ELEX 7660 Team #1 Report

Neil Dehoog
Joshua Lang
Set 6T

Automated Fishing Hook

9th April 2018

Introduction and Objectives

This project was initiated as a means of increasing the efficiency of downrigger fishing devices. The efficiency of these devices is directly related to their ability to maintain a set distance off of the bottom of the ocean floor. We made a proof of concept model that could achieve a fixed distance above a floating reference point. To do this we utilized the following hardware:

- Close range IR sensor to prevent over reeling in
- Ultrasonic distance sensor to gauge ground distance
- Stepper motor to accurately control line length
- SPST switch to toggle between manual mode and autonomous
- 4x4 keypad to control line level manually
- 7-segment display to show which key is being utilized
- Cyclone IV FPGA board as a control system

For the construction of the control system we programmed the FPGA board using Quartus v17.1 and monitored key sensor outputs using the Quartus signaltap addon.

Achievement

We have achieved all of our set objectives for the automated fishing hook:

- Using the code we written for the keypad to control the motor in manual operations of the fishing hook.
- Used an ultrasonic distance sensor to measure distance to the artificial ground.
- In order to reel in the fishing line we used a stepper motor for the controllability.
- An optic sensor was used to stop the reel going to far.
- An basic SPST switch was used to control operation modes

Operation

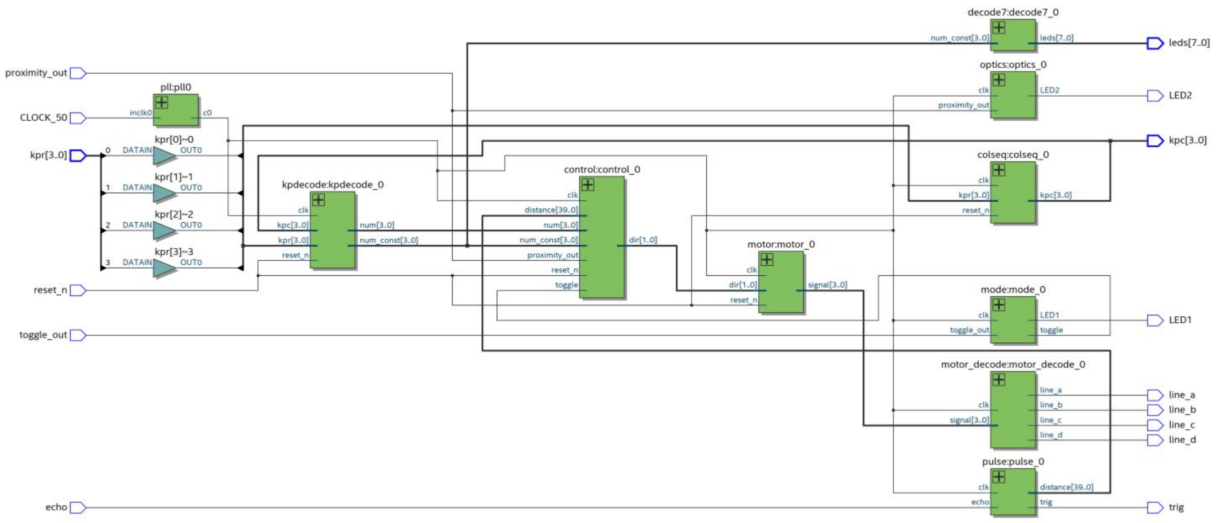

Figure 1: Block diagram of the overall system

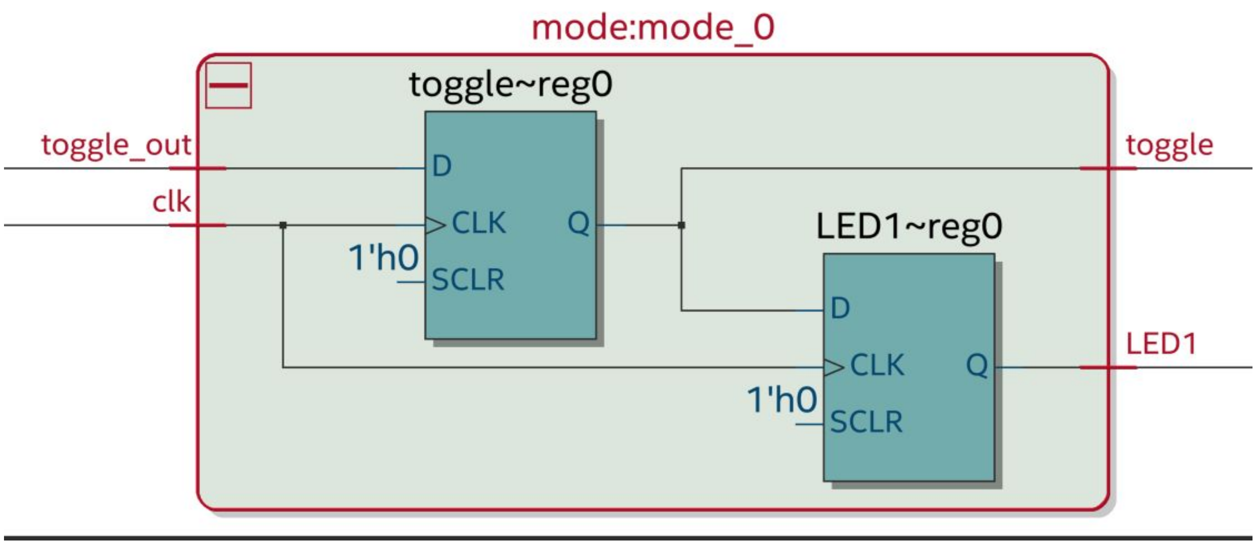

Figure 2: Block diagram of Mode.sv

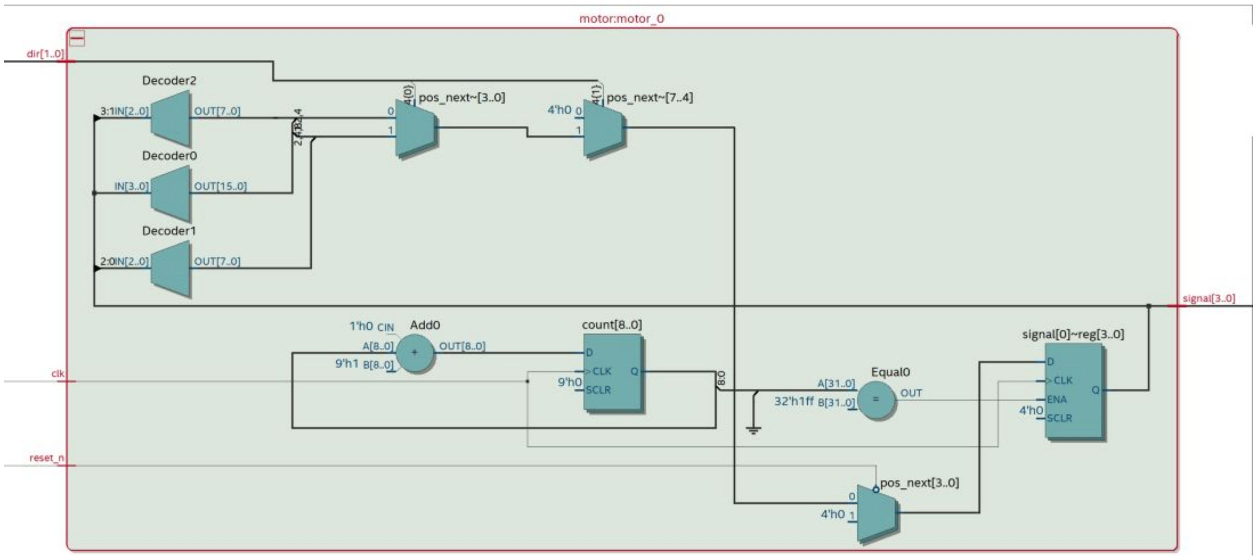

Figure 3: Block diagram of Motor.sv

We had problems initially with the stepper motor because neither of us knew how it operated but debugging using the oscilloscope eventually gave us the right waveform. The next obstacle to figure out the timing for which the motor to operate. We determined that the highest division for the pll is 25000 with took 50 MHz clock down to 2 KHz signal for or motor. After additional testing for other modules we concluded that an 20 KHz signal with an clock divider for the motor resulted for optimum operation.

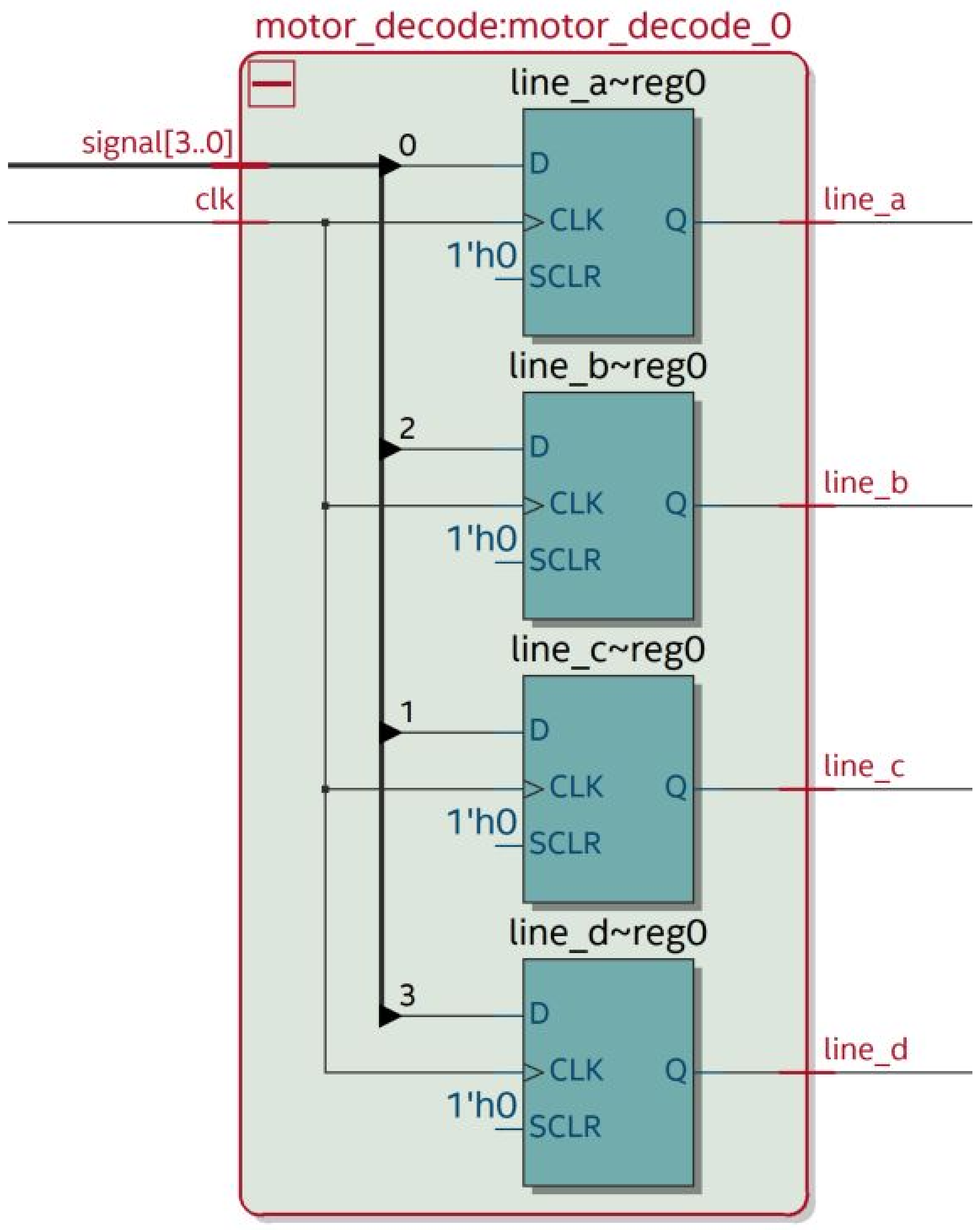

Figure 4: Block diagram Motor_decode.sv

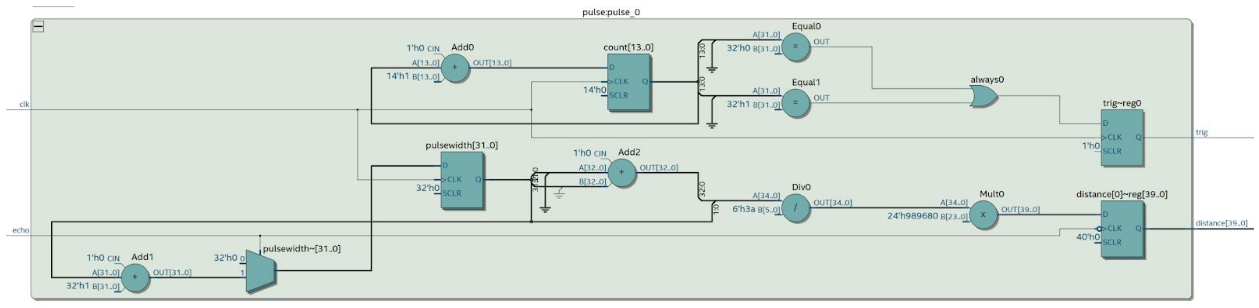

Figure 5: Block diagram for the pulse.sv

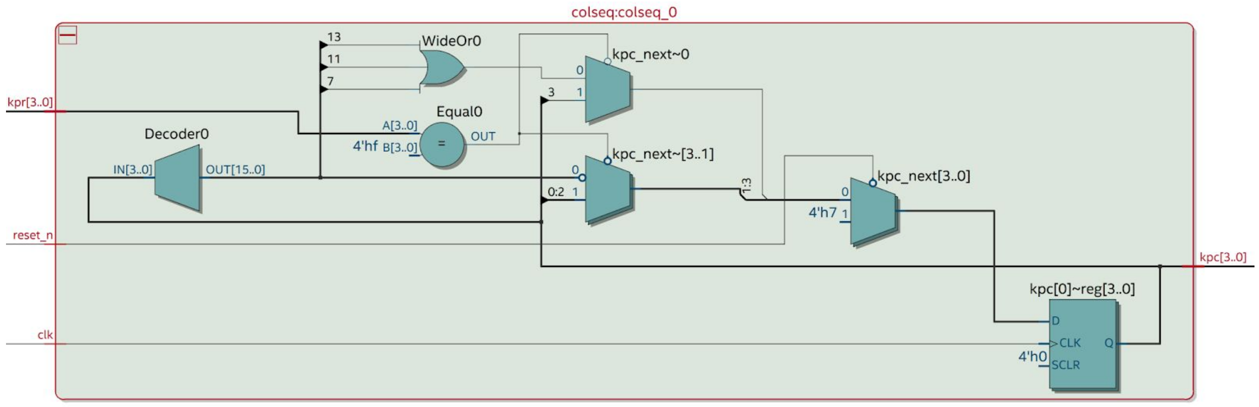

Figure 6: Block diagram of colseq.sv

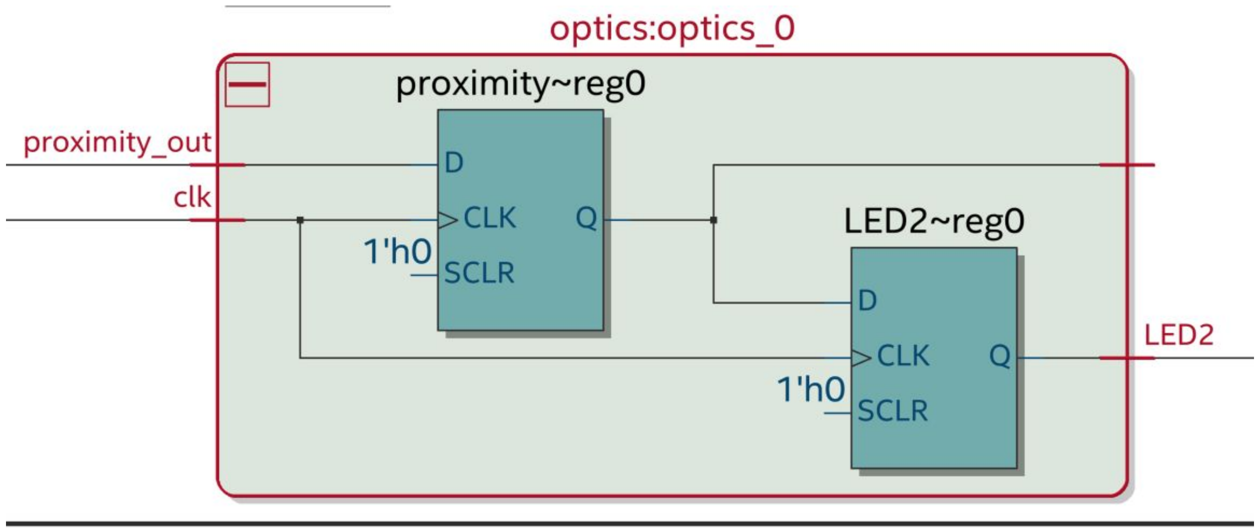

Figure 7: Block diagram for optics.sv

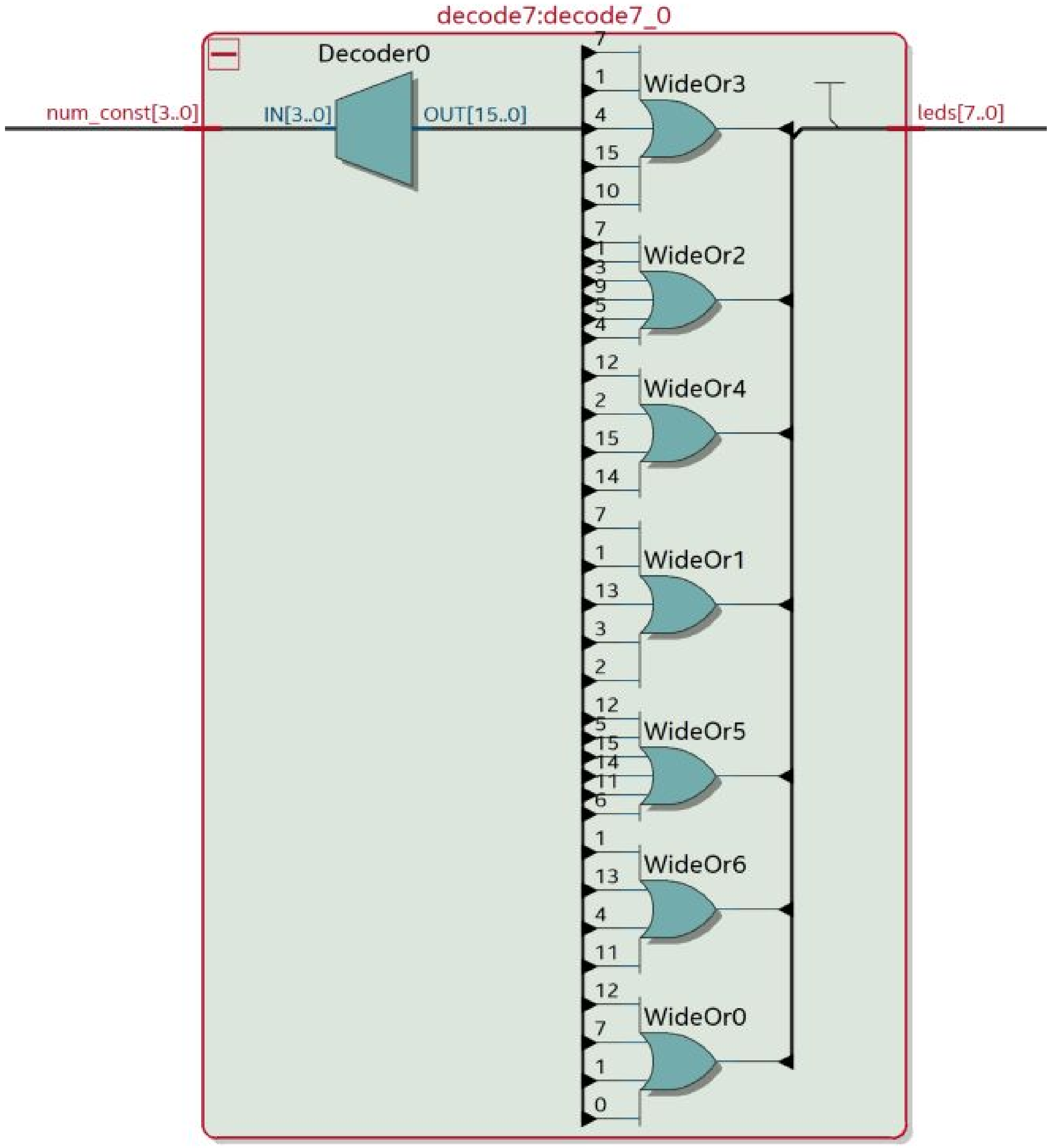

Figure 8: Block diagram for decode7.sv

Resources

HC-SR04 Datasheet

L298N Motor Driver Datasheet

Stepper Motor Datasheet

Conclusion

Our system functioned as expected. The IR sensor prevented the motor from reeling in too far and damaging the chassis. The ultrasonic sensor maintained distance readings with approximately 1 cm of resolution. The 4x4 keypad, switch and 7-segment display all performed within expectation. Finally, the stepper motor was able to accurately raise or lower the line in accordance with changing ground depth. All of these pieces of hardware working together produced a system that maintained a hook at a set distance above the ground, even with changing ground levels, while preventing damage to the system.

Appendix A: Code

The following section which includes all the code used in this project.

SV Module: Fish.sv Top Level

```
module fish ( output logic [3:0] kpc,   // column select, active-low
            (* altera_attribute = "-name WEAK_PULL_UP_RESISTOR ON" *)
            input logic  [3:0] kpr,     // rows, active-low w/ pull-ups
            output logic [7:0] leds,    // active-low LED segments
            output logic [3:0] ct,      // " digit enables
                output logic toggle_in,     // Input to the toggle switch.
Always High
                input logic toggle_out,     // Output from the toggle switch.
Requires a pull up or down resistor
            input logic reset_n, CLOCK_50,
                output logic proximity_in,
                input logic proximity_out,
                output logic LED1,
                output logic LED2,
                output logic LED3,
                output logic LED4,
                output logic LED5,
                output logic LED6,
                output logic LED7,
                output logic LED8,
                output logic line_a,
                output logic line_b,
                output logic line_c,
                output logic line_d,
                input logic PW,
                output logic trig,
                input logic echo);

    logic clk ;                // 2kHz clock for keypad scanning
    logic kphit ;              // a key is pressed
    logic [3:0] num ;           // value of pressed key
    logic [3:0] num_const = 0;  // stored value of the last pressed key
    logic toggle;                   // logic value of the output from the
toggle switch
    logic proximity;                // logic value of the input from the
proximity sensor
    logic [1:0] dir = 2'b00;    // Motor direction variable
    logic [3:0] signal;         // Motor driver control signal
    logic [39:0] distance;
    logic [31:0] dist_test;
    logic [31:0] leng_test;


    assign ct = { {3{1'b0}}, 1'b1 } ;
    assign toggle_in = 0;
    assign proximity_in = 1;
    pll pll0 ( .inclk0(CLOCK_50), .c0(clk) ) ;
    decode7 decode7_0   (.num_const, .leds);
    kpdecode    kpdecode_0  (.kpc, .kpr, .kphit, .num, .num_const, .clk,
.reset_n);
    colseq  colseq_0    (.kpr, .clk, .reset_n, .kpc);
    mode mode_0 (.toggle_out, .toggle, .clk, .LED1);
    optics optics_0 (.LED2, .proximity_out, .proximity, .clk);
    motor motor_0 (.clk, .reset_n, .dir, .signal);
    motor_decode motor_decode_0 (.signal, .clk, .line_a, .line_b, .line_c,
.line_d);
    control control_0 (.clk, .toggle, .reset_n, .num, .num_const, .distance,
.dir, .proximity_out, .dist_test, .leng_test);
    pulse pulse_0 (.clk, .trig, .echo, .distance);

endmodule

// megafunction wizard: %ALTPLL%
// ...
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
// ...

module pll ( inclk0, c0);

        input     inclk0;
        output    c0;

        wire [0:0] sub_wire2 = 1'h0;
        wire [4:0] sub_wire3;
        wire  sub_wire0 = inclk0;
        wire [1:0] sub_wire1 = {sub_wire2, sub_wire0};
        wire [0:0] sub_wire4 = sub_wire3[0:0];
        wire  c0 = sub_wire4;

        altpll altpll_component ( .inclk (sub_wire1), .clk
          (sub_wire3), .activeclock (), .areset (1'b0), .clkbad
          (), .clkena ({6{1'b1}}), .clkloss (), .clkswitch
          (1'b0), .configupdate (1'b0), .enable0 (), .enable1 (),
          .extclk (), .extclkena ({4{1'b1}}), .fbin (1'b1),
          .fbmimicbidir (), .fbout (), .fref (), .icdrclk (),
          .locked (), .pfdena (1'b1), .phasecounterselect
          ({4{1'b1}}), .phasedone (), .phasestep (1'b1),
          .phaseupdown (1'b1), .pllena (1'b1), .scanaclr (1'b0),
          .scanclk (1'b0), .scanclkena (1'b1), .scandata (1'b0),
          .scandataout (), .scandone (), .scanread (1'b0),
          .scanwrite (1'b0), .sclkout0 (), .sclkout1 (),
          .vcooverrange (), .vcounderrange ());

        defparam
                altpll_component.bandwidth_type = "AUTO",
                altpll_component.clk0_divide_by = 250,
                altpll_component.clk0_duty_cycle = 50,
                altpll_component.clk0_multiply_by = 1,
                altpll_component.clk0_phase_shift = "0",
                altpll_component.compensate_clock = "CLK0",
                altpll_component.inclk0_input_frequency = 20000,
                altpll_component.intended_device_family = "Cyclone IV E",
                altpll_component.lpm_hint = "CBX_MODULE_PREFIX=lab1clk",
                altpll_component.lpm_type = "altpll",
                altpll_component.operation_mode = "NORMAL",
                altpll_component.pll_type = "AUTO",
                altpll_component.port_activeclock = "PORT_UNUSED",
                altpll_component.port_areset = "PORT_UNUSED",
                altpll_component.port_clkbad0 = "PORT_UNUSED",
                altpll_component.port_clkbad1 = "PORT_UNUSED",
                altpll_component.port_clkloss = "PORT_UNUSED",
                altpll_component.port_clkswitch = "PORT_UNUSED",
                altpll_component.port_configupdate = "PORT_UNUSED",
                altpll_component.port_fbin = "PORT_UNUSED",
                altpll_component.port_inclk0 = "PORT_USED",
                altpll_component.port_inclk1 = "PORT_UNUSED",
                altpll_component.port_locked = "PORT_UNUSED",
                altpll_component.port_pfdena = "PORT_UNUSED",
                altpll_component.port_phasecounterselect = "PORT_UNUSED",
                altpll_component.port_phasedone = "PORT_UNUSED",
                altpll_component.port_phasestep = "PORT_UNUSED",
                altpll_component.port_phaseupdown = "PORT_UNUSED",
                altpll_component.port_pllena = "PORT_UNUSED",
                altpll_component.port_scanaclr = "PORT_UNUSED",
                altpll_component.port_scanclk = "PORT_UNUSED",
                altpll_component.port_scanclkena = "PORT_UNUSED",
                altpll_component.port_scandata = "PORT_UNUSED",
                altpll_component.port_scandataout = "PORT_UNUSED",
                altpll_component.port_scandone = "PORT_UNUSED",
                altpll_component.port_scanread = "PORT_UNUSED",
                altpll_component.port_scanwrite = "PORT_UNUSED",
                altpll_component.port_clk0 = "PORT_USED",
                altpll_component.port_clk1 = "PORT_UNUSED",
                altpll_component.port_clk2 = "PORT_UNUSED",
                altpll_component.port_clk3 = "PORT_UNUSED",
                altpll_component.port_clk4 = "PORT_UNUSED",
                altpll_component.port_clk5 = "PORT_UNUSED",
                altpll_component.port_clkena0 = "PORT_UNUSED",
                altpll_component.port_clkena1 = "PORT_UNUSED",
                altpll_component.port_clkena2 = "PORT_UNUSED",
                altpll_component.port_clkena3 = "PORT_UNUSED",
                altpll_component.port_clkena4 = "PORT_UNUSED",
                altpll_component.port_clkena5 = "PORT_UNUSED",
                altpll_component.port_extclk0 = "PORT_UNUSED",
                altpll_component.port_extclk1 = "PORT_UNUSED",
                altpll_component.port_extclk2 = "PORT_UNUSED",
                altpll_component.port_extclk3 = "PORT_UNUSED",
                altpll_component.width_clock = 5;


endmodule
```

SV Module: Control.sv

```
module control (input logic clk,
                input logic toggle,
                input logic proximity_out,
                input logic reset_n,
                input logic [3:0] num,
                input logic [3:0] num_const,
                input logic [39:0] distance,
                output logic [1:0] dir,
                output logic [31:0] dist_test,
                output logic [31:0] leng_test);

        logic reset = 1;                        // Required to reel the wire all
the way in on reset
        logic [31:0] max_length = 50;   // Max length of the wire
        logic [1:0] dir_next;           // Next motor direction
        logic [31:0] test = 0;          // Test clock
        logic [31:0] length;            // Length of the wire currently out
        logic [31:0] length_next = 0;



    always_comb begin

        dir_next = dir;
        length_next = length;

        dist_test = distance/10000000;
        leng_test = length/10000000;

        // Manual mode
        if (toggle) begin

            unique case(num)
                10: dir_next = 2'b10;
                11: dir_next = 2'b11;
                default: dir_next = 2'b00;
            endcase

        end


        // Automatic mode
        if (!toggle) begin

            if ((length/10000000) < (distance/10000000))
                dir_next = 2'b11;

            if ((length/10000000) > (distance/10000000))
                dir_next = 2'b10;

            if ((length/10000000) == (distance/10000000))
                dir_next = 2'b00;

        end
        // Length Adjustments
        unique case(dir)
            2'b00: length_next = length_next;
            2'b10: length_next = length_next - 500;
            2'b11: length_next = length_next + 500;
        endcase

        if (!proximity_out)
            length_next = 0;

        if (length_next[31])
            length_next = 0;

    end


    always_ff @(posedge clk) begin

        if (!proximity_out && dir_next == 2'b10)
            dir <= 2'b00;
        else if ((length/10000000 > max_length) && (dir_next == 2'b11))
            dir <= 2'b00;
        else
            dir <= dir_next;

        length <= length_next;


    end

endmodule
```

SV Module: Colseq.sv

```
module colseq   (input logic [3:0] kpr,
                input logic clk,
                input logic reset_n,
                output logic [3:0] kpc);

    logic [3:0] kpc_next;

    always_comb begin

        // Check for a reset press
        if(!reset_n)
            kpc_next = 4'b0111;

        // Hold the current column if any rows are low
        else if (kpr != 4'b1111)
            kpc_next = kpc;

        // Change the low column to the next in sequence
        else
```

```
            case(kpc)
                4'b0111: kpc_next = 4'b1011;
                4'b1011: kpc_next = 4'b1101;
                4'b1101: kpc_next = 4'b1110;
                4'b1110: kpc_next = 4'b0111;
                default: kpc_next = 4'b0111;
            endcase
    end

    always @(posedge clk) begin
        kpc <= kpc_next;
    end

endmodule
```

SV Module: Decode7.sv

```
module decode7  (input logic [3:0] num_const,
                output logic [7:0] leds);

    always_comb begin
        unique case (num_const) // Segments are active low
            0: leds = 8'b1100_0000; // Segments for a 0
            1: leds = 8'b1111_1001; // Segments for a 1
            2: leds = 8'b1010_0100; // Segments for a 2
            3: leds = 8'b1011_0000; // Segments for a 3
            4: leds = 8'b1001_1001; // Segments for a 4
            5: leds = 8'b1001_0010; // Segments for a 5
            6: leds = 8'b1000_0010; // Segments for a 6
            7: leds = 8'b1111_1000; // Segments for a 7
            8: leds = 8'b1000_0000; // Segments for a 8
            9: leds = 8'b1001_0000; // Segments for a 9
            10: leds = 8'b1000_1000; // Segments for a A
            11: leds = 8'b1000_0011; // Segments for a B
            12: leds = 8'b1100_0110; // Segments for a C
            13: leds = 8'b1010_0001; // Segments for a D
            14: leds = 8'b1000_0110; // Segments for an E
            15: leds = 8'b1000_1110; // Segments for a F
            default: leds = 8'b1100_0000; // Defaut is 0
        endcase
    end

endmodule
```

SV Module: kpdecode.sv

```
module kpdecode (input logic [3:0] kpc,
                input logic [3:0] kpr,
                input logic clk, reset_n,
                output logic kphit,
                output logic [3:0] num,
                output logic [3:0] num_const);

    always_comb begin
        // Selects the correct LED output
    // for a given row and column input
    unique case (kpr) // Active low
        4'b1110:    unique case (kpc) // Active low
                        4'b1110: num = 13;
                        4'b1101: num = 15;
                        4'b1011: num = 0;
                        4'b0111: num = 14;
                        default: num = 99;
                    endcase

        4'b1101:    unique case (kpc) // Active low
                        4'b1110: num = 12;
                        4'b1101: num = 9;
                        4'b1011: num = 8;
                        4'b0111: num = 7;
                        default: num = 99;
                    endcase

        4'b1011:    unique case (kpc) // Active low
                        4'b1110: num = 11;
                        4'b1101: num = 6;
                        4'b1011: num = 5;
                        4'b0111: num = 4;
                        default: num = 99;
                    endcase

        4'b0111:    unique case (kpc) // Active low
                        4'b1110: num = 10;
                        4'b1101: num = 3;
                        4'b1011: num = 2;
                        4'b0111: num = 1;
                        default: num = 99;
                    endcase

        default:    num = 99;
    endcase

    // Modifies kbhit based on if a key is
    // being pressed or not
    if (kpr == 4'b1111)
        kphit = 0;
    else
        kphit = 1;

end

always_ff @(posedge clk) begin
    if (kphit)
        num_const <= num;
    else if (!reset_n)
        num_const <= 0;
    else
        num_const <= num_const;
end
```

```
endmodule
```

SV Module: mode.sv

```
module mode (input logic toggle_out,
                input logic clk,
                output logic LED1,
                output logic toggle);

    always_ff @(posedge clk) begin

        toggle <= toggle_out;
        LED1 <= toggle;

    end

endmodule
```

SV Module: motor.sv

```
module motor(input logic clk,
                input logic reset_n,
                input logic [1:0] dir,
                output logic [3:0] signal);


    localparam pos_stop = 0;
    localparam pos_1 = 1;
    localparam pos_2 = 2;
    localparam pos_3 = 4;
    localparam pos_4 = 8;

    logic[3:0] pos_next;
    logic [8:0] count;

    always_comb begin

        if(!reset_n) begin

            pos_next = pos_stop;

        end

        else begin

            if (dir[1]) begin

                if (dir [0]) begin
                    unique case (signal)
                        pos_stop: pos_next = pos_1;
                        pos_1: pos_next = pos_2;
                        pos_2: pos_next = pos_3;
                        pos_3: pos_next = pos_4;
                        pos_4: pos_next = pos_1;
                        default: pos_next = '0;
                    endcase
                end

                else begin

                    unique case (signal)
                        pos_stop: pos_next = pos_4;
                        pos_1:  pos_next = pos_4;
                        pos_2: pos_next = pos_1;
                        pos_3: pos_next = pos_2;
                        pos_4: pos_next = pos_3;
                        default: pos_next = '0;
                    endcase

                end

            end

            else begin

                    pos_next = pos_stop;

            end

        end

    end


    always_ff @(posedge clk) begin

        if (count == 511)
            signal <= pos_next;
        else
            signal <= signal;

        count <= count + 1;

    end

endmodule
```

SV Module: motor_decode.sv

```
module motor_decode(input logic [3:0] signal,
                    input logic clk,
                    output logic line_a,
                    output logic line_b,
                    output logic line_c,
                    output logic line_d);


    always_ff @(posedge clk) begin
        line_a <= signal[0];
        line_b <= signal[2];
        line_c <= signal[1];
        line_d <= signal[3];
    end

endmodule
```

SV Module: optics.sv

```
module optics   (input logic proximity_out,
                    input logic clk,
                    output logic proximity,
                    output logic LED2);


    always_ff @(posedge clk) begin

        proximity <= proximity_out;
        LED2 <= proximity;

    end

endmodule
```

SV Module: pulse.sv

```
module pulse (input logic clk,
                        output logic trig,
                        input logic echo,
                        output logic [39:0] distance);

    logic [13:0] count= '0;
    logic [31:0] pulsewidth = '0;

    always_ff @(posedge clk) begin

        if (count == 0 || count == 1)
            trig <= '1;
        else
            trig <= '0;

        count <= count + 1;

        if (echo)
            pulsewidth <= pulsewidth + 1;

        else if (~echo)
            pulsewidth <= 0;

    end


    always_ff @(negedge echo) begin
        distance <= pulsewidth * 5 / 58 * 10000000;
    end
```

```
endmodule
```