Junior Design Project: Group 10

Ariel Khatchatourian, Dylan Roney, Nicole Santoro, Matthew Simiele

May 10, 2019

1 Abstract

In our project, we constructed a rover that would autonomously navigate an arena to detect four metal mines. We had to travel for at least six feet and display how far we had traveled as well as display the amount of mines we had found. In this paper we outline the various components we used and how they are all connected, the code we used to program the various components as well as the power expenditure of each component. We go through each requirement we had for the project and how we verified that each were achieved.

Contents

List of Figures

2 Project Overview

The goal of our project was to construct a rover that used autonomous navigation to detect the amount of metal mines in a 6x6 foot arena. Our rover used a metal detector that outputted zero volts when not over a piece of metal, and then when hovering over metal would output anywhere between three and four volts. In order to count the amount of metal mines detected we used an Atmega 328 to light up four LEDs once a metal mine was detected. The rover navigated the arena by using sensors to detect any walls or obstacles. Once detecting a boundary in the way the rover would then turn left or right, depending on which sensor the object was blocking. To start the movement of the rover in the arena, we had a push button connected to a transmitter that would send a signal to the receiver that would indicate the start to the pin it was connected to on the papilio board. Once the rover was on, it would then navigate the arena using the search algorithm, that determined the movement of the rover.

3 Block Diagrams

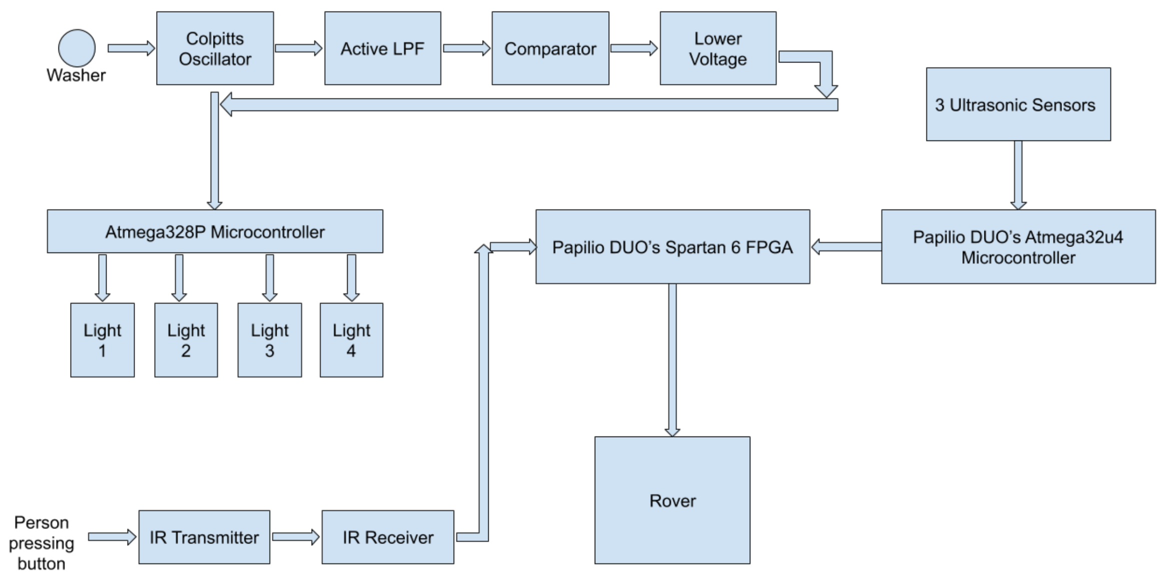

Figure 1: Block Diagram of the System

4 Requirements

4.1 System Requirements Higher Level

[ROVER 1.1] The rover shall navigate through a 6x6 foot arena.

[ROVER 1.2] The rover shall detect metal.

[ROVER 1.3] The rover shall start and stop.

[ROVER 1.4] The rover shall be able to run for 3 minutes.

[ROVER 1.5] The rover shall move at least six feet.

[ROVER 1.6] The rover shall not use lithium battery technologies.

[ROVER 1.7] The Papilio Duo should serve as the rover's main controller.

[ROVER 1.8] The rover shall be at most 2 feet wide by 2 feet long.

[ROVER 1.9] The rover shall not cause harm to anything or anyone while in use.

[ROVER 1.10] The rover shall be self contained.

4.2 System Requirements Lower Level

[NAV 1.1] The rover shall not run into obstacles. {ROVER 1.1}

[NAV 1.2] The rover shall detect the obstacle when it's within a certain distance. {ROVER 1.1}

[MOV 1.1] An Infrared transmitter shall be used to signal the infrared receiver. {ROVER 1.4}

[MOV 1.2] An infrared receiver shall signal the papilio board to start. {ROVER 1.4}

[MET 1.1] The rover shall find four metal mines. {ROVER 1.1 & 1.2}

[MET 1.2] The rover shall count and display four metal mines. {ROVER 1.2}

[DST 1.1] The rover shall display distance traveled on the Papilio board's Seven Segment Display. {ROVER 1.5}

5 Verification of Requirements

[ROVER 1.1]

This requirement was verified by test. The Computer Engineers designed a program that would allow the rover to maneuver through the arena, and successfully avoid objects. We implemented this designed with two separate projects, one in VHDL using Xilinx ISE, and the other in C using Atmel Studio. The rover's motors were controlled by the VHDL project from inputs received from the C code, which determined the configuration of the motor pins. This configuration was calculated according to the state of three ultrasonic sensors adhered to the rover. To reduce blind spots we attached the sensors to the front of the rover, one facing forward and the others on side angles. Our C program interpreted data from the sensors in the following way; If any of the sensors were within a certain distance of an object or wall, it would call the desired motor function (forward, left, right, or brake), and perform that function until the path called for a different movement. The navigation algorithm was as follows: when the front and right sensor was within 6 inches of an obstacle, the rover would turn left. If the front and left sensor was within 6 inches of a wall, the rover turned right. When only the front sensor was within 6 inches, the rover performed a right turn. In the case where the sensors all returned a value greater than 6 inches, the rover would move forward until otherwise. Though simple, this algorithm prompted the rover to cover all of the maze. Ultimately, our rover would avoid walls and obstacles and use them to change their course of travel. Logically speaking, this algorithm is similar to that of an idle TV with its logo bouncing around on the screen; Pivoting off of walls and into other ones ensures that all of the maze will be covered.

[ROVER 1.2]

This requirement was verified by test. After constructing the metal detector circuit the output was tested based on the presence of metal. The circuit will output a logical zero with no metal present (see figure 2) and a logical one with metal present (see figure 3). Since the output of the comparator is too high of a voltage for the Atmega 328 to handle a pull down resistor of 1 Mega Ohms was used (see figure 4).

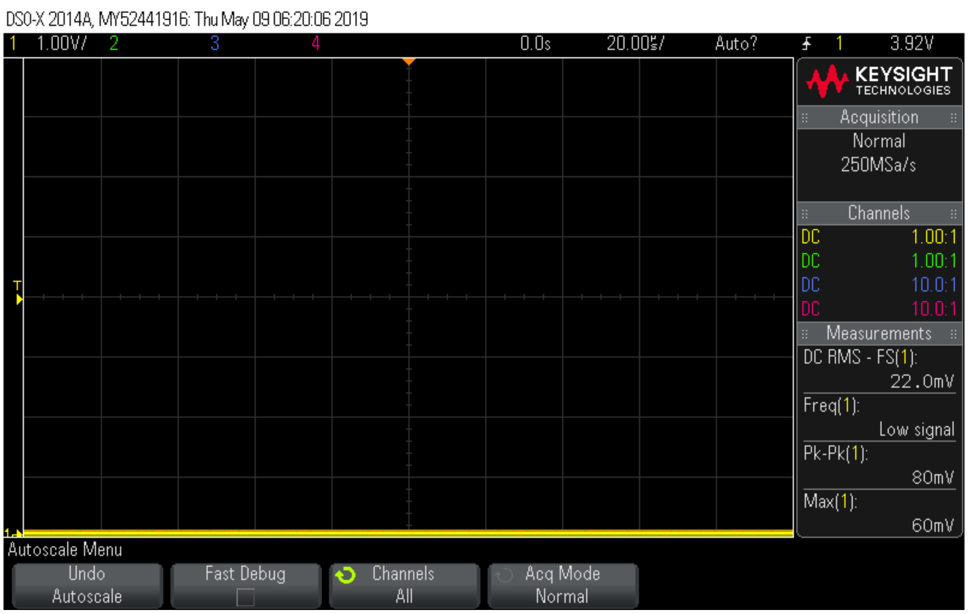

Figure 2: Output of the Comparator w/o Metal

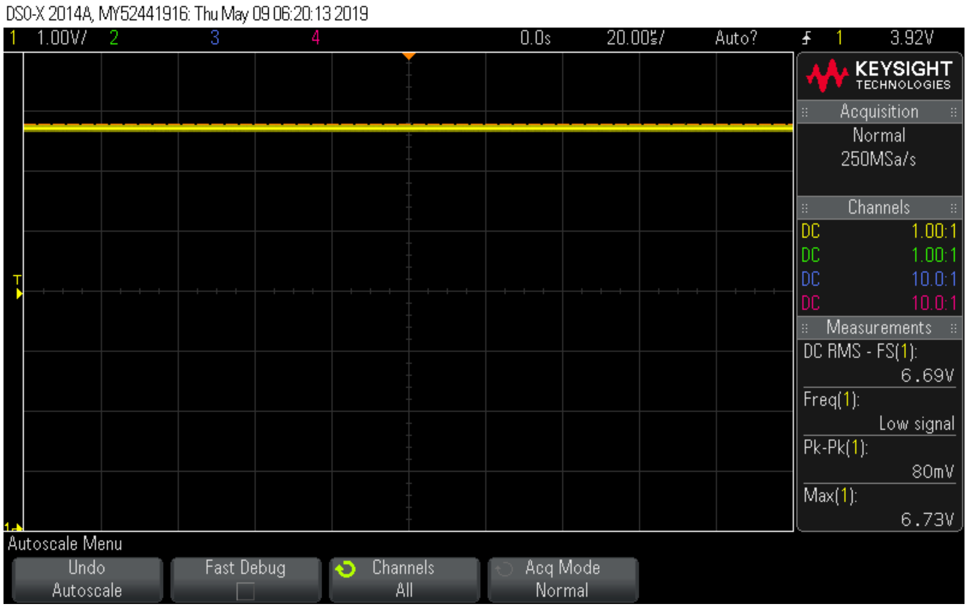

Figure 3: Output of the Comparator w/ Metal

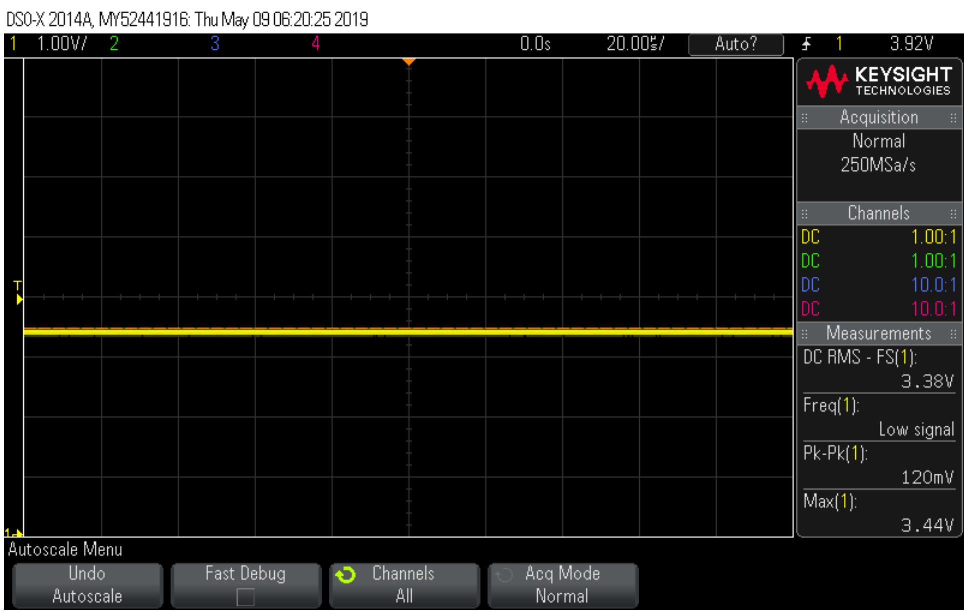

Figure 4: Output of the Comparator after the 1M Ohm Resistor

[ROVER 1.3]

To verify that the rover would start and stop on the press of a button, we tested our infrared remote. This remote was realized using an infrared transmitter and receiver. In order to confirm that each infrared component was working as anticipated, we inserted LEDs in their circuits to visualize their outputs. When pressing the push button on the transmitter, we ensured that the output of the receiver circuit would pulse high and illuminate the LED. Furthermore we verified that the LED on the receiver would remain off when the push-button remote was not being pressed, to make sure that our rover would not constantly start and stop during its mission. Moreover, we analyzed our circuits on an oscilloscope to further confirm the outputs of both the transmitter and receiver. In figure 5 and figure 8 the difference in the transmitters state is illustrated for the cases in which the button is not being pressed, and when it is. The same distinction is depicted in figure 9 and figure 10 for the receiver (A few of the figures are shown in a subsystem requirement's verification [MOV 1.1] and [MOV 1.2]).

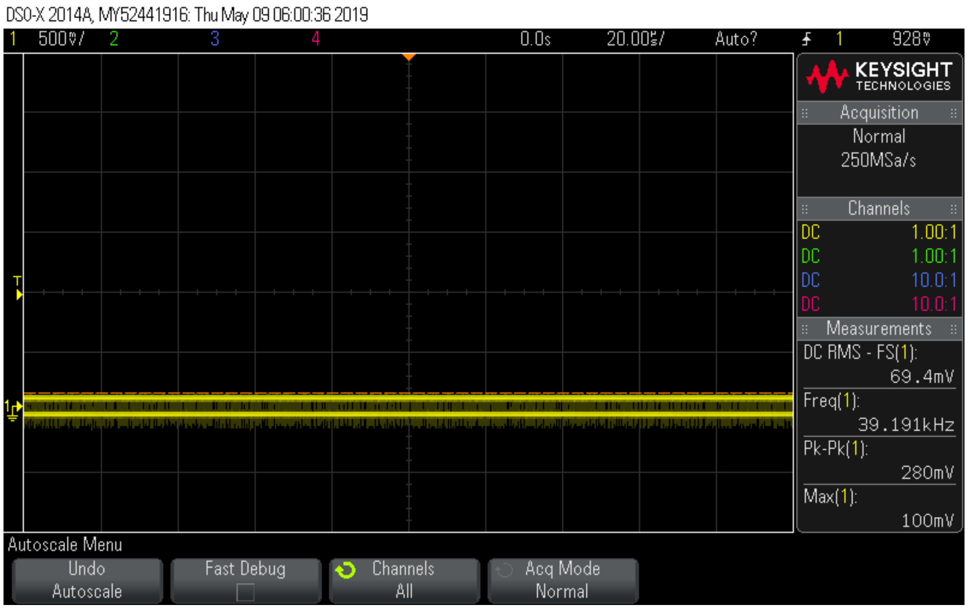

Figure 5: Output of the IR Transmitter when off (Essentially Zero)

[ROVER 1.4]

To verify this requirement, we analyzed all components of the rovers circuitry (motors, Papilio Board, Metal Detection circuit and IR Start/Stop). As seen in the appendices power calculation section, all of the components based on our choice of power supply were designed to last the full 3 minute demonstration. Moreover, the component with the briefest power supply lifespan will still last for just under two hours, proven by calculations.

[ROVER 1.5]

Testing and inspection allowed us to verify this requirement. To ensure that the rover would travel at least six feet, we designed it to move through the maze efficiently. Through interpretation of the ultrasonic sensors in our C program, the robot navigated through the arena while avoiding collision with obstacles and walls. This allowed the rover to travel as far as possible within the three given minutes. With the help of the left motor's two encoder pins, the HEXon7segDisp component in our VHDL project tracked the distance the rover traveled (in inches). This distance was then relayed to the Papilio's seven segment display to be displayed in hex digits.

[ROVER 1.6]

Inspection provided the certification of this requirement. The rover and all other components were powered either by a AA battery pack or a 9V battery. Since these power sources utilize Alkaline technologies and not Lithium, this requirement was satisfied.

[ROVER 1.7]

This requirement is verified through inspection. Besides the Papilio Duo the only other piece of hardware fixed on the rover is the Metal Detection and Counting Circuit. Since this system is self contained it does not contribute to controlling the rover, the Papilio Duo can be verified as the main controller of the rover.

[ROVER 1.8]

Through inspection, this requirement was demonstrated verified. The only external component that was attached out in front or off to the side of the rover was the metal detection coil. This coil was 2.5 inches long and less than the width of the rover wide. In total it added will about 3 inches to the front of the rover making it 12.5 inches long which is less than 2ft, satisfying this requirement.

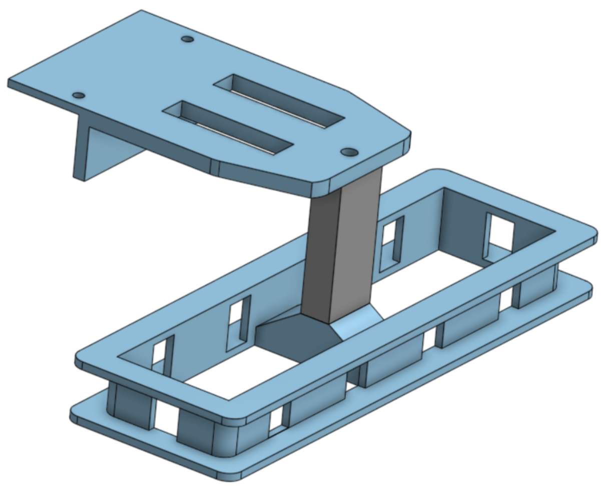

Figure 6: Metal Detecting Coil and Mount

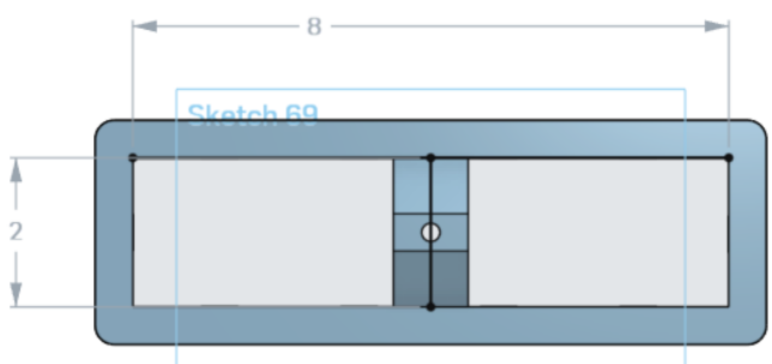

Figure 7: Dimensions of the Metal Detecting Coil

[ROVER 1.9]

Demonstration and testing authenticated this requirement. Since through extensive testing the rover did not cause harm to anything or anyone, it will presumably ensure the same level of safety in the future.

[ROVER 1.10]

This requirement was verified through testing and demonstration. The rover does not send or receive outside information or data from any source other than the IR Stop/Start remote.

[NAV 1.1]

Through many trials of navigation, this requirement was satisfied. To ensure the rover wouldn't run into any obstacles or walls during its run, we attached three ultrasonic sensors to the front of the rover. By means of repeated testing, we determined that the rover required 6 inches of clearance on all sensors to avoid collision. This 6 inches was realized by the return value of 153 for each of the sensor functions. For the instances in which at least one of the sensors was within 6 inches of an obstacle, the rover would then turn based on which of the sensors were within this distance. Permitting 6 inches of clearance allowed enough buffer for the rover to be able to turn without bumping into the walls. If the front or right sensor was within 6 inches, the rover would turn left. If the front or left sensor was within 6 inches, the rover would turn right. If none of the sensors were within 6 inches, the rover would move forward until otherwise.

[NAV 1.2]

While we extensively tested the ultrasonic sensors we affixed to the front of the rover, the notion of using these sensors to detect an obstacle was lamented in earlier labs in this course. Both of these implementations of such sensors demonstrated that the requirement was achieved. We integrated our three ultrasonic sensors with individual functions in our C program, to be called by the main function. The sensors functioned by way of a universal trigger and individual echo's. Measuring the amount of time between the trigger pulse and the echo pulse, the distance between the sensors and obstacles/walls was concluded. To further improve the spacial awareness of our rover, we mounted all the sensors at the front of the rover, one facing forward and the others angled to the sides. According to the SparkFun ultrasonic sensor datasheet, these sensors are capable of determining distance ranging from 2 centimeters up to 4 meters.

[MOV 1.1]

This requirement is verified through analysis and test. The transmitter circuit was designed to output pulses with frequency 38kHz based on the rating of the TSOP 1736 (IR Receiver). As shown in figure 8 the pulses sent to the transmitting diode are in fact 38kHz in frequency and will work with the TSOP 1736.

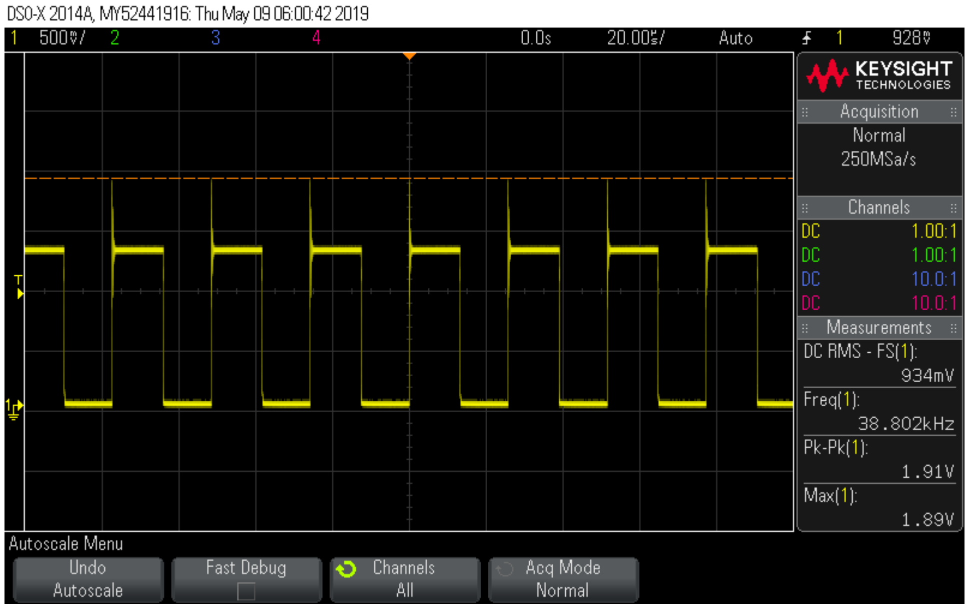

Figure 8: Output of the IR Transmitter Circuit

The transmitter built communicated to our receiver using the same frequency with the transmitting LED and TSOP1738. Therefore, whenever the push button for the transmitter was pressed it would send a wave signal to the TSOP1738 with the correct frequency and would then signal that circuit.

[MOV 1.2]

This requirement is verified through test. After constructing the IR Transmitter circuit the IR Receiver circuit was then tested in order to determine if the system was working properly. The reciever circuit outputs an active low signal (about $2.6V$ when off and $0.1V$ when on). This can be seen in figure 9 and figure 10.

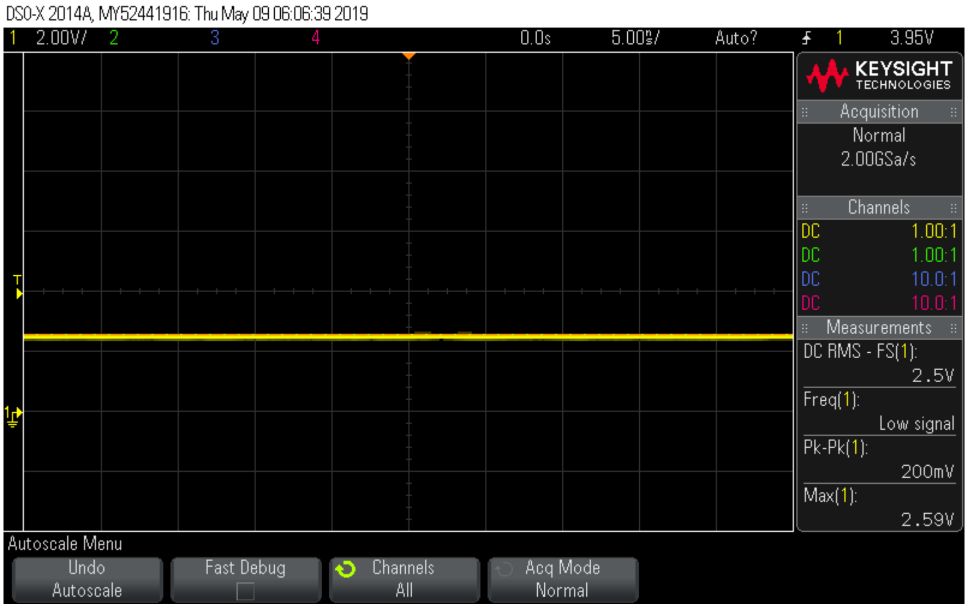

Figure 9: Output of the IR Receiver Circuit Off

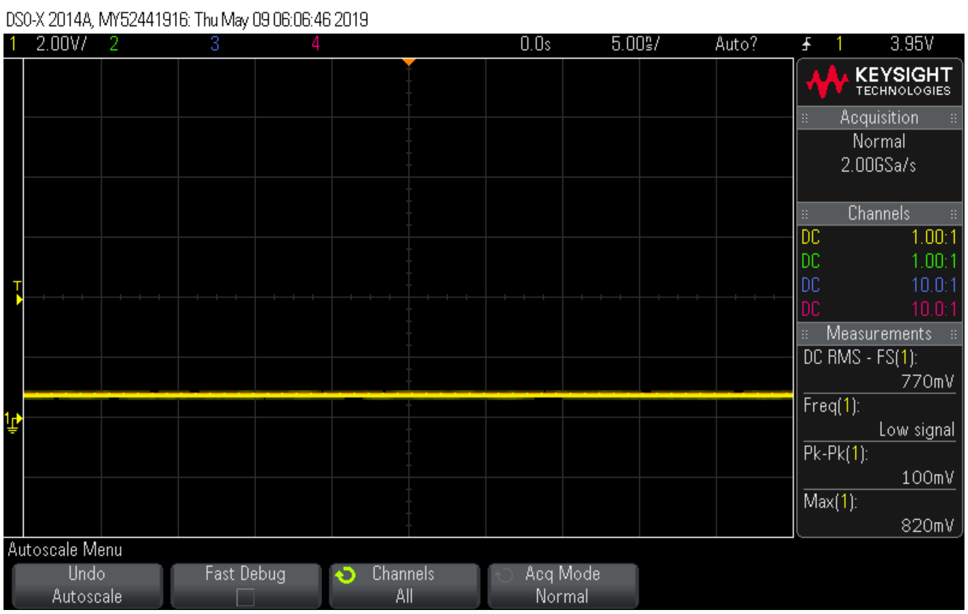

Figure 10: Output of the IR Receiver Circuit On

The circuit for the receiver outputted high to the pin tied to the pin on the papilio board. We checked this by having an LED connected to the pin going to the papilio. Each time we pressed the push button on the transmitter we made sure the LED would light up on the receiver.

[MET 1.1]

The way we ensured that the rover would be able to find four mines within the arena was by using a random search algorithm based on off of our sensor values. The robot turned right if the front or left sensor was within a certain distance, and turned left when the left or right sensor was within a certain distance. Our rover was right turn biased(might be left), so it would always be checked the left and front sensor first before it considered the values of the right sensor. When all the sensors weren't within the certain distance (6 inches), the rover would fall into the else condition in the code. The else condition had a case statement that was determined which case was used based on a random number between zero and two. Based on the certain condition, the rover would go straight for a given amount of time, then do a delay, and turn right or left for a short period of time, before doing another delay, and then going straight again.

[MET 1.2]

This requirement is verified through test. To achieve this requirement, we used an Atmega 328 with four LEDs on output pins and the output from the metal detector on the input pin. In our C code uploaded to the Atmega, we had an array with each output pin all initially set to zero. We would set one pin high once a voltage was detected on the input pin and step through the array. To make sure that the Atmega would only count once for each metal mine found we would delay the code before continuously go through. We found this delay value by experimenting with the metal detector and rover. We verified this was achieved by holding the rover in place and placing metal under the metal detector. When holding it under there for a while we made sure that the LEDs would each light up.

[DST 1.1]

In order to output the distance the rover travelled on the seven segment display, we created the HEXon7segDisp VHDL component. This instance used the two left motor encoder pins to determine the distance travelled. To simplify our calculations, we assigned one central encoder signal to be the exclusive or of the two encoder pins. By way of trial and error, we determined that for every 59 ticks of the encoder signal the rover traveled one inch. Following this conclusion, we created a counter process that kept count of the encoder ticks; When the preliminary count reached 59, the official distance counter was incremented, and the encoder tick counter reset to 0.

Once our final distance counter was reliable, we used several additional processes to display the distance on the seven segment display. Such processes include four multiplexers, one for each of the four character slots (anodes) on the display. These slots were to be filled with the hex equivalent of the 4 corresponding distance signal bits; Such that the first four least significant bits of the 16 bit distance signal were displayed in hex on the leftmost anode. This same logic was used to determine the appropriate hex digits for the three following anodes. These multiplexer processes used their assigned 4 bits of the distance signal as their select signals. For simplicity sake, we created constants containing a 7 bit segment sequence for each possible hex character. When selected, the segment string set each of the 7 reverse active segment cathodes necessary to display the hex representation of the select. Though only one cathode configuration may be present on any of the anodes at a given time, we were able to overcome this limitation to seemingly display four different digits on each of the display anodes. The remaining two processes of the hex display component used the most significant bits of a rapidly increasing clock base counter as their selects. Due to the cyclic nature of these selects, the processes continually stepped through each of the cases. Rapidly switching the 7 segment value of the cathodes and which singular anode was on at a given time created the illusion of 4 distinct hex digits existing on the display. This cycling technique permitted us to display the amount of inches the rover traveled on the seven segment display, in hex.

6 Technical Design

6.1 EE Metal Detector

The metal detector circuit is based off an application of a colpitts oscillator. The Colpitts Oscillator main driving component is an LC tank circuit. A gain device (in this case an NPN Transistor) is used to stabilize the circuit and prevent unwanted frequencies. The inductance of the coil was determined using the formula in equation. This inductance value was computed based on the oscillating frequency noticed in the lab and is verified by using PSpice (see figure 11) the frequency is a bit off from what is observed in lab but not enough to revoke the calculation of the inductance of the coil (see equation 1). Since introducing metal to a coil changes it's properties such as inductance, which will then change the oscillation frequency and DC voltage, this can be exploited in order to properly detect metal mines. The oscillations with and without metal are shown in figure 12 and 13.

$$L = \frac{R \tan \phi}{2\pi f} = \frac{10\Omega \tan 87.4°}{2\pi \cdot 120kHz} = 292.07\mu H \tag{1}$$

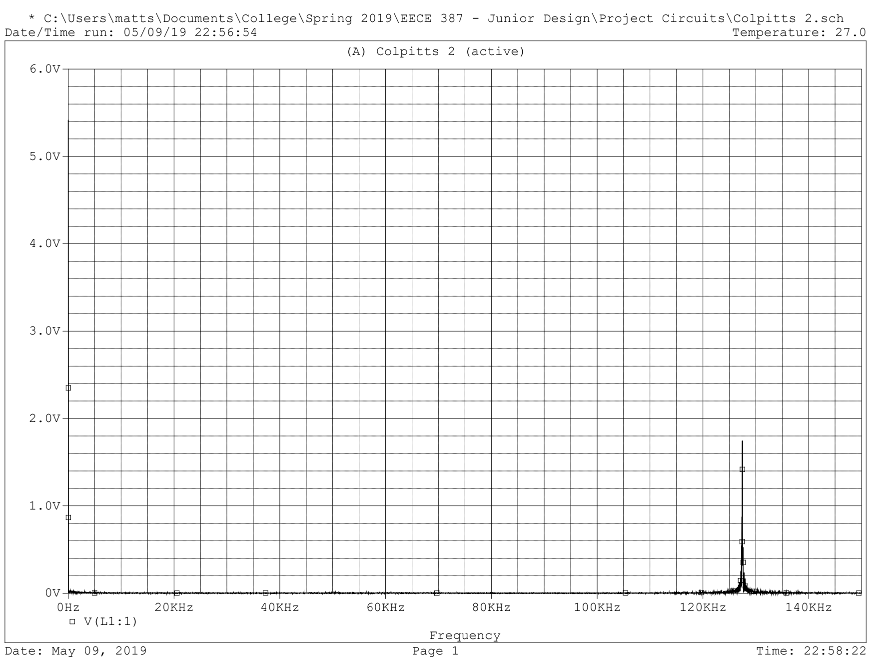

Figure 11: Frequency Domain Analysis of the Colpitts Oscillator

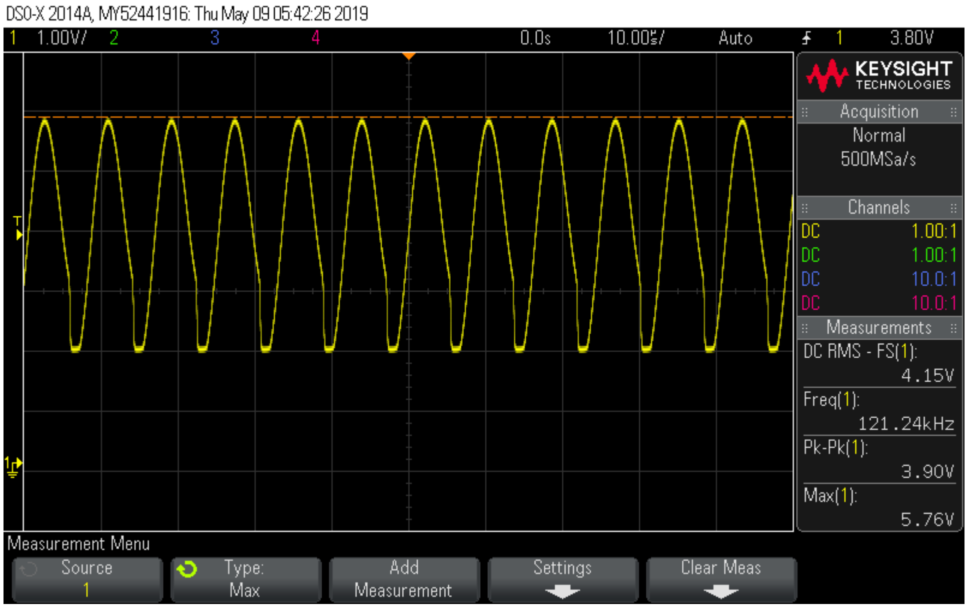

Figure 12: Output of the Colpitts w/o Metal

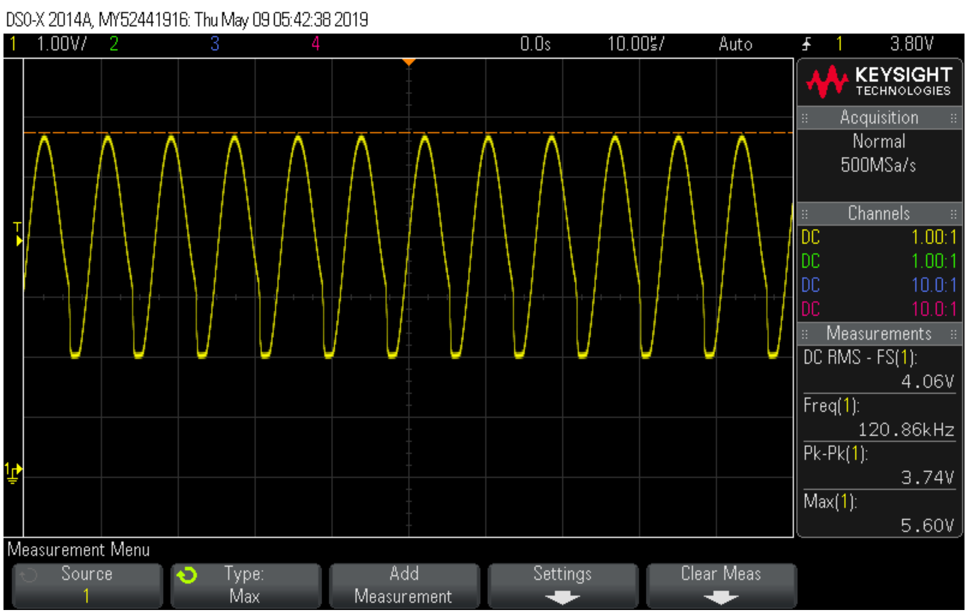

Figure 13: Output of the Colpitts w/ Metal

Since there is a change in the DC voltage on the output of the oscillator an active lowpass filter can be designed to remove all sinusoidal components. Using equation 2 a filter was designed to only pass very small frequencies as to only get a DC voltage at the output. The filter is designed to pass frequencies lower than 1,000Hz which is small enough to filter out our 120kHz oscillation. The difference in DC voltage on the output of the lowpass filter is shown in figure 14 without metal and figure 15 with metal.

$$f_c = \frac{1}{RC} = \frac{1}{100k\Omega \cdot 10nF} = 1000Hz \tag{2}$$

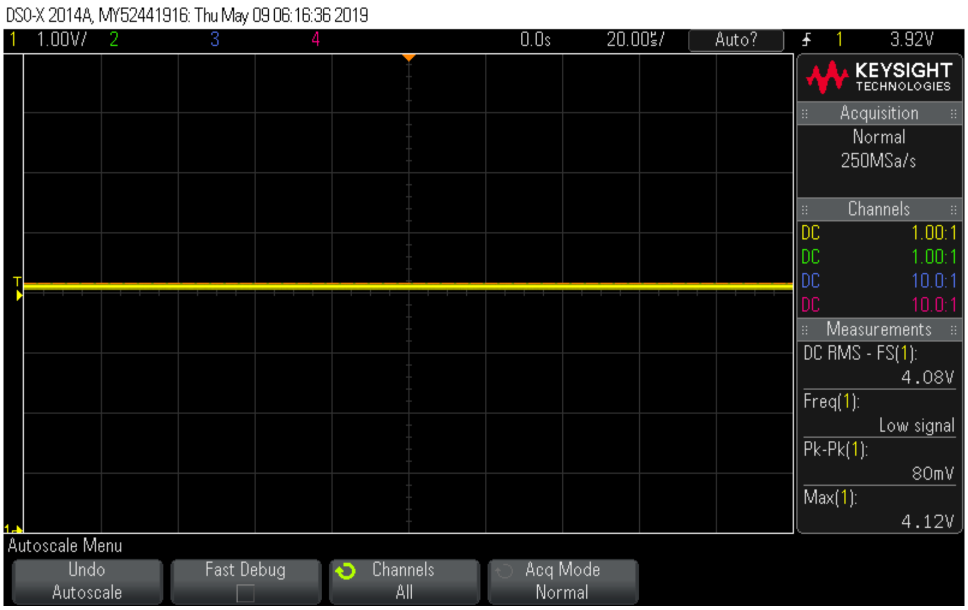

Figure 14: Output of the LPF w/o Metal

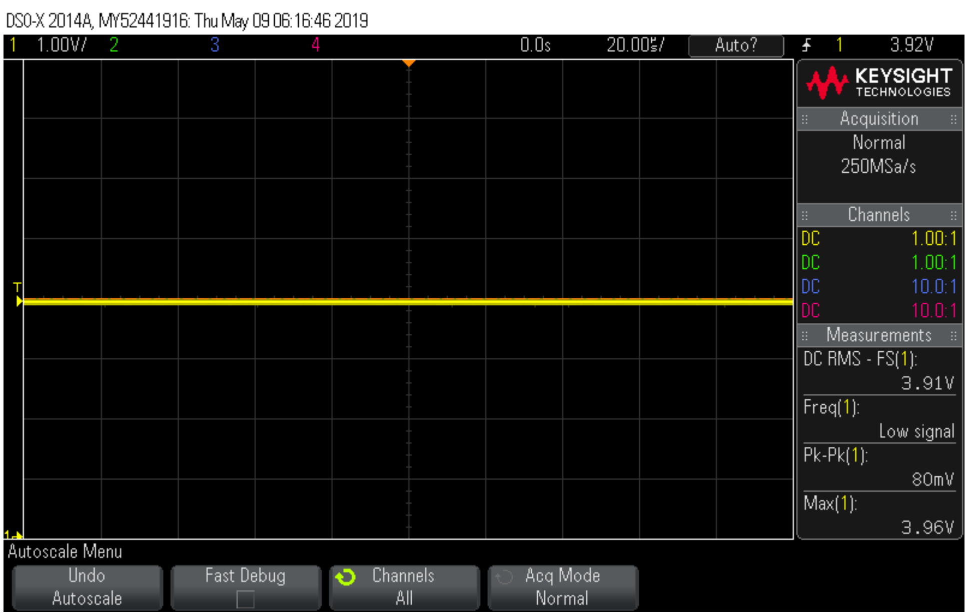

Figure 15: Output of the LPF w/ Metal

Since the output of the active low-pass filter passes only a DC voltage a comparator can then be used to make sure the output of the circuit is a logical one when metal is present and a logical zero when metal is not present. Since a battery was being used to power the circuit a potentiometer was used in the Voltage divider so that as the battery drains throughout the day the circuit will still function as desired. Since the voltage on the output of the comparator is too large for the Atmega to handle a pull down resistor was used during the demo to limit the current and voltage. In the PSpice schematic a voltage divider since it is a better practice. The result of the pulldown resistor on the input of the Atmega is shown in figure 16. The full schematic for the metal detector circuit can be found in figure 17.

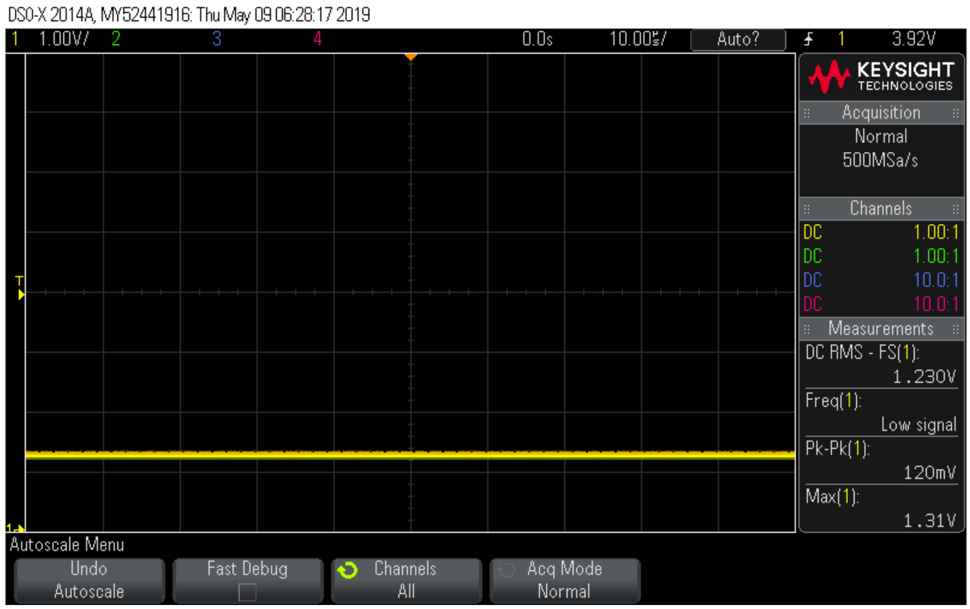

Figure 16: Input Signal to the Atmega w/ Metal

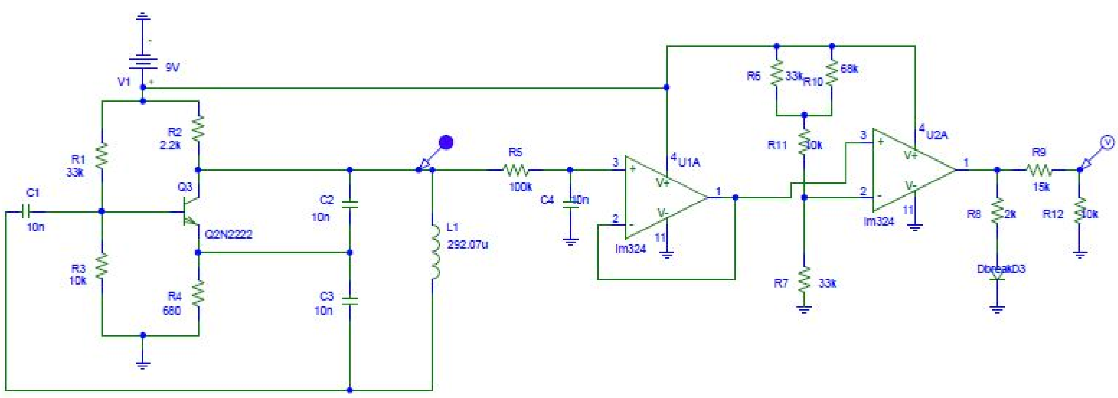

Figure 17: Metal Detector Circuit Schematic

6.2 EE Infrared Stop/Start

We used an infrared transmitter and receiver to start and stop the rover's movements. The infrared transmitter was constructed using a 555 timer. The transmitter is a stand alone circuit treated as a remote. The schematic for this transmitter can be found in figure 19. On our output pin we had a push button that when pressed completed the open circuit and would send the voltage and current to the special IR LED. This special IR LED would emit an infrared wave that would be received by the TSOP 1736. The 555 timer was used in the transmitter circuit to create a square wave of 38kHz. When the push button was pressed it connected these two circuits together which allowed for the signal to be sent.

The receiver circuit that we constructed can also be seen in figure 18. We used a TSOP 1736 to receive the signal sent by the IR LED. Normally, the TSOP 1736 output remains high and once the signal is received the output goes high. A PNP transistor was used as a switch to output a logical 1 when no signal is present from the transmitter and a logical 0 when a signal is detected by the TSOP 1736. The PNP transistor acts as an open circuit when a positive voltage is applied to the base, so when the TSOP is not receiving a signal it is giving a positive output and therefore causes an open circuit not lighting up the LED. When it receives a signal it does not apply a voltage to the base and therefore completes the circuit causing the LED to go on and a signal to be sent to the papilio to begin.

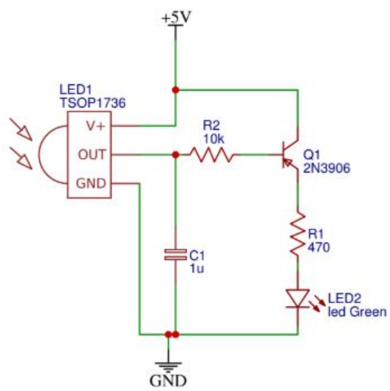

Figure 18: IR Receiver Schematic

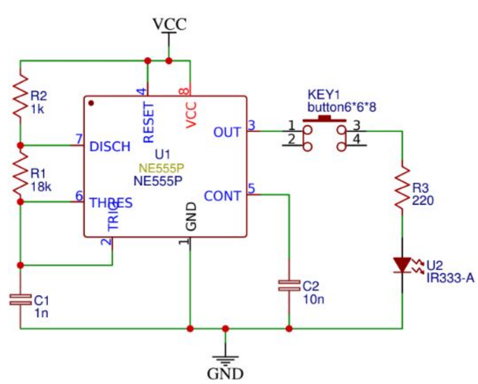

Figure 19: IR Transmitter Schematic

7 Design Integration

To integrate our metal detector with the rover, we had a 3D printed a holder for the coil. This 3D printed platform was able to screw into the rover base and would stick the coil out in front. It was close enough to the ground that it would be able to detect the metal it would go over.

In order to track the amount of metal mines found we had to integrate the metal detector with the metal mine counter. We used the output pin of the metal detector as the input to our metal mine counter. We mapped a pin as the input and connected the two with a wire, no resistor was necessary.

Our infrared start to the rover was integrated by connecting the receiver circuit to the papilio board. The transmitter would send a start signal to the receiver, which would (once signaled) tell the papilio to turn on and start. The output of the receiver was connected to pin 98 of the papilio. When the button for the for the infrared sensor remote was pressed, a 0 was outputted onto pin 98, active low.

8 Power Management

To power up our various circuits we had to use a few battery packs. We had a 9V battery pack used to power up our rover's motors as well as the papilio board. We needed a 5V regulator to lower the voltage for the papilio to ensure that it wouldn't exceed the maximum voltage requirement of the board. Another 9V battery was used to power up the metal detector, the receiver circuit and the metal mine counter circuit a 5V regulator was used to regulate the voltage entering the Atmega328. A 9V battery was also used to power up the transmitter circuit. This used its own battery to allow us to hold the circuit as a remote and not be connected to everything else. Therefore, a total of 1 9V battery pack and 2 9V batteries were used in our construction, and as seen in our power calculations in figure 23 no one circuit used too much power and allowed our rover to easily run for three minutes without any issues.

9 Software

9.1 Main.c

Everything we coded in C was in our main.c program, made using Atmel Studio software. In this program, we created four different motor functions. Additionally, we constructed sensor functions for each of the three sensors. These rudimentary functions allowed us to design and implement a search algorithm successfully, from the ground up. At the start off our code, we included all necessary C libraries, defined the speed of the clock processor and declared all function signatures. Following these declarations, we defined our ports for the Atmega32u4 microprocessor. Our main.c program accepts echo inputs for the three sensors on the ports for Switches 3-1 (Ports B4 B5 & D3). Moreover, our project uses the ports for LEDs 6-3 (Ports D0 D1 D4 & D6) to output the motor pin sequence to be routed through the FPGA and the port for Switch 0 (Port D2) to send out the universal trigger for the 3 ultrasonic sensors. We were able to define these ports as either inputs (pins) or outputs (ports) with the use of bitmasking.

In order to outline the basic rover movements, we implemented four functions: forward, right_turn, left_turn, and brake. All four of these functions assign the motor ports to their necessary values for the rover to perform the specific navigational movement. Bitmasking allowed us to assign the ports their values. To set a specific port number to 0, we used the bitwise AND operation. Conversely, using the bitwise OR operation allowed us to set a specific port number to 1.

Each of the sensor functions return an 8 bit unsigned variable (uint8_t), when called. The value they return is determined by the distance from the sensor to the obstacle in front of it. To keep track of this distance, a variable is initialized to 0 in each of these functions. Following this declaration, a 15us pulse is realized on the universal trigger. In order to measure the sensor distance, after the trigger is pulsed, the function remains in a while loop until the sensor's echo pin has a rising edge. The distance variable is then incremented after a delay, until the sensor's echo pin has a falling edge or the variable is greater than 255, as this value (about 12 feet) is out of range for the sensor. After calculation, the sensor function returns this distance variable. Although the values aren't represented in inches, we determined its conversion factor. Our rover requires six inches of clearance, which equates to the distance variable value of 153.

Once our basic sensor and movement functions were tested and implemented, we designed our rover's search algorithm. For the sake of keeping track of the distance values returned by the sensor functions, we initialized three 8 bit unsigned variables for each sensor. Our navigation pattern was then implemented in a while loop in our main function to be infinitely executed. Starting off our algorithm, we call each of the sensor functions and set their return values to their corresponding distance variables. In between each of these function calls, as well as in between many other lines of code, we use the _delay_ms() function. This function is defined in the util/delay C library and causes a timing delay of the specified amount of microseconds. Such delays are necessary in ensuring that our code is executed by the Papilios microprocessor as we expect, in the order we expect.

Succeeding the call of the sensor functions, our program analyzes the values of the left right and front distance variables. As part of our algorithm we declared a right turn bias. This bias prompted our rover to prioritize right turns where available. Our bias was retained in most scenarios. The first exception to this bias occured when both the front and right sensor were within 6 inches of an obstacle, the rover turned left in this instance. Finally, if all sensors returned distances greater than 6 inches, the rover moved forward to traverse across the middle of the arena. Due to the nature of the while loop with a condition that is constantly true (1), the algorithm was continually executed. By means of this execution, the microcontroller constantly received updated sensor distances. These updates allowed our rover to have spacial awareness and reliably execute the desired movement. Though our C program outputs values for the motor pins on LEDs 6-3, these values were routed through the FPGA before actually being sent to the motors. In programming the Papilio's microcontroller, this C program allowed our rover to successfully move through the maze and avoid obstacles, while being able to detect all 4 mines.

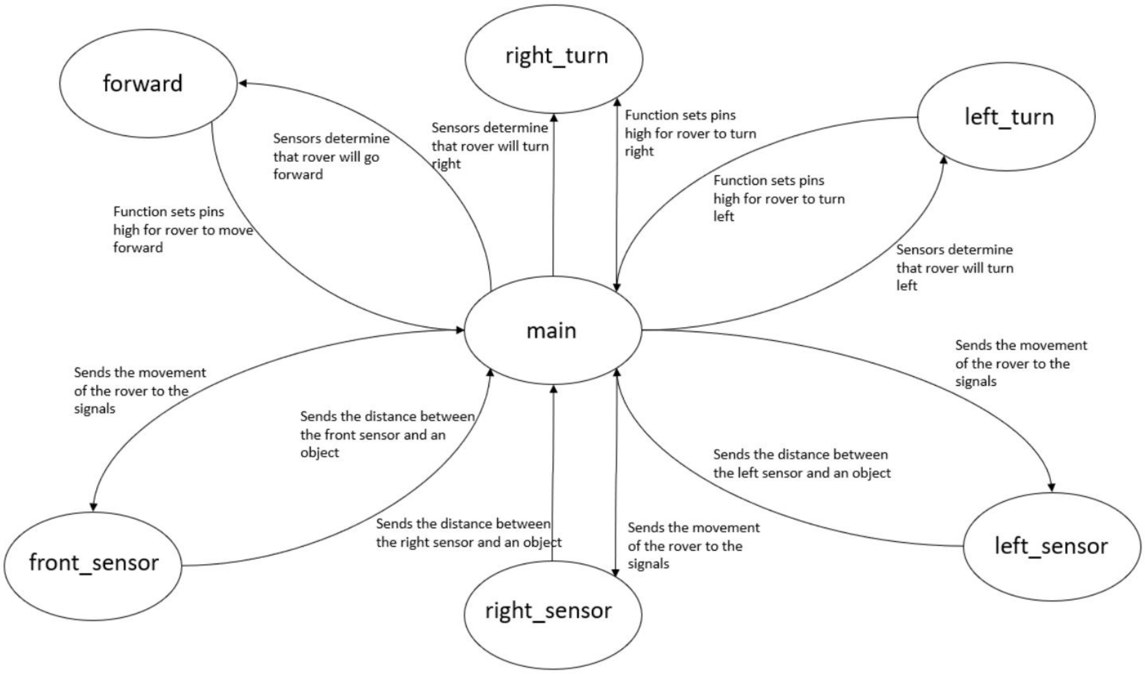

Figure 20: Flowchart for C code

9.2 VHDL Code

Using Xilinx ISE software, we created a VHDL project with two components: Motors and HEXon7segDisp. Due to the nature of VHDL programming, these instances are connected through a top level file, which receives inputs and sends outputs on the pins specified in the UCF file.

9.2.1 Motors.vhd

As routed by the C programmed microcontroller, this component receives the desired values for the four motor pins. Additionally, this component receives a reverse active input from the infrared receiver circuit, on VGA pin Red1. Before deciding which values to forward to the rover's motor pins, the program considers the mode of operation as specified by Red1. Each time the infrared transmitter remote is pressed, Red1 experiences a falling edge. The first process in the instance of Motors creates two signals for the press, each of which are delayed by one clock cycle. The program is able to detect a falling edge on Red1 by comparing the two delayed signals; If the first delayed signal has a value of one and the second delayed signal has a value of zero, a falling edge has occurred. Once the button press has been detected, the mode of operation switches between start and stop. When the rover is in its stop mode, the FPGA sends the value of one to each of the four motor pins, this halts the rover. Otherwise in start mode, the Motor component forwards the motor values from the C code to the motor pins on the rover. The routing of motor pins is necessary, as only the FPGA has compatible pins.

9.2.2 HEXon7segDisp.vhd:

For efficiency's sake, the HEXon7segDisp component is multipurpose. This component accepts inputs from the left motor's two encoder pins and outputs 7 bit HexSel signal and 3 bit an_out signal to the Papilio's seven segment display. Initially, this component uses the two encoder inputs to create a singular encoder signal, the exclusive or of the two inputs. After significant trial and error, we determined that every 59 "ticks" of this encoder signal, counted by the signal data_sig equated to one inch traveled by the rover. To perform this conversion, we created a process that incremented the data signal whenever data_sig reached the value of 59, this dummy signal was then reset to perform the next count. Once the distance traveled has been determined, the component forwards this value to be displayed on the Papilio's seven segment display. Due to the nature of the display, multiple processes were required to achieve this display. The most glaring limitation of the seven segment display is that each of the 4 character slots are connected. Though you may illuminate any number of the anodes at a time, they all share the same cathode values. The display consists of 7 cathodes, one for each of the seven segments comprising the hex digit. In order to overcome the limitation of only being able to display one hex digit at a time, we used multiple VHDL processes.

Such processes include four multiplexers, one for each of the four character slots (powered by anodes) on the display. These slots were to be filled with the hex equivalent of the 4 corresponding distance signal bits; Such that the first four least significant bits of the 16 bit distance signal were displayed in hex on the leftmost anode. This same logic was used to determine the appropriate hex digits for the three following anodes. These multiplexer processes used their assigned 4 bits of the distance signal as their select signals. For simplicity sake, we created constants containing a 7 bit segment sequence for each possible hex character. When selected, the segment string set each of the 7 reverse active segment cathodes necessary to display the hex representation of the select. Though only one cathode configuration may be present on any of the anodes at a given time, we were able to overcome this limitation to seemingly display four different digits on each of the display anodes. The remaining two processes of the hex display component used the most significant bits of a rapidly increasing clock base counter as their selects. Due to the cyclic nature of these selects, the processes continually stepped through each of the cases. Rapidly switching the 7 segment value of the cathodes and which singular anode was on at a given time created the illusion of 4 distinct hex digits existing on the display. This cycling technique permitted us to display the amount of inches the rover traveled on the seven segment display, in hex.

9.2.3 PapilioDUO-LogicSheild-general.ucf

The final "component" of our VHDL project was the most vital, the user constraints file (UCF). Though the top level file was able to assemble and connect our Motors and HEXon7segDisp, it required a UCF to assign variables to their input and output pins. This file contains the variables for the inputs and outputs of the entire VHDL project and their corresponding FPGA pin mappings. Implementing the UCF file permitted communication between the microcontroller and the FPGA as well as the FPGA and the rover.

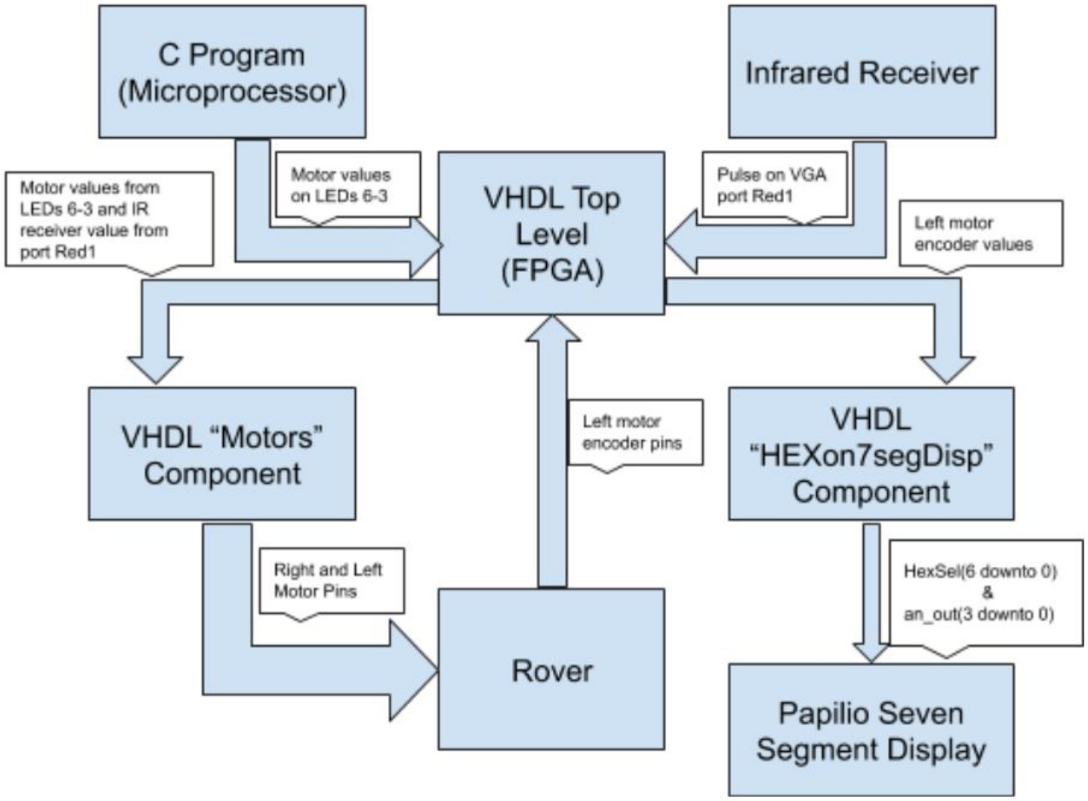

Figure 21: Flowchart for VHDL code

9.3 Mine Counter

In constructing our metal mine counter, we programmed an Atmega 328P microcontroller in C using Atmel Studio. Though we had initially planned to use the Papilio to display the amount of mines found; Due to push button malfunction, we ran out of available microcontroller pins.

Similarly to the main C code for the Papilio's microcontroller, we included all integral C libraries, defined the speed of the clock processor. Additionally, we defined macros for the MetalDetector and each of the four mines. These macros were assigned values based on their respective port/pin number. Following these preprocessor declarations, we utilized bit masking to define the direction of our ports. Firstly, we initialized ports C0, C1, C2, & C3 as outputs (ports) and port B1 as an input (pin) by setting them to 0 and 1 respectively (Mine Count.c). After this, we constructed an array containing each output pin, representing the four mines to be found. In the infinite while loop, we started our index i at 0, which is the first position in the array and the first LED on the board. While looping through, once a metal mine was detected, in this case a value was found on the input pin, the code would set the i-th index in the array to one. The index was then incremented. In order to ensure each mine was only counted once, we implemented delays in the while loop. Such delays prevented the mine detection loop from starting again and counting a single mine more than once. Through experimentation, we decided that a delay of 500ms would allow each mine to be counted only once, while still allowing the detection of another metal washer if it was found fairly quickly in succession. Each output port of the Atmega328P microcontroller had a 220 ohm resistor soldered to an LED connected to ground. We used a resistor in series with a LED to ensure there wasn't too much current going through it, as a precautionary measure. Once a rising edge occurred on any of the four output ports, its respective LED would light up, signaling that a metal mine was detected. The illumination of all four LEDs indicated that all four metal mines were successfully detected.

10 Appendices

10.1 Bill of Materials

ID	Type	Name/Value	Quantity	Manufacturer Part	Manufacturer	Supplier	Supplier Part
1	Resistor	220	5	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
2	Resistor	470	1	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
3	Resistor	680	1	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
4	Resistor	1k	1	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
5	Resistor	2k	1	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
6	Resistor	2.2k	1	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
7	Resistor	6.8k	1	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
8	Resistor	10k	2	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
9	Resistor	15k	1	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
10	Resistor	18k	1	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
11	Resistor	33k	2	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
12	Resistor	68k	1	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
13	Resistor	100k	2	S 0603 C P X 150 J 20 - TR	State of the Art, INC.	LAB	
14	Capacitor	1n	1	TCJ-A-226-M-004-R-0300-E	AVX	LAB	
15	Capacitor	10n	5	TCJ-A-226-M-004-R-0300-E	AVX	LAB	
16	Capacitor	1u	1	TCJ-A-226-M-004-R-0300-E	AVX	LAB	
17	LED	LED Green	6	led Green	KENTO	LCSC	C14641
18	IR LED	IR333-A	1	950nm	Everlight Elec	LCSC	C95407
19	IR Reciever	TSOP1736	1	TSOP1736	VISHAY	LCSC	C5125
20	Magnet Wire	50ft Mag Wire	1		BNTECHGO	Amazon	
21	NPN Transistor	2N2222A	1	2N2222A	Foshan Blue Rocket Elec	LCSC	C358533
22	PNP Transistor	2N3906	1	2N3906	CJ	LCSC	C9809
23	OP AMP	LM324	1	LM324	PUOLOP	LCSC	C351427
24	556 Timer	NA556	1	NA556	TI	LCSC	C46749
25	Push Button	button6*6*8	1	button6*6*8	ReliaPro	LCSC	C59982
26	Ultrasonic Sensors		3			LAB	
27	Logic Start Shield for FPGA	Logic Start Shield	1		Gadget Factory	LAB	
28	FPGA	Papilio Duo	1		Gadget Factory	LAB	
29	Atmega328	Atmega328p	1	ATMEGA328P-PU	Atmel	LAB	
30	Voltage Regulator	5V Regulator	2	L7805	ST	Lab	
31	Battery	9V Battery	2	9V Battery	Rayovac	Walmart	
32	Battery	9V Battery Pack	1			Lab	

Figure 22: Bill of Materials

10.2 Power Calculations

Circuit	Part	V (Volts)	I (A)	R (Ohms)	Power (Watts)	Watt Hours of Batteries (Wh)	
IR Transmitter	556 Timer	9	0.00023077		0.00207693	6 AA	9
		4.227	0.01591		0.06725157	9V	5.49
	18k Ohm		0.000000000009	18000	0.00000000000000000014580		
	1k Ohm		0.000000004067	1000	0.0000000000000165404890		
	220 Ohm		0.01591	220	0.0556881820		
	LED (Diode)	0.72825	0.01591		0.0115864575		
	Total Power in Watts				0.1366031395		
	Hours Able to be Used				40.18941307		
Metal Detector	33k Ohm	6.999	0.00021211	33000	0.00148455789		
	2.2k Ohm	4.302	0.001956	2200	0.008414712		
	10k Ohm	2.001	0.00020005	10000	0.00040030005		
	680 Ohm	1.338	0.001968	680	0.002633184		
	2N2222	3.36	0.001956		0.00657216		
		0.663	0.00001206		0.00000799578		
	100k Ohm		0.0000004466	100000	0.0000000019945156		
	LM324	9	0.001006		0.009054		
		4.702	0.0000004582		0.0000021544564		
		9	0.00099056		0.00891504		
		8.113	0.00372		0.03018036		
	33k Ohm	3.065	0.00009288	33000	0.0002846772		
	68k Ohm	3.065	0.00004507	68000	0.00013813955		
	10k Pot	1.38	0.00013795	10000	0.000190371		
	33k Ohm (2)	4.555	0.00013804	33000	0.0006287722		
	LED (Diode)	0.68939	0.003712		0.00255901568		
	2k Ohm	7.42361	0.003712	2000	0.02755644032		
	15k Ohm	4.868	0.00032451	15000	0.00157971468		
	10k Ohm	3.245	0.00032451	10000	0.00105303495		
	ATMEGA 328	5	0.0015		0.0075		
	220 Ohm (*4)	2.3	0.000375	4	0.00345		
	LED (Diode*4)	0.7	0.000375	4	0.00105		
	Max Power in Watts				0.1136526909		
	Hours able to be Used				48.30505951		
Motors, Sensors, IR Reciever and Papilio	Motor 1	9	0.21		1.89		
	Motor 2	9	0.21		1.89		
	Sensors				0		
	FPGA	5	0.142		0.71		
	TSOP 1736	5	0.00033		0.00165		
	PNP	0.7	0.00033		0.000231		
		2.6	0.00329		0.008554		
	470 Ohm	1.7	0.00362		0.006154		
	LED (Diode)	0.7	0.00362		0.002534		
	Total Power in Watts				4.509123		
	Total Hours for Use				1.995953537		

Figure 23: Power Calculations

10.3 Source Code

UCF.vhd

```
## Junior Design Team 10
## UCF file for the Papilio DUO board
```

```
##
## Main board wing pin [] to FPGA pin Pxx map
## -------C-------    -------B-------    -------A-------
## [GND] [C00] P114   [GND] [B00] P99    P100 [A15]
## [2V5] [C01] P115   [2V5] [B01] P97    P98 [A14]
## [3V3] [C02] P116   [3V3] [B02] P92    P93 [A13]
## [5V0] [C03] P117   [5V0] [B03] P87    P88 [A12]
##       [C04] P118         [B04] P84    P85 [A11] [5V0]
##       [C05] P119         [B05] P82    P83 [A10] [3V3]
##       [C06] P120         [B06] P80    P81 [A09] [2V5]
##       [C07] P121         [B07] P78    P79 [A08] [GND]
## [GND] [C08] P123   [GND] [B08] P74    P75 [A07]
## [2V5] [C09] P124   [2V5] [B09] P95    P67 [A06]
## [3V3] [C10] P126   [3V3] [B10] P62    P66 [A05]
## [5V0] [C11] P127   [5V0] [B11] P59    P61 [A04]
##       [C12] P131         [B12] P57    P58 [A03] [5V0]
##       [C13] P132         [B13] P55    P56 [A02] [3V3]
##       [C14] P133         [B14] P50    P51 [A01] [2V5]
##       [C15] P134         [B15] P47    P48 [A00] [GND]
#
                                                                                                   # Atmega32u4 Pins Atmega32u4 Pin
NET ARDUINO_RESET         LOC="P139"  | IOSTANDARD=LVTTL;
NET CLK                  LOC="P94"   | IOSTANDARD=LVTTL | PERIOD=31.25ns;
NET an_out(3)           LOC="P83"   | IOSTANDARD=LVTTL;
NET an_out(2)                  LOC="P81"   | IOSTANDARD=LVTTL;
NET an_out(1)           LOC="P75"   | IOSTANDARD=LVTTL;
NET an_out(0)           LOC="P67"   | IOSTANDARD=LVTTL;
NET hexsel(6)               LOC="P114"  | IOSTANDARD=LVTTL;
NET hexsel(5)               LOC="P111"  | IOSTANDARD=LVTTL;
NET hexsel(4)              LOC="P102"  | IOSTANDARD=LVTTL;
NET hexsel(3)               LOC="P97"   | IOSTANDARD=LVTTL;
NET hexsel(2)              LOC="P112"  | IOSTANDARD=LVTTL;
NET hexsel(1)               LOC="P115"  | IOSTANDARD=LVTTL;
NET hexsel(0)               LOC="P105"  | IOSTANDARD=LVTTL;
NET Red1                  LOC="P98"   | IOSTANDARD=LVTTL;
NET Motor_La                   LOC="P39"   | IOSTANDARD=LVTTL;
NET Motor_Lb                   LOC="P48"   | IOSTANDARD=LVTTL;
NET Motor_Ra                   LOC="P58"   | IOSTANDARD=LVTTL;
NET Motor_Rb                   LOC="P61"   | IOSTANDARD=LVTTL;
NET Encoder_La                 LOC="P51"   | IOSTANDARD=LVTTL;
NET Encoder_Lb                 LOC="P56"   | IOSTANDARD=LVTTL;
NET LED3                       LOC="P121"  | IOSTANDARD=LVTTL; # D6
NET LED4                       LOC="P123"  | IOSTANDARD=LVTTL; # D4
NET LED5                       LOC="P124"  | IOSTANDARD=LVTTL; # D1
NET LED6                       LOC="P126"  | IOSTANDARD=LVTTL; # D0
```

HEXon7segDisp.vhd

```vhdl
--------------------------------------------------------------------------------
-- Junior Design Team 10
-- Inputs: clk, left motor encoder pins
-- Outputs: hexsel (sequence of A-G segment values for cathodes), an_out (power to 7 segment display anodes)
-- Converts encoder pulses to inches and displays this value in hex on the seven segment display
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity HEXon7segDisp is
        Port (
                clk : IN  STD_LOGIC;
                Encoder_La : IN std_logic;
                Encoder_Lb : IN std_logic;
                      hexsel : OUT  STD_LOGIC_VECTOR (6 downto 0);
                an_out : OUT  STD_LOGIC_VECTOR (3 downto 0)
        );
end HEXon7segDisp;

architecture Behavioral of HEXon7segDisp is

-- Signals for counter, Encoder_L and data to be displayed on the 7 segment display
        signal Counter : unsigned (10 downto 0) := (others => '0') ;
        signal Encoder_L: std_logic;
        signal data_in : std_logic_vector(31 downto 0);
        signal data : std_logic_vector(31 downto 0) := (others => '0');
        signal data_sig : std_logic_vector(31 downto 0) := (others => '0');

-- Defining data[] signals their corresponding bits of the data string to be sent to [] anode
        alias data0 : std_logic_vector(3 downto 0) is data(3 downto 0);
        alias data1 : std_logic_vector(3 downto 0) is data(7 downto 4);
        alias data2 : std_logic_vector(3 downto 0) is data(11 downto 8);
        alias data3 : std_logic_vector(3 downto 0) is data(15 downto 12);

-- Declaring MuxSel[] to their corresponding bits of Counter
        alias MuxSel1 : unsigned (1 downto 0) is Counter(10 downto 9);
        alias MuxSel2 : unsigned (3 downto 0) is Counter(10 downto 7);

-- Signals hex_[] are strings of anode segment values (A-G) needed to display the [] digit (segments are active low)
-- EX: hex_0 = "1000000" All segments are on except segment A, displays the digit "0"
        constant hex_0 : std_logic_vector(6 downto 0) := "1000000";
        constant hex_1 : std_logic_vector(6 downto 0) := "1111001";
        constant hex_2 : std_logic_vector(6 downto 0) := "0100100";
        constant hex_3 : std_logic_vector(6 downto 0) := "0110000";
        constant hex_4 : std_logic_vector(6 downto 0) := "0011001";
```

```
        constant hex_5 : std_logic_vector(6 downto 0) := "0010010";
        constant hex_6 : std_logic_vector(6 downto 0) := "0000010";
        constant hex_7 : std_logic_vector(6 downto 0) := "1111000";
        constant hex_8 : std_logic_vector(6 downto 0) := "0000000";
        constant hex_9 : std_logic_vector(6 downto 0) := "0010000";
        constant hex_a : std_logic_vector(6 downto 0) := "0001000";
        constant hex_b : std_logic_vector(6 downto 0) := "0000011";
        constant hex_c : std_logic_vector(6 downto 0) := "1000110";
        constant hex_d : std_logic_vector(6 downto 0) := "0100001";
        constant hex_e : std_logic_vector(6 downto 0) := "0000110";
        constant hex_f : std_logic_vector(6 downto 0) := "0001110";
begin

-- Encoder_L pulses when Encoder_La and Encoder_Lb have opposite values
Encoder_L <= Encoder_La XOR Encoder_Lb;

-- Process that increments the data signal every 59 Encoder_L ticks (1 inch)
process(Encoder_L, data_sig)
begin
        if rising_edge(Encoder_L) then
                data_sig <= std_logic_vector(unsigned(data_sig) + 1);
                if data_sig = "00000000000000000000000000111011" then
                        data <= std_logic_vector(unsigned(data) + 1);
                        data_sig <= (others => '0');
                end if;
        end if;
end process;

-- Create an upcounter signal to be used by MuxSel1 and MuxSel2
process (clk)
begin
        if rising_edge(clk) then
                Counter <= ((Counter) + 1);
        end if;
end process;

-- Process that assigns the first 7 bits of data_in with its anode segment string based on the value of data0
-- EX: data0 = "0000", to display this value on anode 0, assign data_in(6 downto 0) with hex_0 constant
process(data0)
begin
        case data0 is
                when "0000" =>
                        data_in(6 downto 0) <= hex_0;
                when "0001" =>
                        data_in(6 downto 0) <= hex_1;
                when "0010" =>
                        data_in(6 downto 0) <= hex_2;
                when "0011" =>
                        data_in(6 downto 0) <= hex_3;
                when "0100" =>
                        data_in(6 downto 0) <= hex_4;
                when "0101" =>
                        data_in(6 downto 0) <= hex_5;
```

```
        when "0110" =>
            data_in(6 downto 0) <= hex_6;
        when "0111" =>
            data_in(6 downto 0) <= hex_7;
        when "1000" =>
            data_in(6 downto 0) <= hex_8;
        when "1001" =>
            data_in(6 downto 0) <= hex_9;
        when "1010" =>
            data_in(6 downto 0) <= hex_a;
        when "1011" =>
            data_in(6 downto 0) <= hex_b;
        when "1100" =>
            data_in(6 downto 0) <= hex_c;
        when "1101" =>
            data_in(6 downto 0) <= hex_d;
        when "1110" =>
            data_in(6 downto 0) <= hex_e;
        when others =>
            data_in(6 downto 0) <= hex_f;
    end case;
end process;

-- Process that assigns bits 14-8 of data_in with its anode segment string based on the value of data1
-- EX: data1 = "0000", to display this value on anode 1, assign data_in(14 downto 8) with hex_0 constant
process(data1)
begin
    case data1 is
        when "0000" =>
            data_in(14 downto 8) <= hex_0;
        when "0001" =>
            data_in(14 downto 8) <= hex_1;
        when "0010" =>
            data_in(14 downto 8) <= hex_2;
        when "0011" =>
            data_in(14 downto 8) <= hex_3;
        when "0100" =>
            data_in(14 downto 8) <= hex_4;
        when "0101" =>
            data_in(14 downto 8) <= hex_5;
        when "0110" =>
            data_in(14 downto 8) <= hex_6;
        when "0111" =>
            data_in(14 downto 8) <= hex_7;
        when "1000" =>
            data_in(14 downto 8) <= hex_8;
        when "1001" =>
            data_in(14 downto 8) <= hex_9;
        when "1010" =>
            data_in(14 downto 8) <= hex_a;
        when "1011" =>
            data_in(14 downto 8) <= hex_b;
        when "1100" =>
```

```
                        data_in(14 downto 8) <= hex_c;
                when "1101" =>
                        data_in(14 downto 8) <= hex_d;
                when "1110" =>
                        data_in(14 downto 8) <= hex_e;
                when others =>
                        data_in(14 downto 8) <= hex_f;
        end case;
end process;

-- Process that assigns bits 22-16 of data_in with its anode segment string based on the
↪  value of data2
-- EX: data2 = "0000", to display this value on anode 2, assign data_in(22 downto 16)
↪  with hex_0 constant
process(data2)
begin
        case data2 is
                when "0000" =>
                        data_in(22 downto 16) <= hex_0;
                when "0001" =>
                        data_in(22 downto 16) <= hex_1;
                when "0010" =>
                        data_in(22 downto 16) <= hex_2;
                when "0011" =>
                        data_in(22 downto 16) <= hex_3;
                when "0100" =>
                        data_in(22 downto 16) <= hex_4;
                when "0101" =>
                        data_in(22 downto 16) <= hex_5;
                when "0110" =>
                        data_in(22 downto 16) <= hex_6;
                when "0111" =>
                        data_in(22 downto 16) <= hex_7;
                when "1000" =>
                        data_in(22 downto 16) <= hex_8;
                when "1001" =>
                        data_in(22 downto 16) <= hex_9;
                when "1010" =>
                        data_in(22 downto 16) <= hex_a;
                when "1011" =>
                        data_in(22 downto 16) <= hex_b;
                when "1100" =>
                        data_in(22 downto 16) <= hex_c;
                when "1101" =>
                        data_in(22 downto 16) <= hex_d;
                when "1110" =>
                        data_in(22 downto 16) <= hex_e;
                when others =>
                        data_in(22 downto 16) <= hex_f;
        end case;
end process;

-- Process that assigns bits 30-24 of data_in with its anode segment string based on the
↪  value of data3
```

```
-- EX: data3 = "0000", to display this value on anode 3, assign data_in(30 downto 24)
↪  with hex_0 constant
process(data3)
begin
        case data3 is
                when "0000" =>
                        data_in(30 downto 24) <= hex_0;
                when "0001" =>
                        data_in(30 downto 24) <= hex_1;
                when "0010" =>
                        data_in(30 downto 24) <= hex_2;
                when "0011" =>
                        data_in(30 downto 24) <= hex_3;
                when "0100" =>
                        data_in(30 downto 24) <= hex_4;
                when "0101" =>
                        data_in(30 downto 24) <= hex_5;
                when "0110" =>
                        data_in(30 downto 24) <= hex_6;
                when "0111" =>
                        data_in(30 downto 24) <= hex_7;
                when "1000" =>
                        data_in(30 downto 24) <= hex_8;
                when "1001" =>
                        data_in(30 downto 24) <= hex_9;
                when "1010" =>
                        data_in(30 downto 24) <= hex_a;
                when "1011" =>
                        data_in(30 downto 24) <= hex_b;
                when "1100" =>
                        data_in(30 downto 24) <= hex_c;
                when "1101" =>
                        data_in(30 downto 24) <= hex_d;
                when "1110" =>
                        data_in(30 downto 24) <= hex_e;
                when others =>
                        data_in(30 downto 24) <= hex_f;
        end case;
end process;


-- Mux process which selects one of the hex constants from data[] to be displayed on the
↪  anodes
-- Since Muxsel1 is the 2 most significant bits of the rapidly changing Counter signal,
↪  HexSel updates frequently
process(MuxSel1,data_in)
begin
        case MuxSel1 is
                when "00" =>
                   HexSel<=(data_in(30 downto 24));
                when "01" =>
                        HexSel<=(data_in(22 downto 16));
                when "10" =>
                        HexSel<=(data_in(14 downto 8));
                when others =>
```

```
                        HexSel<=(data_in(6 downto 0));
        end case;
end process;

-- Mux process that enables one of the anodes at a time according to MuxSel2 (anodes are active low)
-- Since MuxSel2 is the 4 most significant bits of the rapidly changing Counter signal, an_out updates frequently
-- The cyclic nature of MuxSel2 causes anodes to alternate being on and off, with only one being on at any given time
-- Rapidly cycling through the anodes allows us to seemingly "display" different values on all 4 anodes at the same time
process(MuxSel2)
begin
        case MuxSel2 is
                when "0000" | "0011" | "0100" | "0111" | "1000" | "1011" | "1100" | "1111" =>
                        an_out<="1111";
                when "0001" | "0010" =>
                        an_out<="1110";
                when "0101" | "0110" =>
                        an_out<="1101";
                when "1001" | "1010" =>
                        an_out<="1011";
                when "1101" | "1110" =>
                        an_out<="0111";
                when others =>
                        an_out<="1111";
        end case;
end process;

end Behavioral ;
bm@sncyVerbLine0
```

Toplevel.vhd

```
----------------------------------------------------------------------------------
-- Junior Design Team 10
-- Inputs: IR Transmitter (Red1), motor sequence from C code (LED3-6), clk, Encoder_La, Encoder_Lb
-- Outputs: Arduino Reset, Motor pins, hexsel, an_out
-- Top level integrates Motor and Hexon7segDisp components
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Top_level is
        PORT(
                clk  : IN std_logic;
                Red1 : IN std_logic;
                Led3 : IN std_logic;
                Led4 : IN std_logic;
```

```
                Led5 : IN std_logic;
                Led6 : IN std_logic;
                Encoder_La : IN std_logic;
                Encoder_Lb : IN std_logic;
                Arduino_Reset : OUT std_logic;
                Motor_La : OUT std_logic;
                Motor_Lb : OUT std_logic;
                Motor_Ra : OUT std_logic;
                Motor_Rb : OUT std_logic;
                hexsel : OUT std_logic_vector(6 downto 0);
                an_out : OUT std_logic_vector(3 downto 0)
        );
end Top_level;

architecture Behavioral of Top_level is

-- Component Motor inputs and outputs
COMPONENT Motors
        PORT(
                clk: IN std_logic;
                Red1 : IN std_logic;
                Led3 : IN std_logic;
                Led4 : IN std_logic;
                Led5 : IN std_logic;
                Led6 : IN std_logic;
                Arduino_Reset : OUT std_logic;
                Motor_La : OUT std_logic;
                Motor_Lb : OUT std_logic;
                Motor_Ra : OUT std_logic;
                Motor_Rb : OUT std_logic
        );
        END COMPONENT;

-- Component HEXon7segDisp inputs and outputs
COMPONENT HEXon7segDisp
        PORT(
                Encoder_La : IN std_logic;
                Encoder_Lb : IN std_logic;
                clk : IN std_logic;
                hexsel : OUT std_logic_vector(6 downto 0);
                an_out : OUT std_logic_vector(3 downto 0)
        );
        END COMPONENT;

begin

-- Instance of Motors, mapping ports to their top level signals
Inst_Motors: Motors PORT MAP(
        clk => clk,
        Red1 => Red1,
        Led3 => Led3,
        Led4 => Led4,
        Led5 => Led5,
        Led6 => Led6,
        Arduino_Reset => Arduino_Reset,
```

```
        Motor_La => Motor_La,
        Motor_Lb => Motor_Lb,
        Motor_Ra => Motor_Ra,
        Motor_Rb => Motor_Rb
        );

-- Instance of HEXon7segDisp, mapping ports to their top level signals
Inst_HEXon7segDisp: HEXon7segDisp PORT MAP(
        Encoder_La => Encoder_La,
        Encoder_Lb => Encoder_Lb,
        hexsel => hexsel,
        an_out => an_out,
        clk => clk
        );

end Behavioral;
```

Motors.vhd

```
----------------------------------------------------------------------------------
-- Junior Design Team 10
-- Inputs: clk, Red1 (IR transmitter), Led3-6 (motor sequence from C code),
-- Outputs: Motors (motor sequence to motor pins), Arduino Reset
-- Determines operation mode based on IR transmitter, halts motors in stop mode and
↪  forwards them their motor sequence from C code in start mode
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Motors is
        Port(
                clk: in std_logic;
                Red1 : in std_logic;
                Led3 : in std_logic;
                Led4 : in std_logic;
                Led5 : in std_logic;
                Led6 : in std_logic;
                Arduino_Reset: out std_logic;
                Motor_La: out std_logic;
                Motor_Lb: out std_logic;
                Motor_Ra: out std_logic;
                Motor_Rb: out std_logic
                );

end Motors;

architecture Behavioral of Motors is

signal press, press_d1, press_d2: std_logic;
signal mode: std_logic; -- 0: stop  1: start
```

```
begin

Arduino_Reset <= '1';

-- IR transmitter active low
press <= NOT Red1;

-- Delay the IR pulse by two clock signals
process(clk)
begin
	if rising_edge(clk) then
		press_d1 <= press;
		press_d2 <= press_d1;
	end if;
end process;

-- Change mode if there was a falling edge on the IR transmitter
process(clk, press, press_d1, press_d2)
begin
	if rising_edge(clk) then
		if (press_d1 = '1' AND (press_d2 = '0')) then
			mode <= NOT mode;
		end if;
	end if;
end process;

-- Halt motors if in stop mode, forward their motor sequence (LED3-6) if in start mode
process(mode)
begin
	if mode = '0' then
		Motor_La <= '1';
		Motor_Lb <= '1';
		Motor_Ra <= '1';
		Motor_Rb <= '1';
	else
		Motor_La <= Led3;
		Motor_Lb <= Led4;
		Motor_Ra <= Led5;
		Motor_Rb <= Led6;
	end if;
end process;

end Behavioral;
```

Main.c

```
/*
 * Junior Design Team 10
 * C Code to program Papilio DUO's atmgea32_4 micro controller
 * Inputs: 3 sensor echos (SW1-3)
```

```c
 * Outputs: Universal sensor trigger (SW0) 4 bit motor movement sequence (LED3-6)
 * Receives distances from sensors and decides which movement sequence to send to the motor pins based on the Rover's position in the course
 */

//Includes necessary C headers and defines the clock speed of the processor
#include <avr/io.h>
#include <stdint.h>
#include <stdbool.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#define F_CPU 16000000ul

//Declarations of motor functions and sensor functions
void forward();
void left_turn();
void right_turn();
void brake();
uint8_t right_sensor();
uint8_t front_sensor();
uint8_t left_sensor();

//Main function that defines atmega pins, calls sensor functions and decides which motor function to call
int main(void) {

        DDRD |= (1<<6); //Config as output for LED3 in port D6 (Motor_La)
        DDRD |= (1<<4); //Config as output for LED4 in port D4 (Motor_Lb)
        DDRD |= (1<<1); //Config as output for LED5 in port D1 (Motor_Ra)
        DDRD |= (1<<0); //Config as output for LED6 in port D0 (Motor_Lb)

        DDRD |= (1<<2); //Config as output for SW0 in port D2 (Universal trigger for right, left and front sensors)

        DDRB &= ~(1<<4); //Config as input for SW3 in port B4 (Echo for right sensor)
        PORTB &= ~(1<<4); //Disable pull up port B4

        DDRB &= ~(1<<5); //Config as input for SW2 in port B5 (Echo for front sensor)
        PORTB &= ~(1<<5); //Disable pull up port B5

        DDRD &= ~(1<<3); //Config as input for SW1 in port D3 (Echo for left sensor)
        PORTD &= ~(1<<3); //Disable pull up port D3

        //Initializes variables for distances from right, left and front sensors
        uint8_t front_dist = 0;
        uint8_t right_dist = 0;
        uint8_t left_dist = 0;

        while(1){
                //Call the sensor functions and set their return value equal to their distance variable
                right_dist = right_sensor();
```

```
        _delay_ms(100);
        left_dist = left_sensor();
        _delay_ms(100);
        front_dist = front_sensor();
        _delay_ms(100);

        //If the front and left sensors are within 6 inches of an obstacle, turn right
        if (front_dist <= 153 && left_dist <= 153){
            _delay_ms(100);
            right_turn();
            _delay_ms(100);
        }
        //Else, if the front and right sensors are within 6 inches of an obstacle, turn left
        else if (right_dist <= 153 && front_dist <= 153){
            _delay_ms(100);
            left_turn();
            _delay_ms(100);
        }
        //Else, if the front sensor is within 6 inches of an obstacle, turn right
        else if (front_dist <= 153){
            _delay_ms(100);
            right_turn();
            _delay_ms(100);
        }
        //If all sensors are more than 6 inches away from an obstacle, go forward
        else{
            _delay_ms(100);
            forward();
            _delay_ms(100);
        }
    }
}

//Sets LED3-6 to motor sequence (1010) to go forward
void forward(){
    PORTD |= (1<<6);
    PORTD &= ~(1<<4);
    PORTD |= (1<<1);
    PORTD &= ~(1<<0);
}

//Sets LED3-6 to motor sequence (0110) to turn left
void left_turn(){
    PORTD &= ~(1<<6);
    PORTD |= (1<<4);
    PORTD |= (1<<1);
    PORTD &= ~(1<<0);
}

//Sets LED3-6 to motor sequence (1001) to turn right
void right_turn(){
    PORTD |= (1<<6);
    PORTD &= ~(1<<4);
```

```
        PORTD &= ~(1<<1);
        PORTD |= (1<<0);
}

//Sets LED3-6 to motor sequence (1111) to brake
void brake(){
        PORTD |= (1<<6);
        PORTD |= (1<<4);
        PORTD |= (1<<1);
        PORTD |= (1<<0);
}

//Measures right sensor distance from an object and returns this value
uint8_t right_sensor() {
        uint8_t distance = 0;

        //15us trigger pulse on SW0
        PORTD |= (1<<2);
        _delay_us(15);
        PORTD &= ~(1<<2);

        //Wait for echo pulse (SW3) = 1
        while (!(PINB & (1<<4))){}

        //Measure amount of time to receive echo for distance
        while((PINB & (1<<4)) && (distance < 255)) {
                _delay_us(7);
                distance++;
        }

        //Return distance from sensor
        return distance;
}

//Measures front sensor distance from an object and returns this value
uint8_t front_sensor() {
        uint8_t distance = 0;

        //15us trigger pulse on SW0
        PORTD |= (1<<2);
        _delay_us(15);
        PORTD &= ~(1<<2);

        //Wait for echo pulse (SW2) = 1
        while (!(PINB & (1<<5))){}

        //Measure echo for distance
        while((PINB & (1<<5)) && (distance < 255)) {
                _delay_us(7);
                distance++;
        }
        return distance;
}

//Measures front sensor distance from an object and returns this value
```

```
uint8_t left_sensor() {
	uint8_t distance = 0;

	//15us trigger pulse on SW0
	PORTD |= (1<<2);
	_delay_us(15);
	PORTD &= ~(1<<2);

	//Wait for echo pulse (SW1) = 1
	while (!(PIND & (1<<3))){}

	//Measure amount of time to receive echo for distance
	while((PIND & (1<<3)) && (distance < 255)) {
		_delay_us(7);
		distance++;
	}
	return distance;
}
```

MineCount.c

```
/*
 * Junior Design Team 10
 * C Code to program atmgea328p micro controller
 * Inputs: Metal Detector on PORT B1
 * Outputs: PORTS C0 C1 C2 C3 for LEDs on breadboard
 * Increments Mine Count for each mine found and outputs this count to the LEDs
 */

//Includes necessary C headers and defines the clock speed of the processor
#include <avr/io.h>
#include <stdint.h>
#include <stdbool.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#define F_CPU 16000000ul
#define MetalDetector 1
#define Mine1 0
#define Mine2 1
#define Mine3 2
#define Mine4 3

//Main function that defines atmega pins/ports, counts number of mines found and outputs
↪  this value to the LEDs
int main(void)
{
	int i = 0;

	//Config as outputs for LEDs on PORTs C0 C1 C2 C3
```

```
	DDRC |= (1<<Mine1) | (1<<Mine2) | (1<<Mine3) | (1<<Mine4);

	//Config as input for Metal Detector on PORT B1
	DDRB &= ~(1<<MetalDetector);

	//Initialize array of the 4 mine count LEDs
	char mines[] = {Mine1,Mine2,Mine3,Mine4};

	//
	PORTC &= ~(1<<Mine1) & ~(1<<Mine2) & ~(1<<Mine3) & ~(1<<Mine4);

	//Infinite loop to detect mines and output mine count (i) to LEDs (ensures count
	↪  only increments once for each mine)
	while(1){
		if(PINB & 0x02){
			PORTC |= (1<<mines[i]);
			_delay_ms(500);
			i++;
		}
	}
}
```