Massachusetts Institute of Technology

# SELF-PARKING CAR

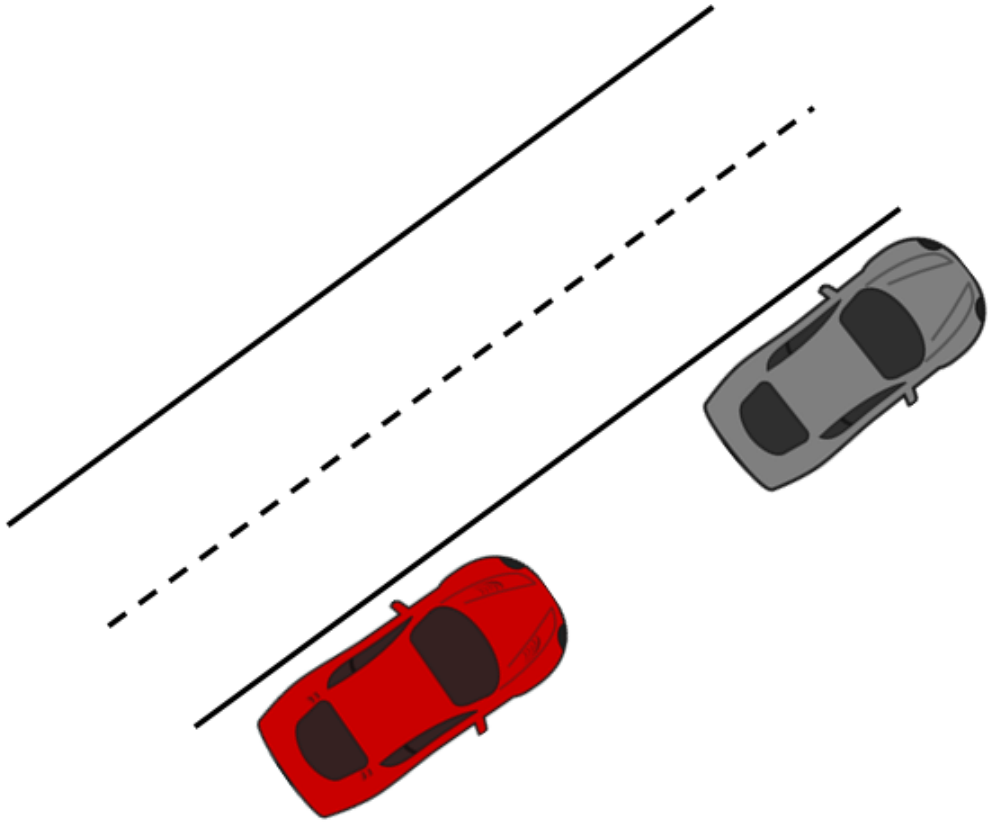
Fei "Frank" Ni and Kevin Hsiue

12/12/12

6.111 Final Project

## Abstract

We believe in the simplicity of day-to-day actions. We want to make what was once difficult to perform an easy task for the average individual. So for our final project, we aim to create a car that parallel parks itself. The car will use 4 infrared sensors to provide "eyes", which will feed data to an on-board FPGA that will control its motors, using a finite state machine, to safely guide the car from the moment a driver decides to park to the point when the car is in the correct position.

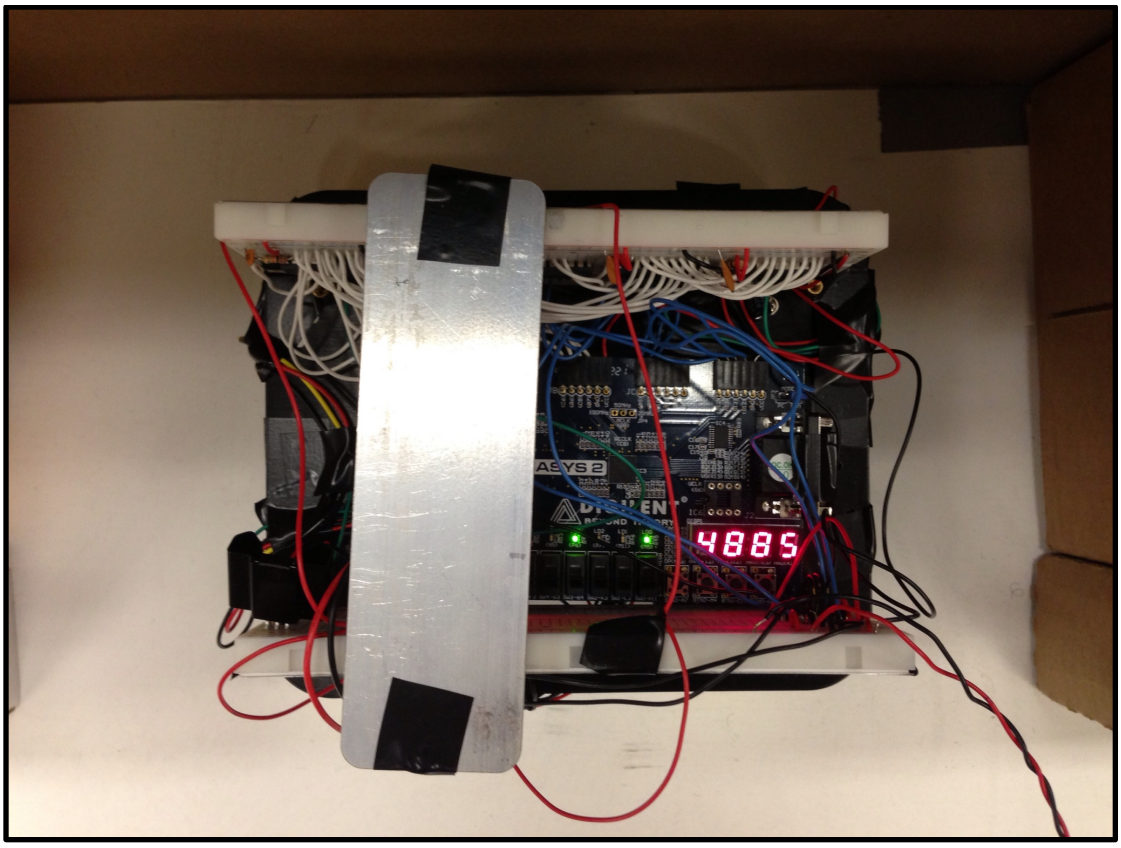


(Figure 0): A typical parallel parking challenge

# Table Of Contents

## 1. Design Overview

As the number of cars increase on the road, it becomes increasingly difficult to find a parking space. However, the final option of parallel parking is usually a driver's worst nightmare because not only of the driver's own skills but also the possibility of other drivers bumping into their parked vehicle. As such, people will drive around for half an hour to avoid facing this challenge. In an attempt to provide more simplicity into people's lives, our project will develop a system that enables a car to parallel-park itself.

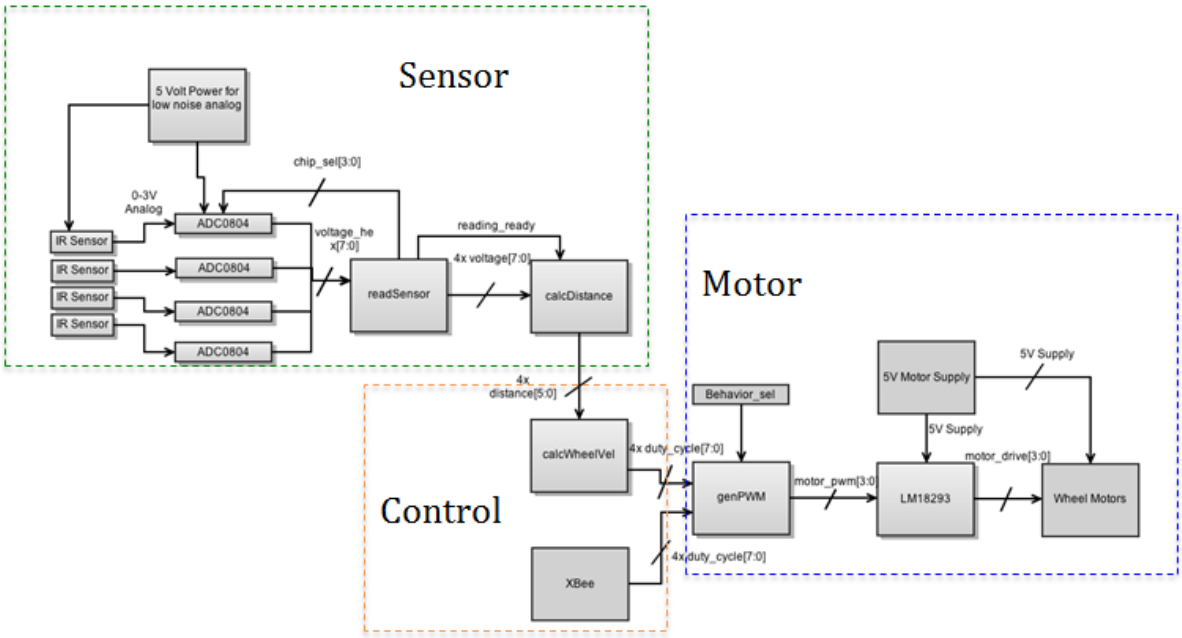



(Figure 1): The vehicle with the 7-segment display showing the distance reading

Of course, it would be very expensive to attempt this project on a real car, so we will be using a four-wheel drive miniature vehicle to simulate the mechanics behind a real car as shown in Figure 1. The parking space itself will be predefined with the car pulled up to the "car" before the space itself. From there, the automated control system, using inputs from

four IR distance sensors and feedback loops, will direct the vehicle into the desired position within the parking space.

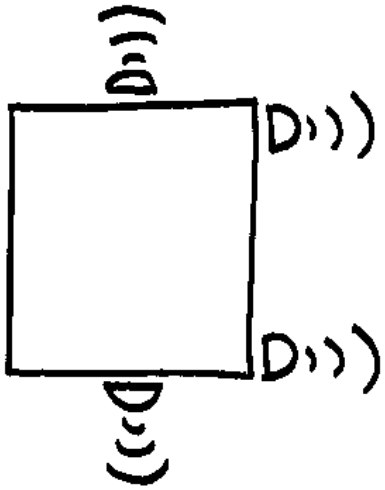
Overall, this project will demonstrate the ability of an FPGA to host a very precise control system to navigate a vehicle into a desired position. The entire system will be self-confined because we'll be using the Basys FPGA board, which, along with other necessary circuitry, can be mounted onto the car itself. The motors will be powered by a lab bench supply and the board will be powered by an on-board battery pack. In addition, the state machine will be programmed to simulate the driver initially pulling up to the parking space, and initiating the self-parking procedure.




(Figure 2): Block diagram of major and minor modules

## 2. Sensors (Author: Frank Ni)

The overall sensing block will provide distance data to the control block. It will involve interfacing with the IR sensors and converting it into a format that is easy to deal with in the control step.



(Figure 3): Infrared sensors that measure distances are mounted on the vehicle

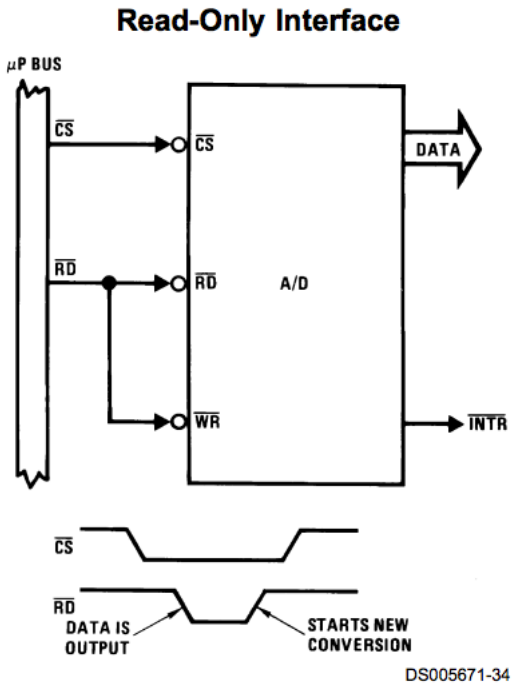
### 2.1 IR Sensors

The IR sensors, purchased from Sparkfun, will be mounted as shown in Figure 3. Each of these requires a 5 volt supply and outputs a voltage between 0 and 3V. This output voltage is inversely proportional to the distance seen by the sensors, so we will keep this in mind when using the reading to control the FSM. The range on these sensors is from 3 cm to 30 cm, which is great for the distances we will be measuring during a parking procedure. The sensors are connected uses JST connectors, which are soldered to wires that can be connected to breadboards. The voltages are connected to the V_IN of the ADC0804s. These sensors were tested by wiring them up to a voltmeter while changing the distance measured.

### 2.2 ADC0804 chip interfacing

The voltages from the IR sensors will be fed individually into an ADC0804 chip, which will convert the analog voltage into a digital format that the FPGA can understand. The problem with these chips is that each of them requires 8 lines of data, which means we

need 32 I/O ports on the FPGA. However, the Basys2 FPGA only has sixteen. In order to resolve this, we decided to use the chip enable pins on the chips and read each of them sequentially rather than in parallel. As a result, each of the chips will be wired up in Read-Only mode as shown below.




(Figure 4): the ADC0804 chip wired in read-only mode and the timing diagram
*- Image Courtesy of National Instruments*

Instead of the chip select line coming from the FPGA, it is tied to ground so that the chip is continuously selected. However, when RD* is high, the DATA out is tri-state, which allows only one A/D to be driving the input pins on the Basys board. As such, all the DATA ports of the AD0804 are tied together and fed into the Basys board. The control pins of these chips come from a multiplexor that determines which convertor to read. Using this MUX allows us to free up one more pin for future purposes (ie. serial communication) because originally each A/D would require a RD line which in total is 4 pins and now by using the MUX, we can use 2 pins to select which A/D to read and another pin to send the correct RD pulses.

The clocks on the ADCs are created using the on-chip clock generator by wiring up a 10k resistor and a 150 pF capacitor. This generates a 600 kHz clock for the chips themselves.

## 2.3 Sensor Reading Module

The readSensor module will cycle through each of the ADCs and generate the necessary protocol to allow the ADCs enough conversion time to convert the analog voltages. After collecting all four voltages, it will create a reading_ready signal to let the next module know that it has finished collecting. In addition this module filters the sensor readings of any possible noise.

A clock divider is used to generate a 300 kHz clock. Then this module start counting to 100, and when the count hits 0, 1, 2, and 3, the module sets *chip_read* to low and selects each ADC respectively. Waiting for 100 counts is to allow the ADC enough time to convert the analog signal to a digital signal. The conversion time according to the data sheets is 114 microseconds at 600 kHz. In the case of this module, 100 cycles at 300 kHz is roughly 300 microseconds which is plenty for each of the ADCs to convert. Upon obtaining all the readings, the module then creates a *dirty_RR*, which lets other modules know that the un-cleaned sensor readings are ready.

A helper module is used to filter the sensor reading. The sharp sensors introduce a lot of noise into the system, which causes the reading to occasionally spike as well as change rapidly. In order to remove the noise, this helper module implements a rolling average with a window of 8 to filter out the noise. This filter creates a window on the dirty sensor readings, and takes the average of the readings within that window and outputs the averaged reading. When a new reading is ready, the window is shifted by 1 and a new average is outputted. In addition, this module makes sure that the sensor reading is stable for 0.04 seconds before considering it a real reading which removes any sudden jumps in reading due to voltage spikes or noise. We chose 0.04 seconds because we know the motors are relatively slow, which means the vehicle won't be able to move far in 0.04 seconds so the sensor readings are still fast enough to detect any sudden change in contour of the surfaces.

When the readings have been filtered, the *reading_ready* signal is asserted and the readings from each of the 4 sensors are held on the output buses of the sensor-reading

module. These readings are fed into the next module as well as a 7-segment display module that converts the high nibble and display it onto one of four displays on the Basys board. This allows us to visually see what the sensors are reading when they are in a certain state.

This block is tested along with the ADC chips first on the lab kit by outputting the readings onto the hex display. Then it was tested on the Basys board using the 7-segment display.

### 2.4 Calculate Distance Module

This module will take each of the four readings and do the necessary calculations to invert the voltages to obtain the actual distance between the sensor and the obstacle. The distances are then passed onto the FSM module of the control block. However, taking the inverse of the distances decreases the accuracy of the sensor readings. This is fine for further distances, but for closer distances, more accuracy is needed for precision control. As a result this module was not implemented and the sensor inputs are used directly to control the vehicle.

# 3. Motor and Chassis (Author: Kevin Hsiue)

## 3.1 Vehicle chassis

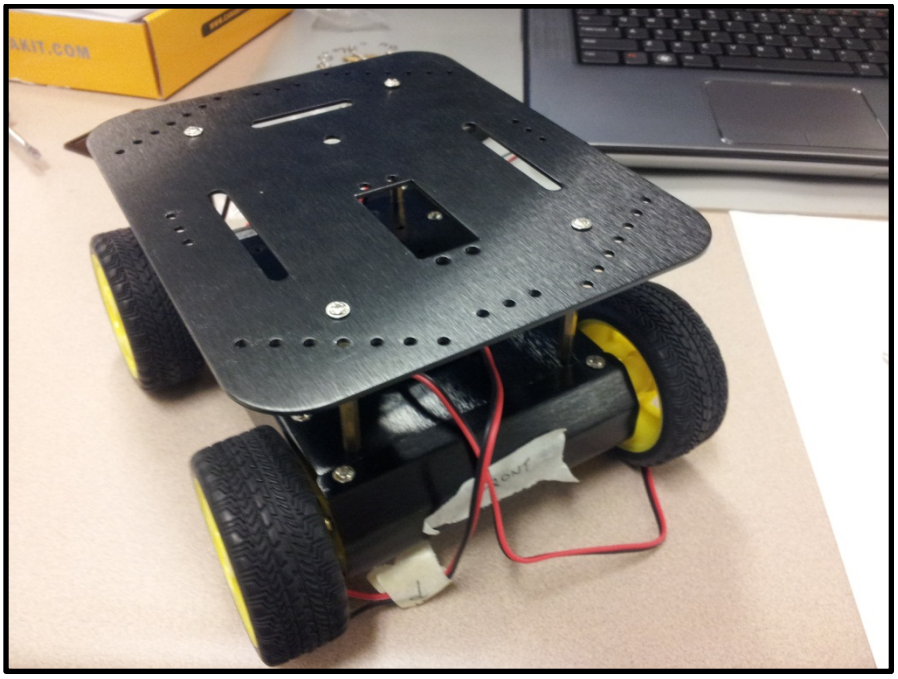
The chassis used in this project was the DFRobot 4WD Arduino Mobile Platform, which was easily accessible and straightforward to construct. The chassis itself allowed had abundant room for various expansions, which we used for adding two prototype breadboards, the Basys2 FPGA, four IR Sharp sensors, as well as the battery supply.

One interesting issue that we had to confront was how to accurately simulate a car's behavior given a chassis with four independently operated DC motors and wheels. This project required a chassis that was capable of faithfully emulating the rack-and-pinion steering of an actual car. However, most robotics platforms do not replicate this behavior, instead often independently driving each DC motor on the chassis.

Therefore, in order to accurately portray the steering of the car, software was used to program the behavior of the model car to drive in the correct behavior, as opposed to being capable of rotating in place like a tank. We did purchase a rack-and-pinion style turning chassis, the i-Racer from SparkFun Electronics, but ultimately decided that the DFRobot chassis was more reliable. Given the demanding schedule, we justified the decision to use a 4WD chassis due to its increased reliability and more robust platform capabilities.

(Figure 5): 4WD Platform before assembly

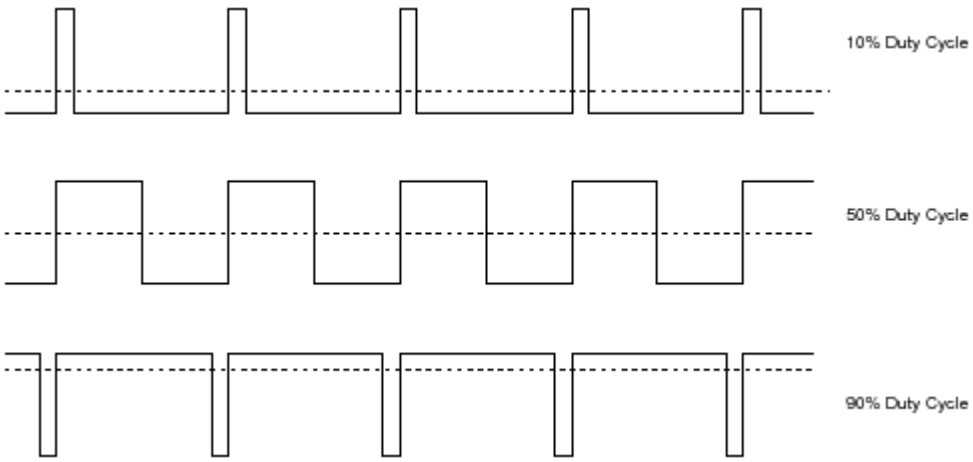
Assembling the chassis was very straightforward. The kit came with no instructions but was very intuitive with how to assemble the DC motors and solder the appropriate connections. Once the base of the chassis was assembled, the platform was installed by adding the larger spacers to suspend the upper level platform over the base of the chassis. We attached the sensors using electrical tape to the front, right side, and back. The two breadboards with the circuitry were propped on their sides on top of the platform, and the Basys2 FPGA was placed in the middle of the two breadboards.



(Figure 6): 4WD Platform after assembly

## 3.2 Pulse Width Modulation

Before discussing the hardware used to drive the chassis motors, it is necessary to discuss how this project used pulse width modulation. Pulse width modulation is a technique used to control electrical devices and is controlled by a square wave cycle that has a variable 'on' period, or duty cycle. By varying the duty cycle at the appropriate frequency unique to the particular electrical device, an average value based on the frequency can be obtained, correlating to that percentage of the supply voltage. For example, if a pulse width has a fifty percent duty cycle with a supply voltage of 5 volts, then the output voltage would be 2.5 volts.




(Figure 7): Varying pulse width modulation duty cycles

– *Image courtesy of [www.best-microcontroller-projects.com](http://www.best-microcontroller-projects.com)*

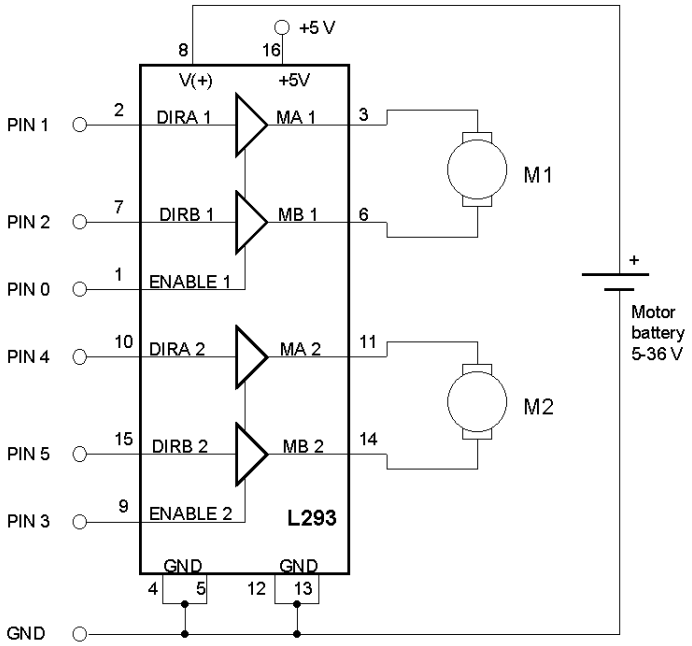
In our project, a specific Verilog module, *pwmgen.v,* was written to generate varying duty cycles. The Basys2 has an onboard clock of 50 MHz, therefore we had to use the *divider.v* module to obtain the necessary frequency of 5 kHz required by the L293 motor driver chip.

The pulse width modulation was achieved by a state machine that took an input ranging from 0 to 10. The state machine would then count to that amount based on the clock frequency with a high output, then change to an output of low for whatever ten minus that input value. This would successfully achieve a varying duty cycle. The high and low outputs were implemented as states, and the span of time in each state was determined by a count variable. This simple state machine would check if the count had expired in that state, then automatically switch to the other state and begin the appropriate count. Given this interface, the project could simply provide an input ranging from 0 to 10 and produce a pulse width modulated signal of that appropriate percentage duty cycle.

### 3.3 L293 Motor Chips

The motor driver chip used in this project was the L293. The L293 is a quadruple high-current half-H driver; it is designed for bidirectional drive currents up to 1 amp for voltages ranging from 4.5 volts to 36 volts. The chip is TTL compatible and therefore is deal

for this project; we can supply a signal voltage, or specific pulse width, to create a certain duty cycle and control the amount of the supply voltage.




(Figure 8): The L293 schematic

– *Image courtesy of [www.me.umn.edu](http://www.me.umn.edu)*

While the chip is often used for driving a motor in both directions by reversing the polarity (signal and ground) on either input, as shown in the schematic above, we opted to ground one input of each motor. This was due to the constraint of the FPGA input/output limit; we had to frugal with the outputs and decided to only drive on set of wheels (front or back) in one direction. While this did save four input/output ports, it also limited the capability of the vehicle since now only two wheels are spinning for each direction.
From there, the L293 took four unique inputs and supplied the DC motors with a driving signal. Therefore, we could independently control each motor and drive it at a different speed based on the duty cycle of the pulse width modulation.

# 4. Finite State Machine (Authors: Frank Ni and Kevin Hsiue)

The finite state machine (FSM) is divided into 10 states; each of which correlates to a step that is taken in parallel parking performed by human drivers. This module takes the four clean sensor readings as input and then outputs a duty cycle for each of the wheels to control their speeds. In order to keep track of which state the vehicle is in, the states are displayed on the LEDs.

This module waits for a second after the start/reset button is pushed. This allows time for the sensors to initialize so that the state machine doesn't begin in an undesired state.

## 4.1 States

*Prestart_step1:*

This is the first step taken before starting the parking procedure. We are simulating the point in which the driver decided that he/she wants to park and starts toward the curb full of cars diagonally. If the vehicle is close enough to the parked cars, which is determined by the side_front sensor reading, the FSM transitions to the "prestart_step2" state.

*Prestart_step2:*

In this state, the car straightens itself out so that it is parallel to the row of parked cars. The vehicle continuously turns left until both side sensors have the same reading, which would indicate that the car is parallel to the parked cars. At this point, the state machine begins the parking procedures by transitions to the "start" state.

*Start*:

Once the procedure enters the start state, the vehicle drives forward and looks for a parking spot that is large enough for the vehicle to fit. This is indicated by the side sensors both reading a far distance reading. If one sensor reads a far reading, but the other one does that, then the space that was detected is too small for the car to fit. When a suitable spot is detected, the FSM is transitioned to the "middle" state.

*Middle:*

This state determines when the open spot has ended and when the vehicle has pulled up to the parked car in front of the open space. In this state, the car continues to drive forward. When the side sensors both register a "close" reading, then the car knows that it is next to the car in front of the space. At this point, the car enters the "back_up" state.

*Back_up:*

In this state, the car backs up straight until the back sensor is been reading far for at least 0.1 seconds. Then it calculates how far the side_front sensor is from the car in front of the space. This information is used to determine how much the car should turn inward in the next state before it is angled correctly into the open spot. Once 0.1 seconds has passed, the FSM transitions to the "turn_in" state.

*Turn_in:*

The vehicle at this point should begin turning into the open spot by driving the left side of the vehicle faster than the right side while moving the wheels in reverse. Similar to the back_up state, if the side_front sensor is a certain threshold away from the car in front of the open spot, the vehicle transitions to the "back_in" state.

*Back_in:*

In this state, the car starts correcting its angle into the spot by driving the right side wheel faster than the left side wheel in proportion to how close it is to the car behind it using the back sensor. Once the back sensor registers a "close" reading to the car behind it, the state machine enters the "straight" state.

*Straight:*

The car corrects its position by wiggling itself until the side sensors both read the same reading, which indicates that the car is parallel to the curb. At this point the car enters the "end" state.

*End:*

In the "end" state the car drives forward until it is close to the car in front of it. Then the car stops and it is considered parked.

## 5. Conclusion (Authors: Frank Ni and Kevin Hsiue)

The project was ultimately successful; we were able to build the model car, interface between sensors and motor drivers, and in the end, program parallel-parking behavior. However, there were several challenges that arose during the process.

The largest problem that came up during the project was the reliability of the chassis itself. It is a common problem in robotics for the physical plant to not be cooperative; we experienced this issue to its fullest extent. The tires of the chassis slipped on the tile surface of the lab floor, and even would slip on a piece of paper that we used as a more tactile surface. This made it extremely difficult to consistently obtain experimental results and appropriately change our algorithm to deal with the issue. We ended up using an aluminum weight to push the model car down, which helped to a certain extent.

The Verilog code even took this unreliable behavior into account and in many instances depended on this drag of the tires to turn appropriately. Again, this variability resulted in the difficulty of pinpointing the solution in both hardware and software. Regardless, we found the balance between the unreliable physical world and definite software world and produced a working result.

Another issue that arose was power supply. The Basys2 FPGA took three AA batteries which were powered on board. However, the motor supply consumed a huge amount of power. Using a power supply train of six AA batteries also allowed for unreliable behavior; the model car would behave differently as the voltage supply decreased. Therefore, we chose to tether the model car to the lab voltage supply in order to guarantee repeatable results.

While the previous two challenges did inhibit our experimental results to a certain extent, we were ultimately successful and being able to see somewhat repeatable parallel parking behavior in the model car. With so many factors that could contribute to incorrect behavior, we were satisfied with the end result of this project.

# Appendix A: Verilog Code

## Labkit.v

```
module labkit (clk_50mhz, seg, dp, Led, digit, sw, btn, user1, user2);
input clk_50mhz;
output [6:0] seg;
output dp;
output [7:0] Led;
output [3:0] digit;

input [7:0] sw;
input [3:0] btn;

inout [7:0] user1, user2;

////////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////////////////////////////////////////////

//Use these if you want to use outputs to display stuff
//Comment them out if you need to use them
//assign seg = 7'b1111111;
assign dp = 1'b1;
assign Led[7:4] = 4'd0;
//these are used to select which LED segment display is used
//I think it's 0 to select the display
//assign digit = 4'b0000;

//io ports
//assign user1 = 8'hz;
assign user2[3] = 1'hz;

//use btn[3:0] and sw[7:0] for inputs

wire reset;
wire clk600;
wire clk_seg;
wire clkmotor;

wire [7:0] display1, display2, display3, display4;

debouncer btn_r(.reset(0), .clock(clk_50mhz), .noisy(btn[0]), .clean(reset));

clockMaker #(.DIV(180)) clk_600(.clock(clk_50mhz), .reset(reset), .div_clk(clk600));
clockMaker #(.DIV(10000)) clkseg(.clock(clk_50mhz), .reset(reset), .div_clk(clk_seg));
```

```
clockMaker #(.DIV(100000)) clk_motor(.clock(clk_50mhz), .reset(reset),
.div_clk(clkmotor));

displaySeg disp1(.display1(display1), .display2(display2), .display3(display3),
.display4(display4),
                                                .clk(clk_seg), .reset(reset), .seg(seg),
.digit(digit));

wire [7:0] distF, distS1, distS2, distB;
wire reading_ready;

adcRead adc1(.reset(reset), .clock(clk600), .value(user1[7:0]), .v_out0(distF),
                                    .v_out1(distS1), .v_out2(distS2), .v_out3(distB),
                                    .chip_read(user2[0]), .chip_sel(user2[2:1]),
.reading_ready(reading_ready));

wire [3:0] dc_BL, dc_BR, dc_FL, dc_FR;
wire [7:0] clean_dF, clean_dS1, clean_dS2, clean_dB;

assign display1 = clean_dF;
assign display2 = clean_dS1;
assign display3 = clean_dS2;
assign display4 = clean_dB;

testFSM FSM1(.distF(distF), .distS1(distS1), .distS2(distS2), .distB(distB),
                                    .clk(clk_50mhz), .reset(reset),
.reading_ready(reading_ready),
                                    .dc_BL(dc_BL), .dc_BR(dc_BR), .dc_FL(dc_FL), .dc_FR(dc_FR),
                                    .st(Led[3:0]),
                                    .clean_dF(clean_dF), .clean_dS1(clean_dS1),
.clean_dS2(clean_dS2), .clean_dB(clean_dB));

pwm_gen right_front(.clk(clkmotor), .duty_cycle(dc_FR), .reset(reset), .pwm(user2[4]));
pwm_gen left_front(.clk(clkmotor), .duty_cycle(dc_FL), .reset(reset), .pwm(user2[5]));
pwm_gen right_back(.clk(clkmotor), .duty_cycle(dc_BR), .reset(reset), .pwm(user2[6]));
pwm_gen left_back(.clk(clkmotor), .duty_cycle(dc_BL), .reset(reset), .pwm(user2[7]));

//pwm_gen right_back(.clk(clkmotor), .duty_cycle(10), .reset(reset), .pwm(user2[6]));
//pwm_gen left_back(.clk(clkmotor), .duty_cycle(0), .reset(reset), .pwm(user2[7]));

//assign user2[5:4] = 0;

endmodule
```

## adcRead.v

```
module adcRead(input reset, clock,
                                        input [7:0] value,
                                        output [7:0] v_out0, v_out1, v_out2, v_out3,
                                        output reg chip_read,
                                        output reg [1:0] chip_sel,
                                        output reading_ready);
        //clock will be 300Khz
        wire ready;
        reg [7:0] count = 0;
        wire [7:0] vout0, vout1, vout2, vout3;
        assign vout0[7:0] = ((chip_sel == 2'b00)&(~chip_read)) ? value[7:0]: vout0[7:0];
        assign vout1[7:0] = ((chip_sel == 2'b01)&(~chip_read)) ? value[7:0]: vout1[7:0];
        assign vout2[7:0] = ((chip_sel == 2'b10)&(~chip_read)) ? value[7:0]: vout2[7:0];
        assign vout3[7:0] = ((chip_sel == 2'b11)&(~chip_read)) ? value[7:0]: vout3[7:0];
        assign ready = ((chip_sel == 2'b11)&(~chip_read)) ? 1 : 0;

        wire rr0, rr1, rr2, rr3;

        rollingAve v0(.v_in(vout0), .clk(clock), .ready(ready),
                                                .v_out(v_out0), .reading_ready(rr0));

        rollingAve v1(.v_in(vout1), .clk(clock), .ready(ready),
                                                .v_out(v_out1), .reading_ready(rr1));

       rollingAve v2(.v_in(vout2), .clk(clock), .ready(ready),
                                                .v_out(v_out2), .reading_ready(rr2));

       rollingAve v3(.v_in(vout3), .clk(clock), .ready(ready),
                                                .v_out(v_out3), .reading_ready(rr3));

       assign reading_ready = rr0 & rr1 & rr2 & rr3;

       always @(posedge clock) begin
               if (reset) begin
                       count <= 8'd0;
                       chip_read <= 1;
                       chip_sel <= 2'b00;
               end
               else begin
                       if (count == 8'd100)
                               begin
                                       count <= count + 8'd1;
                                       chip_read <= 0;
                                       chip_sel <= 2'b00;
                               end
                       else if (count == 8'd101)
                               begin
                                       count <= count + 8'd1;
```

```
                        chip_read <= 0;
                        chip_sel <= 2'b01;
                    end
                else if (count == 8'd102)
                    begin
                        count <= count + 8'd1;
                        chip_read <= 0;
                        chip_sel <= 2'b10;
                    end
                else if (count == 8'd103)
                    begin
                        count <= 0;
                        chip_read <= 0;
                        chip_sel <= 2'b11;
                    end
                else
                    begin
                        count <= count + 8'd1;
                        chip_read <= 1;
                    end
            end
        end

endmodule
```

### Debouncer.v

```
module debouncer (
 input wire reset, clock, noisy,
 output reg clean
);
 reg [19:0] count;
 reg new;

 always @(posedge clock)
  if (reset) begin
   count <= 0;
   new <= noisy;
   clean <= noisy;
  end
  else if (noisy != new) begin
   // noisy input changed, restart the .01 sec clock
   new <= noisy;
   count <= 0;
  end
  else if (count == 500000)
   // noisy input stable for .01 secs, pass it along!
   clean <= new;
  else
   // waiting for .01 sec to pass
   count <= count+1;
endmodule
```

## DisplaySeg.v

```
module displaySeg(input [7:0] display1, display2, display3, display4,
                                        input clk, input reset,
                                        output [6:0] seg, output reg [3:0] digit);

    reg [1:0] count = 2'd0;
    reg [3:0] switch = 2'd0;
    reg [6:0] temp = 7'd0;
    assign seg = temp;

    always @(posedge clk) begin
        count <= count + 2'd1;

        case (count[1:0])
            2'b00: begin
                                        switch[3:0] = display1[7:4];
                                        digit[3:0] = 4'b1110;
                    end
            2'b01: begin
                                        switch[3:0] = display2[7:4];
                                        digit[3:0] = 4'b1101;
                    end
            2'b10: begin
                                        switch[3:0] = display3[7:4];
                                        digit[3:0] = 4'b1011;
                    end
            2'b11: begin
                                        switch[3:0] = display4[7:4];
                                        digit[3:0] = 4'b0111;
                    end
        endcase

        case (switch[3:0])
            4'b0000: temp[6:0] = 7'b1000000; //0
            4'b0001: temp[6:0] = 7'b1111001; //1
            4'b0010: temp[6:0] = 7'b0100100; //2
            4'b0011: temp[6:0] = 7'b0110000; //3

            4'b0100: temp[6:0] = 7'b0011001; //4
            4'b0101: temp[6:0] = 7'b0010010; //5
            4'b0110: temp[6:0] = 7'b0000010; //6
            4'b0111: temp[6:0] = 7'b1111000; //7

            4'b1000: temp[6:0] = 7'b0000000; //8
            4'b1001: temp[6:0] = 7'b0010000; //9
            4'b1010: temp[6:0] = 7'b000_1000;    //a
            4'b1011: temp[6:0] = 7'b000_0011;    //b
```

```
                4'b1100: temp[6:0] = 7'b100_0111;    //c
                4'b1101: temp[6:0] = 7'b100_0001;    //d
                4'b1110: temp[6:0] = 7'b000_0110;    //e
                4'b1111: temp[6:0] = 7'b000_1110;    //f

                default: temp[6:0] = 7'b111_1111;    //blank
            endcase
        end

endmodule
```

### Display_tb.v

```
module display_tb;
	// Inputs
	reg [7:0] display1;
	reg [7:0] display2;
	reg [7:0] display3;
	reg [7:0] display4;
	reg clk;
	reg reset;
	// Outputs
	wire [6:0] seg;
	wire [3:0] digit;
	// Instantiate the Unit Under Test (UUT)
	displaySeg uut (
		.display1(display1),
		.display2(display2),
		.display3(display3),
		.display4(display4),
		.clk(clk),
		.reset(reset),
		.seg(seg),
		.digit(digit)
	);
	always #1 clk = ~clk;
	initial begin
		// Initialize Inputs
		display1 = 0;
		display2 = 0;
		display3 = 0;
		display4 = 0;
		clk = 0;
		reset = 0;

		// Wait 100 ns for global reset to finish
		#100;
		reset = 1;
		#2;
		reset = 0;
		display1 = 8'd10;
		display2 = 8'd20;
		display3 = 8'd30;
		display4 = 8'd40;

		// Add stimulus here

	end
endmodule
```

## Divider.v

```
module divider #(parameter DELAY=27000000)   // 1 sec with a 27Mhz clock
                (input clock, reset, output reg one_hz_enable);

  reg [24:0] count;

  always @(posedge clock) begin
   if (reset) begin
          count <= 0;
          one_hz_enable <= 0;
           end
   else if (count == DELAY) begin
    one_hz_enable <= 1;
                      count <= 0;
           end
   else begin
    count <= count+1;
    one_hz_enable <= 0;
   end
          end

endmodule
```

## PWMgen.v

```
module pwm_gen(
  input clk,
  input duty_cycle,
  input reset,
        output reg pwm
  );

        // State parameters
        parameter HIGH = 1;
        parameter LOW          = 0;

        // Registers
        reg state = 1;
        reg high_count = duty_cycle;
        reg low_count = (10 - duty_cycle);

        always @ (posedge clk) begin
                case(state)
                        HIGH: begin
                                pwm <= HIGH;
                                if (count == high_count) begin
                                        count <= 0;
                                        state <= LOW;
                                end

                                else begin
                                        count = count + 1;
                                end
                        end

                        LOW: begin
                                pwm <= LOW;
                                if (count == low_count) begin
                                        count <= 0;
                                        state <= HIGH;
                                end

                                else begin
                                        count = count + 1;
                                end
                        end

                        default: state <= HIGH;
                endcase
        end
endmodule
```

## RollingAve.v

```
module rollingAve(input [7:0] v_in,
                                                input clk, input ready,
                                                output [7:0] v_out,
                                                output reg reading_ready);

        reg [7:0] window [7:0];
        wire [10:0] sum;
        wire [10:0] average;
        reg [2:0] offset = 3'd0;
        reg [2:0] k;
        //reg [2:0] index;

        initial
                begin
                        for (k = 0; k < 7; k = k + 1)
                                begin
                                        window[k] = 8'd0;
                                end
                        window[7] = 8'd0;
        end

        //wire [2:0] addr;

        //assign addr = offset+index;

        assign sum = window[0] + window[1] + window[2] + window[3] + window[4] +
window[5] + window[6] + window[7];
        assign average = sum/8;
        assign v_out = average[7:0];

        always @(posedge clk) begin
                 if (ready) begin
                        offset <= offset + 1;
                        window[offset] <= v_in;
                        reading_ready <= 1;
                        end
                 else
                        reading_ready <= 0;
        end
endmodule
```

## Average_tb.v

```
module average_tb;

	// Inputs
	reg [7:0] v_in;
	reg clk;
	reg ready;
	// Outputs
	wire [7:0] v_out;
	wire reading_ready;
	// Instantiate the Unit Under Test (UUT)
	rollingAve uut (
		.v_in(v_in),
		.clk(clk),
		.ready(ready),
		.v_out(v_out),
		.reading_ready(reading_ready)
	);
	always #1 clk = !clk;
	initial begin
		// Initialize Inputs
		v_in = 0;
		clk = 0;
		ready = 0;

		// Wait 100 ns for global reset to finish
		#100;
		v_in = 100;
		ready = 1;
		#2;
		ready = 0;
		#10;

		v_in = 200;
		ready = 1;
		#2;
    ready = 0;
		#10;

		v_in = 50;
		ready = 1;
		#2;
    ready = 0;
		#10;

		v_in = 150;
		ready = 1;
		#2;
    ready = 0;
```

```
                #10;

                v_in = 100;
                ready = 1;
                #2;
    ready = 0;
                #10;

                v_in = 130;
                ready = 1;
                #2;
    ready = 0;
                #10;

                v_in = 120;
                ready = 1;
                #2;
    ready = 0;
                #10;

                v_in = 150;
                ready = 1;
                #2;
    ready = 0;
                #10;

                v_in = 200;
                ready = 1;
                #2;
    ready = 0;
                #10;

                v_in = 30;
                ready = 1;
                #2;
    ready = 0;
                #10;

                v_in = 80;
                ready = 1;
                #2;
    ready = 0;
                #10;
                // Add stimulus here

        end
endmodule
```

## ClockMaker.v

```
module clockMaker #(parameter DIV = 2)       //choose what to div 27mhz clock by

                    //default if 27mhz/2
                                                (input clock, reset, output div_clk);

                    //wire real_div[7:0];
                    //assign real_div = DIV/2;
                    reg d_clk = 0;
                    assign div_clk = d_clk;
                    wire new_clk_en;
                    divider #(.DELAY(DIV/2)) new_clk(.clock(clock), .reset(reset),
.one_hz_enable(new_clk_en));
                    always @(posedge new_clk_en) begin
                              d_clk <= ~d_clk;
                    end

endmodule
```

## TestFSM.v

```
module testFSM(input [7:0] distF, distS1, distS2, distB,
                                        input clk, start, reading_ready,
                                        output reg [3:0] dc_BL, dc_BR, dc_FL, dc_FR,
                                        output [6:0] st,
                                        output [7:0] clean_dF, clean_dS1, clean_dS2,
clean_dB);

        wire [1:0] front_state, side1_state, side2_state, back_state;
        wire [7:0] clean_distF, clean_distS1, clean_distS2, clean_distB;
        reg [6:0] state = 9; //9 = prestart
        reg start1 = 0;
        reg start2 = 0;

        thresholder fr_st(.distance(distF), .clk(clk), .state(front_state),
.clean_d(clean_distF));
        thresholder s1_st(.distance(distS1), .clk(clk), .state(side1_state),
.clean_d(clean_distS1));
        thresholder s2_st(.distance(distS2), .clk(clk), .state(side2_state),
.clean_d(clean_distS2));
        thresholder bk_st(.distance(distB), .clk(clk), .state(back_state),
.clean_d(clean_distB));

        parameter START = 0;
        parameter MIDDLE = 1;
        //parameter END = 2;
        parameter BACKUP = 2;
        parameter TURNIN = 3;
        parameter BACKIN = 4;
        parameter CORRECT = 5;
        parameter STRAIGHT = 6;
        parameter WIGGLE = 7;
        parameter END = 8;
        parameter PRESTART1 = 9;
        parameter PRESTART2 = 10;

        assign st = state;
        assign clean_dF = clean_distF;
        assign clean_dS1 = clean_distS1;
        assign clean_dS2 = clean_distS2;
        assign clean_dB = clean_distB;

        reg [26:0] count = 0;
        reg startF = 0;
        reg [7:0] temp = 0;

        always @(posedge clk) begin
                        //wait 0.1 seconds before starting FSM
                        if (start) begin
```

```
                startF <= ~startF;
                start1 <= 0;
                start2 <= 0;
                state <= PRESTART1;
                dc_BL <= 4'd0;
                dc_BR <= 4'd0;
                dc_FL <= 4'd0;
                dc_FR <= 4'd0;
        end

        if (reading_ready & startF) start1 <= 1;

        if (start1) count <= count + 1;

        if (count == 50000000) start2 <= 1;

        if (start2) begin
                case(state)
                        PRESTART1: begin
                                dc_BL <= 4'd7;
                                dc_BR <= 4'd7;
                                dc_FL <= 4'd0;
                                dc_FR <= 4'd0;
                                if (clean_distS1 >= 8'h50) begin
                                        state <= PRESTART2;
                                end
                        end

                        PRESTART2: begin
                                dc_BL <= 4'd1;
                                dc_BR <= 4'd10;
                                dc_FL <= 4'd0;
                                dc_FR <= 4'd0;
                                if (clean_distS1 == clean_distS2) begin
                                        state <= START;
                                end
                        end

                        START: begin
                                if (side1_state < 2 || side2_state < 2) begin
                                        dc_BL <= 4'd7;
                                        dc_BR <= 4'd7;
                                        dc_FL <= 4'd0;
                                        dc_FR <= 4'd0;
                                        state <= START;
                                    end
                                else if (side1_state == 2 || side2_state == 2)
begin
                                        dc_BL <= 4'd7;
                                        dc_BR <= 4'd7;
```

```
                        dc_FL <= 4'd0;
                        dc_FR <= 4'd0;
                        state <= MIDDLE;
                    end
            end

            MIDDLE: begin
                if (side1_state == 2 || side2_state == 2) begin
                        dc_BL <= 4'd7;
                        dc_BR <= 4'd7;
                        dc_FL <= 4'd0;
                        dc_FR <= 4'd0;
                        state <= MIDDLE;
                    end

                else if (side1_state < 2 && side2_state <2)
begin
                        dc_BL <= 4'd7;
                        dc_BR <= 4'd7;
                        dc_FL <= 4'd0;
                        dc_FR <= 4'd0;
                        state <= BACKUP;
                    end
            end

            BACKUP: begin
                if (side1_state < 2 && side2_state <2) begin
                        dc_BL <= 4'd0;
                        dc_BR <= 4'd0;
                        dc_FL <= 4'd10;
                        dc_FR <= 4'd10;
                        count <= 0;
                    end
                    else if (side2_state == 2) begin
                        dc_BL <= 4'd0;
                        dc_BR <= 4'd0;
                        dc_FL <= 4'd10;
                        dc_FR <= 4'd10;
                        temp <= clean_distS1 - 8'd28;
                        if (count == 1000000) state <=
TURNIN;

                    end
            end

            /*TURNIN: begin
                if (back_state > 0) begin
                        dc_BL <= 4'd0;
                        dc_BR <= 4'd4;
                        dc_FL <= 4'd10;
                        dc_FR <= 4'd1;
```

```
                end

                else if (back_state == 0) begin
                        dc_BL <= 4'd0;
                        dc_BR <= 4'd0;
                        dc_FL <= 4'd0;
                        dc_FR <= 4'd0;
                        state <= END;
                end
        end*/

        TURNIN: begin
                //if ((clean_distS1-clean_distS2 > 8'h28) &
(clean_distS2 - clean_distS1 > 8'h28)) begin
                if ((clean_distS1 > temp)) begin
                        dc_BL <= 4'd0;
                        dc_BR <= 4'd1;
                        dc_FL <= 4'd10;
                        dc_FR <= 4'd2;
                end

                else begin
                        dc_BL <= 4'd0;
                        dc_BR <= 4'd0;
                        dc_FL <= 4'd0;
                        dc_FR <= 4'd0;
                        state <= BACKIN;
                end
        end

        BACKIN: begin
                if ((clean_distB < 8'h60) & (side2_state > 0))
begin
                        dc_BL <= 4'd0;
                        dc_BR <= 4'd0;
                        dc_FL <= 4'd2;
                        dc_FR <= 4'd10;
                end

                else begin
                        dc_BL <= 4'd0;
                        dc_BR <= 4'd0;
                        dc_FL <= 4'd0;
                        dc_FR <= 4'd0;
                        state <= CORRECT;
                end
        end

        CORRECT: begin
```

```
                        //if ((clean_distS1-clean_distS2 > 8'h04) &
(clean_distS2 - clean_distS1 > 8'h04)) begin
                        if (back_state > 0) begin
                                dc_BL <= 4'd0;
                                dc_BR <= 4'd0;
                                dc_FL <= 4'd1;
                                dc_FR <= 4'd10;
                        end

                        else begin
                                dc_BL <= 4'd0;
                                dc_BR <= 4'd0;
                                dc_FL <= 4'd0;
                                dc_FR <= 4'd0;
                                state <= STRAIGHT;
                        end
                end

                STRAIGHT: begin
                        //if ((clean_distS1-clean_distS2 > 8'h01) &
(clean_distS2 - clean_distS1 > 8'h01)) begin
                        //if ((clean_distS1 < 8'h40) | (clean_distS2 <
8'h40)) begin
                        if ((clean_distS1-clean_distS2 > 8'h04) &
(clean_distS2 - clean_distS1 > 8'h01)) begin
                                dc_BL <= 4'd10;
                                dc_BR <= 4'd0;
                                dc_FL <= 4'd0;
                                dc_FR <= 4'd8;
                        end

                        else begin
                                dc_BL <= 4'd0;
                                dc_BR <= 4'd0;
                                dc_FL <= 4'd0;
                                dc_FR <= 4'd0;
                                state <= END;
                        end
                end

                WIGGLE: begin
                        if (back_state == 0) begin
                                if (clean_distS1 > clean_distS2) begin
                                        dc_BL <= 4'd1;
                                        dc_BR <=
4'd10;//(clean_distS1 - clean_distS2)*3;
                                        dc_FL <= 4'd1;
                                        dc_FR <= 4'd0;
                                end
                                else begin
```

```
                                        dc_BL <= 4'd10;
                                        dc_BR <= 4'd1;
                                        dc_FL <= 4'd0;
                                        dc_FR <= 4'd1;
                                end
                        end

                        else if (front_state == 0) begin
                                if (clean_distS1 > clean_distS2) begin
                                        dc_FL <= 4'd9;
                                        dc_FR <= 4'd1;
                                        dc_BL <= 4'd0;
                                        dc_BR <= 4'd0;
                                end
                                else begin
                                        dc_FL <= 4'd1;
                                        dc_FR <= 4'd9; //(clean_distS2
- clean_distS1)*3;

                                        dc_BL <= 4'd0;
                                        dc_BR <= 4'd0;
                                end
                        end

                        else if ((clean_distS1 > 8'ha0) | (clean_distS2
> 8'ha0)) begin
                                        dc_BL <= 4'd0;
                                        dc_BR <= 4'd0;
                                        dc_FL <= 4'd0;
                                        dc_FR <= 4'd0;
                                        //state <= END;
                        end

                        else begin
                                        dc_BL <= dc_BL;
                                        dc_BR <= dc_BR;
                                        dc_FL <= dc_FL;
                                        dc_FR <= dc_FR;
                        end
                end

                END: begin
                        if (front_state == 0) begin
                                        dc_BL <= 4'd0;
                                        dc_BR <= 4'd0;
                                        dc_FL <= 4'd0;
                                        dc_FR <= 4'd0;
                        end
                        else begin
                                        dc_BL <= 4'd10;
                                        dc_BR <= 4'd2;
```

```
                                        dc_FL <= 4'd0;
                                        dc_FR <= 4'd0;
                                    end
                                end

                                default: begin
                                        dc_BL <= 4'd0;
                                        dc_BR <= 4'd0;
                                        dc_FL <= 4'd0;
                                        dc_FR <= 4'd0;
                                    end
                            endcase
                        end


                end
            endmodule
```

### Thresholder.v

```
module thresholder(input [7:0] distance,
                                                input clk,
                                                output [1:0] state,
                                                output [7:0] clean_d);

                //0 = close, 1 = medium, 2 = far, 3 = error
                parameter T1 = 8'hA0;
                parameter T2 = 8'h54;

                reg [1:0] st = 2'd3;
                reg [22:0] count;
                reg [7:0] clean_dist;
                reg [7:0] new;

                assign state = st;
                assign clean_d = clean_dist;

                always @(posedge clk) begin
                        if (new[7:4] != distance[7:4]) begin
                                new <= distance;
                                count <= 0;
                        end
                        else if (count == 2000000)
                                clean_dist <= new;
                        else
                                count <= count +1;


                        if (clean_dist >= T1) st <= 2'd0;
                        else if ((clean_dist < T1) && (clean_dist >= T2)) st <= 2'd1;
                        else st <= 2'd2;
                end
endmodule
```

## Labkit.ucf

```
# This file is a general .ucf for Basys2 rev C board
# To use it in a project:
# - remove or comment the lines corresponding to unused pins
# - rename the used signals according to the project

# clock pin for Basys2 Board
NET "clk_50mhz" LOC = "B8"; # Bank = 0, Signal name = MCLK
NET "clk_50mhz" CLOCK_DEDICATED_ROUTE = FALSE;



# Pin assignment for DispCtl
# Connected to Basys2 onBoard 7seg display
NET "seg<0>" LOC = "L14"; # Bank = 1, Signal name = CA
NET "seg<1>" LOC = "H12"; # Bank = 1, Signal name = CB
NET "seg<2>" LOC = "N14"; # Bank = 1, Signal name = CC
NET "seg<3>" LOC = "N11"; # Bank = 2, Signal name = CD
NET "seg<4>" LOC = "P12"; # Bank = 2, Signal name = CE
NET "seg<5>" LOC = "L13"; # Bank = 1, Signal name = CF
NET "seg<6>" LOC = "M12"; # Bank = 1, Signal name = CG
NET "dp" LOC = "N13"; # Bank = 1, Signal name = DP


#turns on each digit
NET 'digit<3>' LOC = 'F12';
NET 'digit<2>' LOC = 'J12';
NET 'digit<1>' LOC = 'M13';
NET 'digit<0>' LOC = 'K14';

# Pin assignment for LEDs
NET "Led<7>" LOC = "G1" ; # Bank = 3, Signal name = LD7
NET "Led<6>" LOC = "P4" ; # Bank = 2, Signal name = LD6
NET "Led<5>" LOC = "N4" ;  # Bank = 2, Signal name = LD5
NET "Led<4>" LOC = "N5" ;  # Bank = 2, Signal name = LD4
NET "Led<3>" LOC = "P6" ; # Bank = 2, Signal name = LD3
NET "Led<2>" LOC = "P7" ; # Bank = 3, Signal name = LD2
NET "Led<1>" LOC = "M11" ; # Bank = 2, Signal name = LD1
NET "Led<0>" LOC = "M5" ;  # Bank = 2, Signal name = LD0

# Pin assignment for SWs
NET "sw<7>" LOC = "N3";  # Bank = 2, Signal name = SW7
NET "sw<6>" LOC = "E2";  # Bank = 3, Signal name = SW6
NET "sw<5>" LOC = "F3";  # Bank = 3, Signal name = SW5
NET "sw<4>" LOC = "G3";  # Bank = 3, Signal name = SW4
NET "sw<3>" LOC = "B4";  # Bank = 3, Signal name = SW3
```

```
NET "sw<2>" LOC = "K3";  # Bank = 3, Signal name = SW2
NET "sw<1>" LOC = "L3";  # Bank = 3, Signal name = SW1
NET "sw<0>" LOC = "P11";  # Bank = 2, Signal name = SW0

NET "btn<3>" LOC = "A7";  # Bank = 1, Signal name = BTN3
NET "btn<2>" LOC = "M4";  # Bank = 0, Signal name = BTN2
NET "btn<1>" LOC = "C11"; # Bank = 2, Signal name = BTN1
NET "btn<0>" LOC = "G12"; # Bank = 0, Signal name = BTN0


#IO Ports:

NET "user1<0>" LOC = "B2"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA1
NET "user1<1>" LOC = "A3"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA2
NET "user1<2>" LOC = "J3"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA3
NET "user1<3>" LOC = "B5"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA4

NET "user1<4>" LOC = "C6"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB1
NET "user1<5>" LOC = "B6"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB2
NET "user1<6>" LOC = "C5"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB3
NET "user1<7>" LOC = "B7"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB4

NET "user2<0>" LOC = "A9"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC1
NET "user2<1>" LOC = "B9"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC2
NET "user2<2>" LOC = "A10" | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC3
NET "user2<3>" LOC = "C9"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC4

NET "user2<4>" LOC = "C12"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JD1
NET "user2<5>" LOC = "A13"  | DRIVE = 2  | PULLUP ; # Bank = 2, Signal name = JD2
NET "user2<6>" LOC = "C13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JD3
NET "user2<7>" LOC = "D12"  | DRIVE = 2  | PULLUP ; # Bank = 2, Signal name = JD4

##################################################################
###
#UNUSED STUFF:
##################################################################
###

# Pin assignment for EppCtl
# Connected to Basys2 onBoard USB controller
#NET "EppAstb" LOC = "F2"; # Bank = 3
#NET "EppDstb" LOC = "F1"; # Bank = 3
#NET "EppWR"      LOC = "C2"; # Bank = 3

#NET "EppWait" LOC = "D2"; # Bank = 3


# NET "EppDB<0>" LOC = "N2"; # Bank = 2
# NET "EppDB<1>" LOC = "M2"; # Bank = 2
# NET "EppDB<2>" LOC = "M1"; # Bank = 3
```

```
# NET "EppDB<3>" LOC = "L1"; # Bank = 3
# NET "EppDB<4>" LOC = "L2"; # Bank = 3
# NET "EppDB<5>" LOC = "H2"; # Bank = 3
# NET "EppDB<6>" LOC = "H1"; # Bank = 3
# NET "EppDB<7>" LOC = "H3"; # Bank = 3

# Loop Back only tested signals
# NET "PIO<72>" LOC = "B2"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA1
# NET "PIO<73>" LOC = "A3"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA2
# NET "PIO<74>" LOC = "J3"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA3
# NET "PIO<75>" LOC = "B5"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA4
#
# NET "PIO<76>" LOC = "C6"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB1
# NET "PIO<77>" LOC = "B6"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB2
# NET "PIO<78>" LOC = "C5"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB3
# NET "PIO<79>" LOC = "B7"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB4
#
# NET "PIO<80>" LOC = "A9"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC1
# NET "PIO<81>" LOC = "B9"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC2
# NET "PIO<82>" LOC = "A10" | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC3
# NET "PIO<83>" LOC = "C9"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC4
#
# NET "PIO<84>" LOC = "C12"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JD1
# NET "PIO<85>" LOC = "A13"  | DRIVE = 2  | PULLUP ; # Bank = 2, Signal name = JD2
# NET "PIO<86>" LOC = "C13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JD3
# NET "PIO<87>" LOC = "D12"  | DRIVE = 2  | PULLUP ; # Bank = 2, Signal name = JD4


# Loop back/demo signals
# Pin assignment for PS2
# PS2 not used in 6.111 lab
# NET "PS2C"    LOC = "B1"   | DRIVE = 2  | PULLUP ; # Bank = 3, Signal name = PS2C
# NET "PS2D"    LOC = "C3"   | DRIVE = 2  | PULLUP ; # Bank = 3, Signal name = PS2D

# Pin assignment for VGA
# VGA output not used in 6.111 lab
# NET "HSYNC"   LOC = "J14"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = HSYNC
# NET "VSYNC"   LOC = "K13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = VSYNC
#
# NET "OutRed<2>"  LOC = "F13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = RED2
# NET "OutRed<1>"  LOC = "D13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = RED1
# NET "OutRed<0>"  LOC = "C14"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = RED0
# NET "OutGreen<2>"  LOC = "G14"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name =
GRN2
# NET "OutGreen<1>"  LOC = "G13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name =
GRN1
# NET "OutGreen<0>"  LOC = "F14"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = GRN0
# NET "OutBlue<2>"  LOC = "J13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = BLU2
# NET "OutBlue<1>"  LOC = "H13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = BLU1
```

```
#NET "an<3>" LOC = "K14"; # Bank = 1, Signal name = AN3
#NET "an<2>" LOC = "M13"; # Bank = 1, Signal name = AN2
#NET "an<1>" LOC = "J12"; # Bank = 1, Signal name = AN1
#NET "an<0>" LOC = "F12"; # Bank = 1, Signal name = AN0

#NET "uclk" LOC = "M6"; # Bank = 2, Signal name = UCLK
#NET "uclk" CLOCK_DEDICATED_ROUTE = FALSE;
```