THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF ELECTRICAL ENGINEERING

# FIELD PROGRAMMABLE GATE ARRAY CONTROLLED AUTONOMOUS CAR

AUSTIN CRAIN
FALL 2014

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Electrical Engineering
with honors in Electrical Engineering

Reviewed and approved* by the following:

Hal Scholz
Instructor in Physics and Engineering
Thesis Supervisor

Jeff Mayer
Associate Professor of Electrical Engineering
Honors Adviser

* Signatures are on file in the Schreyer Honors College.

## ABSTRACT

As industries, such as the auto-industry, continue to add computer aided control to their products, more and more is required from the hardware that is embedded in their products. New designs must work faster and handle more situations which require more processing power. Currently, the preferred method of computer aided control utilizes microprocessors. While this method is effective to handle simple situations, more complicated situations that require parallel processing are becoming a reality.

One of the possible alternatives to using microprocessors to handle parallel processing is a Field Programmable Gate Array (FPGA). An FPGA is an integrated circuit made up of a collection of logic gates. An FPGA is programmable in the sense that the logic gates it contains can be programmatically wired together to create a system that responds to inputs predictably. The logic gates can be wired together using a dataflow language, such as Verilog. Since logic gates are independently connected to each other, they can operate in parallel. This enables FPGAs to operate in parallel.

The goal of this project is to build an autonomous FPGA controlled car. The car would have Ultrasonic Distance Sensors to gather input about the surroundings. The data would be forwarded from the distance sensors to the FPGA. The FPGA would then use the data obtained from the distance sensors to make decisions about how to move the car. The FPGA moves the car using an H-bridge connected to a drive motor and a steering motor. The main focus of this project is the algorithms used to decide which directions to go and their effectiveness.

# TABLE OF CONTENTS

List of Figures .......... iii

List of Tables .......... iv

Acknowledgements .......... v

Chapter 1 Introduction .......... 1

- 1.1: Principles of Boolean Algebra .......... 1
- 1.2: Logic Gates .......... 2
- 1.3: Dataflow Programming .......... 5
- 1.4: Look Up Tables .......... 6
- 1.5: Field Programmable Gate Array (FPGA) .......... 8
- 1.6: Ultrasonic Distance Sensors .......... 8
- 1.7: H-Bridge .......... 9

Chapter 2 Hardware Implementation .......... 11

- 2.1: Car .......... 12
- 2.2: Ultrasonic Distance Sensors .......... 12
- 2.3: Motors .......... 14
- 2.4: H-Bridge .......... 14

Chapter 3 Algorithms .......... 15

- 3.1: Sensor Interface .......... 15
- 3.2: Averaging Data .......... 16
- 3.3: Deciding a Direction .......... 18

Chapter 4 Results .......... 21

BIBLIOGRAPHY .......... 28

ACADEMIC VITA .......... 30

# LIST OF FIGURES

Figure 1: AND gate symbol....3
Figure 2: Image of the Ultrasonic Distance Sensor....9
Figure 3: Example H-bridge....10
Figure 4: Sample H-bridge configuration for forward current flow....10
Figure 5: Sample H-bridge configuration for reverse current flow....11
Figure 6: Timing diagram of the US 020 Ultrasonic Distance Sensor....13
Figure 7: This is the schematic of the H-bridge used in the project where R7 is the motor. ...15
Figure 8: Derivation of the average function....17
Figure 9: The response of the averaging function to constant input....17
Figure 10: Averaging function Verilog Code....18
Figure 11: Averaging function response to impulse....18
Figure 12: Algorithm Layout....19
Figure 13: Module layout for another possible algorithm. ....19
Figure 14: C-like code depicting decision unit....22
Figure 15: Verilog code implementing design unit....23
Figure 16: C-code depicting decision unit with added module....25
Figure 17: Verilog code implementing decision unit with added module....26

## LIST OF TABLES

Table 1: AND operation truth table ........................................................................................1

Table 2: OR operation truth table ..........................................................................................1

Table 3: NOT operation truth table..........................................................................................2

Table 4: NAND operation truth table .......................................................................................3

Table 5: NOR operation truth table..........................................................................................4

Table 6: XOR operation truth table..........................................................................................4

Table 7: XNOR operation truth table.......................................................................................4

Table 8: Lookup table example for AND gate, OR gate, and XOR gate.................................7

## ACKNOWLEDGEMENTS

**To Dr. Hal Scholz:** Thank you for taking your time and working with me on both the research and my thesis.  I learned a lot from you and I look forward to applying what I learned towards a career and a Master's Degree.

**To Shawn Waggoner:** Thank you for being a role model for me and guiding me through the Schreyer's Honor College system.  I'm not sure I would have figured it out on my own.

**To my friends and family:** Thank you for all your support.  You kept pushing me to go farther when I slowed down.

# Chapter 1

# Introduction

## 1.1: Principles of Boolean Algebra

Boolean algebra is used to control the FPGA.  Boolean values can be either “on” or “off” and the rules that govern variable constants storing these values are defined by Boolean algebra.  In Boolean algebra, there are three basic operations: AND, OR, and NOT.  A combination of these three operations is used to create complex equations that can model the behavior some systems.  Truth tables are used explain the operation of the Boolean operators.  A truth table lists every possible combination of operands and corresponding output of an operation.  The Truth Tables for the AND, OR, and NOT operations can be seen in Table 1, Table 2, and Table 3, respectively.

**Table 1: AND operation truth table**

| *AND Truth Table* | | |
|---|---|---|
| **Output** | Operand 1 | Operand 0 |
| **ON** | ON | ON |
| **OFF** | OFF | ON |
| **OFF** | ON | OFF |
| **OFF** | OFF | OFF |

**Table 2: OR operation truth table**

| *OR Truth Table* | | |
|---|---|---|
| **Output** | Operand 1 | Operand 0 |
| **ON** | ON | ON |
| **ON** | OFF | ON |
| **ON** | ON | OFF |
| **OFF** | OFF | OFF |

**Table 3: NOT operation truth table**

| *NOT Truth Table* | |
|---|---|
| **Output** | Operator 0 |
| **ON** | ON |
| **OFF** | ON |

## 1.2: Logic Gates

Logic gates, or modules, are an implementation of Boolean equations. Like Boolean algebra, there are three main operations: AND, OR, and NOT. Logic gates can be connected and bundled together to create a module. A collection of logic gates and modules can be called a logic circuit. Modules can model time invariant systems. Unlike Boolean algebra equations, which use variables, logic gates and modules have inputs and outputs. Logic gates are typically represented on schematics with a symbol. A symbol of a logic gate can be seen in **Figure 1** where A and B are inputs and out is an output.

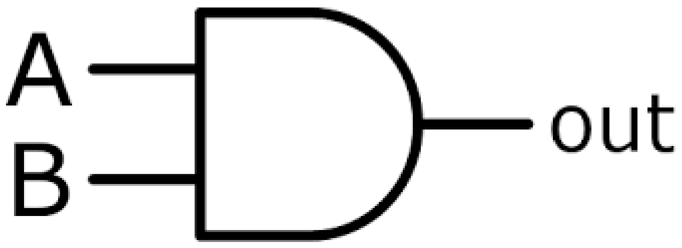


Figure 1: AND gate symbol

By connecting logic gates, modules can be created to model more complex logic gates. A few of the more complex logic gates are: NAND, NOR, XOR, XNOR, and FLIP-FLOPS. For example, a NAND gate is composed of an AND gate were the output is connected to the input of a NOT gate. The output of the module is the output of the NOT gate and the inputs of the module are the inputs of the AND gate. The truth tables for the NAND, NOR, XOR, and XNOR gates can be seen below in **Table 4**, **Table 5**, **Table 6**, and **Table 7**, respectively.

Table 4: NAND operation truth table

| *NAND Truth Table* | | |
|---|---|---|
| **Output** | Input 1 | Input 0 |
| **OFF** | ON | ON |
| **ON** | OFF | ON |
| **ON** | ON | OFF |
| **ON** | OFF | OFF |

**Table 5: NOR operation truth table**

| *NOR Truth Table* | | |
|---|---|---|
| **Output** | Input 1 | Input 0 |
| **OFF** | ON | ON |
| **OFF** | OFF | ON |
| **OFF** | ON | OFF |
| **ON** | OFF | OFF |

**Table 6: XOR operation truth table**

| *XOR Truth Table* | | |
|---|---|---|
| **Output** | Operand 1 | Operand 0 |
| **OFF** | ON | ON |
| **ON** | OFF | ON |
| **ON** | ON | OFF |
| **OFF** | OFF | OFF |

**Table 7: XNOR operation truth table**

| *XNOR Truth Table* | | |
|---|---|---|
| **Output** | Operand 1 | Operand 0 |
| **ON** | ON | ON |
| **OFF** | OFF | ON |
| **OFF** | ON | OFF |
| **ON** | OFF | OFF |

Ideally, logic gates respond instantaneously to inputs; however, this is not the case in reality. Each logic gate has a delay associated with it due to supply voltage and temperature. Therefore, a system of logic gates has a non-zero delay from the input to the output that depends on the number of logic gates and which logic gates are used.

FLIP-FLOP modules are special because are a memory element and do not respond immediately to data inputs. They have an input, an output, and a clock input. As the clock input transitions from OFF to ON, the output transitions to the value measured on the input. The output maintains this value until the next clock transition from OFF to ON. FLIP-FLOP modules give a logic circuit memory.

Typically, logic gates are implemented in hardware using silicon. Switches, called transistors, can be connected together to behave like logic gates. These transistors are implemented in silicon and are powered by the supply voltage (VDD). ON/OFF inputs and outputs become voltage levels. ON corresponds to approximately 3.3V while OFF corresponds to approximately 0V.

## 1.3: Dataflow Programming

A programming paradigm that is used to describe the movement of information throughout a program is known as a dataflow programming language. These languages can be used to model logic circuits because logic circuits have information that is moving through each logic gate. Logic circuits respond nearly instantaneously to data inputs unless a FLIP-FLOP is used. Thus, a dataflow programming language, such as Verilog or LabVIEW, can be used to model and test logic circuits before they are physically wired together on a circuit board. Some of the characteristics of a dataflow programming language are that all instructions are executed at the same time. This is in contrast to other popular programming paradigms that execute instructions serially.

## 1.4: Look Up Tables

Look up tables are a way to implement a Boolean operation. Look up tables are a memory with a mapping from an address to the information stored in the location addressed. To use the lookup table as a logic gate, the address is treated as the inputs to a logic gate, and the output of the logic gate is the data stored in the location that is addressed.

An example of this can be seen in **Table 8**. In this example, there is a 16 word memory. To reach all 16 addresses, 4 inputs are required. The inputs to the memory are A, B, C, and D. At each address, there is a bit of data stored. There is an example output stored in the lookup table for the AND, OR, and XOR gates. It is possible to change which kind of gate the lookup table is behaving as by changing what data is stored in the memory addresses. ON states are represented with a 1 and OFF states are represented with a 0.

Lookup tables are useful because complex logic gates can be represented without the need for a large number of smaller gates. This can reduce architectural complexity of an FPGA.

**Table 8: Lookup table example for AND gate, OR gate, and XOR gate.**

| *Inputs* | | | | | *Output* | | |
|---|---|---|---|---|---|---|---|
| **A** | **B** | **C** | **D** | Address | AND gate | OR gate | XOR gate |
| **0** | **0** | **0** | **0** | 0x0 | 0 | 0 | 0 |
| **0** | **0** | **0** | **1** | 0x1 | 0 | 1 | 1 |
| **0** | **0** | **1** | **0** | 0x2 | 0 | 1 | 1 |
| **0** | **0** | **1** | **1** | 0x3 | 0 | 1 | 1 |
| **0** | **1** | **0** | **0** | 0x4 | 0 | 1 | 1 |
| **0** | **1** | **0** | **1** | 0x5 | 0 | 1 | 1 |
| **0** | **1** | **1** | **0** | 0x6 | 0 | 1 | 1 |
| **0** | **1** | **1** | **1** | 0x7 | 0 | 1 | 1 |
| **1** | **0** | **0** | **0** | 0x8 | 0 | 1 | 1 |
| **1** | **0** | **0** | **1** | 0x9 | 0 | 1 | 1 |
| **1** | **0** | **1** | **0** | 0xA | 0 | 1 | 1 |
| **1** | **0** | **1** | **1** | 0xB | 0 | 1 | 1 |
| **1** | **1** | **0** | **0** | 0xC | 0 | 1 | 1 |
| **1** | **1** | **0** | **1** | 0xD | 0 | 1 | 1 |
| **1** | **1** | **1** | **0** | 0xE | 0 | 1 | 1 |
| **1** | **1** | **1** | **1** | 0xF | 1 | 1 | 0 |

## 1.5: Field Programmable Gate Array (FPGA)

Field Programmable Gate Arrays (FPGA) are a commercially available hardware implementations on silicon used to implement logic circuits. FPGAs are made up of a number of components including Look-up tables (LUT) and FLIP-FLOPs, which act as fast memory. Most importantly, they are made up of multiplexers (MUX), which route the data from the LUTs and FLIP-FLOPs to other LUTs and FLIP-FLOPS. To control the FLIP-FLOPS, there is a clock signal which repeatedly oscillates from ON to OFF as long as power is supplied to the board. On some FPGAs, the clock is integrated into the board, while on other FPGAs, the clock needs to be provided by an external input. An FPGA is reprogrammable primarily as a result of the LUTs and MUXs. The LUTs can be programmed to implement different gates and MUXs can be programmed to connect the different LUTs together.

To program the FPGA, a dataflow language is used. The dataflow language describes the Boolean operations to encode in the LUT and where to use the FLIP-FLOPs. This project uses Verilog to program the FPGA. Once the FPGA is programmed, it can be treated like a module. It has input wires, called pins, and output wires. By writing different ON/OFF values to the input pins, you get different ON/OFF values at the output. However, FLIP-FLOPS store the current state of the system which impacts the next state and output of the system.

## 1.6: Ultrasonic Distance Sensors

Ultrasonic Distance Sensors are sensors that use high frequency sound to determine distances. They send pulses with frequencies higher than 20 kHz as sound waves and record the time it takes for the sound pulses to be read by the receiver. The sensors then return the time from when the sensor first sent data to the time when data is received. One of the problems with using an ultrasonic sensor is the possibility of not receiving sound pulses that have bounced off of a surface in a direction that will not return to the receiver.  A picture of an ultrasonic distance sensor can be seen in **Figure 2**.

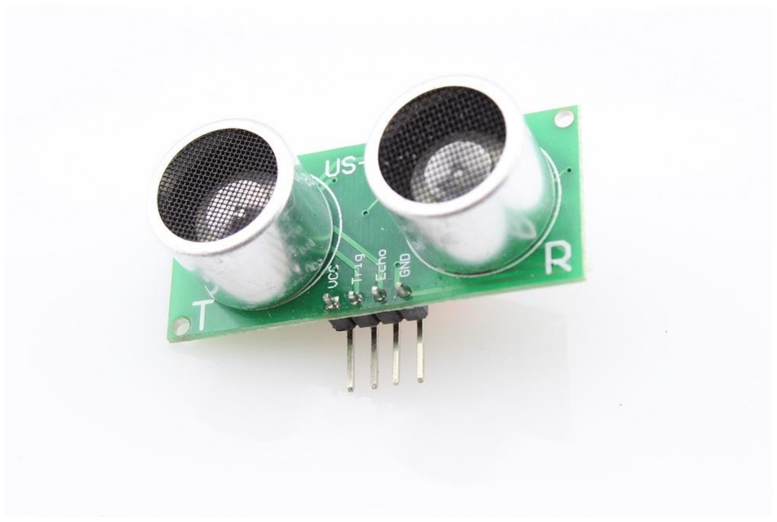


**Figure 2: Image of the Ultrasonic Distance Sensor**

## 1.7: H-Bridge

H-bridges are circuits made up of transistors that allow a two wire input to control the direction of current flow across a motor.  A diagram of an H-bridge can be seen in **Figure 3**.  Usually the third wire is replaced by a motor so that different combination of switches flow current both directions across the motor.  Another added benefit is that inputs to the H-bridge do not need to supply the current to the motor.  This allows circuits that are much more or less powerful than the motor to control the H-bridge.  The configurations of the H-bridge for forward and reverse can be seen in **Figure 4** and **Figure 5**, respectively.  In the diagrams, the switches are being flipped to choose which direction current flows across the motor.

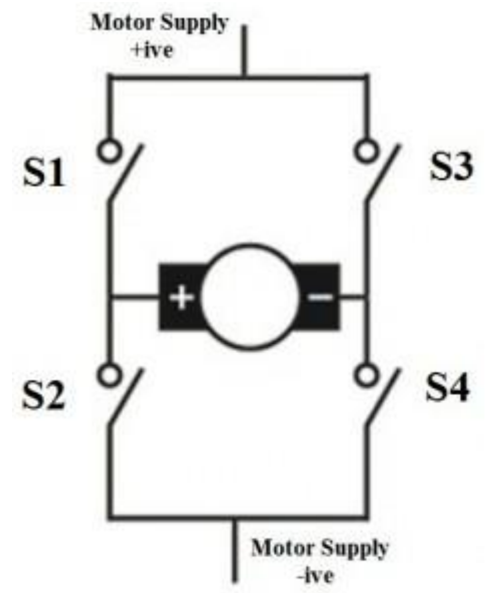


Figure 3: Example H-bridge

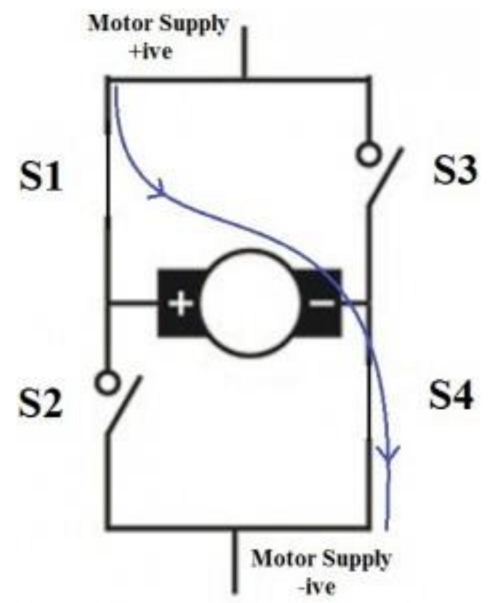


Figure 4: Sample H-bridge configuration for forward current flow

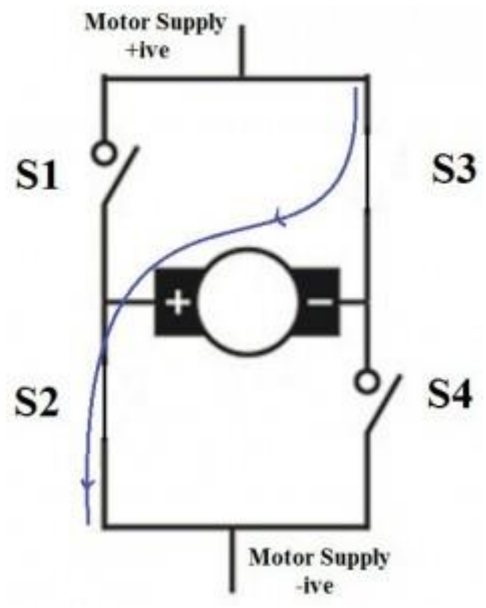


Figure 5: Sample H-bridge configuration for reverse current flow

# Chapter 2

# Hardware Implementation

To test the varying algorithms on the FPGA, the FPGA is integrated with a remote control (RC) car. The RC component would be removed from the car and the FPGA would be connected to the motors. Ultrasonic distance sensors would be added to the front and sides of the car and connected to the FPGA. This gives the FPGA the ability to gather information about its surroundings and drive the motors. Since the motors are driven at a higher voltage that can be switched between positive and negative, an H-bridge is placed between the FPGA and the motors. The FPGA controls the H-bridge and the H-bridge the motors.

The FPGA used for this project is the iCEblink40 HX1K. This FPGA comes with 1280 LUTs and FLIP-FLOPS and it comes with 100 pins on the board. The clock was built into the FPGA and oscillated at 3.3 MHz.

## 2.1: Car

The car was implemented using an RC car as a base. Its ability to turn left and right, its approximate speed, and its ability to go forward and backwards are some of the functions that were tested for initially. The approximate speed of the car is about one meter per second. Since the FPGA was going to control the car, the RC components were able to be disconnected from the motors. The pins for the motors were labelled as positive and negative. The motors were attached to an H-bridge and FPGA later.

## 2.2: Ultrasonic Distance Sensors

The ultrasonic distance sensors were the next aspect to be connected. The US-020 ultrasonic distance sensors had 4 pins each: One pin was for VDD, one pin was for ground, one pin for TRIG, and one pin for ECHO. The TRIG pin was for sending electric pulses 10 µs long to initiate the distance measurement while the ECHO pin was for reading a pulse that was the length of the time it took to send and receive the high frequency sound pulse.

The sensors were hot glued to screws which attached to the front and sides of the car. The four wires that attached to the sensor were bundled together to make a bus. Each bus was wired to a connecter that could be plugged in and out of the sensors. Four of these busses were created for each of the 4 sensors that were attached, two of which were in the front and one on each side.

Once the sensors were wired in, the sensor interface needed to be determined. The sensors did not come with a datasheet and, after doing research on other distance sensors, a standard protocol was determined. To initiate a distance measurement, an electronic pulse would be sent to the sensor on the TRIG pin. The distance sensor would return information about the distance on the ECHO pin. A pulse with the same length as the time taken for the pulse to go to the wall and back would be sent across the ECHO pin.

The sensor interface could be done directly on the FPGA since the sensor and the FPGA operate at 3.3 V. The FPGA would initially send out a 300 nanosecond pulse to the sensor on the TRIG pin to initiate a distance measurement. The FPGA would then wait until the sensor starts responding on the ECHO pin. Once the sensor starts sending a pulse across the ECHO pin, the FPGA would start counting how long the pulse is. When the pulse ends, the length of the pulse in binary would be stored in the FPGA for use in decision algorithm later.

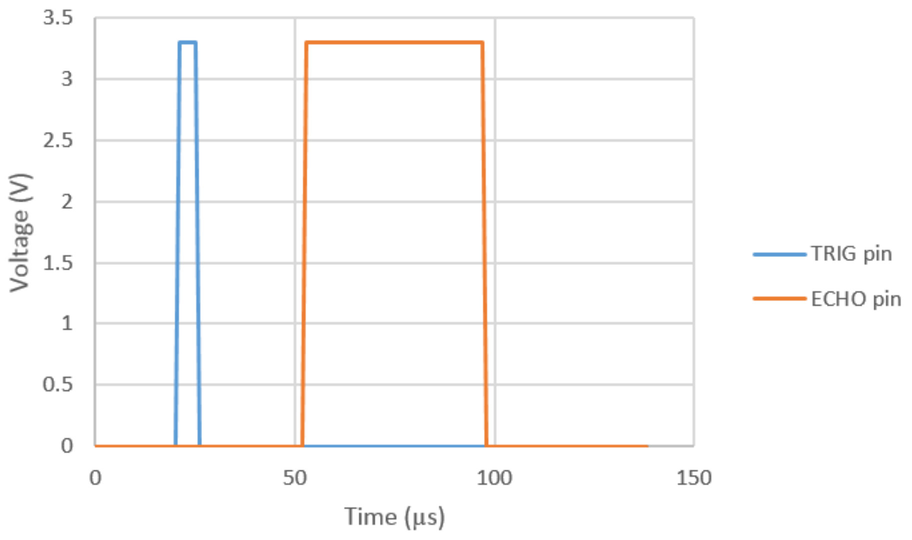


**Figure 6: Timing diagram of the US 020 Ultrasonic Distance Sensor**

## 2.3: Motors

The motors for this project were included in the RC car. When they were tested, they operated between positive and negative ten volts. When the voltage across them was positive, the drive motors would drive forward and the steering would turn to the right. When a negative voltage was applied, the drive motor would drive backward and the steering would turn to the left. When zero voltage was applied to the motors, the drive motor would not move and the steering motor would remain in a neutral position. To start the motors moving in either direction, an initial voltage needed to be close to either positive or negative 10 volts. Anything less than that value would not be a high enough to initially start the motors.

Since the FPGA operates at 3.3 volts and low current, there was a problem driving the motors directly from the FPGA. The motors needed to be powered by ten volts with high current. To solve this, an H-bridge needs to be used as an interface between the FPGA and the motors

## 2.4: H-Bridge

Once the sensors had been connected to the FPGA, the FPGA needed to control the motors. However, because the FPGA only operates ON or OFF and the motors operate OFF, FORWARD, or BACKWARD, an H-bridge was chosen to translate the two. The H-bridge would include 2 transistors to transition from the 3.3V that the FPGA provides to the 10V that is required to power the motors. There is one H-bridge for the steering motor and one H-bridge for the drive motor.

It would require 2 pins from the FPGA to control one H-bridge. The value of both pins being OFF would imply that the motor is OFF. If one of the pins is ON, the motor is either moving FORWARD or BACKWARD, depending which pin is ON. Both pins cannot be on at the same time without damaging the H-bridge. A schematic of the one H-bridge can be seen in **Figure 7**.

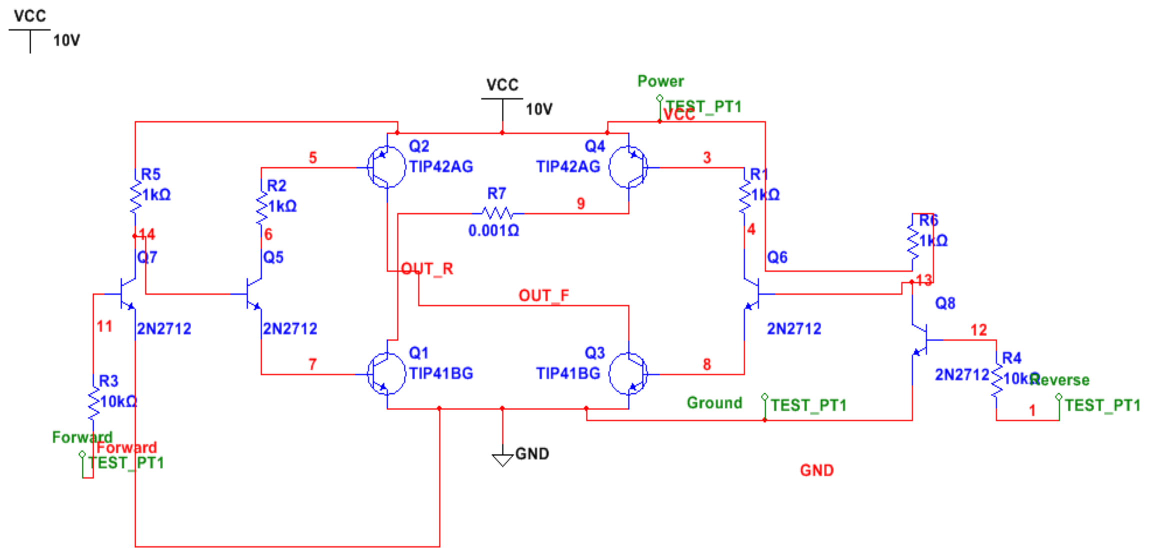


Figure 7: This is the schematic of the H-bridge used in the project where R7 is the motor.

# Chapter 3

# Algorithms

The primary goal of this research is to determine the algorithms. The algorithms make up the main difference between FPGAs and microcontrollers. In this project, the car was coded to be able to navigate a room and a hallway without human assistance. The first couple algorithms are aimed at controlling the cars hardware, similar to device drivers in a computer. These algorithms are the Sensor Interface and the Motor Interface. The algorithms that follow are aimed at making decisions about the surroundings.

## 3.1: Sensor Interface

The sensor interface was a simple interface used to gather data about the surroundings. The sensor module was intended to create a pulse seen in **Figure 6** on the TRIG pin. The sensor module would then read the pulse seen in **Figure 6** coming back on the ECHO pin. The module worked by sending out a short pulse. The pulse was about as long as thirty clock cycles where the clock frequency is 3.3 MHz. Once the pulse was sent out, a 17 bit counter, initially cleared to be all zeros, starts counting up one increment. The counter would count up until it started reading a pulse. Once it started reading a non-zero voltage, the counter would reset to zero and start incrementing again. When the pulse stopped, the value of the counter would be stored in memory. If the module did not start reading a pulse back from the sensor before the clock reached its maximum value, the module would reset and try again.

Each sensor module required 31 FLIP-FLOPS. In order to save LUTs in the FPGA, only one sensor module was incorporated. A MUX would cycle through which sensor the FPGA listened to. The maximum time required to read each sensor was about 0.029 seconds. This time is based on how long it takes the ultrasonic sensor to measure 5 meters. 0.029 seconds is how long it takes sound to go 5 meters to an obstacle and back. The total time to record information about the surroundings from the four sensors was about 0.116 seconds.

## 3.2: Averaging Data

Ultrasonic sensors are not always perfect. Occasionally, the sensors glitch and create incorrect values for data. Also, there is a chance that the ultrasonic sensors can bounce off of something other than the walls and create values that are larger or smaller than is correct. Rather than responding to the data immediately, an averaging algorithm was implemented. This function weights the average much higher than the input in order to save gates. Overtime, if the results from the sensor are consistent, the average will slowly approach the new input. Outliers caused by glitches would get blended into the average without having a significant effect.

The equation for the averaging function can be seen in **Figure 8**.

$$Average := \frac{1}{N}\sum_{i=1}^{N} a_i = \frac{1}{2^n}\sum_{i=1}^{2^n} a_i = \frac{(2^n - 1) \times Average + Input}{2^n}$$

$$Average = \frac{2^n \times Average + Input - Average}{2^n} = Average + \frac{(Input - Average)}{2^n}$$

$$Average := Average + \frac{(Input - Average)}{2^n}$$

**Figure 8: Derivation of the average function**

The smaller that the value of n was, the more quickly the average distance would respond to the input read from the sensors. To determine an adequate value, n, graphs were produced to compare how this function performed using different values of n. The average was initialized to 1 and the input was constantly a 10. Calculations were made every 0.116 seconds, the time it took for each sensor to be read. The value, n, varied from 3 to 5 for each of the graphs in **Figure 9**.

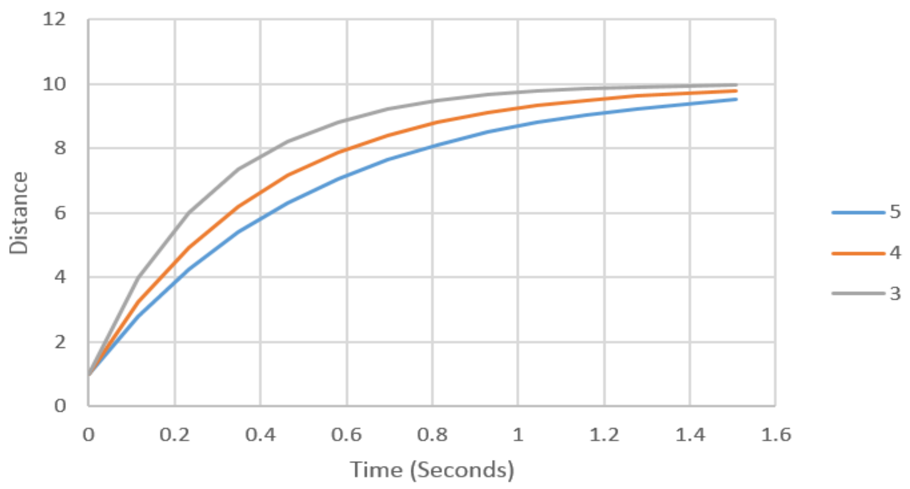
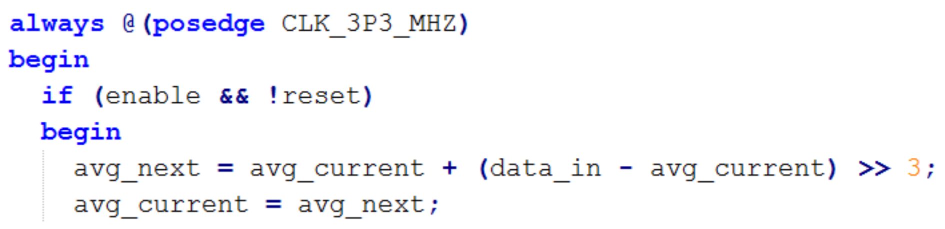


**Figure 9: The response of the averaging function to constant input**

From the graph, n was chosen to be 3 for a quick response. The other values were too large and would require too much time for the car to determine the distance. A snippet of the Verilog code used to implement the function can be seen in **Figure 10**. In Figure 10, the >> operator can be seen. This operator divides the left operand by two to the power of the right operand. For example, 8 >> 3 is equal to 1.



```
always @(posedge CLK_3P3_MHZ)
begin
  if (enable && !reset)
  begin
    avg_next = avg_current + (data_in - avg_current) >> 3;
    avg_current = avg_next;
```

Figure 10: Averaging function Verilog Code

The benefit of the averaging function can be seen in **Figure 11**. In **Figure 11**, a glitch on the input only causes a minimal jump with the averaging functions. The glitch jumps up to 10 on the distance. The averaging function jumps up as high as 4 in distance. This creates a more stable input.

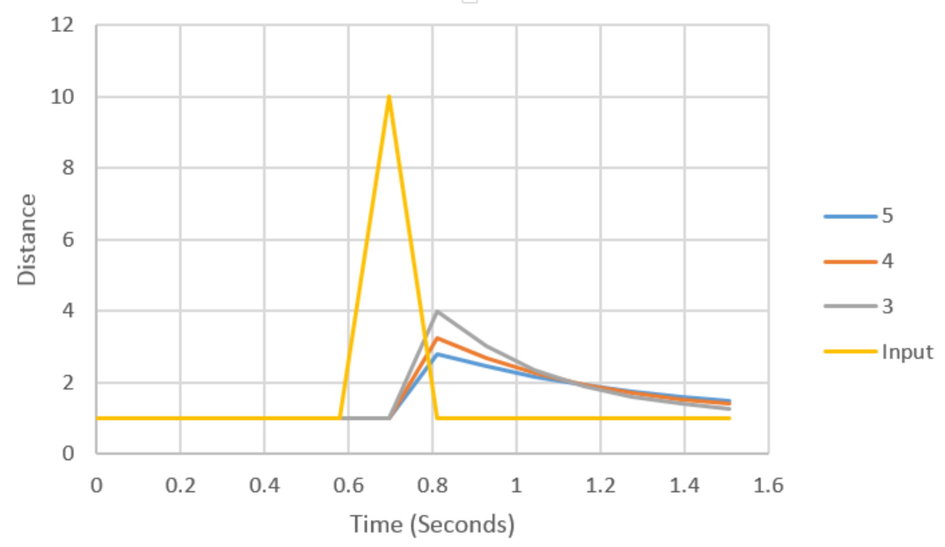


Figure 11: Averaging function response to impulse

## 3.3: Deciding a Direction

The decision unit of the FPGA is an independent module that can be replaced easily. The inputs to the decision module are the outputs of the average module. The outputs of the decision module are the inputs of the motor control module. An overall diagram of the how the modules link together can be seen in **Figure 12.**

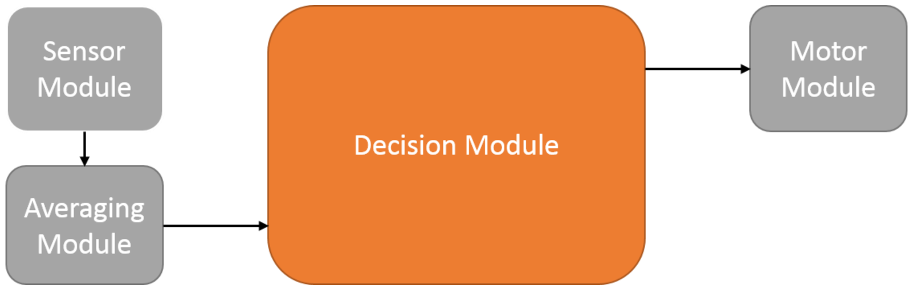


Figure 12: Algorithm Layout

The decision unit that is currently implemented is a randomized direction decider. This algorithm works by checking all the sensors average data to determine where there are openings big enough for the car to navigate through. Once the module has a list of all possible openings, it chooses an opening and goes through it. The car continues until it runs into a dead end. Once the car reaches a dead end, it stops.

This model was chosen to test the functionality of the car and show that it is possible to create a car that navigates a room. The decision algorithm is intended to be as simple as possible.

A more in-depth algorithm that had been discussed was to plot the points of locations where there are barriers in memory. The algorithm could then choose routes based on remembering where it had been. The algorithm could either choose to explore new locations or navigate old locations. This algorithm would require another module to record speed and location using the sensor data. This new module could be seen in **Figure 13.**

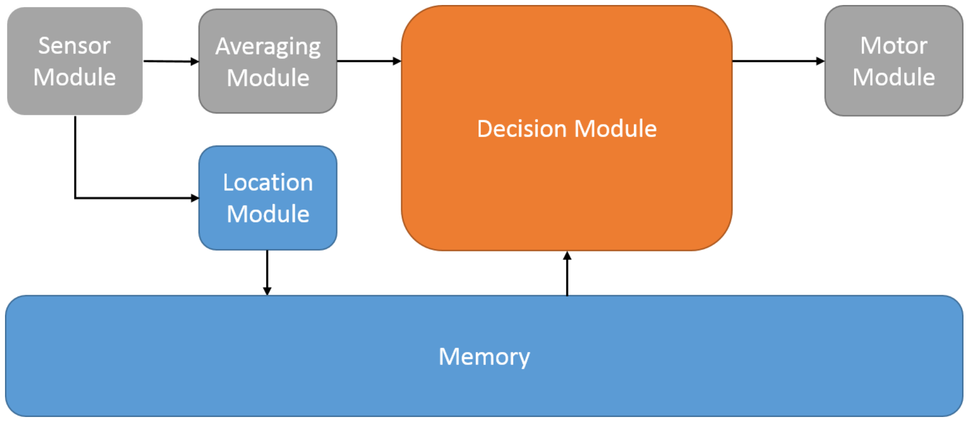


Figure 13: Module layout for another possible algorithm.

From the **Figure 13**, it can be seen that more modules would be needed to be added to the algorithm which could require a lot more space. However, this would give the decision module a lot more information to work with and make decisions about. The location module would collect the sensor data to generate the coordinates of where walls were located. The coordinates would then be saved into memory to be used by the decision unit later. The location module would keep track of the speed and direction of the car. This would allow the location module to keep track of the cars position relative to the cars initial position. Using the cars position and information coming from the sensors, the coordinates of the different obstacles could be plotted and stored in memory. The decision module could read the coordinates from memory to determine the possible pathways to take.

# Chapter 4

# Results

After these modules had been connected together and loaded onto the FPGA, the car was tested in a hallway setting.  The car was able to navigate a lab for just over 5 minutes.  The car could consistently identify and avoid obstacles.  The algorithm used to decide where to go was the randomized decider.  This decision module was simple to implement and it showed that only minimal programming was required to accomplish the task of navigating a room.  A C-like code snippet in **Figure 14** shows the procedure that the algorithm was following.

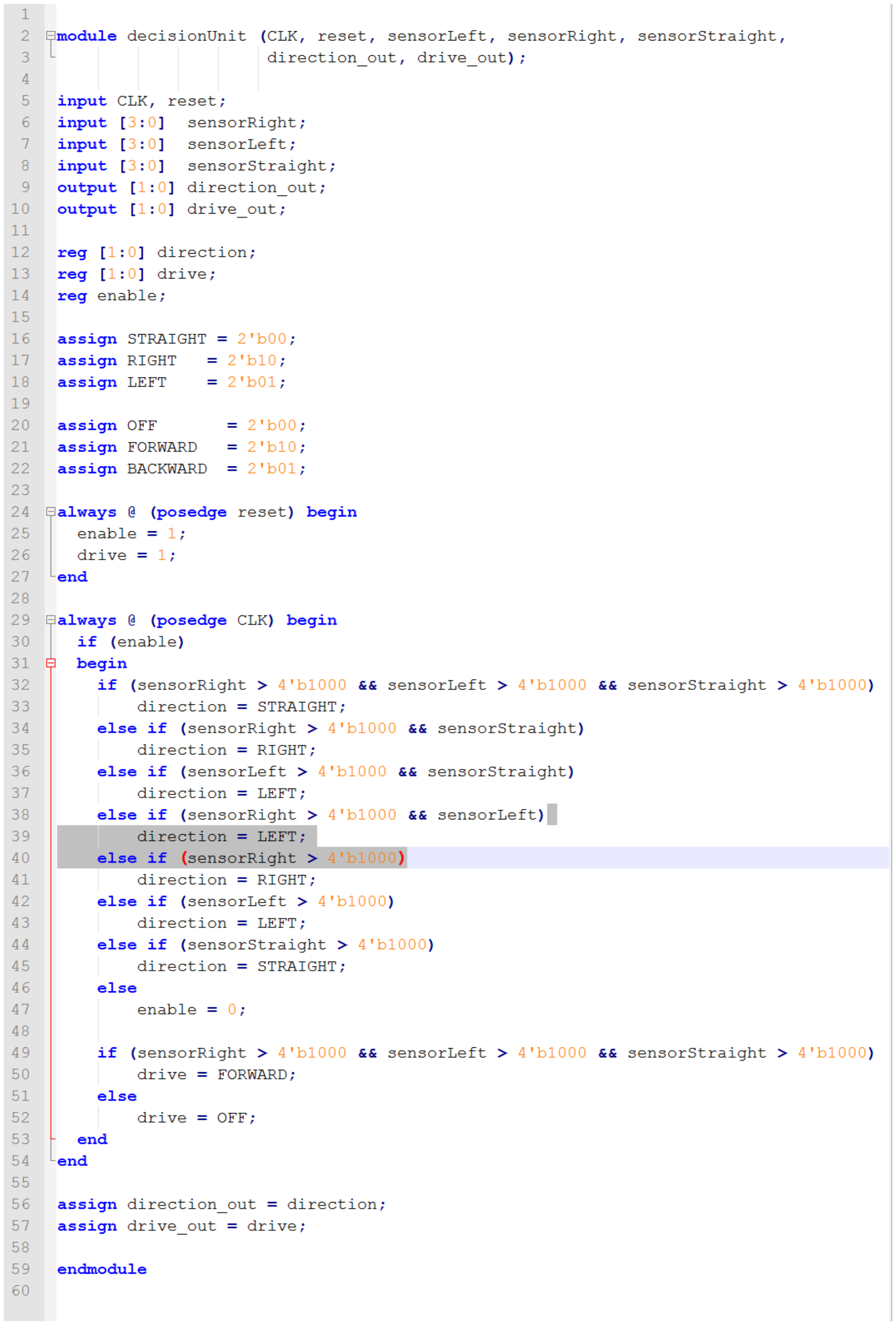
In **Figure 15**, the actual Verilog code used to implement the decision unit can be seen.  Although the if-statements look similar between the two code snippets, Verilog calculates the if-statement conditions at the same time and determines the direction in one clock cycle while C will calculate each if-statement individually, which costs more than one clock cycle.

Unfortunately, if the car sees openings in all directions, the car will continue driving straight until it sees another surface.  This is a good algorithm when there are wide open spaces where the car would need to travel to find a new barrier, it can be troublesome in a confined space.  In confined spaces, the openings might be a result of the ultrasonic sound from the sensors bouncing off a wall and not returning.

```
main ()
  {
    bool stop = false;
    drive = FORWARD;

    while (!stop)
      {
        if (sensorRight > 8 meters && sensorLeft > 8 meters && sensorStraight)
          {
            direction = STRAIGHT;
          }
        else if (sensorRight > 8 && sensorStraight)
          {
            direction = RIGHT;
          }
        else if (sensorLeft > 8 && sensorStraight)
          {
            direction = LEFT;
          }
        else if (sensorRight > 8 && sensorLeft)
          {
            direction = LEFT;
          }
        else if (sensorRight > 8)
          {
            direction = RIGHT;
          }
        else if (sensorLeft > 8)
          {
            direction = LEFT;
          }
        else if (sensorStraight > 8)
          {
            direction = Straight;
          }
        else
          {
            stop = true;
            drive = OFF;
          }
      }
    return 0;
  }
```

**Figure 14: C-like code depicting decision unit**



```verilog
module decisionUnit (CLK, reset, sensorLeft, sensorRight, sensorStraight,
                     direction_out, drive_out);

input CLK, reset;
input [3:0]  sensorRight;
input [3:0]  sensorLeft;
input [3:0]  sensorStraight;
output [1:0] direction_out;
output [1:0] drive_out;

reg [1:0] direction;
reg [1:0] drive;
reg enable;

assign STRAIGHT = 2'b00;
assign RIGHT   = 2'b10;
assign LEFT    = 2'b01;

assign OFF       = 2'b00;
assign FORWARD   = 2'b10;
assign BACKWARD  = 2'b01;

always @ (posedge reset) begin
  enable = 1;
  drive = 1;
end

always @ (posedge CLK) begin
  if (enable)
  begin
    if (sensorRight > 4'b1000 && sensorLeft > 4'b1000 && sensorStraight > 4'b1000)
        direction = STRAIGHT;
    else if (sensorRight > 4'b1000 && sensorStraight)
        direction = RIGHT;
    else if (sensorLeft > 4'b1000 && sensorStraight)
        direction = LEFT;
    else if (sensorRight > 4'b1000 && sensorLeft)
        direction = LEFT;
    else if (sensorRight > 4'b1000)
        direction = RIGHT;
    else if (sensorLeft > 4'b1000)
        direction = LEFT;
    else if (sensorStraight > 4'b1000)
        direction = STRAIGHT;
    else
        enable = 0;

    if (sensorRight > 4'b1000 && sensorLeft > 4'b1000 && sensorStraight > 4'b1000)
        drive = FORWARD;
    else
        drive = OFF;
  end
end

assign direction_out = direction;
assign drive_out = drive;

endmodule
```

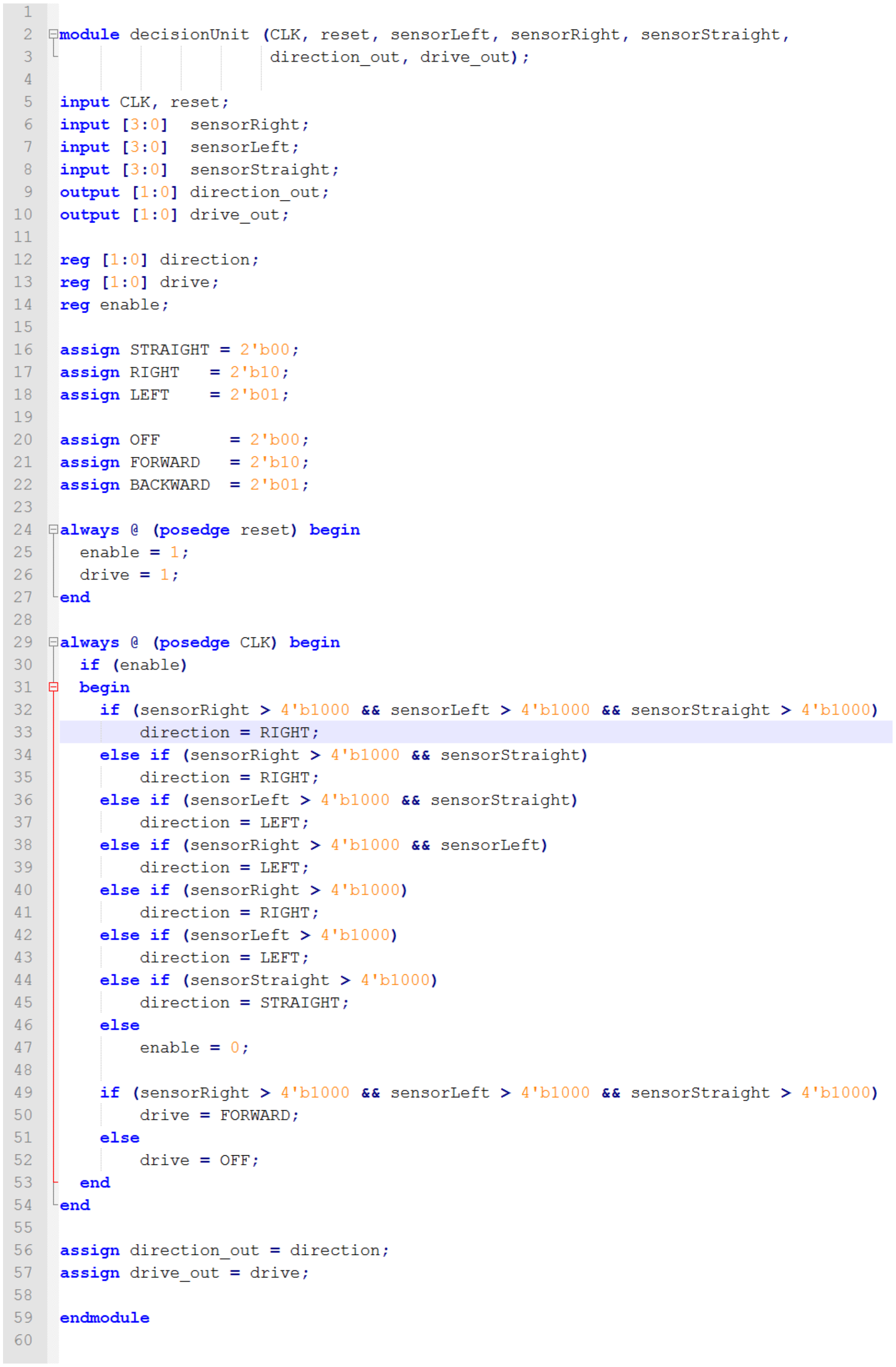
Figure 15: Verilog code implementing design unit

Therefore, the code was modified in the decision unit to allow the car to realign itself with a flat surface that it could follow.  With the addition of this module, the car was able to navigate a hallway with better results than the lab.  The car was able to navigate the hallway for 10 minutes.  Other than the addition of the module to help the car realign itself with a flat surface, the decision unit was still using a random decider.  A C-like code snippet in **Figure 16** shows the new procedure that the algorithm was following.  The corresponding Verilog code can be seen in **Figure 17.**

```
main ()
  {
    bool stop = false;
    drive = FORWARD;

    while (!stop)
      {
        if (sensorRight > 8 meters && sensorLeft > 8 meters && sensorStraight)
          {
            direction = RIGHT; // Until a flat surface is detected
          }
        else if (sensorRight > 8 && sensorStraight)
          {
            direction = RIGHT;
          }
        else if (sensorLeft > 8 && sensorStraight)
          {
            direction = LEFT;
          }
        else if (sensorRight > 8 && sensorLeft)
          {
            direction = LEFT;
          }
        else if (sensorRight > 8)
          {
            direction = RIGHT;
          }
        else if (sensorLeft > 8)
          {
            direction = LEFT;
          }
        else if (sensorStraight > 8)
          {
            direction = Straight;
          }
        else
          {
            stop = true;
            drive = OFF;
          }
      }
    return 0;
  }
```

**Figure 16: C-code depicting decision unit with added module**



```
module decisionUnit (CLK, reset, sensorLeft, sensorRight, sensorStraight,
                     direction_out, drive_out);

input CLK, reset;
input [3:0]  sensorRight;
input [3:0]  sensorLeft;
input [3:0]  sensorStraight;
output [1:0] direction_out;
output [1:0] drive_out;

reg [1:0] direction;
reg [1:0] drive;
reg enable;

assign STRAIGHT = 2'b00;
assign RIGHT   = 2'b10;
assign LEFT    = 2'b01;

assign OFF       = 2'b00;
assign FORWARD   = 2'b10;
assign BACKWARD  = 2'b01;

always @ (posedge reset) begin
  enable = 1;
  drive = 1;
end

always @ (posedge CLK) begin
  if (enable)
  begin
    if (sensorRight > 4'b1000 && sensorLeft > 4'b1000 && sensorStraight > 4'b1000)
        direction = RIGHT;
    else if (sensorRight > 4'b1000 && sensorStraight)
        direction = RIGHT;
    else if (sensorLeft > 4'b1000 && sensorStraight)
        direction = LEFT;
    else if (sensorRight > 4'b1000 && sensorLeft)
        direction = LEFT;
    else if (sensorRight > 4'b1000)
        direction = RIGHT;
    else if (sensorLeft > 4'b1000)
        direction = LEFT;
    else if (sensorStraight > 4'b1000)
        direction = STRAIGHT;
    else
        enable = 0;

    if (sensorRight > 4'b1000 && sensorLeft > 4'b1000 && sensorStraight > 4'b1000)
        drive = FORWARD;
    else
        drive = OFF;
  end
end

assign direction_out = direction;
assign drive_out = drive;

endmodule
```

Figure 17: Verilog code implementing decision unit with added module

Since the FPGA was not used to its full potential, the amount of data that is required to process increases, the speed of the FPGA did not appear to slow down. For instance, if a camera sensor were to be included, the FPGA would break down the image into several sections which could independently be broken down and searched rather than searching the entire image one pixel at a time.

If a microprocessor had a clock speed of 1 GHz and an FPGA had a clock speed of 100 MHz, and the microprocessor is running the C algorithm while the FPGA runs the Verilog algorithm, it can be seen that the FPGA will be faster. If the FPGA read the sensors data at the same time, the total time to execute the algorithm will be approximately 0.029 seconds, the maximum time it takes to read the sensors. The microprocessor will need to read each sensor individually. This will take a maximum of 0.116 seconds, four times as long as the FPGA.

Even after the data is read from the sensors, the decision unit of the FPGA would run faster than the microcontroller. The FPGA would complete the algorithm in one clock cycle, 10 nanoseconds. The microcontroller would need to calculate each of the 8 if-statements individually. Each if statement requires more than 1 clock cycle to determine whether the condition is true. Once the microcontroller chooses an if-statement, there are a maximum of 2 instructions that need to be executed inside, each one takes about a clock cycle more. If each if-statement requires 2 clock cycles to determine which if-statement is true, then the microcontroller needs 18 clock cycles to complete its operation. 18 microcontroller clock cycles takes about 18 nanoseconds. That is nearly double the time that it takes the FPGA.

Since the decision unit could run while data was being gathered from the sensors, the FPGA is able to handle more complex situations. It is possible that the sensors could all be read at the same time, allowing the FPGA to react to situations even more quickly than before. The decision unit is also able to calculate faster because each if-statement that C would need to go through one-at-a-time is calculated at the same time.

## BIBLIOGRAPHY

1. Gomes, L. (2014). *Hidden Obstacles for Google's Self-Driving Cars*. MIT Technology Review. http://www.technologyreview.com/news/530276/hidden-obstacles-for-googles-self-driving-cars/
2. Erico, G. (2013). *How Google's Self-Driving Car Works*. *IEEE Spectrum, 18* *http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/how-google-self-driving-car-works*
3. Patterson, D. A., Hennessy, J. L. (2014). Computer Organization and Design (Fifth Edition). Waltham, MA: Elsevier.
4. *What is an FPGA?* Altera. http://www.altera.com/products/fpga.html
5. *Logic Gate.* Wikipedia. http://en.wikipedia.org/wiki/Logic_gate
6. *Logic Gate (AND, OR, XOR, NOT, NAND, NOR, and XNOR).* TechTarget. http://whatis.techtarget.com/definition/logic-gate-AND-OR-XOR-NOT-NAND-NOR-and-XNOR
7. *Look-up Tables*. All About Circuits. http://www.allaboutcircuits.com/vol_4/chpt_16/2.html
8. *Dataflow Programming.* Wikipedia. http://en.wikipedia.org/wiki/Dataflow_programming
9. *How Does an Ultrasonic Sensor Work?* Teach Engineering. https://www.teachengineering.org/view_lesson.php?url=collection/umo_/lessons/umo_sensorswork/umo_sensorswork_lesson06.xml
10. *Boolean Algebra.* Hong Kong Institute of Education. https://www.ied.edu.hk/has/phys/de/de-ba.htm

11. *DC Motor Driving using H Bridge*. electroSome.

http://electrosome.com/dc-motor-driving-using-h-bridge/

12. *Ultrasound*. SensorWiki.

http://www.sensorwiki.org/doku.php/sensors/ultrasound

13. *AND gate*. Wikitronics.

http://electronics.wikia.com/wiki/AND_gate

14. *US-020 Ultrasonic Ranging Sensor*. Electrow.

http://www.elecrow.com/us020-ultrasonic-ranging-sensor-p-687.html

15. *H BRIDGE CONTROL OF A MOTOR – LOW SIDE SWITCH*. Solutions Cubed.

http://blog.solutions-cubed.com/h-bridge-control-of-a-motor-low-side-switch/

16. *iCEblink40HX1K Evaluation Kit.* Lattice Semiconductor.

http://www.latticesemi.com/iceblink40-hx1k

17. *iCE40 LP/HX/LM*. Lattice Semiconductor.

http://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40.aspx

18. *How FPGAs Work*. Fpga4fun.

http://www.fpga4fun.com/FPGAinfo2.html

## ACADEMIC VITA

**Austin T. Crain**
501 Vairo Blvd Apt. # 2222, State College, PA 16803 ♦ 610-554-0906 ♦ aoc5329@psu.edu

### Objective

Electrical engineer with a background in computer science pursuing a hardware or software engineering position in computer security.

### Education

- **Pennsylvania State University** University Park, PA Graduate in December 2014
  **Schreyer's Honor College**
  *Bachelor of Science in Electrical Engineering*

### Employment

- **Intel Corporation Intern** Folsom, CA 5/2014-8/2014
  Management Engine Firmware Debug
  - Replicated issues seen on customer systems
  - Interacted with customers to narrow issues down and resolve them quickly
  - Facilitated debugging process by embedding a website in an eclipse application
    - Converted JavaScript code to Java
    - Drove Intel groups from Germany to add eclipse plug-in to main distribution
- **Intel Corporation Intern** Folsom, CA 5/2013-8/2013
  Debug Tools
  - Helped simplify a debugging program by standardizing XML files
  - Wrote a binary parse to map firmware debug messages to a standard application programming interface (API)
    - Collaborated with Intel engineers in Israel to validate the binary parse
- **Pennsylvania State University** Allentown, PA 8/2012-5/2013
  Tutor: Math, Programming, and Physics

### Activities

- Prepared and presented "Online Tools in the Classroom" to an audience of professors

### Achievements

- Dean's List: Pennsylvania State University

### Additional Skills or Qualifications

| | |
|---|---|
| *Proficient in C, C++, C#, Java, Python, Verilog* | *Certified Labview Programmer* |
| *Proficient in English and Spanish* | *CPR Certified* |
| *US Citizen* | *Word, Excel, Powerpoint* |