ELEX 7660
Project Report

Friendo Finder

Vladimir Cvjetan

Kyle Soroczka

ELEX 7660 Winter 2017

“Friendo Finder”

Group 1

3/31/17

Contents

Figures

Overview

For the ELEX 7660 final project we designed and built the Friendo Finder, which detects objects within a range of 20-70 cm. It uses a stepper motor with an infrared sensor mounted on top to sweep in a 180 degree arc to detect objects. If an object is detected, the distance it was detected and the angle it was detected at is displayed on a 16x2 LCD screen.

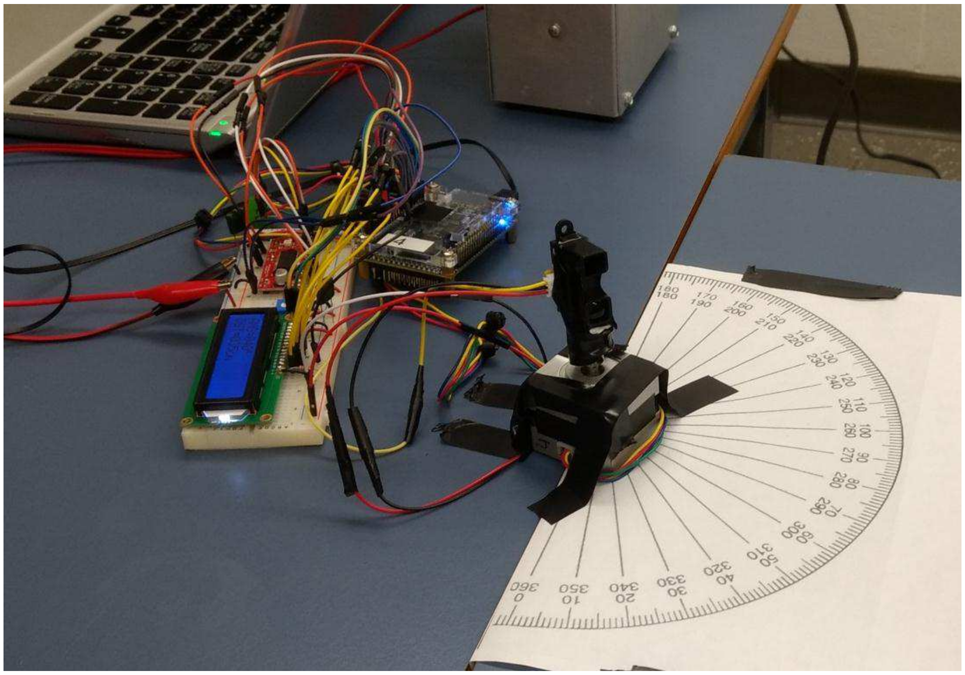

Figure 1: Friendo Finder

Project Motivation

We originally set out to make something similar to the motion tracker from the Alien movie series. It used ultrasonic waves to detect objects and put them onto a radar screen. We quickly came to terms with scaling the project back to deal with time and budgetary constraints. After some preliminary research, we had decided that accurate ultrasonic sensors were outside of our budget range and went with an infrared sensor instead. Since the infrared sensor detects objects in a line, we decided to sweep the sensor over a range with a stepper motor. Instead of the CRT screen featured in the Alien series, a 16x2 LCD screen would be used instead. As a stretch goal, we could implement the display onto the LCD screen used in lab 4 that we could make it look more like a radar screen.

High Level Design:

The design can be divided into three components: the FPGA, a stepper motor, and an infrared sensor. The final construction is shown in the picture below.

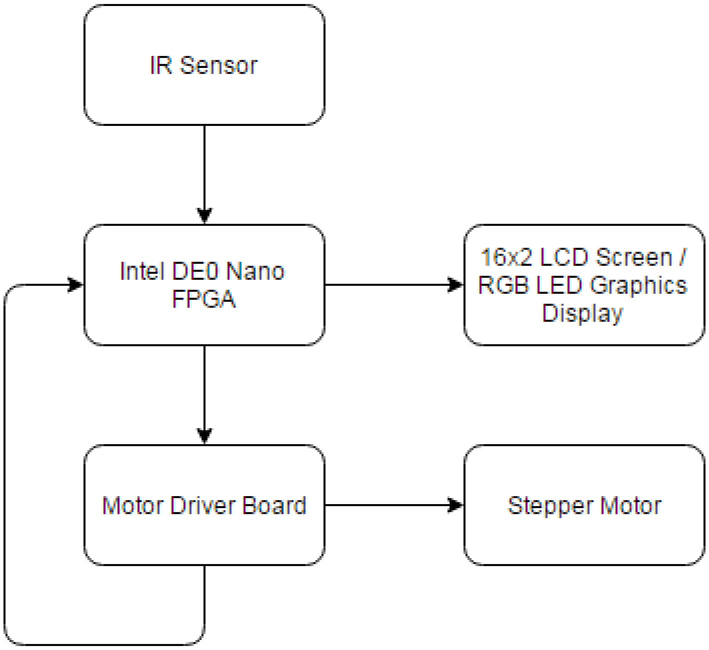

Figure 2: High-level flowchart

Data is read through the infrared sensor and provided to the FPGA as a voltage level. The FPGA interprets the voltage level as a distance measurement and decides if an object has been detected. If an object is detected, the motor stops moving and the distance and angle is displayed on the LCD screen. A high-level diagram showing the interactions is shown below.

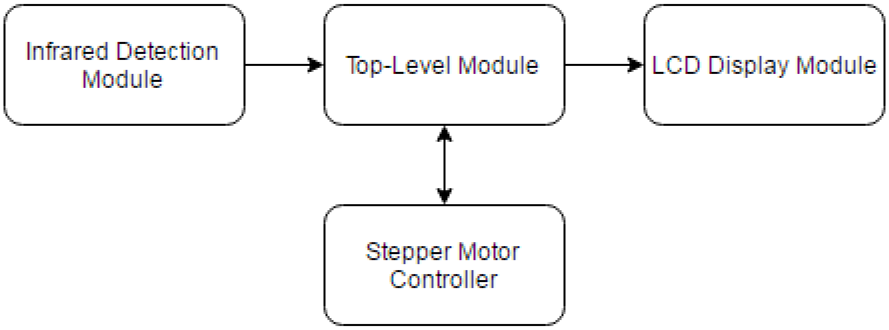

Figure 3: Module interactions

Hardware

The Friendo Finder is composed 3 pieces of hardware besides the FPGA: an infrared sensor, a stepper motor with a driver board and a 16x2 LCD screen. The hardware is connected as per the high-level flowchart. A complete schematic will be shown at the end of the hardware section.

Infrared Sensor

The sensor we decided upon was the Sharp GP2Y0A02YK0F. It has a advertised measuring distance of 20 to 150cm. It has 3 wires: power, ground, and a differential voltage signal. This will be fed directly into an ADC on the FPGA to be interpreted as distance. It requires a 5V power source, so the FPGA cannot power this module.

This sensor outputs a nonlinear voltage curve when measuring distance, shown below. To account for this, linear approximation needs to be implemented for accurate results.

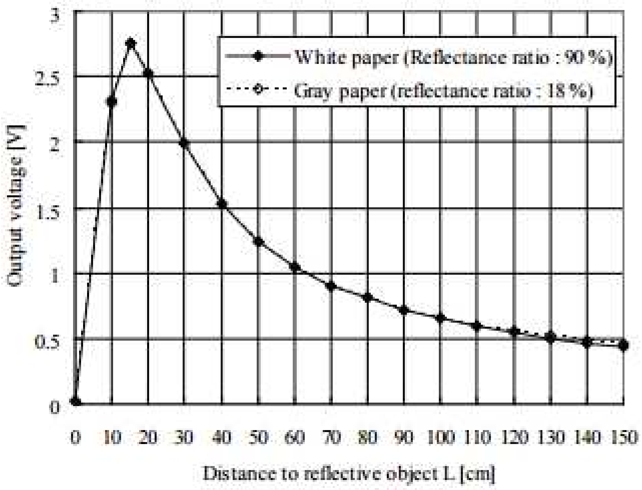

Figure 4: IR sensor distance measuring curve

During initial testing of the sensor, we found it was quite prone to interference from other light sources and reflections. Also, sometimes it had difficulty recognizing dark objects. This is due to the reflective nature of light. More of the light emitted by the IR sensor is absorbed in dark objects, so lighter targets needed to be utilized.

The sensor had to be mounted vertically to ensure that the ADC received and accurate result. When horizontal, the sensor's receive and send sensors are offset and can falsely detect edges. To minimize this, the sensor is mounted vertically and tall target must be used.

Stepper Motor and Driver

A stepper motor and a driver board was chosen for this project. A stepper motor was specifically chosen due to precise steps, which can be converted into an angle for display. The driver board determines various settings for the stepper motor and can fit onto a breadboard. The control signals can be found in detail in the Stepper Motor Module section.

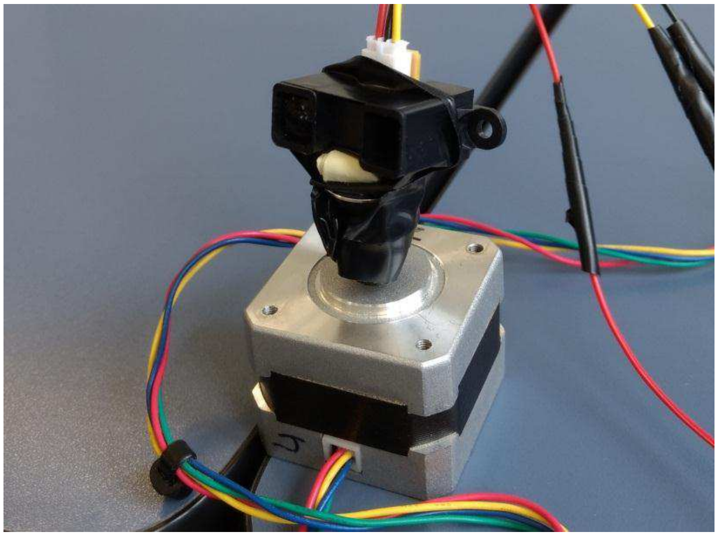

Figure 5: Stepper motor with horizontal IR sensor (Wall-e?)

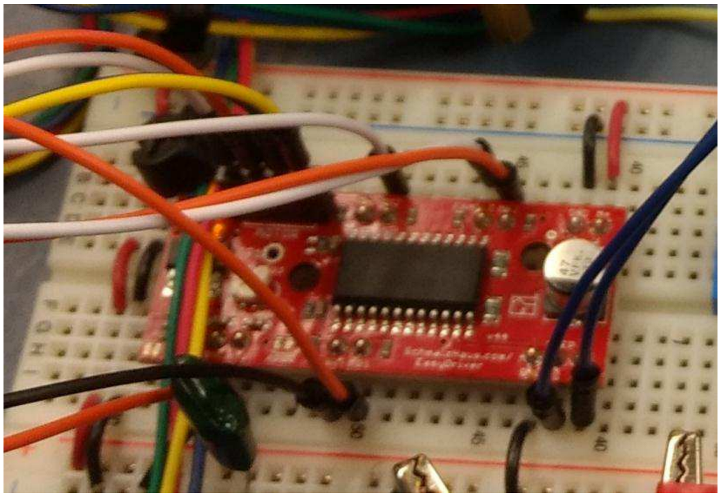

Figure 6: Driver board

16x2 LCD screen

The 1602A-1 LCD screen was used for this project. It's a 16x2 LCD screen with a display font of 5x8 pixels. It has the capability of 4 or 8 bit interface for data input. The display is powered by 5V and accepts 5V data signals. However, the lower threshold for a voltage high is 2.2 V so the FPGA can provide data without having to buffer the signal. A detailed description of the signals can be found in the LCD module section below.

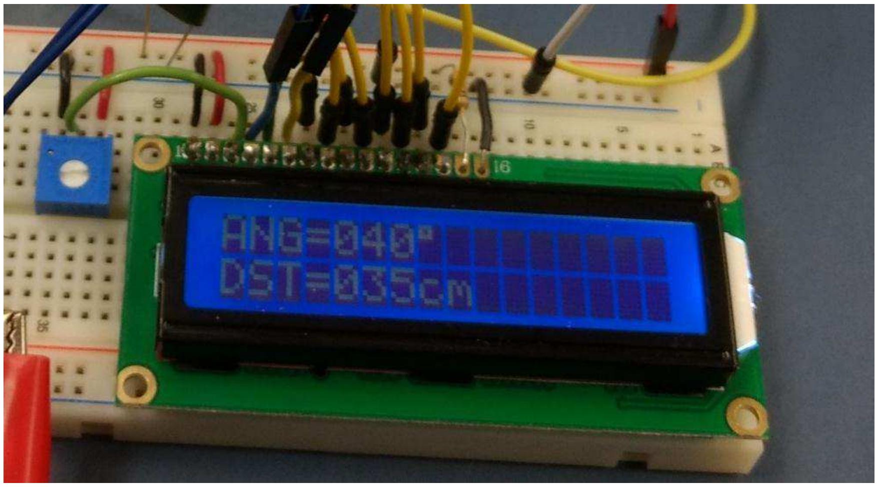

Figure 7: 16x2 LCD screen

Hardware Schematic

Note that Vss is a 5V supply.

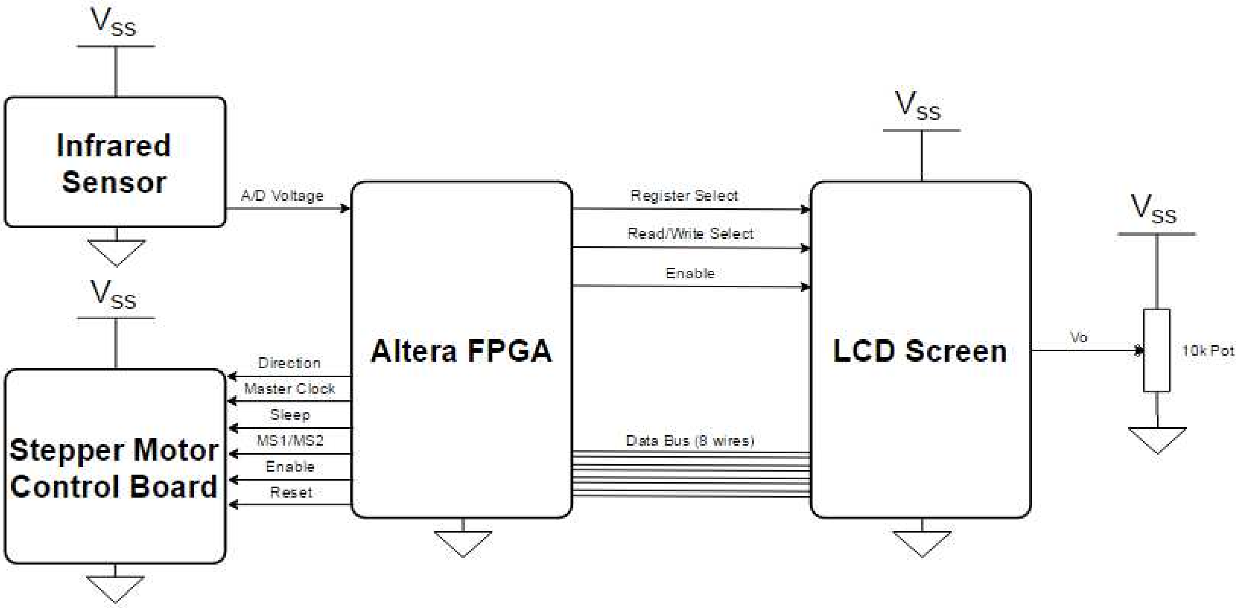

Figure 8: Hardware layout

Modules

A written description of each software module can be found below. We wrote all of the modules in System Verilog without using software (NIOS II) in the hopes of it being easier to implement. We recognize this as being inefficient in terms of hardware usage, but we were not going to do anything else with it so it may as well be implemented in hardware.

ADC Module

The ADC test code from Lab 5 was utilized for this module. It features a 16-bit FIFO buffer, which is filled by and ADC SPI interface, which continuously reads ADC Channel 0 until filled. It checks if the FIFO buffer is loaded by checking a done flag. Once filled, it outputs the 16-bit FIFO onto a 16-bit data out line to be read by the top-level module. The module performs this operation continuously.

Stepper Motor Module

As you might expect, the stepper motor module is in charge of autonomous motor control. The motor sweeps back and forth 180 degrees while counting the number of steps from the initial position. The motor control module has several control signals:

- Direction: selects if the motor steps clockwise or counterclockwise
- Master clock: a square wave which controls how frequent the motor steps, we chose 5Hz to be reasonable
- Sleep: used to halt operation by disregarding input signals
- MS1 and MS2: two bits which control the step size
- Enable: select whether the motor runs or not
- Reset: a switch used to halt the motor step manually and resets the count register

By knowing the change in angle caused by each step and a zeroed position, we can determine the angle at which an object was detected. The number of steps is counted and stored in a 32-bit count register. This is fed into the top-level module to be sent into the LCD display.

16x2 LCD Screen Module

This module is based around four states: power up, initialize, ready, and send. It accepts two signals from the top-level module, an LCD enable signal and a data bus. The module communicates a byte to the LCD screen, an enable signal, a register select, and read or write select. A busy flag communicates if the screen is ready to accept a command. A reset button is used to restart the display if required.

The code is based on a module written for the same LCD screen for ELEX 3305. While the original was written in C, it was converted into System Verilog for this project.

To write a character to the module, an LCD enable signal is required to send a command. The command to be sent is driven along the data bus, with the two most significant bytes signifying where and what the data bus signal is. The two most significant bits are data input or an instruction input, and either to read or write.

As mentioned, the module has four states:

- Power up: waits a relatively long duration to ensure the LCD screen powers up properly
- Initialize: set up the LCD screen settings, such has telling the screen how to interpret the byte or how the cursor behaves
- Ready: ready to receive a command if the screen is not busy
- Send: the process in which the byte is sent

Timing requirements needed to be met in between instructions within the states. This was accommodated for by counting the number of clock cycles from the FPGA clock.

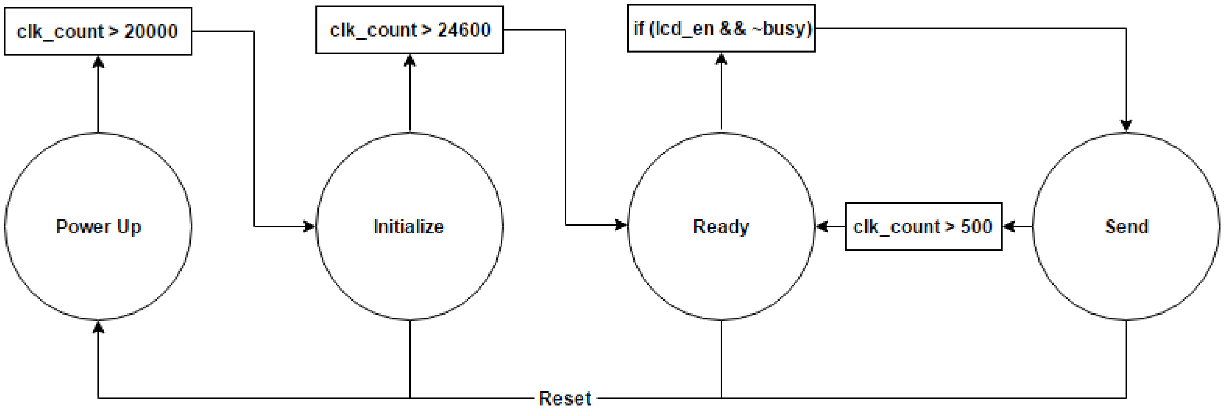

Figure 9: LCD state diagram

Top-Level Module

This module unifies the other three. It gathers information from the ADC and stepper motor and outputs the information onto the LCD screen. It is composed of four states as well: power up, run, set data, and display. It also features two functions which assist writing information to the LCD screen. It can be reset manually using one of the onboard switches.

- Power up: similar to the LCD screen, this allows all modules to power up in the correct sequence and all variables are reset within this state
- Run: the motor is swept back and forth in a 180 degree arc while monitoring the ADC for a valid input
- Set data: Take the valid input from the ADC, calculate the angle, and prepare to output the data to the LCD
- Display: Pulse the enable bit for the LCD for writing

The two functions are implemented as tasks in System Verilog. The first task contains a case list of characters to output to the LCD screen. The case is selected by a 5 bit input variable which gets incremented in the set data state. The second task performs the distance and angle conversions from the motor control module and the ADC. Note that since the infrared sensor is quite non-linear, we had to interpolate between certain voltage ranges to ensure an accurate output.

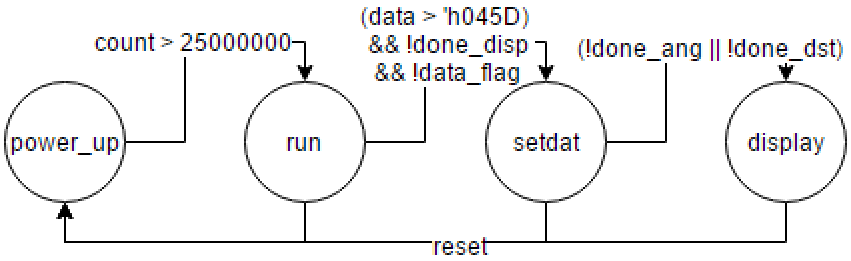

Figure 10: Top level module state diagram

Testing Method

Test bench modules were not written for this project. Instead, the SignalTap 2 Logic Analyzer in Quartus Prime was used heavily to ensure the correct signals were transmitted and received. Each module was independently verified to work using the SignalTap 2 Logic Analyzer and then combined into a top-level module. Once combined, we set up a testing area shown below.

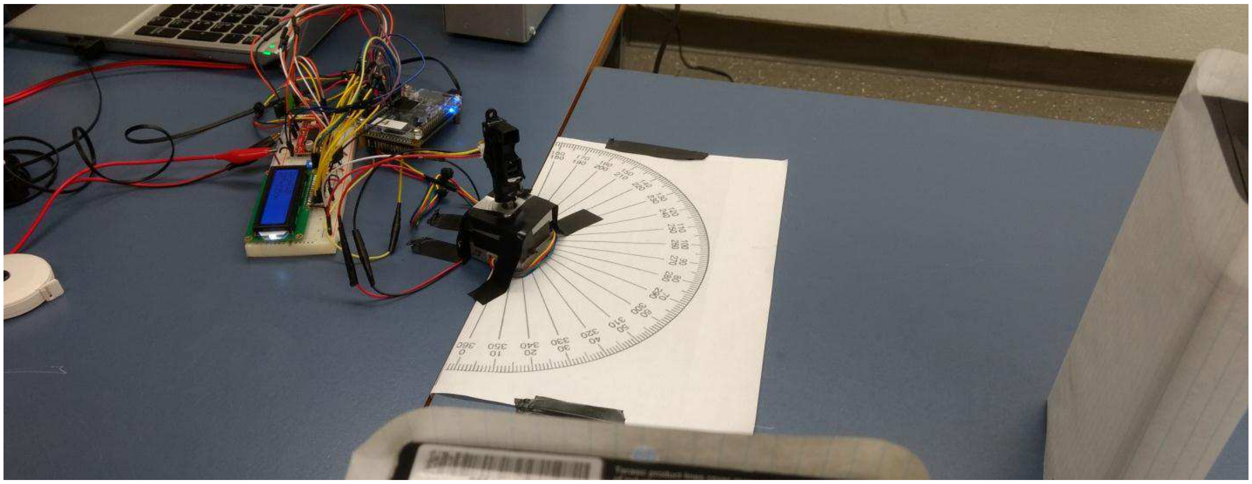

Figure 11: Testing the Friendo Finder

The infrared sensor was mounted on top of the stepper motor and secured in the center of a protractor. Once it powered up, it began to sweep. We would targets at various locations to confirm the angle and distance measurements were correct.

Results

By interpolating the voltage ranges on the IR sensor and setting the stepper motor range to as close to 180 degrees as possible, we could achieve quite accurate results. Within the range of 20-70cm we would achieve results within 1cm and an angle within 5 degrees. With some further refinement on the ADC module and with a stepper motor with smaller steps we would be able to get more accurate results.

Conclusions and Final Thoughts

While the original project idea was to make something closer to what you see in Alien, we are still quite pleased with the results. The measurements ended up being quite accurate and we had a lot to show by the end of the project. If we were to do this project again, we would implement some features in software using NIOS II and perhaps use the LCD screen used in lab 4 for a fancier display. Using software, the timing would be easier to write and we would use less hardware overall.

Appendix (Friendo Finder code)

ADC_Controller.sv

```
// This code is a modification of the ADC_spi Interface written by the
// instructor for ELEX7660 lab5. Code has been edited to read from one
// ADC chanel(0) as well as removing the need for Nios 2.

module adcspi
   (
    output logic sclk, mosi, ssn, // SPI master
    input logic miso,

    output logic [15:0] data, // ready/valid data out

    input logic clk, reset ) ;

      parameter MISO = 16'b0 ;

      // clock/bit counter
      struct packed {
            logic [3:0] bitcnt ;
            logic sclk ;
            logic [3:0] clkcnt ; } cnt, cnt_next ;

      logic [15:0] sr ;            // shift register

      logic rising, falling, done ;

      assign sclk = cnt.sclk ;

      // done all bits

      assign done = cnt ==? '{'1,'1,'1} ;

      // clock/bit counter

      assign cnt_next = ( reset || done ) ? '0 : cnt+1'b1 ;
      always@(posedge clk)
            cnt <= cnt_next ;

      assign rising   =  cnt_next.sclk && ~cnt.sclk ;
      assign falling = ~cnt_next.sclk &&  cnt.sclk ;

      always@(posedge clk)
      begin
            if ( falling )            // shift mosi out
                  mosi <= sr[15] ;
            if ( rising )             // shift miso in
                  sr <= {sr[14:0],miso} ;
            if ( done )
            begin
                  data <= sr ;           // copy to parallel out
                  sr <= MISO ;           // channel select serial out
                  mosi <= 16'd0;
```

```
            end
        end

        always@(posedge clk)            // run continously
            ssn <= reset ;

    endmodule
```

Motor_Controller.sv

```
// This code is a controller for A3967 stepper motor controller.
// This is a modified version of the ELEX7660 lab 3 module.

module motor_control
   (
      input logic clk, reset,

      // Motor control signals mclk is stepper motor square wave,
      // enable_n allows the motor to run, sleep causes motor to
      // disregard all input signals. ms1, ms2 set microsepping
      // ang is number of steps that have been covered used by
      // module friendo_finder in Friendo_Finder.sv

      output logic mclk, mreset_n,
      output logic DIR, enable_n,
      output logic ms1, ms2,
      input logic sleep,
      output logic [31:0] ang
      ) ;

      logic [31:0] fclk = 32'd50000000;     // clock frequancy used by ff's
      logic [31:0] freq = 32'd5;            // output frequency Hz
      logic signed [31:0] count, count_next;
      logic mclk_next, DIR_next;
      logic [31:0] ang_next;

      assign   fclk = 32'd50000000;
      assign   freq = 32'd5;

      always_ff@(posedge clk) begin
            count <= count_next;
            mclk <= mclk_next;
            DIR <= DIR_next;
            ang <= ang_next;
            ms1 = 'd0;
            ms2 = 'd0;
      end

      always_comb begin
            count_next = (reset && sleep) ? fclk : (count >= 0) ? count - (2
* freq) : (count < 0) ? fclk: count;
            mreset_n = reset ? 1'd0 : 1'd1;
            enable_n = reset ? 1'd1 : 1'd0;
            mclk_next = (reset && sleep) ? 1'd0 : (count < 0) ? ~mclk : mclk;
            if(reset) begin
                  DIR_next = 1'd1;
                  ang_next = 32'd0;
            end
            else if((count < 0) && sleep) begin
                  if (ang >= 180) begin
                        ang_next = 32'd0;
                        DIR_next = ~DIR;
                  end
                  else begin
```

```
                        ang_next = ang + 1;
                        DIR_next = DIR;
                  end
            end
            else begin
                  ang_next = ang;
                  DIR_next = DIR;
            end
      end
endmodule
```

LCD_Controller.sv

```
// 16x2 LCD display has 3 control lines besides power:

// lcd_en = latch data to lcd controller
// lcd_rw = read(1)/write(0) select signal
// lcd_rs = register select signal (1 = data input, 0 = instruction input)
// enable = enable for lcd display
// busy = lcd controller busy

// lcd screen has 8 or 4 bit data mode

module lcd_display
   (
      input logic clk, reset,
      input logic [9:0] lcd_bus,    // msb is rs, then rw, then data ms to ls
      input logic lcd_en,
      output logic lcd_rs, lcd_rw, enable, busy,
      output logic [7:0] lcd_data
   ) ;

      int clk_count = 0 ;                 // timing count
      int clk_freq = 50 ;                 // in mhz, for timing
      enum {power_up, initialize, ready, send} state ;

      always_ff@(posedge clk) begin

            if (reset)
            begin
                  state <= power_up ;
                  clk_count <= 0;
            end

            // power_up

            else if(!reset)
            begin
            // wait to make sure lcd is powered on
            unique case (state)
            (power_up):
            begin
                  busy <= '1 ;
                  if (clk_count < (20000 * clk_freq)) begin
                        clk_count <= clk_count + 1 ;
                        state <= power_up ;
                  end
                  else begin
                        clk_count <= 0 ;
                        lcd_rs <= '0 ;
                        lcd_rw <= '0 ;
                        lcd_data <= 8'b00000000 ;
                        state <= initialize ;
                  end
            end

            // initialize
```

```
(initialize):
begin
      busy <= '1 ;
      clk_count <= clk_count + 1 ;

      // set up 8 bit mode first
      if (clk_count < (100 * clk_freq)) begin
            lcd_data <= 8'b00110000 ;           //
            enable <= '1 ;
            state <= initialize ;
      end

      else if (clk_count < (600 * clk_freq)) begin
            lcd_data <= 8'b00000000 ;
            enable <= '0 ;
            state <= initialize ;
      end
      // set lcd for 2 line mode, 5x8 dot mode, and 8bit mode
      else if (clk_count < (700 * clk_freq)) begin
            lcd_data <= 8'b00111000 ;     //
            enable <= '1 ;
            state <= initialize ;
      end

      else if (clk_count < (1200 * clk_freq)) begin
            lcd_data <= 8'b00000000 ;
            enable <= '0 ;
            state <= initialize ;
      end

      // clear up the screen
      else if (clk_count < (1300 * clk_freq)) begin
            lcd_data <= 8'b00000001 ;
            enable <= '1 ;
            state <= initialize ;
      end

      else if (clk_count < (21300 * clk_freq)) begin
            lcd_data <= 8'b00000000 ;
            enable <= '0 ;
            state <= initialize ;
      end

      // assign display on, curson on, blink on
      else if (clk_count < (21400 * clk_freq)) begin
            lcd_data <= 8'b00000010 ;
            enable <= '1 ;
            state <= initialize ;
      end

      else if (clk_count < (22000 * clk_freq)) begin
            lcd_data <= 8'b00000000 ;
            enable <= '0 ;
            state <= initialize ;
      end
```

```
      // assigns display on, cursor on, cursor blink on
      else if (clk_count < (22200 * clk_freq)) begin
            lcd_data <= 8'b00001100 ;
            enable <= '1 ;
            state <= initialize ;
      end

      else if (clk_count < (22800 * clk_freq)) begin
            lcd_data <= 8'b00000000 ;
            enable <= '0 ;
            state <= initialize ;
      end
      // assign cursor shift direction
      else if (clk_count < (23200 * clk_freq)) begin
            lcd_data <= 8'b00010100 ;
            enable <= '1 ;
            state <= initialize ;
      end

      else if (clk_count < (23800 * clk_freq)) begin
            lcd_data <= 8'b00000000 ;
            enable <= '0 ;
            state <= initialize ;
      end

      // return cursor home
      else if (clk_count < (24200 * clk_freq)) begin
            lcd_data <= 8'b00000010 ;     //
            enable <= '1 ;
            state <= initialize ;
      end

      else if (clk_count < (24600 * clk_freq)) begin
            lcd_data <= 8'b00000000 ;
            enable <= '0 ;
            state <= initialize ;
      end

      // complete initialization
      else begin
            clk_count <= '0 ;
            busy <= '0 ;
            state <= ready ;
      end

end

// ready

(ready):
begin
      // load rs, rw, and data from bus
      if (lcd_en == '1) begin
            busy <= '1 ;
            lcd_rs <= lcd_bus[9] ;
            lcd_rw <= lcd_bus[8] ;
            lcd_data <= lcd_bus[7:0] ;
```

```
                        clk_count <= '0 ;
                        state <= send ;
                  end
                  else begin
                        busy <= '0 ;
                        lcd_rs <= '0 ;
                        lcd_rw <= '0 ;
                        clk_count <= '0 ;
                        state <= ready ;
                  end
            end

            // send

            (send):
            begin
                  busy <= '1 ;
                  clk_count <= clk_count + 1 ;
                  // wait to complete sending
                  if ( clk_count < (500 * clk_freq)) begin
                        busy <= '1 ;

                        // pulse enable lcd
                        if (clk_count < clk_freq)
                              enable <= '0 ;
                        else if (clk_count < (150 * clk_freq))
                              enable <= '1 ;
                        else if (clk_count < (300 * clk_freq))
                              enable <= '0 ;
                        else
                              state <= send ;
                  end
                  else begin
                        clk_count <= '0 ;
                        state <= ready ;
                  end

            end
            endcase
            end
      end


endmodule
```

Friendo_Finder.sv

```
// Top level module for Friendo Finder.

module Friendo_Finder (
	input logic CLOCK_50,
	input logic [1:0] KEY,

	// Motor Contrtoller Signals
	output logic mclk, mreset_n,
	output logic DIR, enable_n,
	output logic ms1, ms2, sleep,

	// ADC Interface Signals
	output logic ADC_CS_N,
	output logic ADC_SADDR,
	output logic ADC_SCLK,
	input  logic ADC_SDAT,

	// LCD interface Signals
	output logic lcd_rs,
	output logic lcd_rw,
	output logic enable,
	output logic [7:0] lcd_data) ;


	// Common variables for all moduels
	logic reset_n, clk;

	// ADC Interface variables
	logic [15:0] data;

	// LCD Interface variables
	logic lcd_en;
	logic [9:0] lcd_bus; // msb is rs, then rw, then data ms to ls
	logic busy;

	// Angle calculation variables
	int angle;
	logic [31:0] ang;
	logic [4:0] ang_out [39:0];

	// Distance Calculation variables
	logic [4:0] dst_out [39:0];
	logic [15:0] high_data, range_data, set_data ;

	// Display variables
	logic [6:0] i;
	bit done_ang;
	bit done_dst;
	bit done_disp;

	// Control variables
	bit data_flag;
	bit move_flag;
	logic [31:0] count;
```

```
      enum {power_up, run, setdat, display} mode;

      assign clk = CLOCK_50;
      assign reset_n = KEY[0];

      motor_control m1
   (
            .clk(clk),
            .reset(~reset_n),

            .mclk(mclk),
            .mreset_n(mreset_n),
            .DIR(DIR),
            .enable_n(enable_n),
            .ms1(ms1),
            .ms2(ms2),
            .sleep(sleep),
            .ang(ang)
      );

      adcspi a0
   (
            .sclk(ADC_SCLK),
            .mosi(ADC_SADDR),
            .ssn(ADC_CS_N),
            .miso(ADC_SDAT),
            .data(data),

            .clk(clk),
            .reset(~reset_n)
      );

   lcd_display lcd1
      (
            .clk(clk),
            .reset(~reset_n),

            .lcd_en(lcd_en),
            .lcd_bus(lcd_bus),
            .lcd_rs(lcd_rs),
            .lcd_rw(lcd_rw),
            .enable(enable),
            .busy(busy),
            .lcd_data(lcd_data)
      );

      always_ff@ (posedge clk)
      begin

            // Reset starts the power up process for all the components
involved
            if(~reset_n)
            begin
                  count <= '0;
                  mode <= power_up;
            end
```

```
else if (reset_n)
begin
unique case (mode)

// Gives all modules time to power up, mainly lcd module .
// As well as initializing all variables
(power_up):
begin
      if (count < 25000000)
      begin
            i <= '0;
            sleep <= '0;
            done_ang <= '0;
            mode <= power_up;
            count <= count + 1;
            done_disp <= '0;
            data_flag <= '0;
            move_flag <= '0;
      end
      else mode <= run;
end

// Waits for a valid data input if none is found the motor
// control module will continue to turn if found then motor will
// stop and data will display.
(run):
begin
      if(~KEY[1]) move_flag <= '0;
      if((data > 'h045D) && (!done_disp) && (!data_flag))
      begin
            if (count < 20000000) count <= count + 1;
            else
            begin
                  set_data = data;
                  angle = (DIR == '1) ? ang : (180 - ang);
                  mode <= setdat;
                  count <= '0;
                  count <= '0;
                  i <= '0;
                  done_ang <= '0;
                  done_dst <= '0;
                  sleep <= '0;
                  data_flag <= '1;
                  move_flag <= '1;
                  func1();          // Sets data to be displayed
            end
      end
      else if(~move_flag)
      begin
            sleep <= '1;
            mode <= run;
            count <= '0;
            if(data < 'h045D) data_flag <= '0;
      end
end
```

```
            // Sets the data at the pins of the LCD by feeding data to
            // the LCD_Controller module
            (setdat):
            begin
                  if(!done_ang)
                  begin
                        func(ang_out[i]);
                        i <= i + 1;
                        if(i == 39)
                        begin
                            done_ang <= '1;
                              i <= '0;
                        end
                        else done_ang <= '0;
                        mode <= display;
                  end
                else if(!done_dst)
                  begin
                        func(dst_out[i]);
                        i <= i + 1;
                      if(i == 39)
                        begin
                            done_dst <= '1;
                              done_disp <= '1;
                        end
                        else done_dst <= '0;
                        mode <= display;
                  end
                  else
                  begin
                        lcd_en <= '0;
                        mode <= run;
                        done_disp <= '0;
                  end
            end

            // Trigger the LCD_Controller module and display data on the
lines
            (display):
            begin
                  if(~busy)
                  begin
                        lcd_en <= 1;
                        mode <= setdat;
                  end
                  else
                  begin
                        mode <= display;
                        lcd_en <= 0;
                  end
            end
            endcase
            end
      end

      // This task sets the data bus for the LCD_Controller module
      task func;
```

```
      input [4:0] a;
      begin
            unique case(a)
                  0: lcd_bus = 10'b1000110000; // "0"
                  1: lcd_bus = 10'b1000110001; // "1"
                  2: lcd_bus = 10'b1000110010; // "2"
                  3: lcd_bus = 10'b1000110011; // "3"
                  4: lcd_bus = 10'b1000110100; // "4"
                  5: lcd_bus = 10'b1000110101; // "5"
                  6: lcd_bus = 10'b1000110110; // "6"
                  7: lcd_bus = 10'b1000110111; // "7"
                  8: lcd_bus = 10'b1000111000; // "8"
                  9: lcd_bus = 10'b1000111001; // "9"
                  10: lcd_bus = 10'b1001000001; // "A"
                  11: lcd_bus = 10'b1001001110; // "N"
                  12: lcd_bus = 10'b1001000111; // "G"
                  13: lcd_bus = 10'b1001000100; // "D"
                  14: lcd_bus = 10'b1001010011; // "S"
                  15: lcd_bus = 10'b1001010100; // "T"
                  16: lcd_bus = 10'b1000111101; // "="
                  17: lcd_bus = 10'b1011011111; // "degree symbol"
                  18: lcd_bus = 10'b1001100011; // "c"
                  19: lcd_bus = 10'b1001101101; // "m"
                  default: lcd_bus = 10'b1000100000; // "_"
            endcase
      end
      endtask

      // This task calculates the angle and the distance to be displayed on
the
      // LCD display...NOTE: Highly computationaly intensive find way to trim
down
      task func1;
      begin
            ang_out [0] = 5'd10;
            ang_out [1] = 5'd11;
            ang_out [2] = 5'd12;
            ang_out [3] = 5'd16;
            ang_out [7] = 5'd17;
            ang_out [39:8] = '{32{5'd20}};
            dst_out [0] = 5'd13;
            dst_out [1] = 5'd14;
            dst_out [2] = 5'd15;
            dst_out [3] = 5'd16;
            dst_out [4] = 5'd0;
            dst_out [7] = 5'd18;
            dst_out [8] = 5'd19;
            dst_out [39:9] = '{31{5'd20}};

            ang_out [4] = angle / 100;
            ang_out [5] = (ang_out[4] == 1) ? ((angle - 100) / 10) : (angle /
10);
            ang_out [6] = (ang_out[4] == 1) ? (angle - 100) - ang_out[5] * 10
: angle - ang_out[5] * 10;
            ang_out [6] = (ang_out[6] > 9) ? 5'd0 : ang_out[6];
            if(set_data > 16'h09B2) dst_out[5] = 2;
            if(data <= 16'h09B2 && set_data > 16'h0745) dst_out[5] = 3;
```

```
        if(data <= 16'h0745 && set_data > 16'h060F) dst_out[5] = 4;
        if(data <= 16'h060F && set_data > 16'h0555) dst_out[5] = 5;
        if(data <= 16'h0555 && set_data > 16'h045D) dst_out[5] = 6;
        if(data <= 16'h045D) dst_out[5] = 7;

        unique case (dst_out[5])
            2:
            begin
                high_data = 16'h0C1F;
                range_data = 16'h026C;
            end
            3:
            begin
                high_data = 16'h09B2;
                range_data = 16'h026C;
            end
            4:
            begin
                high_data = 16'h0745;
                range_data = 16'h0136;
            end
            5:
            begin
                high_data = 16'h060F;
                range_data = 16'h00BA;
            end
            6:
            begin
                high_data = 16'h0555;
                range_data = 16'h00F8;
            end
            7:
            begin
                high_data = set_data;
                range_data = 16'hFFFF;
            end
        endcase
        dst_out [6] = (dst_out[5] == 'd7) ? 5'd0: ((high_data - set_data)
* 10) / range_data;
        dst_out [6] = (dst_out[6] > 9) ? 5'd0 : dst_out[6];
    end
    endtask
endmodule
```