5/23/2023

Self-Parking RC

Verilog Design Report

Przekop, Matt

BOSTON UNIVERSITY

Abstract

The driving factor of consumer action is simplicity and efficiency. The goal of my engineering design is to make what was once a complicated task into a simplistic and automated process. So, for my project I aim to create a car that will make the parallel parking maneuver a computer-controlled function rather than a task for the driver. The car will use four IR sensors as input to feed data into an on-board FPGA that will power the car's movement using a finite state machine. From the moment the user decides to park to the point where the car is in position the FPGA will be in complete control.

Table Of Contents

1. Overview

To simplify and quicken the process of parallel parking on the road the goal of the system and project is to enable a car to park itself. However, it would be too costly and risky to incorporate the system on a real car so instead a four-wheel drive RC car will be used as a supplement to the car. The parking space will be a predefined spot and the car will pull up to the spot before parking. The car will also use four IR sensors to measure the distance to the spot and a finite state machine to maneuver the car into place.

The project aims to demonstrate the capabilities of an FPGA to use precise measurements and calculations to perform real world tasks. The plan is to use a mix of battery and lab bench power sources to provide enough energy input to power the system and to maintain the self-contained integrity of the project with the mounted Basys2 FPGA.

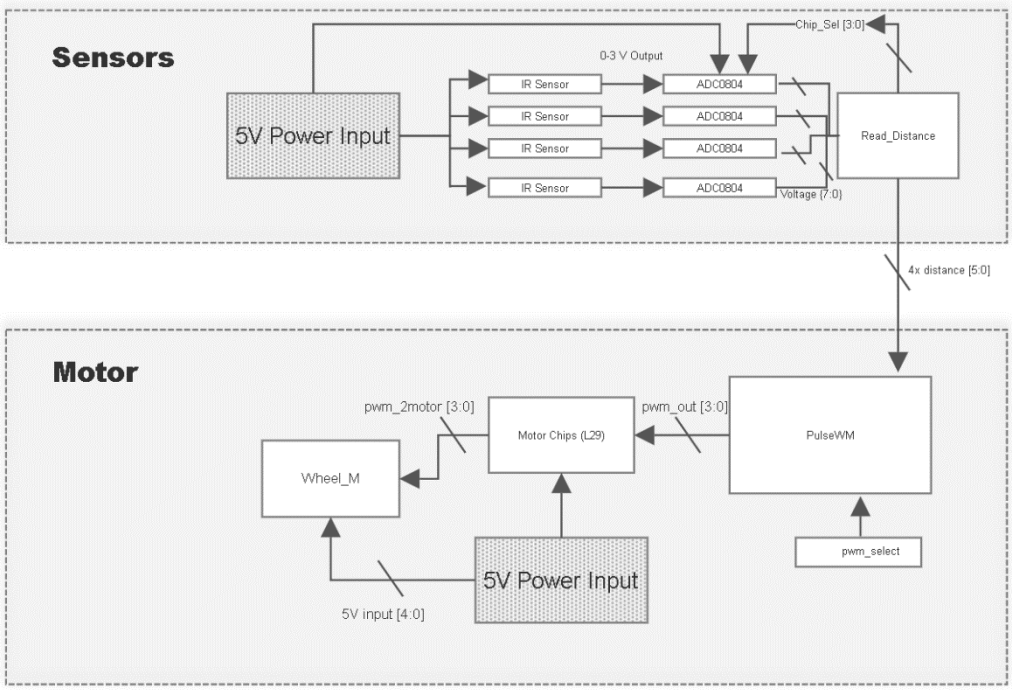

Figure 1: Schematic of System

2. Input Components

The sensing block in the schematic will be driven by the IR sensors that I had leftover from a previous project and a 5-volt battery sourced from either a lab bench or a on board battery pack.

2.1 IR Sensors

The sensors will output an analog signal so it is important to convert this to a digital signal so that the Basys-2 can register the readings. The output, however, will be received as the inverse of the distance proportional to the readings so it may be necessary to account for this when implementing the finite state machine. The IR sensors are accurate up to 30 cm and as close as 3 cm based on the datasheet. This parameter is fine for accuracy required when running testing. These sensors will be mounted around the frame of the car and will be soldered to wires that connect to the inputs of the ADC's which will be used to convert the output readings.

2.2 ADC's

These chips are used to convert the analog output of the IR sensors to a digital input for the Basys-2. The main issue with the chips is that they require eight lines of data, and with four IR sensors each requiring a chip for conversion, the system will need 32 ports on the FPGA for use. But the Basys-2 only has 16. Instead of ordering a new FPGA a solution is to use chip enable pins and read them sequentially rather than in parallel. The chip will be connected to ground so that the chip is continuously reading the data; however, the output will be in tri-state when the input of the chip enable pin is high, which allows only one chip to produce input for the

FPGA. Because of this the outputs of the chips will be connected and inputted into the FPGA as a bus and a MUX will be used to select the input line (control pins). This also increases efficiency as the read lines of each chip would total four ports but utilizing a MUX will save us a pin as it now only requires two pins for selection and another to send the correct read pulses. To make these pulses an embedded clock is functionalized by using a 10k resistor and a 150-pF capacitor – creating a 600 kHz clock for the ADC's.

2.3 Read_Distance

The ADCs require a certain amount of time to convert the analog signal to digital. Because of this a Read_Distance module is created to both allow the necessary conversion time for the ADC's and create a ready signal to the next module that the ADC's are done collecting. In addition to this the module will also filter some noise that the sensitive IR sensors may pick up.

At 600 kHz the ADC data sheet says the chips require 114 microseconds for conversion. Using a clock divider, a 300 kHz clock is generated. The module will count in increments of 100 and set the chip enable pins to low four times to hit every ADC. Using a 100-cycle count at 300 kHz is around 300 microseconds which is more than double the amount of the time the ADC's need to convert the signal. The extent of the overestimation is used so that in the event the battery performance affects the ability of the chips to convert the system failure won't be noticed as early and more tests can be run on each charge. This module will the send another signal to let a different module know to begin filtering through the signals/

The filter module will account for the spikes in the readings of the IR sensors. In order to do this, the module computes a rolling average with a window of 8 to filter out the noise. The

module will take an average of the readings within this window and output this average value. After every average, the module will shift the window by one and compute a new average if new data is available. To compute the average reading of the sensors more accurately, the module will only accept the data if the sensor reading is stable for 0.05 seconds. This will remove any sudden jumps due to voltage spikes or noise. When the readings are done being filtered they will be inputted into the next module.

2.4 Distance Calibration

The distance reading outputs by the chips are the inverse of the distances that are read by the IR sensors. Reinverting these readings increases the inaccuracy of the system, however, so the readings will be directly inputted into the FSM instead of inverting them first. It is also important to note the pulse with modulation used in deciding the power output that will be delivered to the motors. Using the average of an off and on state of a power source, specific voltages can be simulated from a single voltage battery source.

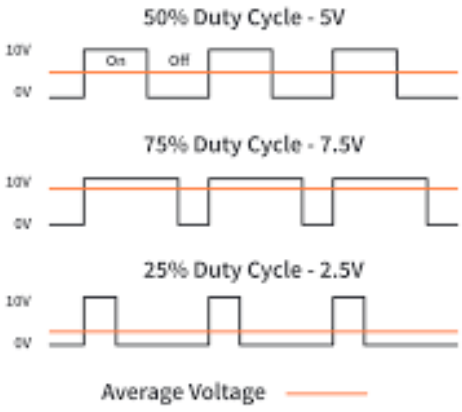

Figure 2: Example Diagram of PWM

In this project, the module PulseWM is used to generate the varying power outputs of the Read_Distance module. The FPGA used has a built-in clock of 50 MHz but the L293 motor

chips require a frequency of 5 kHz, so a clock divider is again implemented to achieve the desired frequency. The PWM will be implemented by using a state machine with inputs from 0-10 as we are using a 5-volt battery. The input number will be double whatever the desired voltage is and the clock will output high for the amount of time that this doubled voltage value is. It will then output low for the amount of time that 10 minus the doubled voltage value is. This will provide the correct voltage output for the motors.

3. Motors and Vehicle Construction

3.1 Vehicle Body

The vehicle's body is a frame purchased from DFRobot and is a four wheel drive Arduino mobile platform. This was used as it as a wide body that is large enough to support multiple bread boards and the FPGA needed in order to have a system that is fully independent from any exterior support. The chassis is a good emulation of the type of performance we see in an actual car as it has a rack and pinion style of steering. The alternative of which is a vehicle that can turn in place.

Figure 3: Vehicle Chassis

3.2 L293 Chips

The L293 chips are used to drive the vehicle. It is a quadruple high-current half-H driver used to drive DC motors. It is designed to drive bidirectional currents up to 1 amp and voltages ranging from 4.5 to 36 volts. The chips can read data for a specific amount of time before it discards and updates. This is very useful in our system as we need the motors to constantly change speed and direction depending on the input readings from the IR sensors.

We use the readings from the PulseWM module as an input into these chips. Based on the limited availability of I/O ports on the Basys2, the motors would only be driven in one direction as opposed to their capability of two. This is done by attaching one input of the motor to the PulseWM module and the other to ground. This means that two wheels will be able to spin forward, and two wheels will be able to spin backwards. Next, the L293 takes four inputs and supplies the motors with a driving signal, meaning it can independently control each motor and drive it at a different speed based on PulseWM.

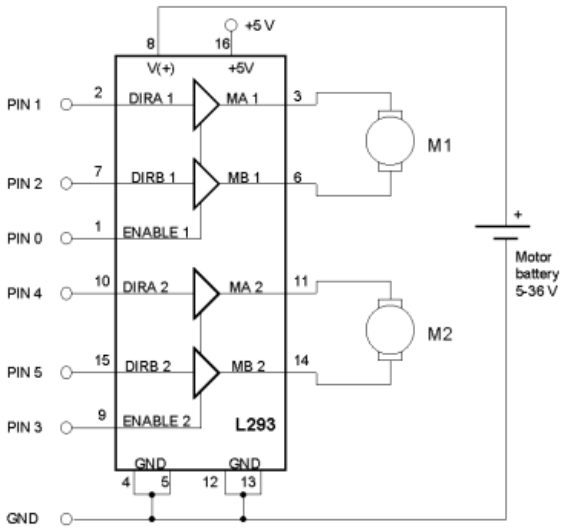

Figure 4: Schematic of L293 Motor Chip

4. Finite State Machine

The Finite State Machine can be divided into smaller steps that outline the entire parking system of the RC car. The module takes the four IR sensor readings, sifts through the noise, and outputs a duty cycle for each of the wheels. In order to begin the system waits for the user to push the start / reset button.

4.1 Steps

1a) The car will is placed in a position away from the parking spot in a diagonal or straight state simulating an actual driver pulling up to a parking spot. If the sensors determine that the car is close enough to the spot already the FSM will go to step 1b.

1b) In this step, the car will position itself parallel to the row of "parked cars" so the parallel parking sequence can begin. The vehicle will shift left until both sensors on the right side of the vehicle read the same distance to the car adjacent to it.

2) Once the car is parallel to the adjacent car, it will drive forward and the sensors will look for a space that can fit. Once a suitable size is fit (i.e. both distance readings return a high value) the FSM transitions to the next step.

3) In this state the car will pull up to the car in front of it. It will know when it has done this as the sensors will both drive a low (close) reading.

4) Next, the car will back up for a specified time and a reading will be taken by the side sensor. This will be used to calculate the angle at which the car will need to turn into the spot with.

5) The car now turns into the spot by moving the left wheels faster than the right side wheels in a reverse direction. If the car becomes too far from the car it was previously adjacent to, it will transition back into the next state.

6) Now the car will back into the spot with its right side wheels moving faster than its left side wheels in order to straighten out into the spot. Once the back sensor crosses a threshold of being to close the machine will move onto the next state.

7) The car will now wiggle itself using the left and right side wheels at various speeds until the side sensors read the same input from the curb.

8) Now the car will drive forward until it is a certain distance from the car in front of it. And the car will then stop and be parked.

5. Verilog

5.1 FSMrun.v

```
module FSMrun(clk50, seg, dp, digit, sw, btn, user0);

output dp;
output [3:0] digit;
output [6:0] seg;
input clk50;
input [3:0] btn;
input [7:0] sw, user0;
assign dp = 1'b1;
assign user[3] = 1'hz;
wire reset, clk600, clkseg, clkmotor;

debouncer btn_r(.reset(0), .clock(clk50), .noisy(btn[0]), .clean(reset));
IniClk #(.DIV(180)) clk600(.clock(clk50), .reset(reset), .divclk(clk600));
IniClk #(.DIV(10000)) clkseg(.clock(clk50), .reset(reset), .divclk(clkseg));
```

```
IniClk #(.DIV(100000)) clkmotor,(.clock(clk50), .reset(reset), .divclk(clkmotor));

wire [7:0] dist0, dist1, dist2, dist3;
wire readyread;

IRRead   IR0 (.reset(reset), .clock(clk600), .value(user0[7:0], .vout0(dist0), .vout1(dist1), .vout2(dist2), .vout3(dist3),
chipread(user0[0]), .chipsel(user0[2:1]), .readyread(readyread));

wire [3:0] wheelBL, wheelBR, wheelFL, wheelFR;
wire [7:0] cleanD0, cleanD1, cleanD2, cleanD3;

testFSM FSM0(.distF(dist0), .dist1(dist1), .dist2(dist2), .distB(dist3), .clk(clk50), .reset(reset), .readyread(readyread),
.wheelBL(wheelBL), .wheelBR(wheelBR), .wheelFL(wheelFL), .wheelFR(wheelFR), .cleanD0(cleanD0),
.cleanD1(cleanD1), .cleanD2(cleanD2), .cleanD3(cleanD3));

PulseWM      RF(.clk(clkmotor), .duty_cycle(wheelFR),    .reset(reset),    .pwm(user0[4]));
PulseWM      LF(.clk(clkmotor), .duty_cycle(wheelFL),    .reset(reset),    .pwm(user0[5]));
PulseWM      RB(.clk(clkmotor),.duty_cycle(wheelBR),     .reset(reset),    .pwm(user0[6]));
PulseWM      LB(.clk(clkmotor), .duty_cycle(wheelBL),    .reset(reset),    .pwm(user0[7]));

endmodule
```

5.2 Debounce.v

```
module Debounce(reset, clock, noisy, clean);
        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
                //if noisy input changes then reset count and new is noisy
                else if(noisy != new)
```

```
                begin
                new <= noisy;
                count <= 0;
                end
        //signals a steady input from sensors
        else if (count==500000)
                clean <= new;
        //waiting 0.01 seconds
        else
                count <= count+1;
endmodule
```

5.3 clkDivider.v

```
module clkDivider #(paramter DELAY=27000000) (clock, reset, one_hz_enable);
        output reg one_hz_enable;
        input clock, reset;
        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
```

5.4 IRRead.v

```
module IRRead(input reset, clock, input [7:0] value, output [7:0] vout0, vout1,vout2, vout3, output reg chipread,
output reg[1:0]chipsel, output readyread);

wire ready;
reg [7:0] vouta, voutb, voutc, voutd;
assign vouta[7:0] = ((chipsel==2'b00)&(~chipread)) ?value[7:0]: vouta[7:0];
assign voutb[7:0] = ((chipsel==2'1)&(~chipread)) ?value[7:0]: voutb[7:0];
assign voutc[7:0] = ((chipsel==2'b10)&(~chipread)) ?value[7:0]: voutc[7:0];
assign voutd[7:0] = ((chipsel==2'b11)&(~chipread)) ?value[7:0]: voutd[7:0];

wire r0, r1, r2, r3;

rollAvg A0(.vin(vouta), .clk(clock), .ready(ready), .vout(vout0), .readyread(r0));

rollAvg A1(.vin(voutb), .clk(clock), .ready(ready), .vout(vout1), .readyread(r1));

rollAvg A2(.vin(voutc), .clk(clock), .ready(ready), .vout(vout2), .readyread(r2));

rollAvg A3(.vin(voutd), .clk(clock), .ready(ready), .vout(vout3), .readyread(r3));

assign readyread = (r0&r1&r2&r3);

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

```
                chipsel <= 2'b00;
                end
        else if(count == 8'd101)
                begin
                count <= count + 8'd1;
                chipread <= 0;
                chipsel <= 2'b01;
                end
        else if(count == 8'd 102)
                begin
                count <= count + 8'd1;
                chipread <= 0;
                chipsel <= 2'b10;
                end
        else if(count == 8'd103)
                begin
                count <= 0;
                chipread <= 0;
                chipsel <= 2'b11;
                end
        else
                begin
                count <= count +8'd1;
                chipread <= 1;
                end
        end
    end
endmodule
```

5.5 PulseWM.v

```
module PulseWM(clk, dutycycle, reset, pwm);

input clk, dutycycle, reset
output reg pwm;

parameter HIGH = 1;
parameter LOW = 0;
```

```
reg state = 1;
reg highc = dutycycle;
reg lowc = (10-dutycycle);

always @(posedge clk)
        begin
        case(state)
                HIGH: begin
                        pwm <= HIGH;
                        if(count == highc)
                                begin
                                count <= 0;
                                state <= LOW;
                                end
                        else
                                begin
                                count = count + 1;
                                end
                        end
                LOW: begin
                        pwm <= LOW;
                        if(count == lowc)
                                begin
                                count <= 0;
                                state <= HIGH;
                                end
                        else
                                begin
                                count = count + 1;
                                end
                        end
        endcase
        end
endmodule
```

5.6 rollingAve.v

```
module rollingAve(vin, clk, ready, vout, readyread);

output [7:0] vout;
output reg readyread;
input clk, ready;
input [7:0] vin;
reg [7:0] window [7:0];
wire [10:0] sum, average;
reg [2:0] offset = 3'd0;
reg [2:0] i;

initial
        begin
        for (i=0; i<7; i=i+1)
                begin
                window[i]=8'd0;
                end
        window[7]=8'd0;
end
assign sum = window[1] +  window[2] +  window[3] +  window[4] + window[5] + window[6] +  window[7];
assign average = sum/8;
assign vout = average[7:0];

always @(psedge clk)
        begin
        if(ready)
                begin
                offset<=offset+1;
                window[offset]<=vin;
                readyread <= 1;
                end
        else
                readyread <=0;
        end
endmodule
```

5.7 IniClk.v

```
module IniClk #(parameter DIV = 2) (clock, reset, divclk);
output divclk;
input clock, reset;

reg dividedclk = 0;
assign divclk = dividedclk;
wire newclk;
divider #(.DELAY(DIV/2)) newclk(.clock(clock), .reset(reset), .onehzenable(newclk));
always @(posedge newclk)
        begin
        dividedclk <= ~dividedclk;
        end
endmodule
```

5.8 thresholder.v

```
module thresholder(state, cleand, clk, distance);

//0 = close; 1 = medium; 2 = far; 3 = error;

parameter T1 = 8'hA0;
parameter T2 = 8'h54;

output [1:0] state;
output [7:0] cleand;

input clk;
input [7:0] distance;

reg [1:0] st = 2'd3;
reg [22:0] count;
reg [7:0] cleand1;
reg [7:0] new;

assign state = st;
assign cleand = cleand1;
```

```
always@(posedge clk);
        if(new[7:4] !=distance [7:4])
                begin
                new <= distance;
                count <= 0;
        end
        else if(count == 200000)
                cleand <= new;
        else
                count <= count+1;
        if(cleand >= T1)
                st <= 2'd0;
        else if((cleand<T1) && (cleand >= T2))
                st <= 2'd1;
        else
                st <= 2d'2;
        end
endmodule
```

5.9 IOpins.ucf

```
NET "clk_50mhz" LOC = "B8"; # Bank = 0, Signal name = MCLK
NET "clk_50mhz" CLOCK_DEDICATED_ROUTE = FALSE;
# Pin assignment for DispCtl

# 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
```