ECE 271, IR Controlled Robot, Group 16

Emily Becher, Thomas Johnson, Nathan Kiely

June 6, 2020

Contents

1 Project Description

Inputs: This design reads an infrared remote.
Outputs: The design controls to motors using pulse width modulation (PWM).

The project is to control a simple robot with an IR remote. An IR receiver will read the signal from the remote. The signal will then be read by the FPGA where the logic to control the two motors will be done. The FPGA has three main tasks. The first is two decode the signal from the IR receiver through an IR decoder block. Next the FPGA will determine what each motor should do. This will be accomplished by the main controller block. The final task is to turn the commands into a PWM signal that controls the speed of the motors. The motor controller block creates the PWM signal for both motors. This operation is shown in the block diagram in Figure 1. The robot will be controlled by six different buttons on the remote: go, stop, speed up, speed down, right, left. The robot is able to move at three different speeds. For a preview of the hardware demonstration please take a look at our video here (https://youtu.be/WcZzuia0Q2Y).

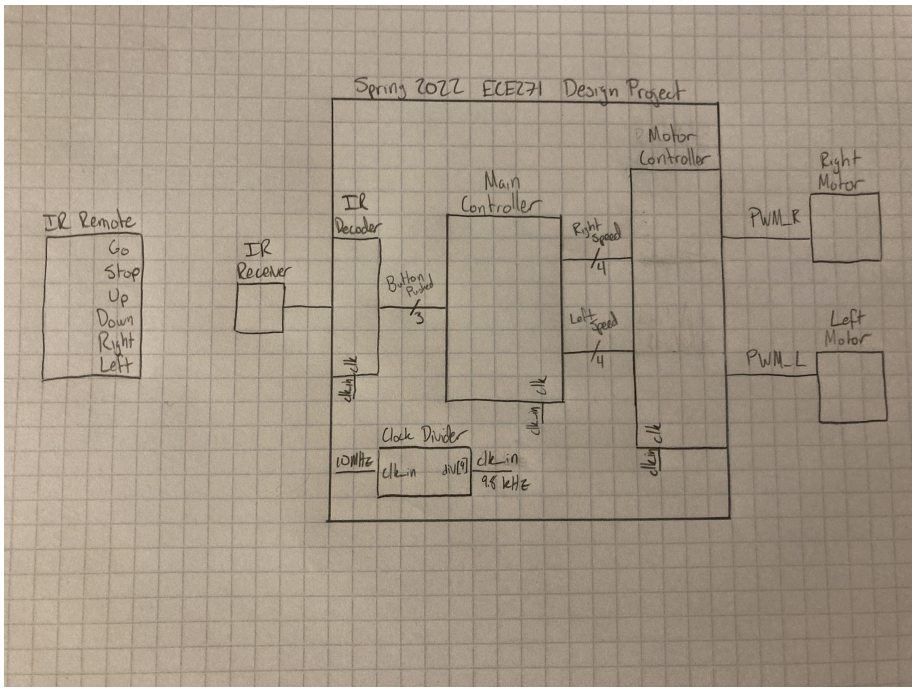

Figure 1: This image shows a high level block diagram. It includes the three main blocks as well as the input and outputs.

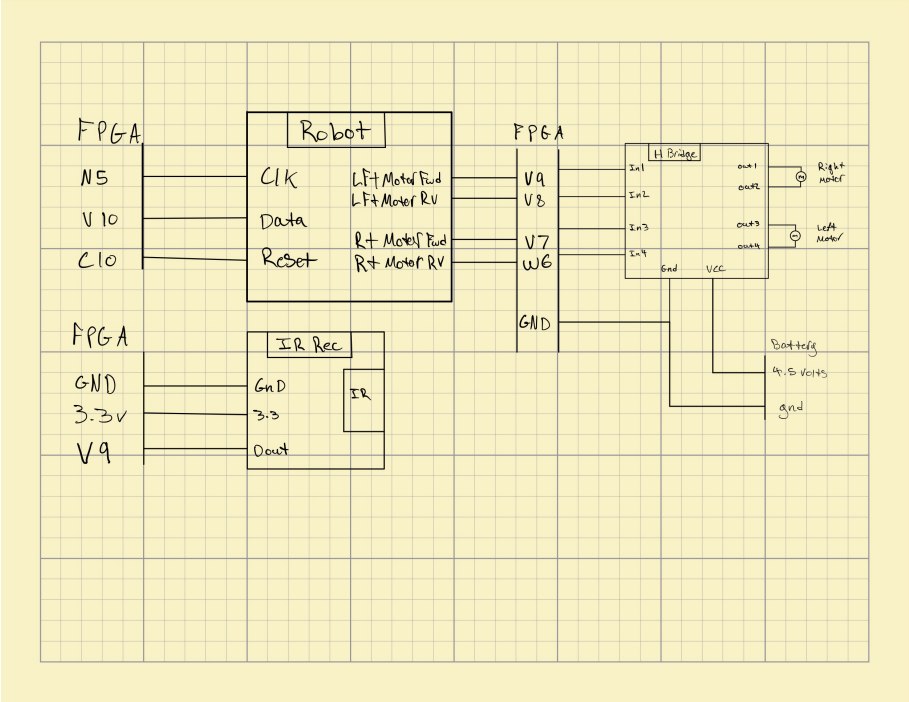

Figure 2: Hardware diagram at the highest level of abstraction to show the physical modules.

2 High Level Description

Inputs: This reads an infrared signal and a 10 MHz clock signal.
Outputs: This controls two motors at varying speeds and directions using four outputs.

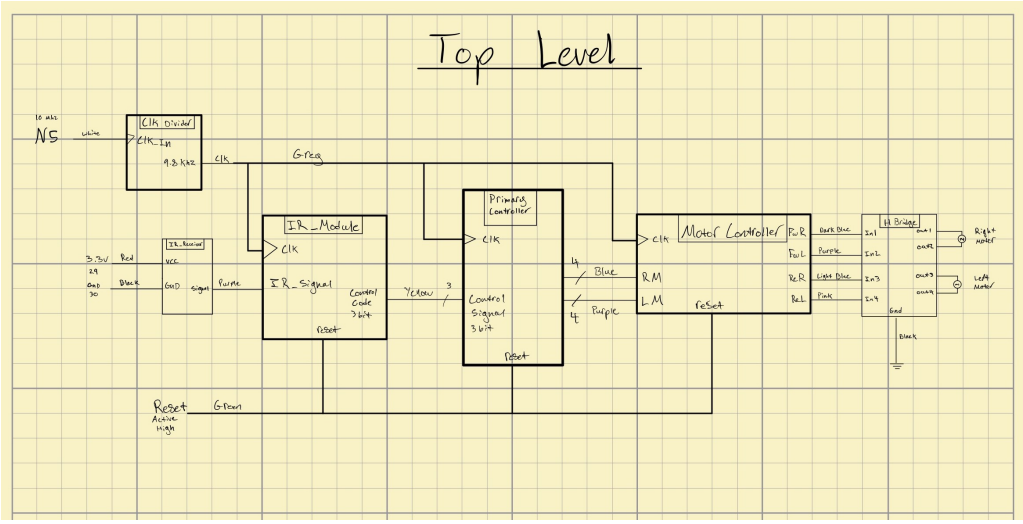

Figure 3: The top level design for the a robot controlled by an IR remote. This design follows basic functionality of reading an IR signal that is translated to a control signal before being sent to the primary controller. The primary controller keeps track of what is and should happen based on the signals it receives. It then passes these signals to a motor controller capable of PWM in order to precisely control two motors.

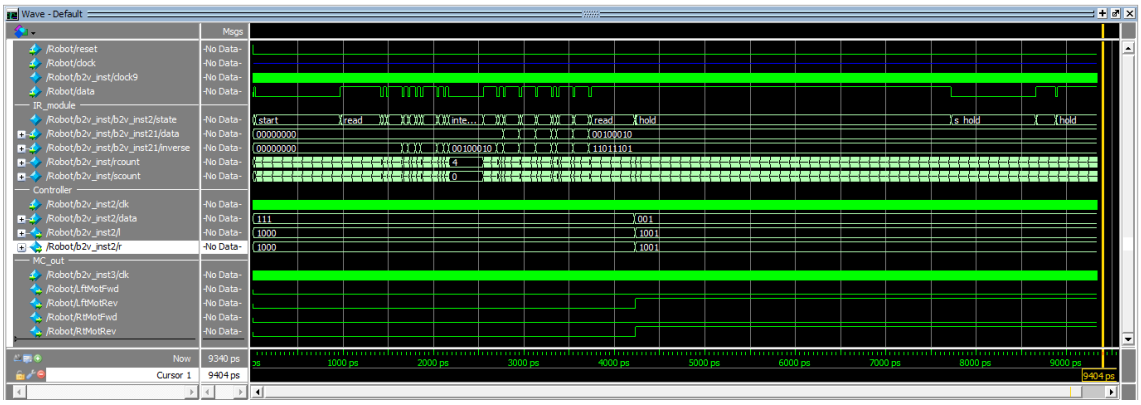

Figure 4: Simulation of all modules working together to run the Robot. This simulation shows a single IR command coming in, being interpreted and finally being executed by both the primary controller and the motor controller. The signal was go from stop position. Something to note is the forward and reverse signals are swapped but otherwise work correctly. The simulation commands can be found at B.0.1.

2.1 IR Reader

Inputs: This reads an IR signal, uses a 10 MHz clock signal, and has an active high reset.

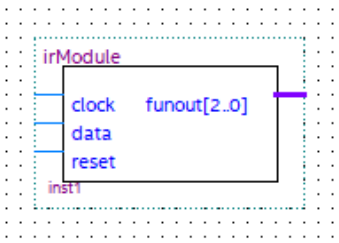

Figure 5: Top Level block diagram of the IR reader module.

Outputs: a 3-bit signal out to the primary controller. The signal consist 7 commands outlined in the table below.

Commands	
Stop	000
Go	001
Speed Up	010
Speed Down	011
Right	100
Left	101
No Input	111

This module consist of a finite state machine that synchronizes a set of counters to run at specific points within the received signal. One counter runs when signal is low and the other counts when the signal is high. The counters stop on the opposing edge and then the bit interpreter converts the counts to a one or zero that is finally shifted into an eight bit register. Once the entire sequence is read inverse data and data bits are checked for errors using the validate module. If valid the a latch sets the eight bit signal to the output. Next the reader proceeds to wait for a hold signal, if found it will repeat keeping the latch with data high, if hold is not found the latch becomes reset and data falls to zero. Finally, data is moves through a secondary interpreter in order to provide a nice code to the primary controller.

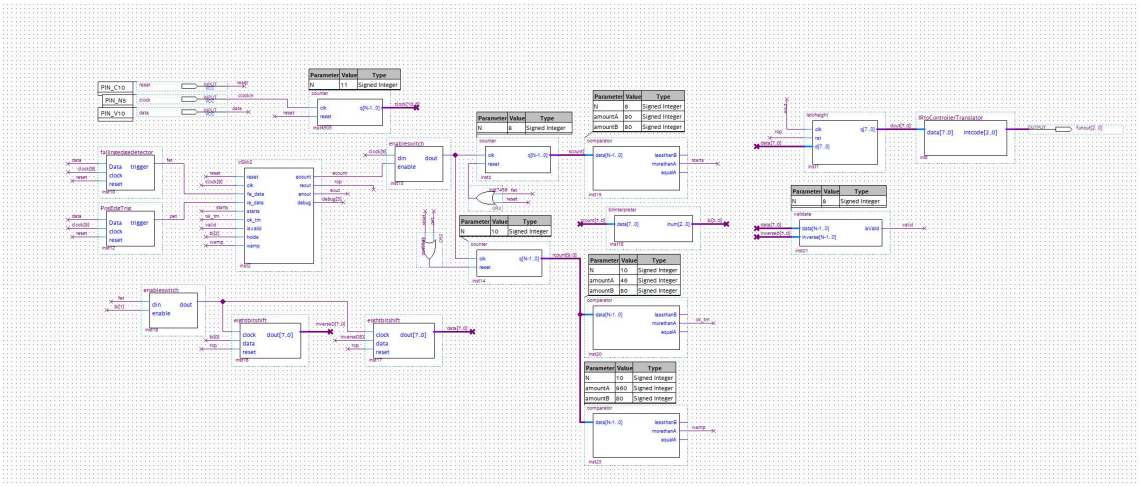

Figure 6: This is an expanded view of the IR module.

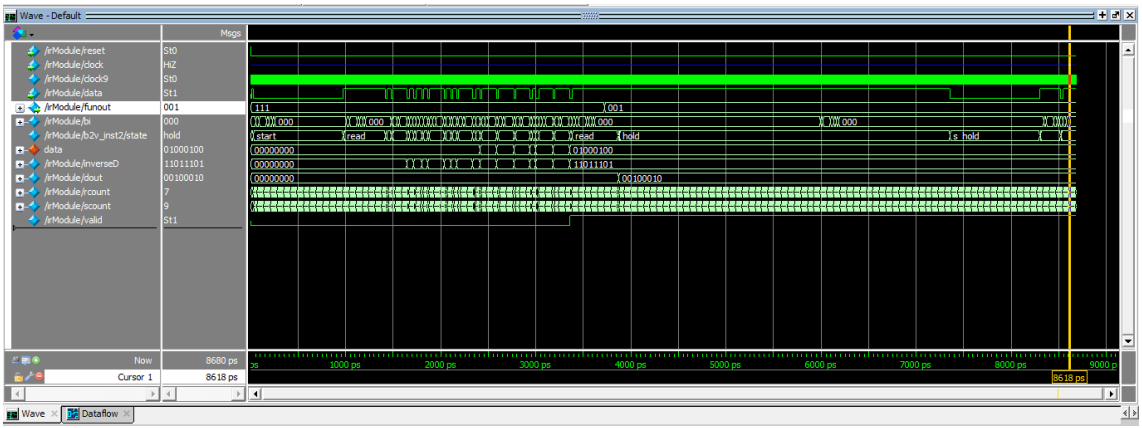

Figure 7: This simulation tested the go command. Looking at the data wave we can see the IR signal coming into the reader, the status of the state of the reader and finally the function out labelled funout. Notice once all of the bits have been read and validated the function output become the binary bit 001 signifying the Go command. The simulation commands can be found at B.1.

2.1.1 IR FSM

Inputs: 9.8 Khz clock signal. Eight active high inputs are used to change the FSMs state.

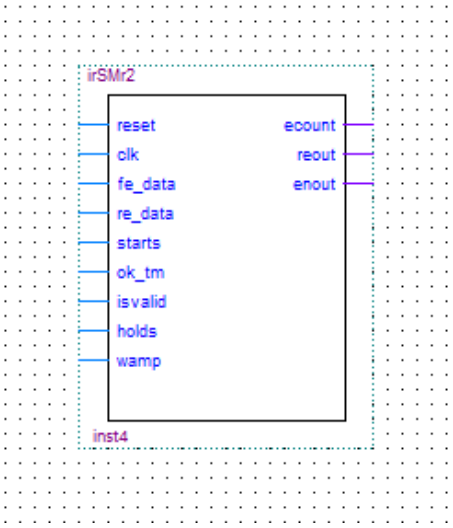

Figure 8: This is IR finite state machine. The purpose of this module is to ensure the counters and interpreters are running at the correct points in time.

Outputs: It has 3 outputs, ecount enables the counters to run, reout resets the output of the reader, and enout enables the output of the reader.

For this block the simulation was not done until the supporting modules were ready. This was to simplify the testing process since this particular module is very reliant on many signals running at precise timings. To see this simulation please reference figure 7.

2.1.2 Bit Interpreter

Inputs: Takes an eight bit number from the counters.

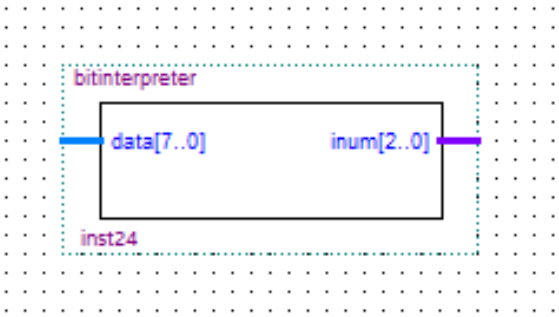

Figure 9: This block takes an eight bit number from the counters and translates it to a one or zero. The first bit is the number. The second bit resembles if the bit is valid and the third bit signals when the hold sequence has been detected. Valid bits are pushed to the shift register as part of decoded IR signal.

Outputs: a 3 bit signal that is split up within the IR module.

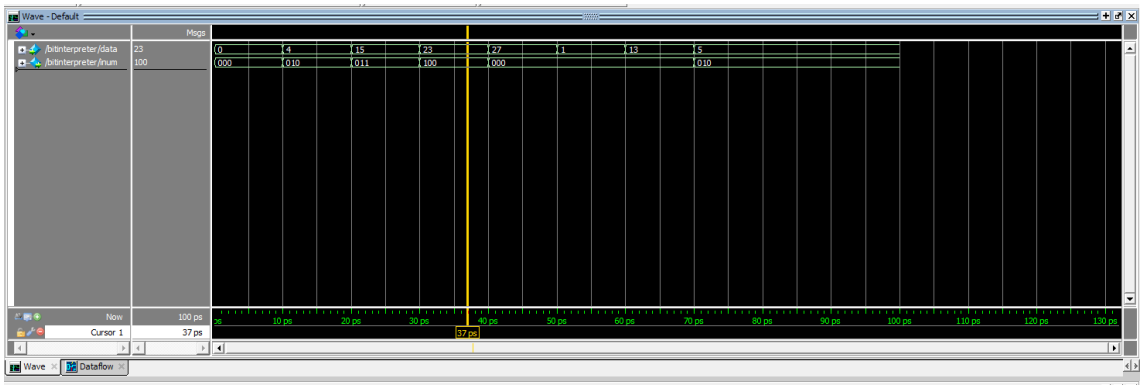

Figure 10: This simulation moved through each possible interpretation. The data represents the incoming decimal number and the inum represents the translated bits. The simulation commands can be found at B.1.4.

2.1.3 Falling Edge Detector

Inputs: Active HIGH reset. Clock signal. Data signal.
Outputs: trigger goes HIGH on detection.

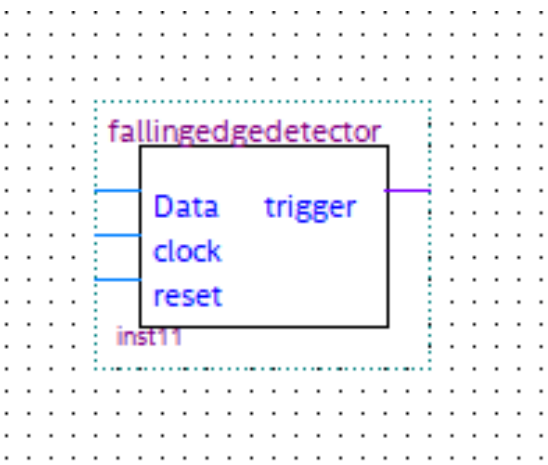

Figure 11: This block looks for a falling edge within the data signal and then outputs HIGH for one clock signal.

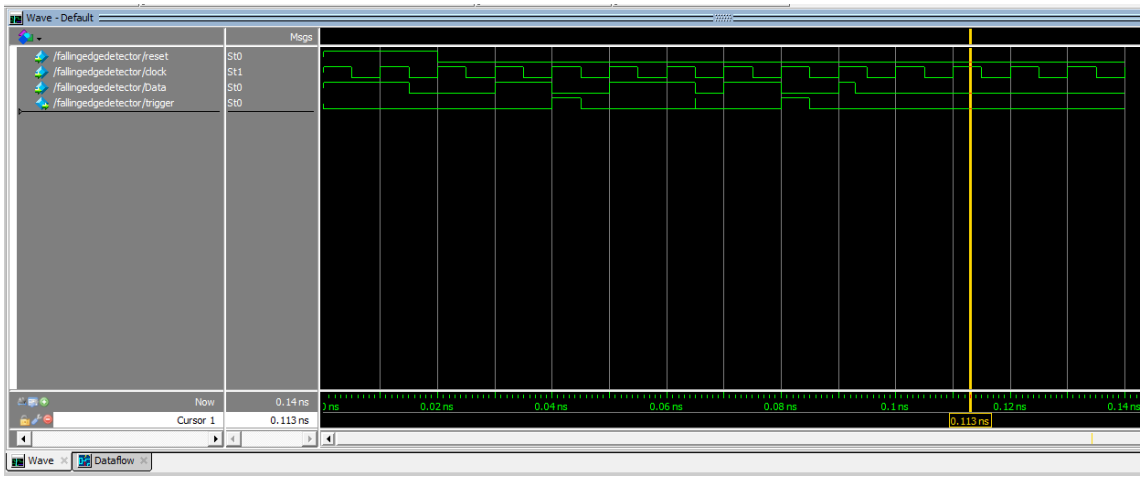

Figure 12: This simulation tested an active HIGH data signal that would fall at various points. Each time the signal became LOW the trigger activated for one clock signal. This module helps the FSM know what is happening in the IR signal in order to change states. The simulation commands can be found at B.1.2.

2.1.4 Positive Edge Trigger

Inputs: Active HIGH reset. Clock signal. Data signal.

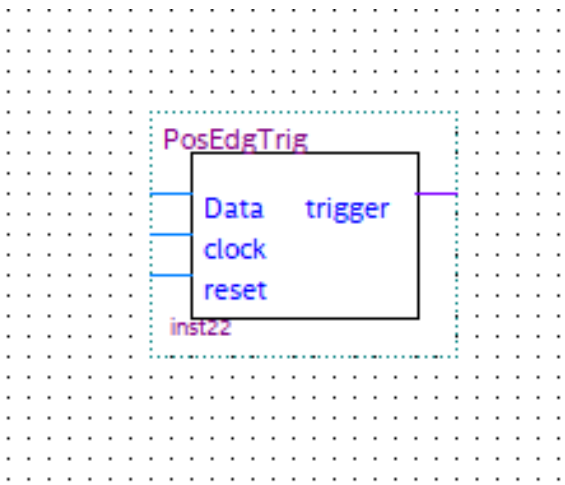

Figure 13: This block looks for a rising edge within the data signal and then outputs high for one clock signal.

Outputs: trigger goes HIGH on detection.

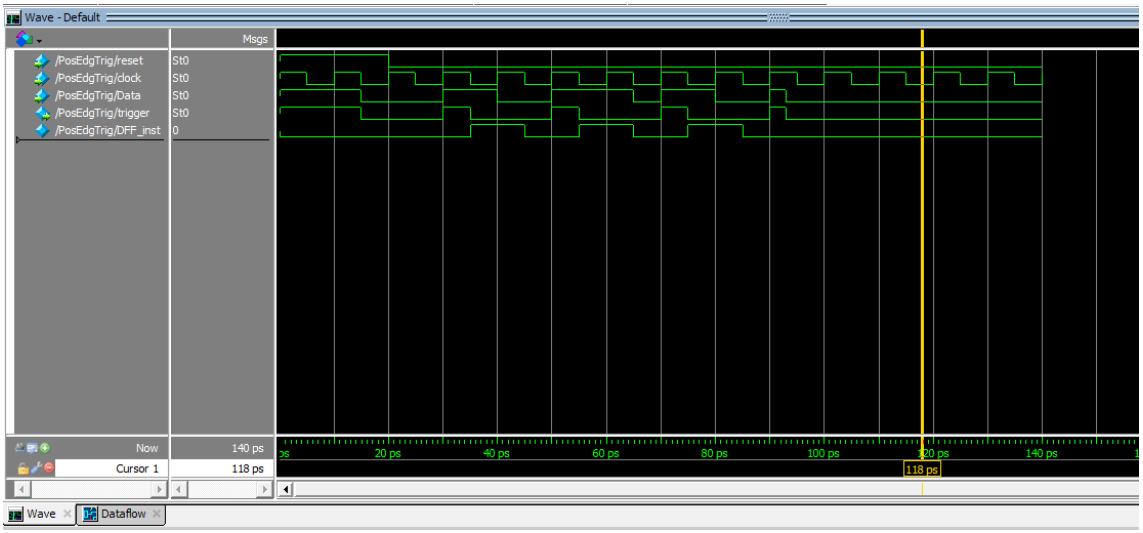

Figure 14: This simulation tested an active LOW data signal that would rise at various points. Each time the signal became HIGH the trigger activated for one clock signal. This module helps the FSM know what is happening in the IR signal in order to change states. The simulation commands can be found at B.1.3.

2.1.5 IR to Controller Translator

Inputs: one 8-bit number.
Outputs: one 3-bit number output one to five and seven.

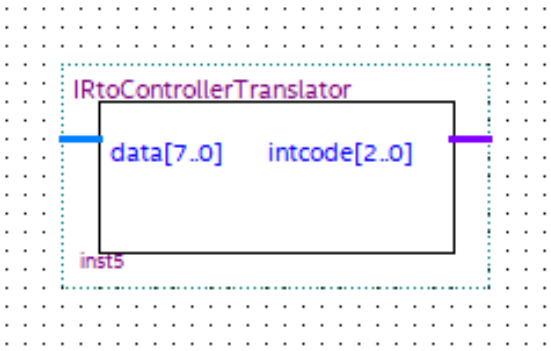

Figure 15: This module translates the IR remote code to a usable 3-bit command for the primary controller. This simplifies the control signal going to the primary controller.

Figure 16: This simulation tested each of needed remote commands for the primary controller. Reference table below to see the number to command. The simulation commands can be found at B.1.1.

Commands

Button	code	out
Stop	69	0
Go	68	1
Speed Up	9	2
Speed Down	21	3
Right	67	4
Left	64	5
No Input	xxx	7

2.1.6 Eight bit Latch

Inputs: one 8-bit binary bus, a clock signal to cycle the latch, and a reset to bring the latch to zero .

Outputs: one 8-bit binary bus.

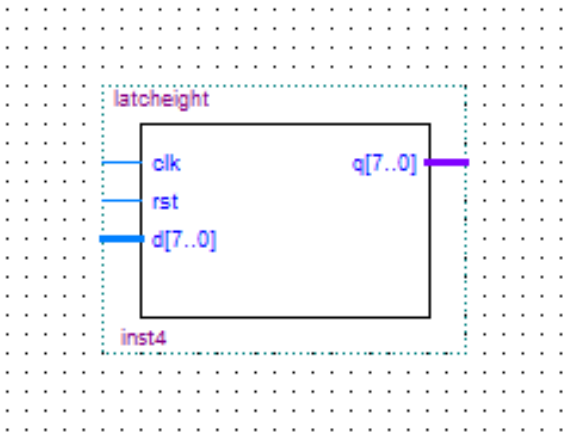

Figure 17: This module simply latches the 8-bit data signal on each clock cycle to the output. This clock signal can is more meant to be a stationary signal like trigger but it can be used as a sync module when paired with a normal alternating signal. In the IR module this is used to activate the output only during a button press or while the hold sequence is active.

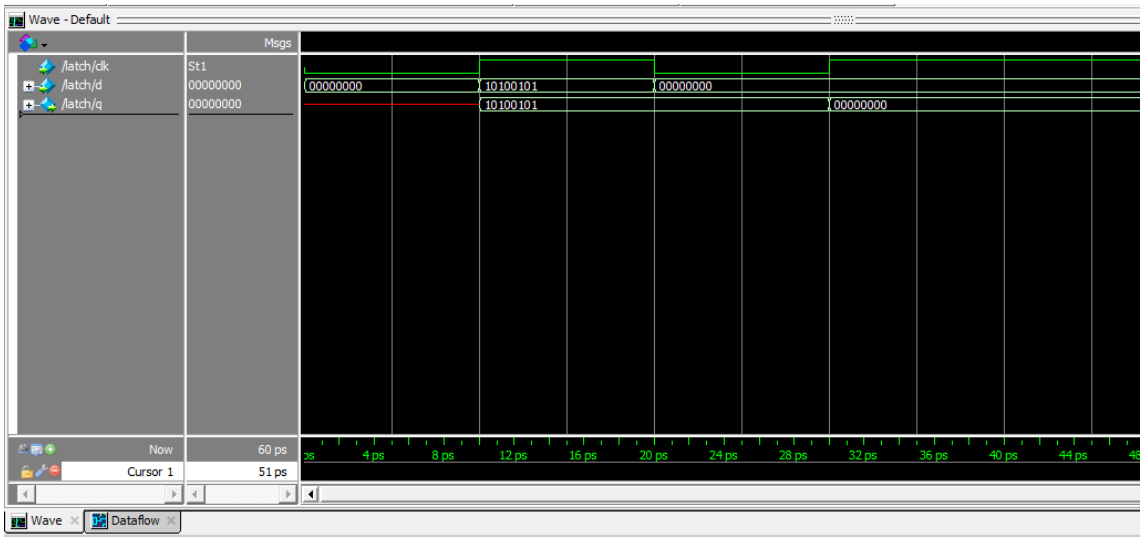

Figure 18: This simulation tested an arbitrary signal that gets latched on the rising edge of each clock signal. Finally, it can be reset at the end to bring the output back to zero. The simulation commands can be found at B.1.5.

2.1.7 Validate - N bit XOR

Inputs: two 8-bit binary buses inverse data and data.
Outputs: a single HIGH/LOW output.

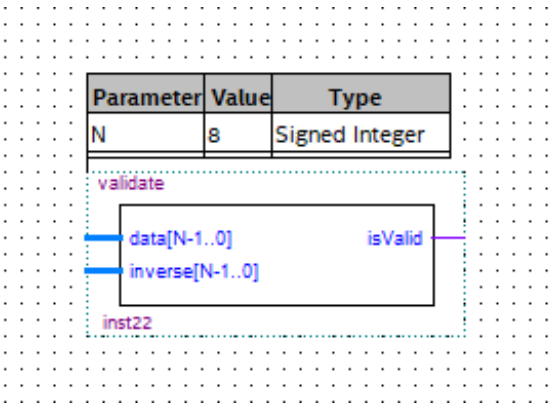

Figure 19: This module is not actually parameterized, instead each bit is connected to an XOR individually then connected to an AND gate to present a valid sequence. This module is used to verify the IR signal which provides two segments inverse data and data. If the sequence is valid then IR module will display the data.

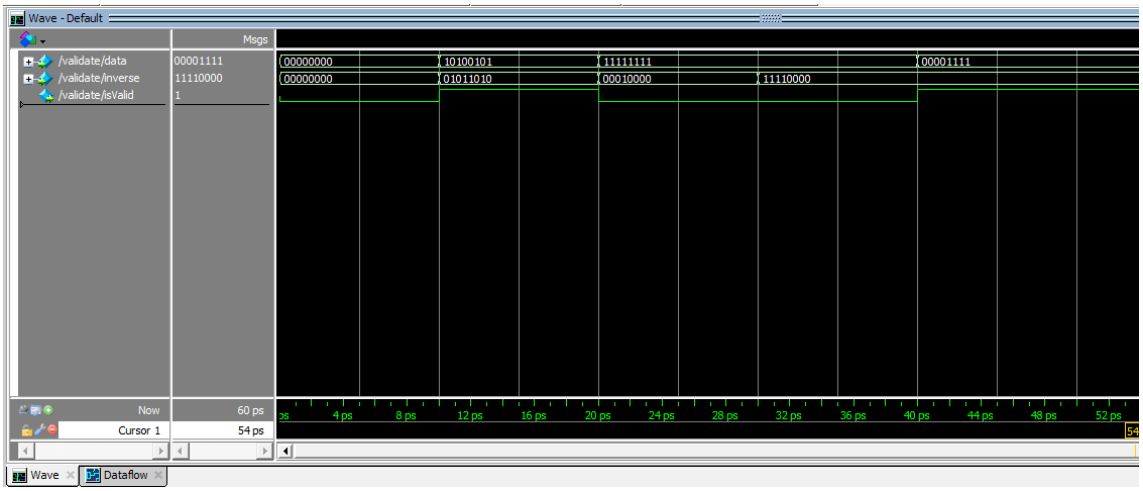

Figure 20: This simulation cycles through a few valid and invalid scenarios. The output becomes high if the signal is valid and low if it's invalid. The simulation commands can be found at B.1.6.

2.2 Main Controller

Inputs: The main controller reads a 3 bit value, uses a 9.8 KHz clock signal, and has an active low reset.

Outputs: The main controller outputs two 4 bit values. One output gives the speed information to the left motor controller and the other gives the speed information to the right motor controller. The most significant bit of the outputs is the direction. The two middle bits represent the speed. The least significant bit is the stop or go. The table below provides some example outputs and their corresponding behaviors.

0000	stopped
0001	lowest speed forward
0011	middle speed forward
0101	highest speed forward
1001	lowest speed in reverse
1011	middle speed in reverse
1101	highest speed in reverse

The role of the main controller is to translate which button is being pressed to the speeds that the motors should go. The main controller is composed of four blocks. The block diagram in figure 21 shows how the blocks, toggle FSM, speed, FSM, right decoder, and left decoder, fit together. The behavior of the main controller is shown in figures 22 and 23.

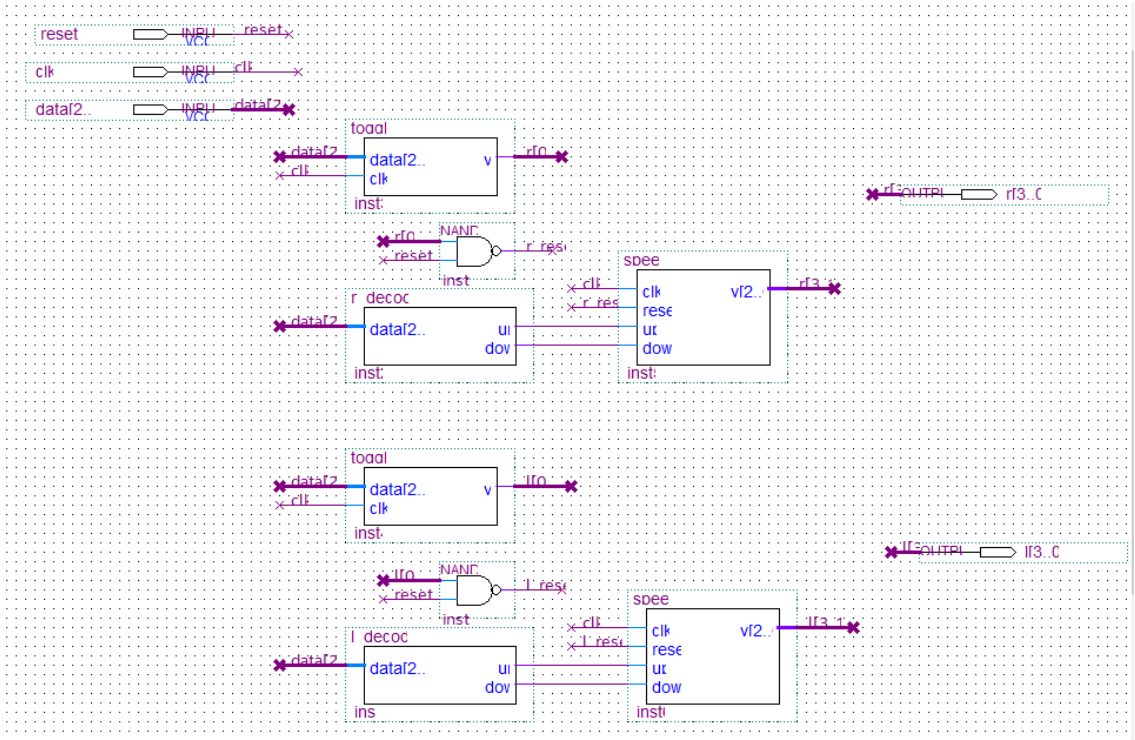

Figure 21: The block diagram for the main controller.

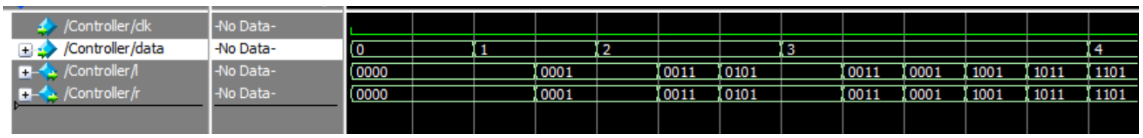

Figure 22: Simulation of the controller showing how the speed of each motor is increased and decreased by the stop (0), go (1), up (2), and down (3) buttons. The do file that created this simulation is in B.2.

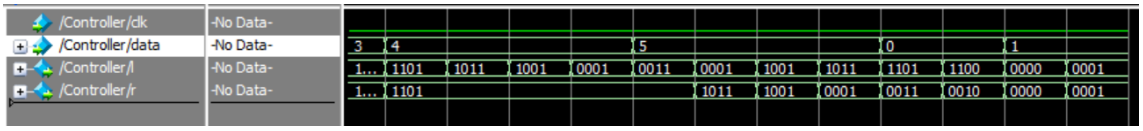

Figure 23: Simulation of the controller showing how the speed of each motor is increased and decreased by the right (4) and left (5) buttons. The do file that created this simulation is in B.2.

2.2.1 Toggle FSM

Inputs: The toggle FSM uses a 9.8 kHz clock signal and a three bit value.

Outputs: The toggle FSM outputs a single bit with 1 meaning go and 0 meaning stop. This bit will end up being the least significant bit in both outputs of the main controller.

The role of toggle FSM is determine if the motors should be on or off. The HDL for the toggle FSM is in A.2.1. Figure 24 shows the operation of the toggle FSM.

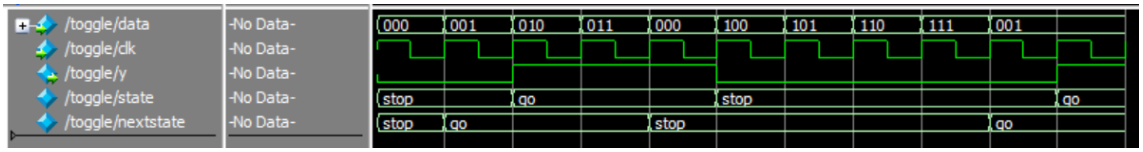

Figure 24: Simulation of the toggle FSM. The state only switches when the stop (0) or go (1) button is pressed. The do file that created this simulation is in B.2.1

2.2.2 Left Decoder

Inputs: The left decoder takes in one three bit value.

Outputs: The left decoder outputs two 1 bit values. The signals represent up and down. The two signals can't be HIGH at the same time.

The role of the left decoder is to determine from the button pressed if the up and down signals should be HIGH or LOW for the left motor. The HDL for the left decoder is in A.2.2. Figure 25 shows the operation of the left decoder.

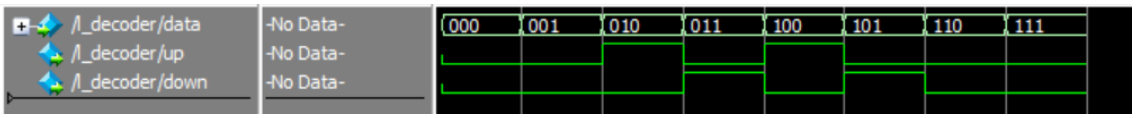

Figure 25: Simulation of the left decoder. The up signal goes HIGH when the pressed button is up (2) and right (4). The down signal goes HIGH when the button pressed is down (3) and left (5). The do file that created this simulation is in B.2.2.

2.2.3 Right Decoder

Inputs: The right decoder takes in one three bit value.

Outputs: The right decoder outputs two 1 bit signals. The signals are up and down. The two signals can't be HIGH at the same time.

The role of the right decoder is to determine when the right motor should be speeding up or slowing down based on the button pressed. The HDL for the right decoder is in A.2.3. Figure 26 shows the operation of the right decoder.

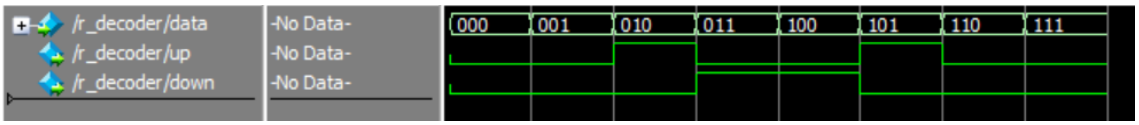

Figure 26: Simulation of the right decoder. The up signal goes HIGH when the pressed button is up (2) and left (5). The down signal goes HIGH when the button pressed is down (3) and right (4). The do file that created this simulation is in B.2.3.

2.2.4 Speed FSM

Inputs: The speed FSM takes in three 1 bit values and uses a 9.8 kHz clock signal.

Outputs: The speed FSM outputs one 3 bit value. The output will become the three most significant bits of the main controller outputs.

The role of the speed FSM is shift between the different speeds of the motors based on how previous blocks have interpreted the pressed button. Specifically, the speed will be reset to low forward after being stopped and change the speed up or down depending on the up and down signals from the decoders. The HDL for the speed FSM is in A.2.4. Figure 27 shows the operation of the speed FSM.

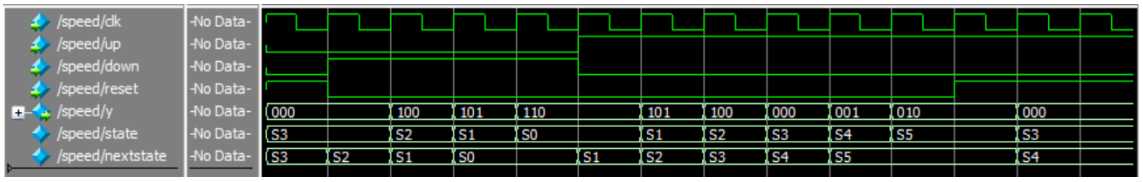

Figure 27: Simulation of the speed FSM. When up is HIGH the speed shifts up to faster states and when down is HIGH the speed shifts down to slower states. The do file that created this simulation is in B.2.4.

2.3 Motor Controller

The module, represented by the block diagram shown below, receives 2 4-bit numbers from the main controller, a clock signal, a ground signal, and a reset input and outputs 2 2-bit outputs, a signal for each of the motors controlling the two wheels.

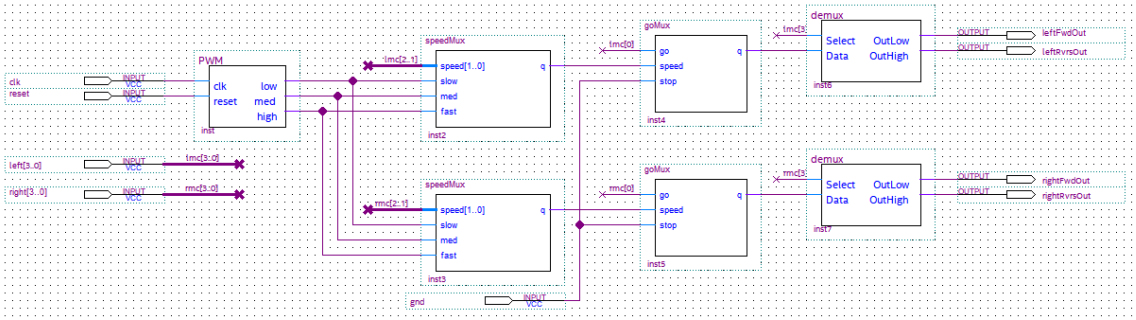

Figure 28: The block diagram of the motor controller.

Inputs: clk reset gnd left[3:0] right[3:0]
Outputs: rightFwdOut, rightRvrsOut, leftFwdOut, leftRvrsOut

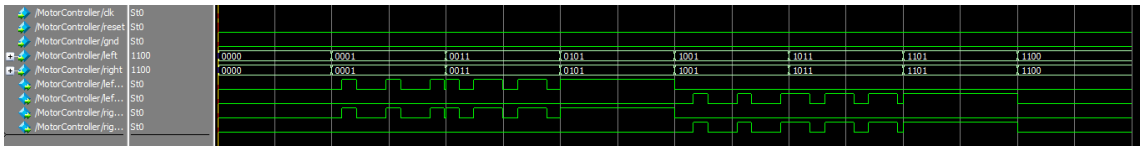

Figure 29: The simulation of the Motor Controller block.

2.3.1 Pulse-Width Modulator

The module, represented by the block diagram shown below, receives a clock input and a reset input and outputs a signal for each of the three speeds, low, med, and high.

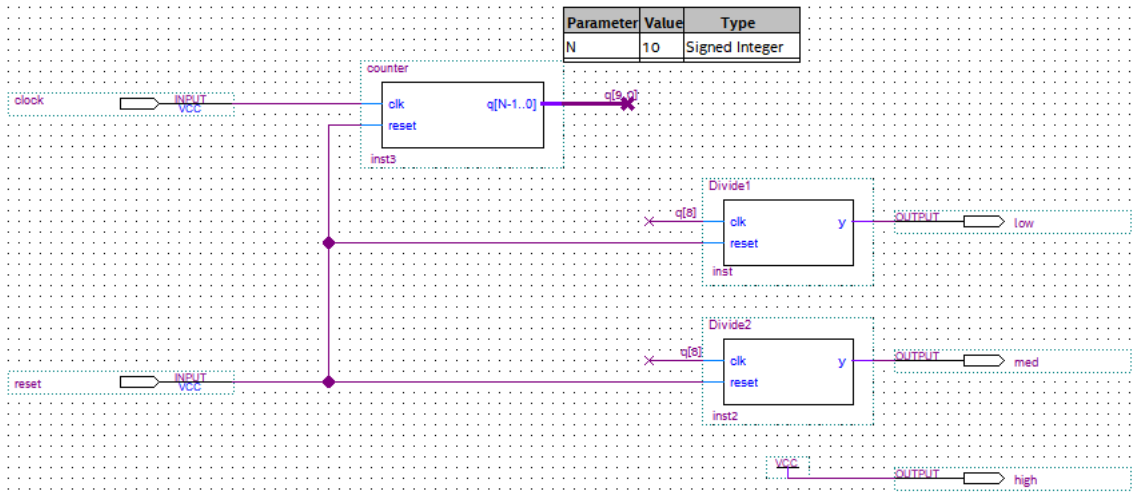

Figure 30: The block diagram of the Motor Controller's PWM.

Inputs: clock reset Outputs: low med high

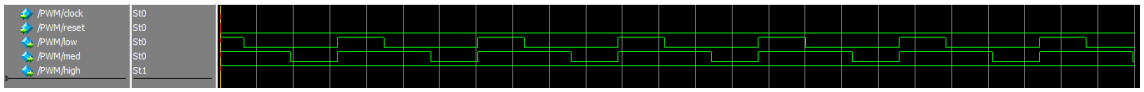

Figure 31: The simulation of the PWM block.

2.3.2 Finite State Machine 1/3 Divider

This module is a finite state machine functioning as a 1/3 divider, that is it will output a clock signal at 1/3 of the input frequency.

Inputs: clk, reset Output: y

Figure 32: The simulation of the 1/3 Divider FSM.

2.3.3 Finite State Machine 2/3 Divider

This module is a finite state machine functioning as a 2/3 divider, that is it will output a clock signal at 2/3 of the input frequency.

Inputs: clk, reset Output: y

Figure 33: The simulation of the 2/3 Divider FSM.

2.3.4 PWM Clock Counter

This module is a counter used to slow down the clock signal before it is sent to the FSMs to be further modified.

Inputs: clk, reset Output: q

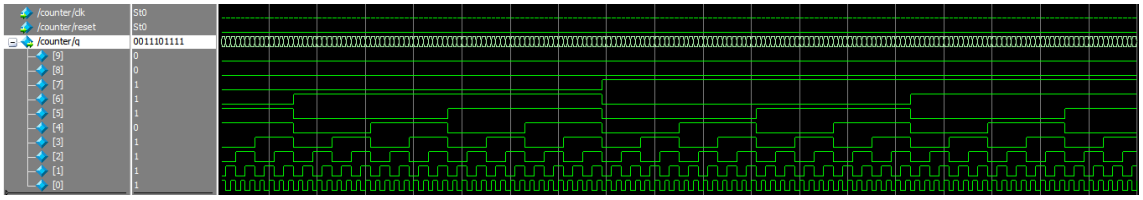

Figure 34: The simulation PWM Clock Counter.

2.3.5 Speed Multiplexer

This module is a multiplexer that is used to determine the appropriate speed to be output. It accepts a 2-bit speed input that is used to determine the desired output, as well as the 3 1-bit speed inputs generated by the PWM.

Inputs: speed[1:0], slow, med, fast Output: q

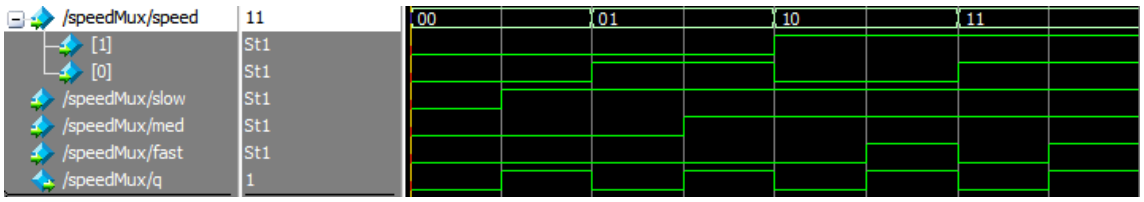

Figure 35: The simulation of the Speed Multiplexer.

2.3.6 Go Multiplexer

This module is a multiplexer that is used to determine whether to stop or move the robot. It accepts a go input, which is used to determine the output, a speed input from the Speed Multiplexer, and a stop input which is generated from a ground input.

Inputs: go, speed, stop Output: q

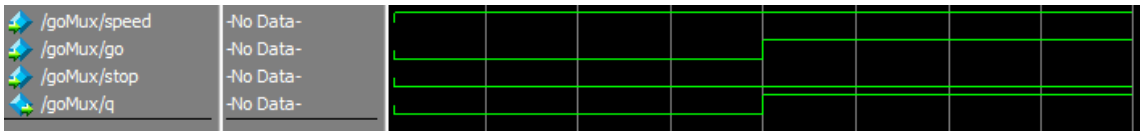

Figure 36: The simulation of the Go Multiplexer.

2.3.7 Direction Multiplexer

This module, represented by the block diagram below, is used to determine the direction each motor should be moving in. It accepts a select input, which determines the direction, and a data input, which is the speed from the Go Multiplexer. The low output is used to represent forward movement and the high output is used to represent reverse movement.

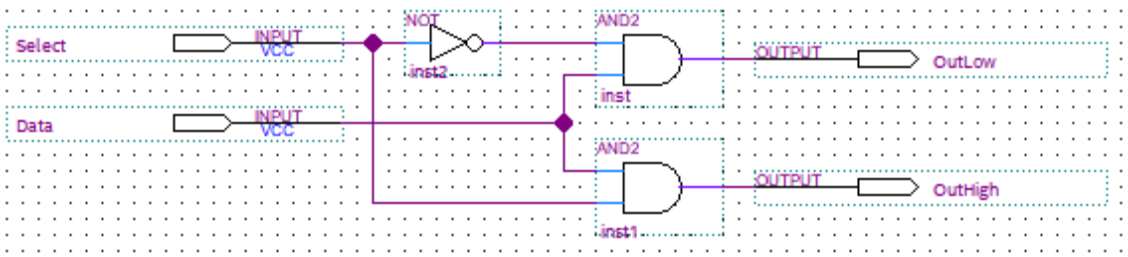

Figure 37: The block diagram of the Motor Controller's Direction Multiplexer.

Inputs: Select, Data Outputs: OutLow, OutHigh

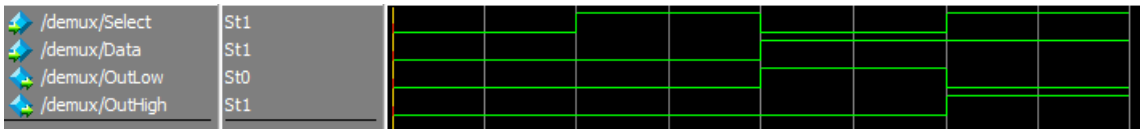

Figure 38: The simulation of the Direction Multiplexer.

A SystemVerilog Files

```
//Putting it together, Top Level Robot
module Robot(
        reset,
        clock,
        data,
        LftMotFwd,
        LftMotRev,
        RtMotFwd,
        RtMotRev
);

input wire      reset;
input wire      clock;
input wire      data;
output wire     LftMotFwd;
output wire     LftMotRev;
output wire     RtMotFwd;
output wire     RtMotRev;

wire    clock0;
wire    clock1;
wire    clock10;
wire    clock2;
wire    clock3;
wire    clock4;
wire    clock5;
wire    clock6;
wire    clock7;
wire    clock8;
wire    clock9;
wire    clockin;
wire    [2:0] SYNTHESIZED_WIRE_0;
wire    SYNTHESIZED_WIRE_1;
wire    [3:0] SYNTHESIZED_WIRE_2;
wire    [3:0] SYNTHESIZED_WIRE_3;

assign  SYNTHESIZED_WIRE_1 = 0;



irModule        b2v_inst(
        .reset(reset),
        .clock(clockin),
        .data(data),
        .funout(SYNTHESIZED_WIRE_0));

Controller      b2v_inst2(
        .clk(clock9),
        .data(SYNTHESIZED_WIRE_0),
        .l(SYNTHESIZED_WIRE_2),
        .r(SYNTHESIZED_WIRE_3));

MotorController b2v_inst3(
        .clk(clockin),
        .reset(reset),
        .gnd(SYNTHESIZED_WIRE_1),
        .left(SYNTHESIZED_WIRE_2),
        .right(SYNTHESIZED_WIRE_3),
        .leftFwdOut(LftMotFwd),
        .leftRvrsOut(LftMotRev),
        .rightFwdOut(RtMotFwd),
        .rightRvrsOut(RtMotRev));


counter b2v_inst4905(
        .clk(clockin),
        .reset(reset),
        .q({clock9}));
        defparam        b2v_inst4905.N = 11;

assign  clockin = clock;

endmodule
```

A.1 IR Reader HDL

```
// Top Level IR Reader

module irModule(
        reset,
        clock,
        data,
        funout
);


input wire      reset;
input wire      clock;
input wire      data;
output wire     [2:0] funout;
```

```
15
16  wire     [2:0] bi;
17  wire     clock0;
18  wire     clock1;
19  wire     clock10;
20  wire     clock2;
21  wire     clock3;
22  wire     clock4;
23  wire     clock5;
24  wire     clock6;
25  wire     clock7;
26  wire     clock8;
27  wire     clock9;
28  wire     clockin;
29  wire     data0;
30  wire     data1;
31  wire     data2;
32  wire     data3;
33  wire     data4;
34  wire     data5;
35  wire     data6;
36  wire     data7;
37  wire     [7:0] dout;
38  wire     ecount;
39  wire     eout;
40  wire     fet;
41  wire     [7:0] inverseD;
42  wire     ok_tm;
43  wire     pet;
44  wire     [9:0] rcount;
45  wire     rop;
46  wire     [7:0] scount;
47  wire     starts;
48  wire     valid;
49  wire     wamp;
50  wire     SYNTHESIZED_WIRE_6;
51  wire     SYNTHESIZED_WIRE_1;
52  wire     SYNTHESIZED_WIRE_7;
53  wire     SYNTHESIZED_WIRE_5;
54
55  wire     [7:0] GDFX_TEMP_SIGNAL_0;
56  wire     [7:0] GDFX_TEMP_SIGNAL_1;
57  wire     [7:0] GDFX_TEMP_SIGNAL_2;
58
59
60  assign   GDFX_TEMP_SIGNAL_0 = {data7,data6,data5,data4,data3,data2,data1,data0};
61  assign   data7 = GDFX_TEMP_SIGNAL_1[7];
62  assign   data6 = GDFX_TEMP_SIGNAL_1[6];
63  assign   data5 = GDFX_TEMP_SIGNAL_1[5];
64  assign   data4 = GDFX_TEMP_SIGNAL_1[4];
65  assign   data3 = GDFX_TEMP_SIGNAL_1[3];
66  assign   data2 = GDFX_TEMP_SIGNAL_1[2];
67  assign   data1 = GDFX_TEMP_SIGNAL_1[1];
68  assign   data0 = GDFX_TEMP_SIGNAL_1[0];
69
70  assign   GDFX_TEMP_SIGNAL_2 = {data7,data6,data5,data4,data3,data2,data1,data0};
71
72
73  IRtoControllerTranslator          b2v_inst(
74          .data(dout),
75          .intcode(funout));
76
77
78  latcheight        b2v_inst1(
79          .clk(eout),
80          .rst(rop),
81          .d(GDFX_TEMP_SIGNAL_0),
82          .q(dout));
83
84
85  fallingedgedetector       b2v_inst10(
86          .Data(data),
87          .clock(clock9),
88          .reset(reset),
89          .trigger(fet));
90
91
92  PosEdgTrig        b2v_inst12(
93          .Data(data),
94          .clock(clock9),
95          .reset(reset),
96          .trigger(pet));
97
98
99  enableswitch      b2v_inst13(
100         .din(clock9),
101         .enable(ecount),
102         .dout(SYNTHESIZED_WIRE_6));
103
104
105 counter b2v_inst14(
106         .clk(SYNTHESIZED_WIRE_6),
107         .reset(SYNTHESIZED_WIRE_1),
108         .q(rcount));
109         defparam          b2v_inst14.N = 10;
110
111
112 comparator        b2v_inst15(
113         .data(scount),
114
115         .morethanA(starts)
116         );
117         defparam          b2v_inst15.amountA = 90;
118         defparam          b2v_inst15.amountB = 80;
119         defparam          b2v_inst15.N = 8;
120
121
122 eightbitshift     b2v_inst16(
123         .clock(SYNTHESIZED_WIRE_7),
124         .data(bi[0]),
125         .reset(rop),
126         .dout(inverseD));
127
128
129 eightbitshift     b2v_inst17(
130         .clock(SYNTHESIZED_WIRE_7),
```

```
131           .data(inverseD[0]),
132           .reset(rop),
133           .dout(GDFX_TEMP_SIGNAL_1));
134
135
136   bitinterpreter  b2v_inst18(
137           .data(rcount[7:0]),
138           .inum(bi));
139
140
141   enableswitch    b2v_inst19(
142           .din(fet),
143           .enable(bi[1]),
144           .dout(SYNTHESIZED_WIRE_7));
145
146
147   irSMr2  b2v_inst2(
148           .reset(reset),
149           .clk(clock9),
150           .fe_data(fet),
151           .re_data(pet),
152           .starts(starts),
153           .ok_tm(ok_tm),
154           .isvalid(valid),
155           .holds(bi[2]),
156           .wamp(wamp),
157           .ecount(ecount),
158           .reout(rop),
159           .enout(eout));
160
161
162   comparator      b2v_inst20(
163           .data(rcount),
164
165           .morethanA(ok_tm)
166           );
167           defparam        b2v_inst20.amountA = 46;
168           defparam        b2v_inst20.amountB = 80;
169           defparam        b2v_inst20.N = 10;
170
171
172   validate        b2v_inst21(
173           .data(GDFX_TEMP_SIGNAL_2),
174           .inverse(inverseD),
175           .isValid(valid));
176           defparam        b2v_inst21.N = 8;
177
178
179   comparator      b2v_inst23(
180           .data(rcount),
181
182           .morethanA(wamp)
183           );
184           defparam        b2v_inst23.amountA = 960;
185           defparam        b2v_inst23.amountB = 80;
186           defparam        b2v_inst23.N = 10;
187
188
189   counter b2v_inst3(
190           .clk(SYNTHESIZED_WIRE_6),
191           .reset(SYNTHESIZED_WIRE_5),
192           .q(scount));
193           defparam        b2v_inst3.N = 8;
194
195
196   counter b2v_inst4905(
197           .clk(clockin),
198           .reset(reset),
199           .q({clock9}));
200           defparam        b2v_inst4905.N = 11;
201
202   assign  SYNTHESIZED_WIRE_1 = reset | pet;
203
204   assign  SYNTHESIZED_WIRE_5 = fet | reset;
205
206   assign  clockin = clock;
207
208   endmodule
```

A.1.1 IR Finite State Machine

```
1   //IR fsm to keep the counters well orchestrated
2   module irSMr2 (input reset, input clk, input fe_data, input re_data,
3           input starts, input ok_tm, input isvalid, input holds, input wamp,
4       output ecount, output reout, output enout);
5
6   typedef enum logic [10:0] {idle, start, i_start, read, interpret, validate,
7   set, hold, s_hold, si_hold, r_hold, i_hold} statetype;
8   statetype state, nextstate;
9
10  // state register
11  always_ff @(posedge clk, posedge reset)
12          if (reset) state <= idle;
13          else state <= nextstate;
14          // next state logic
15
16  always_comb
17          case (state)
18                  idle: if (fe_data) nextstate = start;
19                                  else nextstate = idle;
20
21                  start: if (re_data) nextstate = i_start;
22                                  else nextstate = start;
23
24                  i_start: if (starts) nextstate = read;
25                                  else nextstate = idle;
26
27                  read: if (ok_tm) nextstate = validate;
28                                  else if (fe_data) nextstate = interpret;
29                                  else nextstate = read;
30
31                  interpret: if (re_data) nextstate = read;
32                                  else nextstate = interpret;
33
```

```
                    validate: if (isvalid) nextstate = set;
                                        else nextstate = idle;

                    set: nextstate = hold;

                    hold: if (fe_data) nextstate = s_hold;
                                    else if (wamp) nextstate = idle;
                                    else nextstate = hold;

                    s_hold: if (re_data) nextstate = si_hold;
                                    else nextstate = s_hold;

                    si_hold: if (starts) nextstate = r_hold;
                                    else nextstate = idle;

                    r_hold: if (ok_tm) nextstate = idle;
                                    else if (fe_data) nextstate = i_hold;
                                    else nextstate = r_hold;

                    i_hold: if (holds) nextstate = hold;
                                    else nextstate = idle;

                    default: nextstate = idle;
            endcase

    // output logic
    assign ecount = ((state == start) | (state == read) | (state == hold) | (state == s_hold) | (state ==
        r_hold));
    assign reout = (state == idle);
    assign enout = (state == set);

    endmodule
```

A.1.2 IR bit Interpreter

```
    // Bit interpreter for IR signal
    module bitinterpreter(input logic [7:0] data,
      output logic [2:0]inum);
    always_comb
      case(data)

            4: inum = 3'b010;
            5: inum = 3'b010;
            6: inum = 3'b010;
            15: inum = 3'b011;
            16: inum = 3'b011;
            17: inum = 3'b011;
            21: inum = 3'b100;
            22: inum = 3'b100;
            23: inum = 3'b100;

            default:        inum = 3'b000;
      endcase
    endmodule
```

A.1.3 Code Interpreter Pt 1

```
    // First Step Code Interpreter
    module codeinterepter(input logic [7:0] data,
       output logic stop, go, up, dwn, rt, lt);
    assign stop = (data == 69);
    assign go = (data == 68);
    assign up = (data == 9);
    assign dwn = (data == 21);
    assign rt = (data == 67);
    assign lt = (data == 64);
    endmodule
```

A.1.4 IR Code to Controller Translator

```
    // IR Code to Controller Translator
    module IRtoControllerTranslator(
            data,
            intcode
    );


    input wire      [7:0] data;
    output wire     [2:0] intcode;

    wire    [5:0] ct;




    codeinterepter  b2v_inst(
            .data(data),
            .stop(ct[0]),
            .go(ct[1]),
            .up(ct[2]),
            .dwn(ct[3]),
            .rt(ct[4]),
            .lt(ct[5]));


    controltranslation      b2v_inst31(
            .data(ct),
            .inum(intcode));


    endmodule
```

A.1.5 Counter

```
    // Counter
    module counter #(parameter N = 8)
      (input logic clk,
```

```
    input logic reset,
    output logic [N-1:0] q);
  always_ff @(posedge clk, posedge reset)
if (reset) q <= 0;
else q <= q + 1;
endmodule
```

A.1.6 Eight Bit Shift Register - Temp Memory for reader.

```
// 8-bit shift register
module eightbitshift(
        reset,
        data,
        clock,
        dout
);


input wire      reset;
input wire      data;
input wire      clock;
output wire     [7:0] dout;

reg     [7:0] b;
wire    SYNTHESIZED_WIRE_8;




always@(posedge clock or negedge SYNTHESIZED_WIRE_8)
begin
if (!SYNTHESIZED_WIRE_8)
        begin
        b[7] <= 0;
        end
else
        begin
        b[7] <= data;
        end
end


always@(posedge clock or negedge SYNTHESIZED_WIRE_8)
begin
if (!SYNTHESIZED_WIRE_8)
        begin
        b[6] <= 0;
        end
else
        begin
        b[6] <= b[7];
        end
end


always@(posedge clock or negedge SYNTHESIZED_WIRE_8)
begin
if (!SYNTHESIZED_WIRE_8)
        begin
        b[5] <= 0;
        end
else
        begin
        b[5] <= b[6];
        end
end


always@(posedge clock or negedge SYNTHESIZED_WIRE_8)
begin
if (!SYNTHESIZED_WIRE_8)
        begin
        b[4] <= 0;
        end
else
        begin
        b[4] <= b[5];
        end
end


always@(posedge clock or negedge SYNTHESIZED_WIRE_8)
begin
if (!SYNTHESIZED_WIRE_8)
        begin
        b[3] <= 0;
        end
else
        begin
        b[3] <= b[4];
        end
end


always@(posedge clock or negedge SYNTHESIZED_WIRE_8)
begin
if (!SYNTHESIZED_WIRE_8)
        begin
        b[2] <= 0;
        end
else
        begin
        b[2] <= b[3];
        end
end


always@(posedge clock or negedge SYNTHESIZED_WIRE_8)
begin
if (!SYNTHESIZED_WIRE_8)
        begin
        b[1] <= 0;
        end
```

```
106   else
107           begin
108           b[1] <= b[2];
109           end
110   end
111
112
113   always@(posedge clock or negedge SYNTHESIZED_WIRE_8)
114   begin
115   if (!SYNTHESIZED_WIRE_8)
116           begin
117           b[0] <= 0;
118           end
119   else
120           begin
121           b[0] <= b[1];
122           end
123   end
124
125   assign  SYNTHESIZED_WIRE_8 =  ~reset;
126
127   assign  dout = b;
128
129   endmodule
```

A.1.7 Falling Edge Detector

```
1   // Falling Edge Detector
2   module fallingedgedetector(
3           Data,
4           clock,
5           reset,
6           trigger
7   );
8
9
10  input wire      Data;
11  input wire      clock;
12  input wire      reset;
13  output wire     trigger;
14
15  wire    SYNTHESIZED_WIRE_0;
16  wire    SYNTHESIZED_WIRE_1;
17  wire    SYNTHESIZED_WIRE_2;
18  reg     DFF_inst;
19
20
21
22
23
24  always@(posedge SYNTHESIZED_WIRE_1 or negedge SYNTHESIZED_WIRE_0)
25  begin
26  if (!SYNTHESIZED_WIRE_0)
27          begin
28          DFF_inst <= 0;
29          end
30  else
31          begin
32          DFF_inst <= Data;
33          end
34  end
35
36  assign  trigger = SYNTHESIZED_WIRE_2 & DFF_inst;
37
38  assign  SYNTHESIZED_WIRE_1 =  ~clock;
39
40  assign  SYNTHESIZED_WIRE_2 =  ~Data;
41
42  assign  SYNTHESIZED_WIRE_0 =  ~reset;
43
44
45  endmodule
```

A.1.8 Positive Edge Trigger

```
1   //Positive Edge Trigger
2   module PosEdgTrig(
3           Data,
4           clock,
5           reset,
6           trigger
7   );
8
9
10  input wire      Data;
11  input wire      clock;
12  input wire      reset;
13  output wire     trigger;
14
15  wire    SYNTHESIZED_WIRE_0;
16  wire    SYNTHESIZED_WIRE_1;
17  wire    SYNTHESIZED_WIRE_2;
18  reg     DFF_inst;
19
20
21
22
23
24  always@(posedge SYNTHESIZED_WIRE_1 or negedge SYNTHESIZED_WIRE_0)
25  begin
26  if (!SYNTHESIZED_WIRE_0)
27          begin
28          DFF_inst <= 0;
29          end
30  else
31          begin
32          DFF_inst <= Data;
33          end
34  end
35
36  assign  trigger = Data & SYNTHESIZED_WIRE_2;
37
```

```
38  assign  SYNTHESIZED_WIRE_1 =  ~clock;
39
40  assign  SYNTHESIZED_WIRE_2 =  ~DFF_inst;
41
42  assign  SYNTHESIZED_WIRE_0 =  ~reset;
43
44
45  endmodule
```

A.1.9 Eight bit Latch

```
1  // 8 bit latch
2  module latcheight (input logic clk, rst,
3  input logic [7:0] d,
4  output logic [7:0] q);
5  always_latch
6          if (clk) q <= d;
7          else if (rst) q <= 8'b0;
8  endmodule
```

A.1.10 Validate

```
1  // Validates two signals using XOR gates
2  module validate #(parameter N=8)
3                   (input logic [N-1:0] data, inverse,
4                    output logic isValid);
5
6      assign isValid = ((data[7]^inverse[7]) & (data[6]^inverse[6]) &
7           (data[5]^inverse[5]) & (data[4]^inverse[4]) & (data[3]^inverse[3])
8           & (data[2]^inverse[2]) & (data[1]^inverse[1]) & (data[0]^inverse[0]));
9  endmodule
```

A.2 Main Controller HDL

A.2.1 Toggle FSM

```
1  //speed toggle
2  module toggle(input logic [2:0]data,
3                                      input logic clk,
4                                      output logic y);
5
6          typedef enum logic {stop, go} statetype;
7          statetype state, nextstate;
8
9          always_ff @(posedge clk)
10                 state <= nextstate;
11
12         always_comb
13                 case(state)
14                         stop: if(data == 3'b001) nextstate = go;
15                                 else nextstate = stop;
16                         go: if(data == 3'b000) nextstate = stop;
17                                 else nextstate = go;
18                         default: nextstate = stop;
19                 endcase
20
21         assign y = (state == go);
22
23 endmodule
```

A.2.2 Left Decoder

```
1  //left decoder
2  module l_decoder(input logic [2:0]data,
3                                  output logic up, down);
4
5          always_comb
6                  //up logic
7                  if (data == 3'b010) up = 1'b1;
8                  else if (data == 3'b100) up = 1'b1;
9                  else up = 1'b0;
10
11         always_comb
12                 //down logic
13                 if (data == 3'b011) down = 1'b1;
14                 else if (data == 3'b101) down = 1'b1;
15                 else down = 1'b0;
16
17
18 endmodule
```

A.2.3 Right Decoder

```
1  //right decoder
2  module r_decoder(input logic [2:0]data,
3                                  output logic up, down);
4
5          always_comb
6                  //up logic
7                  if (data == 3'b010) up = 1'b1;
8                  else if (data == 3'b101) up = 1'b1;
9                  else up = 1'b0;
10
11         always_comb
12                 //down logic
13                 if (data == 3'b011) down = 1'b1;
14                 else if (data == 3'b100) down = 1'b1;
15                 else down = 1'b0;
16
17
18 endmodule
```

A.2.4 Speed FSM

```
//Speed State Machine
module speed(input logic clk, reset, up, down,
                                output logic [2:0]y);

                typedef enum logic [2:0] {S5, S4, S3, S2, S1, S0} statetype;
                statetype state, nextstate;

                always_ff @(posedge clk)
                        if (reset) state = S3;
                        else state <= nextstate;

                //nextstate logic
                always_comb
                        case (state)
                                S5: if (down)             nextstate = S4;
                                      else                      nextstate = S5;
                                S4: if (up)               nextstate = S5;
                                      else if (down) nextstate = S3;
                                      else                      nextstate = S4;
                                S3: if (up)               nextstate = S4;
                                      else if (down) nextstate = S2;
                                      else                      nextstate = S3;
                                S2: if (up)               nextstate = S3;
                                      else if (down) nextstate = S1;
                                      else                      nextstate = S2;
                                S1: if (up)               nextstate = S2;
                                      else if (down) nextstate = S0;
                                      else                      nextstate = S1;
                                S0: if (up)               nextstate = S1;
                                      else                      nextstate = S0;
                                default: nextstate = S3;
                        endcase

                //output logic
                always_comb
                        case(state)
                                S5: y = 3'b010;
                                S4: y = 3'b001;
                                S3: y = 3'b000;
                                S2: y = 3'b100;
                                S1: y = 3'b101;
                                S0: y = 3'b110;
                                default: y = 3'b100;
                        endcase


endmodule
```

A.3 Motor Controller HDL

A.3.1 Finite State Machine 1/3 Divider

```
module Divide1(input logic clk,
                    input logic reset,
                    output logic y);

    typedef enum logic [1:0] {S0, S1, S2} statetype;
    logic [1:0] state, nextstate;

    always_ff@(posedge clk, posedge reset)
        if (reset)      state <= S0;
        else             state <= nextstate;

    always_comb
        case(state)
            S0:         nextstate <= S1;
            S1:         nextstate <= S2;
            S2:         nextstate <= S0;
            default:    nextstate <= S0;
        endcase
    assign y = (state == S0);
endmodule
```

A.3.2 Finite State Machine 2/3 Divider

```
module Divide2(input logic clk,
                    input logic reset,
                    output logic y);

    typedef enum logic [1:0] {S0, S1, S2} statetype;
    logic [1:0] state, nextstate;

    always_ff@(posedge clk, posedge reset)
        if (reset)      state <= S0;
        else             state <= nextstate;

    always_comb
        case(state)
            S0:         nextstate <= S1;
            S1:         nextstate <= S2;
            S2:         nextstate <= S0;
            default:    nextstate <= S0;
        endcase
    assign y = (state == S0 | state == S1);
endmodule
```

A.3.3 PWM Clock Counter

```
module counter #(parameter N = 10)
                                        (input  logic clk,
                                         input  logic reset,
                                         output logic [N-1:0] q);

        always_ff @(posedge clk, posedge reset)
                if(reset)       q <= 0;
                else                    q <= q + 1;
endmodule
```

A.3.4 Speed Multiplexer

```
module speedMux(input    logic [1:0] speed,
                                     input  logic   slow, med, fast,
                                     output logic   q);

        assign q = speed[1] ? (speed[0] ? fast : fast)
                                                    : (speed[0] ? med  : slow);
endmodule
```

A.3.5 Go Multiplexer

```
module goMux(input       logic go,
                                 input  logic   speed, stop,
                                 output logic   q);

        assign q = go ? speed : stop;
endmodule
```

B Simulation Files

B.0.1 Robot

```
quietly WaveActivateNextPane {} 0
add wave -noupdate /Robot/reset
add wave -noupdate /Robot/clock
add wave -noupdate /Robot/b2v_inst/clock9
add wave -noupdate /Robot/data
add wave -noupdate -divider IR_module
add wave -noupdate /Robot/b2v_inst/b2v_inst2/state
add wave -noupdate /Robot/b2v_inst/b2v_inst21/data
add wave -noupdate /Robot/b2v_inst/b2v_inst21/inverse
add wave -noupdate -radix unsigned /Robot/b2v_inst/rcount
add wave -noupdate -radix unsigned /Robot/b2v_inst/scount
add wave -noupdate -divider Controller
add wave -noupdate /Robot/b2v_inst2/clk
add wave -noupdate /Robot/b2v_inst2/data
add wave -noupdate /Robot/b2v_inst2/l
add wave -noupdate /Robot/b2v_inst2/r
add wave -noupdate -divider MC_out
add wave -noupdate /Robot/b2v_inst3/clk
add wave -noupdate /Robot/LftMotFwd
add wave -noupdate /Robot/LftMotRev
add wave -noupdate /Robot/RtMotFwd
add wave -noupdate /Robot/RtMotRev
TreeUpdate [SetDefaultTree]
force sim:/Robot/reset 1 0, 0 5
force -freeze sim:/Robot/b2v_inst/clock9 1 0, 0 {5 ps} -r 10
force -freeze sim:/Robot/b2v_inst2/clk 1 0, 0 {5 ps} -r 10
force -freeze sim:/Robot/b2v_inst3/clk 1 0, 0 {5 ps} -r 10
force sim:/Robot/data 0 0, 1 10, 0 30, 1 970, 0 1420
run 1430
force sim:/Robot/data 1 10, 0 50, 1 70, 0 220, 1 240, 0 290, 1 310, 0 370, 1 390, 0 430, 1 450, 0 610, 1
      630, 0 680, 1 700, 0 740
run 1110
force sim:/Robot/data 1 10, 0 160, 1 190, 0 230, 1 250, 0 410, 1 430, 0 590, 1 610, 0 760, 1 780, 0 820,
      1 850, 0 1000, 1 1020, 0 1180, 1 1200, 0 5180, 1 6120, 0 6340, 1 6360
run 6800
```

B.1 IR Reader Simulations

```
quietly virtual signal -install /irModule { (context /irModule )&{data0 , data1 , data2 , data3 , data4
      , data5 , data6 , data7 }} dat
add wave -noupdate /irModule/reset
add wave -noupdate /irModule/clock
add wave -noupdate /irModule/clock9
add wave -noupdate /irModule/data
add wave -noupdate /irModule/funout
add wave -noupdate /irModule/bi
add wave -noupdate /irModule/b2v_inst2/state
add wave -noupdate -label data /irModule/dat
add wave -noupdate /irModule/inverseD
add wave -noupdate /irModule/dout
add wave -noupdate -radix unsigned /irModule/rcount
add wave -noupdate -radix unsigned /irModule/scount
add wave -noupdate /irModule/valid
force sim:/irModule/reset 1 0, 0 5
force -freeze sim:/irModule/clock9 1 0, 0 {5 ps} -r 10
force sim:/irModule/data 0 0, 1 10, 0 30, 1 970, 0 1420
run 1430
force sim:/irModule/data 1 10, 0 50, 1 70, 0 220, 1 240, 0 290, 1 310, 0 370, 1 390, 0 430, 1 450, 0
      610, 1 630, 0 680, 1 700, 0 740
run 750
force sim:/irModule/data 1 10, 0 160, 1 190, 0 230, 1 250, 0 410, 1 430, 0 590, 1 610, 0 760, 1 780, 0
      820, 1 850, 0 1000, 1 1020, 0 1180, 1 1200, 0 5180, 1 6120, 0 6340, 1 6360
run 6500
```

B.1.1 IR Code to Primary Controller Translator Simulation

```
add wave -noupdate -radix unsigned /IRtoControllerTranslator/data
add wave -noupdate -radix unsigned /IRtoControllerTranslator/intcode
force sim:/IRtoControllerTranslator/data 0 0, 01000101 10, 01000100 20, 00001001 30, 00010101 40,
      01000011 50, 01000000 60, 01100101 70
run
```

B.1.2 Falling Edge Detection Simulation

```
add wave -position end  sim:/fallingedgedetector/reset
add wave -position end  sim:/fallingedgedetector/clock
add wave -position end  sim:/fallingedgedetector/Data
add wave -position end  sim:/fallingedgedetector/trigger
```

```
add wave -position end  sim:/fallingedgedetector/DFF_inst
force -freeze sim:/fallingedgedetector/reset 1 0, 0 20
force -freeze sim:/fallingedgedetector/clock 1 0, 0 {5 ps} -r 10
force sim:/fallingedgedetector/Data 1 0, 0 15, 1 30, 0 40, 1 50, 0 65, 1 70, 0 80, 1 90, 0 93
run 90
```

B.1.3 Positive Edge Trigger Simulation

```
add wave -position end  sim:/PosEdgTrig/reset
add wave -position end  sim:/PosEdgTrig/clock
add wave -position end  sim:/PosEdgTrig/Data
add wave -position end  sim:/PosEdgTrig/trigger
add wave -position end  sim:/PosEdgTrig/DFF_inst
force -freeze sim:/PosEdgTrig/reset 1 0, 0 20
force -freeze sim:/PosEdgTrig/clock 1 0, 0 {5 ps} -r 10
force sim:/PosEdgTrig/Data 1 0, 0 15, 1 30, 0 40, 1 50, 0 65, 1 70, 0 80, 1 90, 0 93
run 90
```

B.1.4 Bit Interpreter

```
add wave -position end  sim:/bitinterpreter/data
add wave -position end  sim:/bitinterpreter/inum
force sim:/bitinterpreter/data 0 0, 100 10, 1111 20, 10111 30, 11011 40, 1 50, 1101 60, 101 70, 10110 80
run 90
```

B.1.5 Eight bit Latch

```
add wave -position end  sim:/latch/clk
add wave -position end  sim:/latch/d
add wave -position end  sim:/latch/q
force sim:/latch/d 0 0, 10100101 10, 0 20
force sim:/latch/clk 0 0, 1 10, 0 20, 1 30
run
```

B.1.6 Validate Sequence Simulation

```
add wave -position end  sim:/validate/data
add wave -position end  sim:/validate/inverse
add wave -position end  sim:/validate/isValid
force sim:/validate/data 0 0, 10100101 10, 11111111 20, 00001111 40
force sim:/validate/inverse 0 0, 01011010 10, 00010000 20, 11110000 30
run 50
```

B.2 Main Controller Simulations

```
vsim -gui work.Controller

add wave clk
add wave data
add wave l
add wave r

force clk 1 0, 0 -r 5
force data 0 @ 0, 1 @ 10, 10 @ 20, 11 @ 35, 100 @ 60, 101 @ 80, 0 @ 100, 1 @ 110

run 120
```

B.2.1 Toggle FSM

```
vsim -gui work.toggle

add wave *

force clk 1 0, 0 -r 10
force data 0 @ 0, 1 @ 10, 10 @ 20, 11 @ 30, 0 @ 40, 100 @ 50, 101 @ 60, 110 @ 70, 111 @ 80, 1 @ 90

run 110
```

B.2.2 Left Decoder

```
vsim -gui work.l_decoder

add wave *

force data 0 @ 0, 1 @ 10, 10 @ 20, 11 @ 30, 100 @ 40, 101 @ 50, 110 @ 60, 111 @ 70

run 80
```

B.2.3 Right Decoder

```
vsim -gui work.r_decoder

add wave *

force data 0 @ 0, 1 @ 10, 10 @ 20, 11 @ 30, 100 @ 40, 101 @ 50, 110 @ 60, 111 @ 70

run 80
```

B.2.4 Speed FSM

```
vsim -gui work.speed

add wave *

force clk 1 0, 0 -r 10
force reset 1 @ 0, 0 @ 10, 1 @ 110
force up 0 @ 0, 1 @ 10, 0 @ 50
force down 0 @ 0, 1 @ 50

run 140
```

B.3 Motor Controller Simulations

```
vsim -gui work.MotorController

add wave clk
add wave reset
add wave gnd
add wave left
add wave right
add wave leftFwdOut
add wave leftRvrsOut
add wave rightFwdOut
add wave rightRvrsOut

force clk 1 0, 0 -r 5
force reset 1 @ 0, 0 @ 5
force gnd 0
force left 0000 @ 0, 0001 @ 20000, 0011 @ 40000, 0101 @ 60000, 1001 @ 80000, 1011 @ 100000, 1101 @
    120000, 1100 @ 140000
force right 0000 @ 0, 0001 @ 20000, 0011 @ 40000, 0101 @ 60000, 1001 @ 80000, 1011 @ 100000, 1101 @
    120000, 1100 @ 140000

run 160000
```

B.3.1 Pulse-Width Modulator

```
vsim -gui work.PWM

add wave clock
add wave reset
add wave low
add wave med
add wave high

force reset 1 @ 5, 0 @ 10
force clock 1 0, 0 -r 5
run 50000
```

B.3.2 1/3 FSM Divider

```
vsim -gui work.Divide1

add wave clk
add wave reset
add wave y

force reset 0
force clk 1 0, 0 -r 5
run 120
```

B.3.3 2/3 FSM Divider

```
vsim -gui work.Divide2

add wave clk
add wave reset
add wave y

force reset 0
force clk 1 0, 0 -r 5
run 120
```

B.3.4 PWM Clock Counter

```
vsim -gui work.counter

add wave clk
add wave reset
add wave q

force reset 1 @ 5, 0 @ 10
force clk 1 0, 0 -r 5
run 1200
```

B.3.5 Speed Multiplexer

```
vsim -gui work.speedMux

add wave speed
add wave slow
add wave med
add wave fast
add wave q

force slow 0 @ 0, 1 @ 50
force med 0 @ 0, 1 @ 150
force fast 0 @ 0, 1 @ 250, 0 @ 300, 1 @ 350
force speed 00 @ 0, 01 @ 100, 10 @ 200, 11 @ 300

run 400
```

B.3.6 Go Multiplexer

```
vsim -gui work.goMux

add wave speed
add wave go
add wave stop
add wave q

force speed 1
force go 0 @ 0, 1 @ 200
force stop 0

run 400
```

B.3.7 Direction Multiplexer

```
vsim -gui work.demux

add wave Select
add wave Data
add wave OutLow
add wave OutHigh

force Select 0 @ 0, 1 @ 100, 0 @ 200, 1 @ 300
force Data 0 @ 0, 1 @ 200

run 400
```