig [28] - metal detection simulation circuit

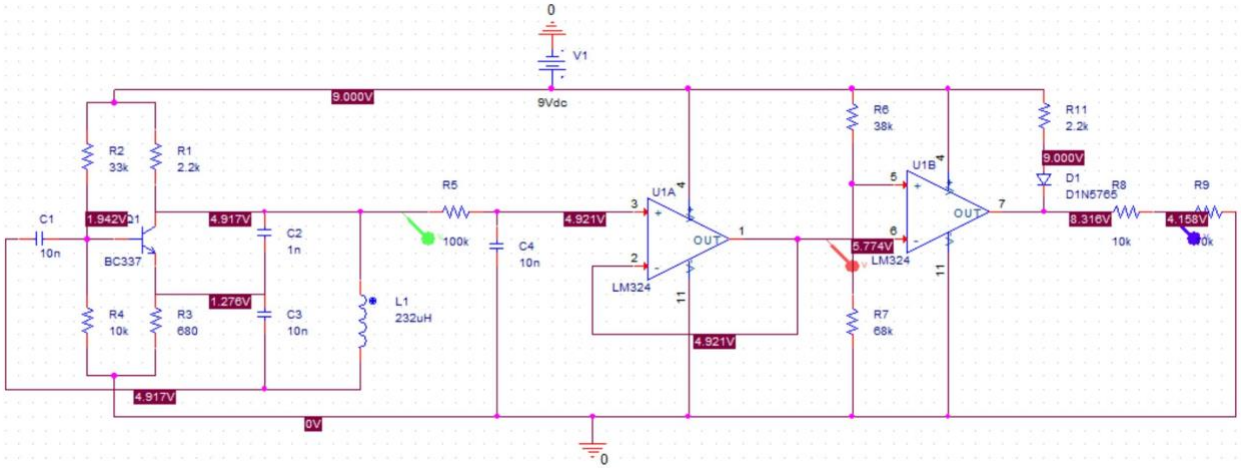


Fig [29] - metal detection simulation circuit with node voltage labeled

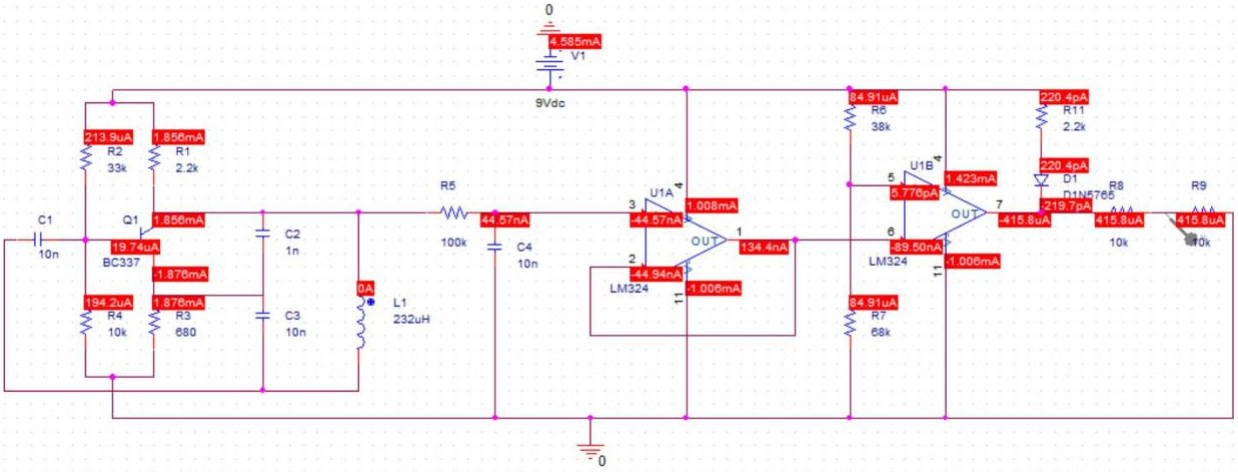


Fig [30] - metal detection simulation circuit with component current labeled

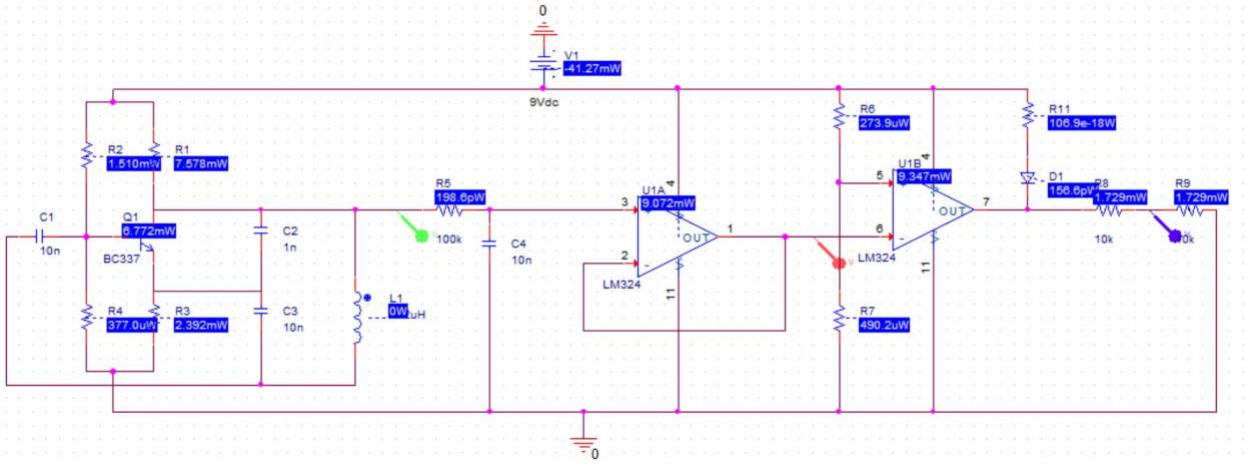


Fig [31] - metal detection simulation circuit with component power labeled

iii. Source Code

a. FPGA

1. Code

```
----------------------------------------------------------------------------------
-- Company:        Binghamton University
-- Engineer:       Milton Peraza
--
-- Create Date:    23:41:43 03/12/2019
-- Design Name:
-- Module Name:    fpga_interface - Behavioral
-- Project Name:   EECE 387 Rover Project
-- Target Devices: Papilio Duo Spartan 6
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-- EECE 387 Rover Project
-- Shift Register, PWM Decoder using LED 6,7 , SSEG Decoder
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity fpga_interface is
   Port ( CLK : in  STD_LOGIC;
                        SH_CLK : in  STD_LOGIC;
          SH_DIN : in  STD_LOGIC;
          REG_CLK : in  STD_LOGIC;
```

```
                OC0A : in STD_LOGIC;
                OC0B : in STD_LOGIC;
                L_ECHO : in STD_LOGIC;
                M_ECHO : in STD_LOGIC;
                R_ECHO : in STD_LOGIC;
                ENCODER_A : in STD_LOGIC;
                ENCODER_B : in STD_LOGIC;
                METAL_DETECTOR : in STD_LOGIC;
                DIR_LEFT : out STD_LOGIC;
                DIR_RIGHT : out STD_LOGIC;
                ARDUINO_RESET : out STD_LOGIC;
                Seg7_SEG : out  STD_LOGIC_VECTOR (6 downto 0);
        Seg7_AN : out  STD_LOGIC_VECTOR (3 downto 0);
        Seg7_DP : out  STD_LOGIC;
                MOTOR_1A : out STD_LOGIC;
                MOTOR_1B : out STD_LOGIC;
                MOTOR_2A : out STD_LOGIC;
                MOTOR_2B : out STD_LOGIC;
                ECHO_OUT : out STD_LOGIC);

end fpga_interface;

architecture Behavioral of fpga_interface is
--:= (others => '0')

COMPONENT HEXon7segDisp
        PORT(
                hex_data_in0 : IN std_logic_vector(3 downto 0);
                hex_data_in1 : IN std_logic_vector(3 downto 0);
                hex_data_in2 : IN std_logic_vector(3 downto 0);
                hex_data_in3 : IN std_logic_vector(3 downto 0);
                dp_in : IN std_logic_vector(2 downto 0);
                clk : IN std_logic;
                seg_out : OUT std_logic_vector(6 downto 0);
                an_out : OUT std_logic_vector(3 downto 0);
                dp_out : OUT std_logic
                );
        END COMPONENT;

        COMPONENT debouncer
```

```
        PORT(
                b : IN std_logic;
                clk : IN std_logic;
                b_debounced : OUT std_logic
                );
        END COMPONENT;

signal shift_reg : std_logic_vector(31 downto 0) := (others => '0');
signal hold_reg : std_logic_vector(31 downto 0) := (others => '0');

signal motor_decode : std_logic_vector(2 downto 0);
signal echo_decode : std_logic_vector(1 downto 0);

signal hex_data0 : std_logic_vector(3 downto 0);
signal hex_data1 : std_logic_vector(3 downto 0 );
signal hex_data2 : std_logic_vector(3 downto 0);
signal hex_data3 : std_logic_vector(3 downto 0 );

signal motor1A_sig : std_logic;
signal motor1B_sig : std_logic;
signal motor2A_sig : std_logic;
signal motor2B_sig : std_logic;

signal echo_sig : std_logic;

signal encoder_xor : std_logic;
signal encoder_divide : std_logic_vector(7 downto 0);
signal encoder_count : std_logic_vector(7 downto 0);

signal metal_detect : std_logic;
signal metal_count : std_logic_vector(3 downto 0);
signal metal_out : std_logic;
signal distance_comp : std_logic;

signal switch_alg : std_logic;

begin
--Shift Register
        process(SH_CLK,SH_DIN,shift_reg)
        begin
```

```
            if (rising_edge(SH_CLK)) then
                shift_reg(31 downto 0) <= SH_DIN & shift_reg(31 downto 1);
            end if;
        end process;

--Buffer Register Interfacing I/O
        process(REG_CLK,hold_reg)
        begin
            if (rising_edge(REG_CLK)) then
                hold_reg(31 downto 0) <= shift_reg(31 downto 0);
            end if;
        end process;

        ARDUINO_RESET <= '1';

        MOTOR_1A <= motor1A_sig;
        MOTOR_1B <= motor1B_sig;
        MOTOR_2A <= motor2A_sig;
        MOTOR_2B <= motor2B_sig;

        ECHO_OUT <= echo_sig;

        --Motor decoder signals
        motor_decode <= hold_reg(2 downto 0);

        --Echo Decoder Signals
        echo_decode <= hold_reg(4 downto 3);

        --Leftmost anodes display encoder counter
        hex_data0 <= encoder_count(7 downto 4);
  hex_data1 <= encoder_count(3 downto 0);
        --Middle right anode displays echo decoder value
        hex_data2 <= metal_count;
        --Far right anode displays motor decoder value
        hex_data3 <= '0' & hold_reg(2 downto 0);

--Motor Select Signal Decoder
        process(motor_decode,OC0A,OC0B)
        begin
```

```
        case motor_decode is
                --Forward
                when "000" =>
                        motor1A_sig <= '1';
                        motor1B_sig <= OC0A;
                        motor2A_sig <= '1';
                        motor2B_sig <= OC0B;
                --Backwards
                when "001" =>
                        motor1A_sig <= OC0A;
                        motor1B_sig <= '1';
                        motor2A_sig <= OC0B;
                        motor2B_sig <= '1';
                --Counterclockwise
                when "010" =>
                        motor1A_sig <= OC0A;
                        motor1B_sig <= '1';
                        motor2A_sig <= '1';
                        motor2B_sig <= OC0B;
                --Clockwise
                when "011" =>
                        motor1A_sig <= '1';
                        motor1B_sig <= OC0A;
                        motor2A_sig <= OC0B;
                        motor2B_sig <= '1';
                --Brake ("100")
                when others =>
                        motor1A_sig <= '1';
                        motor1B_sig <= '1';
                        motor2A_sig <= '1';
                        motor2B_sig <= '1';
        end case;
end process;

process(echo_decode,L_ECHO,M_ECHO,R_ECHO)
begin

        case echo_decode is
                --Left Echo
                when "00" =>
```

```
                    echo_sig <= L_ECHO;
                --Middle Echo
                when "01" =>
                    echo_sig <= M_ECHO;
                --Right Echo
                when others =>
                    echo_sig <= R_ECHO;
        end case;
end process;

--ENCODERS
--xor encoder signals
encoder_xor <= ENCODER_A xor ENCODER_B;

--Clock divider for encoder_xor signal to translate to feet
process(encoder_xor, encoder_divide)
begin
if(rising_edge(encoder_xor)) then
        encoder_divide <= std_logic_vector(unsigned(encoder_divide) + 1);
end if;
end process;

--Encoder up counter
process(encoder_count,encoder_divide)
begin
if(rising_edge(encoder_divide(7))) then
        encoder_count <= std_logic_vector(unsigned(encoder_count) + 1);
end if;
end process;

--display each of the motor IO values on the corresponding sseg anode (0123)
Inst_HEXon7segDisp: HEXon7segDisp PORT MAP(
                hex_data_in0 => hex_data0,
                hex_data_in1 => hex_data1,
                hex_data_in2 => hex_data2,
                hex_data_in3 => hex_data3,
                dp_in => "000",
                seg_out => Seg7_SEG,
                an_out => Seg7_AN,
                dp_out => Seg7_DP,
```

```
                clk => CLK
        );

    Inst_debouncer: debouncer PORT MAP(
        b => METAL_DETECTOR,
        b_debounced => metal_detect,
        clk => CLK
    );

    --metal_detect counter
    process(metal_detect)
    begin
    if(rising_edge(metal_detect)) then
        metal_count <= std_logic_vector(unsigned(metal_count) + 1);
    end if;
    end process;

    --metal_count comparator
--  process(clk,metal_count)
--  begin
--      if ( unsigned(metal_count) > 1 ) then
--          metal_out <= '1';
--      else
--          metal_out <= '0';
--      end if;
--  end process;

    --metal count
    process(clk,distance_comp,encoder_count)
    begin
    --5th bit effectively divides by 16
    if encoder_count(4) = '1'  then
        distance_comp <= '1';
    else
        distance_comp <= '0';
    end if;
    end process;

    DIR_LEFT <=  distance_comp;
```

```
        --switch algorithm signal
        process(clk,switch_alg,encoder_count)
        begin
        --enable when encoder count is greater than 90
        if ( unsigned(encoder_count) > 90)  then
                switch_alg <= '1';
        else
                switch_alg <= '0';
        end if;
        end process;

        DIR_RIGHT <= switch_alg;


end Behavioral;
```

2. UCF

```
 # UCF file for the Papilio DUO board
# Generated by pin_converter, written by Kevin Lindsey
# https://github.com/thelonious/papilio_pins/tree/development/pin_converter

## Prohibit the automatic placement of pins that are connected to VCC or GND for configuration.
CONFIG PROHIBIT=P144;
CONFIG PROHIBIT=P69;
CONFIG PROHIBIT=P60;

NET CLK          LOC="P94"  | IOSTANDARD=LVTTL | PERIOD=31.25ns;          # CLK
#NET RX          LOC="P46"  | IOSTANDARD=LVTTL;                          # RX
#NET TX          LOC="P141" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;     # TX
NET ARDUINO_RESET  LOC="P139" | IOSTANDARD=LVTTL;                         # ARDUINO_RESET
#NET SWITCH(0)     LOC="P116" | IOSTANDARD=LVTTL;                         # A0 SW0/WING1_7
#NET SWITCH(1)     LOC="P117" | IOSTANDARD=LVTTL;                         # A1 SW1/WING1_6
NET METAL_DETECTOR     LOC="P118" | IOSTANDARD=LVTTL;                     # A2 SW2/WING1_5
#NET SWITCH(3)     LOC="P119" | IOSTANDARD=LVTTL;                         # A3 SW3/WING1_4
#NET SWITCH(4)     LOC="P120" | IOSTANDARD=LVTTL;                         # A4 SW4/WING1_3
NET SH_DIN       LOC="P121" | IOSTANDARD=LVTTL;                          # A5 LED(3)
NET SH_CLK       LOC="P123" | IOSTANDARD=LVTTL;                          # A6 LED(4)
NET REG_CLK      LOC="P124" | IOSTANDARD=LVTTL;                          # A7 LED(5)
NET OC0B       LOC="P126" | IOSTANDARD=LVTTL;                            # A8 LED(6)
NET OC0A       LOC="P127" | IOSTANDARD=LVTTL;                            # A9 LED(7)
NET DIR_RIGHT     LOC="P131" | IOSTANDARD=LVTTL;                         # A10
#NET DIR_UP       LOC="P132" | IOSTANDARD=LVTTL;                         # A11
NET ECHO_OUT      LOC="P133" | IOSTANDARD=LVTTL;                         # A12 down_btn
NET DIR_LEFT      LOC="P134" | IOSTANDARD=LVTTL;                         # A13
NET Seg7_SEG(1) LOC="P115" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;     # B0
```

```
NET Seg7_SEG(6) LOC="P114" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
B1
NET Seg7_SEG(2) LOC="P112" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
B2
NET Seg7_SEG(5) LOC="P111" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
B3
NET Seg7_SEG(0) LOC="P105" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
B4
NET Seg7_SEG(4) LOC="P102" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
B5
#NET AUDIO1_RIGHT   LOC="P101" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# B6
#NET AUDIO1_LEFT    LOC="P100" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# B7
NET Seg7_DP LOC="P99"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        # C0
NET Seg7_SEG(3) LOC="P97"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
C1
#NET GPIO0        LOC="P93"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
C2
#NET GPIO1        LOC="P88"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
C3
#NET GPIO2        LOC="P85"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
C4
NET Seg7_AN(3)   LOC="P83"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
C5
NET Seg7_AN(2)   LOC="P81"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
C6
#NET Seg7_AN(4)   LOC="P79"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# C7
NET Seg7_AN(1)   LOC="P75"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
C8
NET Seg7_AN(0)   LOC="P67"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;        #
C9
NET R_ECHO     LOC="P62"  | IOSTANDARD=LVTTL;                    # C10
SW5/WING1_2
NET M_ECHO     LOC="P59"  | IOSTANDARD=LVTTL;                    # C11
SW6/WING1_1
NET L_ECHO     LOC="P57"  | IOSTANDARD=LVTTL;                    # C12
SW7/WING1_0
#NET LED(0)       LOC="P55"  | IOSTANDARD=LVTTL;                    # C13
```

```
#NET LED(1)       LOC="P50"  | IOSTANDARD=LVTTL;                    # C14
#NET LED(2)       LOC="P47"  | IOSTANDARD=LVTTL;                    # C15
#NET VGA_RED(0)    LOC="P98"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D0
#NET VGA_RED(1)    LOC="P95"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D1
#NET VGA_RED(2)    LOC="P92"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D2
#NET VGA_RED(3)    LOC="P87"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D3
#NET VGA_GREEN(3)  LOC="P84"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D4
#NET VGA_GREEN(2)  LOC="P82"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D5
#NET VGA_GREEN(1)  LOC="P80"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D6
#NET VGA_GREEN(0)  LOC="P78"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D7
#NET ENCODER_B   LOC="P74"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D8
#NET ENCODER_A   LOC="P66"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D9
NET MOTOR_2B   LOC="P61"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D10
NET MOTOR_2A  LOC="P58"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;       #
D11
NET ENCODER_B   LOC="P56"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D12
NET ENCODER_A   LOC="P51"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D13
NET MOTOR_1B    LOC="P48"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D14
NET MOTOR_1A    LOC="P39"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# D15
#NET SRAM_ADDR(0)  LOC="P7"   | IOSTANDARD=LVTTL;                    #
SRAM_ADDR0
#NET SRAM_ADDR(1)  LOC="P8"   | IOSTANDARD=LVTTL;                    #
SRAM_ADDR1
#NET SRAM_ADDR(2)  LOC="P9"   | IOSTANDARD=LVTTL;                    #
SRAM_ADDR2
```

```
#NET SRAM_ADDR(3)  LOC="P10"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR3
#NET SRAM_ADDR(4)  LOC="P11"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR4
#NET SRAM_ADDR(5)  LOC="P5"   | IOSTANDARD=LVTTL;                   #
SRAM_ADDR5
#NET SRAM_ADDR(6)  LOC="P2"   | IOSTANDARD=LVTTL;                   #
SRAM_ADDR6
#NET SRAM_ADDR(7)  LOC="P1"   | IOSTANDARD=LVTTL;                   #
SRAM_ADDR7
#NET SRAM_ADDR(8)  LOC="P143" | IOSTANDARD=LVTTL;                    #
SRAM_ADDR8
#NET SRAM_ADDR(9)  LOC="P142" | IOSTANDARD=LVTTL;                    #
SRAM_ADDR9
#NET SRAM_ADDR(10) LOC="P43"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR10
#NET SRAM_ADDR(11) LOC="P41"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR11
#NET SRAM_ADDR(12) LOC="P40"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR12
#NET SRAM_ADDR(13) LOC="P35"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR13
#NET SRAM_ADDR(14) LOC="P34"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR14
#NET SRAM_ADDR(15) LOC="P27"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR15
#NET SRAM_ADDR(16) LOC="P29"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR16
#NET SRAM_ADDR(17) LOC="P33"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR17
#NET SRAM_ADDR(18) LOC="P32"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR18
#NET SRAM_ADDR(19) LOC="P44"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR19
#NET SRAM_ADDR(20) LOC="P30"  | IOSTANDARD=LVTTL;                    #
SRAM_ADDR20
#NET SRAM_DATA(0)  LOC="P14"  | IOSTANDARD=LVTTL;                   #
SRAM_DATA0
#NET SRAM_DATA(1)  LOC="P15"  | IOSTANDARD=LVTTL;                   #
SRAM_DATA1
```

```
#NET SRAM_DATA(2)  LOC="P16"  | IOSTANDARD=LVTTL;                    #
SRAM_DATA2
#NET SRAM_DATA(3)  LOC="P17"  | IOSTANDARD=LVTTL;                    #
SRAM_DATA3
#NET SRAM_DATA(4)  LOC="P21"  | IOSTANDARD=LVTTL;                    #
SRAM_DATA4
#NET SRAM_DATA(5)  LOC="P22"  | IOSTANDARD=LVTTL;                    #
SRAM_DATA5
#NET SRAM_DATA(6)  LOC="P23"  | IOSTANDARD=LVTTL;                    #
SRAM_DATA6
#NET SRAM_DATA(7)  LOC="P24"  | IOSTANDARD=LVTTL;                    #
SRAM_DATA7
#NET SRAM_CE      LOC="P12"  | IOSTANDARD=LVTTL;                  #
SRAM_CE
#NET SRAM_WE      LOC="P6"   | IOSTANDARD=LVTTL;                  #
SRAM_WE
#NET SRAM_OE      LOC="P26"  | IOSTANDARD=LVTTL;                  #
SRAM_OE
#NET JTAG_TMS     LOC="P107" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# JTAG_TMS
#NET JTAG_TCK     LOC="P109" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# JTAG_TCK
#NET JTAG_TDI     LOC="P110" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# JTAG_TDI
#NET JTAG_TDO     LOC="P106" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# JTAG_TDO
#NET FLASH_CS     LOC="P38"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# FLASH_CS
#NET FLASH_CK     LOC="P70"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# FLASH_CK
#NET FLASH_SI     LOC="P64"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST;
# FLASH_SI
#NET FLASH_SO     LOC="P65"  | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=FAST |
PULLUP; # FLASH_SO
```

b. Micro Controller

```
#define F_CPU 16000000UL
#include <util/delay.h>
#include <avr/io.h>
#include <stdint.h>
#include <avr/interrupt.h>

#define FORWARD      0x00
#define BACKWARDS    0x01
#define CNTR_CLKWISE 0x02
#define CLKWISE      0x03
#define BRAKE        0x04


#define V_CLOSE 96
#define M_CLOSE 100
#define CLOSE 160

#define FAR 225

volatile uint8_t button_pressed = 0;
void config_IRsensor();
void isr_initialize();
void config_metal_comp();

void output_byte(uint8_t byte);
void update_IO();

void moveForward();
void moveBackwards();
void rotateLeft();
void rotateRight();
void brake_out();

void idle_state();

uint8_t left_US();
uint8_t mid_US();
uint8_t right_US();
```

```
uint8_t shift_reg[4];

ISR(PCINT0_vect){
	if (!(PINB & (1<<4))){
		PCMSK0 &=~ (1<<4); //enable pin change interrupt on PB4
		button_pressed = 1;
	}
}

int main(void) {
	config_IRsensor();
	isr_initialize();
	config_metal_comp();
	//**********************
	//Timer Peripheral I/O
	//**********************
	//configuring LEDS(7:6) OCCA1 OCCA2
	DDRB |= (1<<7);
	DDRD |= (1<<0);

	//configure timer counter control register
	TCCR0A = 0xA3;
	//clk prescale of 256
	TCCR0B = 0x04;

	//set match for both to output 75% duty cycle
	OCR0A = 0x19;
	OCR0B = 0x19;

	//**************************
	//Shift Register I/O Setup
	//**************************
	//configure D1 LED5 reg_clk output to fpga
	DDRD |= (1<<1);

	//configure D4 LED4 shift clk output to fpga
	DDRD |= (1<<4);

	//configure D6 LED3 buffer shift data in
```

```
	DDRD |= (1<<6);

	//*************************
	//Ultrasonic Sensor Setup
	//*************************
	//configure trigger pin as output D7 on Wing
	DDRD |= (1<<7);

	//configure echo pin as input, no pull up  B3 down_btn
	DDRB &= ~(1<<3);
	PINB &= ~(1<<3);

	//*************************
	//Algorithm Switch Signal
	//*************************
	DDRB &= ~(1<<0);
	PINB &= ~(1<<0);

	//**************
	// "Main" Loop
	//**************
	uint8_t leftDistance;
	uint8_t midDistance;
	uint8_t rightDistance;
	while (1) {
		static uint8_t rotate_multiplier = 10;

		leftDistance = left_US();
		midDistance = mid_US();
		rightDistance = right_US();

		//First stage algorithm right turn bias
		if( !(PINB & (1<<1)) && (!(PINB & (1<<0))) ){
			if ( leftDistance < V_CLOSE ) {
				rotateRight();
			}
			else if ( leftDistance < CLOSE && leftDistance > V_CLOSE &&
midDistance > CLOSE && rightDistance > CLOSE ) {
				moveForward();
			}
```

```
                else if (( leftDistance < CLOSE && midDistance <  M_CLOSE) || (
midDistance < M_CLOSE && leftDistance > CLOSE )) {
                        rotateRight();
                }
                else if ( midDistance < M_CLOSE && leftDistance > CLOSE &&
rightDistance > CLOSE) {
                        rotateRight();
                }
                else if (rightDistance < V_CLOSE){
                        rotateLeft();
                }
                //addition
                else{
                        moveForward();
                }
        }
        //First Stage algorithm left turn bias
        if( PINB & (1<<1) && (!(PINB & (1<<0)))  ){
                if ( rightDistance < V_CLOSE ) {
                        rotateLeft();
                }
                else if ( rightDistance < CLOSE && rightDistance > V_CLOSE &&
midDistance > CLOSE && leftDistance > CLOSE ) {
                        moveForward();
                }
                else if (( rightDistance < CLOSE && midDistance <  M_CLOSE) || (
midDistance < M_CLOSE && rightDistance > CLOSE )) {
                        rotateLeft();
                }
                else if ( midDistance < M_CLOSE && rightDistance > CLOSE &&
leftDistance > CLOSE) {
                        rotateLeft();
                }
                else if (leftDistance < V_CLOSE){
                        rotateRight();
                }
                //addition
                else{
                        moveForward();
                }
```

```
                }

                //Second Stage algorithm right turn bias
                if( PINB & (1<<0) )   {
                        int i;
                        if ( rotate_multiplier < 2 ) {
                                rotate_multiplier = 10;
                        }
                        /*
                        if (rightDistance < FAR) {

                        }*/
                        if ( (leftDistance < V_CLOSE) || (midDistance < V_CLOSE) ||
(rightDistance < V_CLOSE) ) {
                                rotateRight();
                                //rotate for rotate_multiplier delays
                                for(i=0;i<(5 + rotate_multiplier);i++) {
                                        _delay_ms(50);
                                }
                                //decrease the time of the next rotate
                                rotate_multiplier--;
                        }
                        else {
                                moveForward();
                        }


                }

                idle_state();
        }
        return 0;
}

void output_byte(uint8_t byte) {
        uint8_t i;  //mask
        //shift out each bit
        for(i = 1;i != 0;i = i<<1) {
                if(byte & i)
                        PORTD |= (1<<6);
```

```
                else
                        PORTD &= ~(1<<6);
                //toggle shift clk
                PORTD |= (1<<4);
                PORTD &= ~(1<<4);
        }
}

void update_IO() {
        //shift out each of the 4 bytes to the 32 bit shift register
        output_byte(shift_reg[0]);
        output_byte(shift_reg[1]);
        output_byte(shift_reg[2]);
        output_byte(shift_reg[3]);

        //toggle buffer reg clk
        PORTD |= (1<<1);
        PORTD &= ~(1<<1);
}

void moveForward() {
        //Forward "000" bits 2-0
        //update relevant byte and output to registers
        shift_reg[0] &= ~( (1<<2) | (1<<1) | (1<<0) );
        update_IO();
}

void moveBackwards() {
        //Backwards "001"
        //update relevant byte and output to registers
        shift_reg[0] |= (1<<0);
        shift_reg[0] &= ~( (1<<2) | (1<<1) );
        update_IO();
}

void rotateLeft() {
        //Counter Clockwise "010"
        //update relevant byte and output to registers
        shift_reg[0] |= (1<<1);
        shift_reg[0] &= ~( (1<<2) | (1<<0) );
```

```
        update_IO();
}

void rotateRight() {
        //Clockwise "011"
        //update relevant byte and output to registers
        shift_reg[0] |= ( (1<<1) | (1<<0) );
        shift_reg[0] &= ~(1<<2);
        update_IO();
}


void brake_out() {
        //Brake "100"
        //update relevant byte and output to registers
        shift_reg[0] |= (1<<2);
        shift_reg[0] &= ~( (1<<1) | (1<<0) );
        update_IO();
}

uint8_t left_US() {
        //Select Left Ultrasonic Sensor on FPGA
        //Update corresponding register values
        //"00"  bits 4-3
        shift_reg[0] &= ~( (1<<4) | (1<<3) );
        update_IO();

        uint8_t distance = 0;
        //trigger pulse
        PORTD |= (1<<7);
        _delay_us(15);
        PORTD &= ~(1<<7);

        //wait for echo to go high
        while (!(PINB & (1<<3)) );

        //measure echo
        while( (PINB & (1<<3)) && (distance < 255) ) {
                _delay_us(7);
                distance++;
```

```
        }

        //delay to avoid glitches
        _delay_ms(50);

        return distance;
}

uint8_t mid_US() {
        //Select Middle Ultrasonic Sensor on FPGA
        //Update corresponding register values
        //"01"  bits 4-3
        shift_reg[0] |= (1<<3);
        shift_reg[0] &= ~(1<<4);
        update_IO();

        uint8_t distance = 0;
        //trigger pulse
        PORTD |= (1<<7);
        _delay_us(15);
        PORTD &= ~(1<<7);

        //wait for echo to go high
        while (!(PINB & (1<<3)) );

        //measure echo
        while( (PINB & (1<<3)) && (distance < 255) ) {
                _delay_us(7);
                distance++;
        }

        //delay to avoid glitches
        _delay_ms(50);

        return distance;
}

uint8_t right_US() {
        //Select Right Ultrasonic Sensor on FPGA
        //Update corresponding register values
```

```
        //"10"  bits 4-3
        shift_reg[0] |= (1<<4);
        shift_reg[0] &= ~(1<<3);
        update_IO();

        uint8_t distance = 0;
        //trigger pulse
        PORTD |= (1<<7);
        _delay_us(15);
        PORTD &= ~(1<<7);

        //wait for echo to go high
        while (!(PINB & (1<<3)) );

        //measure echo
        while( (PINB & (1<<3)) && (distance < 255) ) {
                _delay_us(7);
                distance++;
        }

        //delay to avoid glitches
        _delay_ms(50);

        return distance;
}

//addition
void config_IRsensor(){
        DDRB &= ~(1<<4);  //IR Input  SW3
        PORTB &= ~(1<<4); //Disable pull-ups
}

void idle_state(){
        while(button_pressed == 1){ //Interrupt driven global variable
                brake_out();
                //PCMSK0 |= (1<<4);  //Enable IOC on pin 4 again
                _delay_ms(20);
                if(!(PINB & (1<<4))){  //detect falling edge on pin B
                        button_pressed = 0;
                        _delay_ms(150);
```

```
                        PCMSK0 |= (1<<4);  //Enable IOC on pin4
                }

        }
}
void isr_initialize(){
        cli();
        PCMSK0 |= ((1<<4)); //enable pin change interrupt on PB4
        PCICR |= (1<<PCIE0); //enable pin change interrupts
        sei();
}

void config_metal_comp(){
        DDRB &= ~(1<<1);
        PINB &= ~(1<<1);
}
```