Final Project Report: digitEyez

6.111 Fall 2019

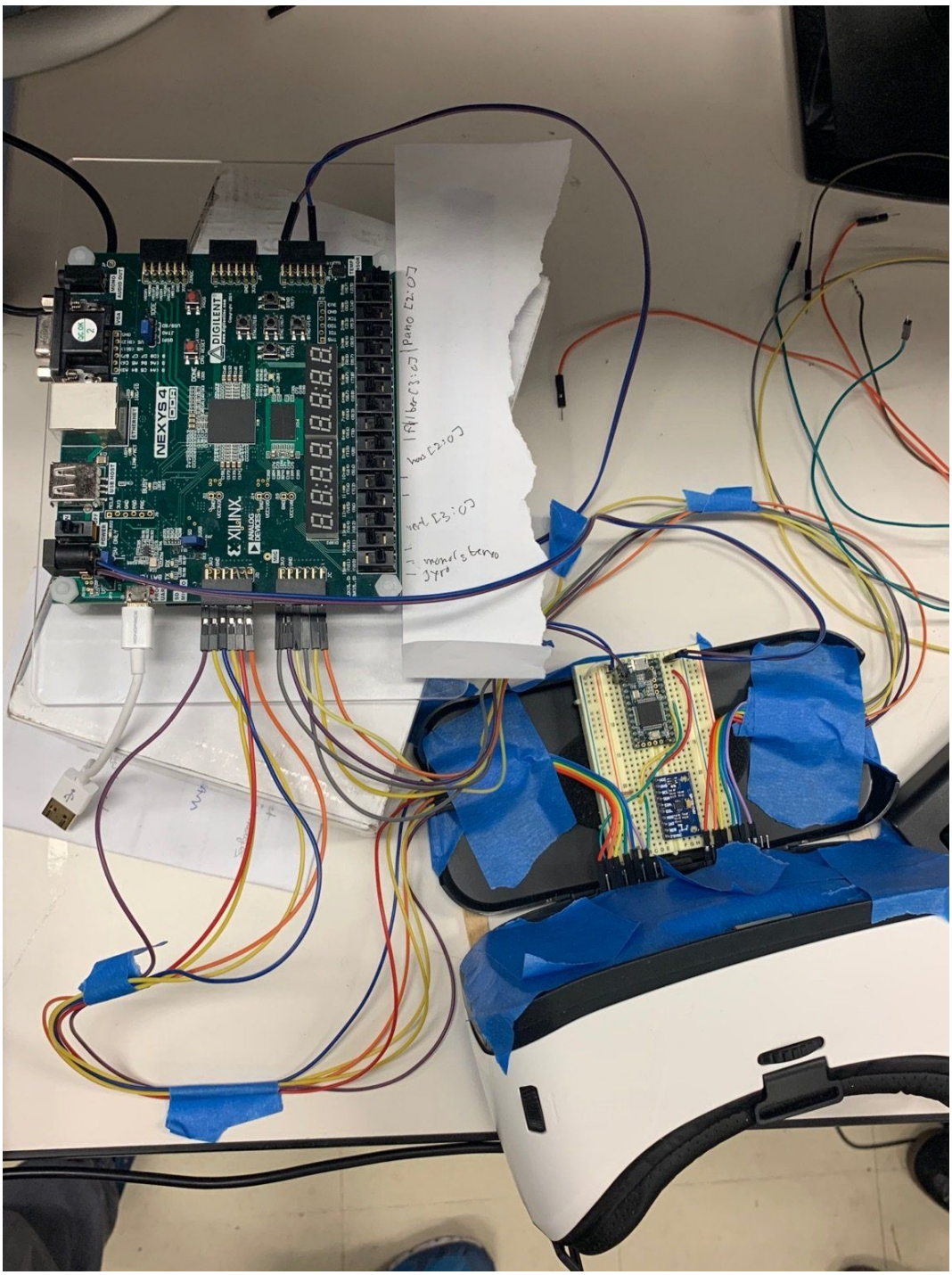

Table of Contents

Overview

Inspired by the View Master, our team decided to build a VR-like headset with two OLED screens. We would use a gyroscope mounted on the headset to determine where the user was facing, and then draw an image on the screen. We used two different types of images: mono and stereo. For mono images, we would just display the same image, giving a relatively flat experience. For stereo images, we would display two slightly different images on the left and right screens, which would give a 3D illusion. We pulled images online from Google street view and Google's Daydream developer resources, in addition to images from NASA and even one of our own panoramic images.

Additional Hardware

- **3 Adafruit ST7789 LCD Displays**: Our final product contained two 240x320 pixel LCD screens for displaying images
- **1 MPU-9250 IMU**: We used the IMU to capture raw accelerometer and gyroscope data.
- **1 Teensy**: The teensey was used to convert raw data from the IMU and send it to the FPGA via the UART protocol.
- **1 "VR" Glass Frame**: we used a VR headset designed to hold a smartphone to mount our screens

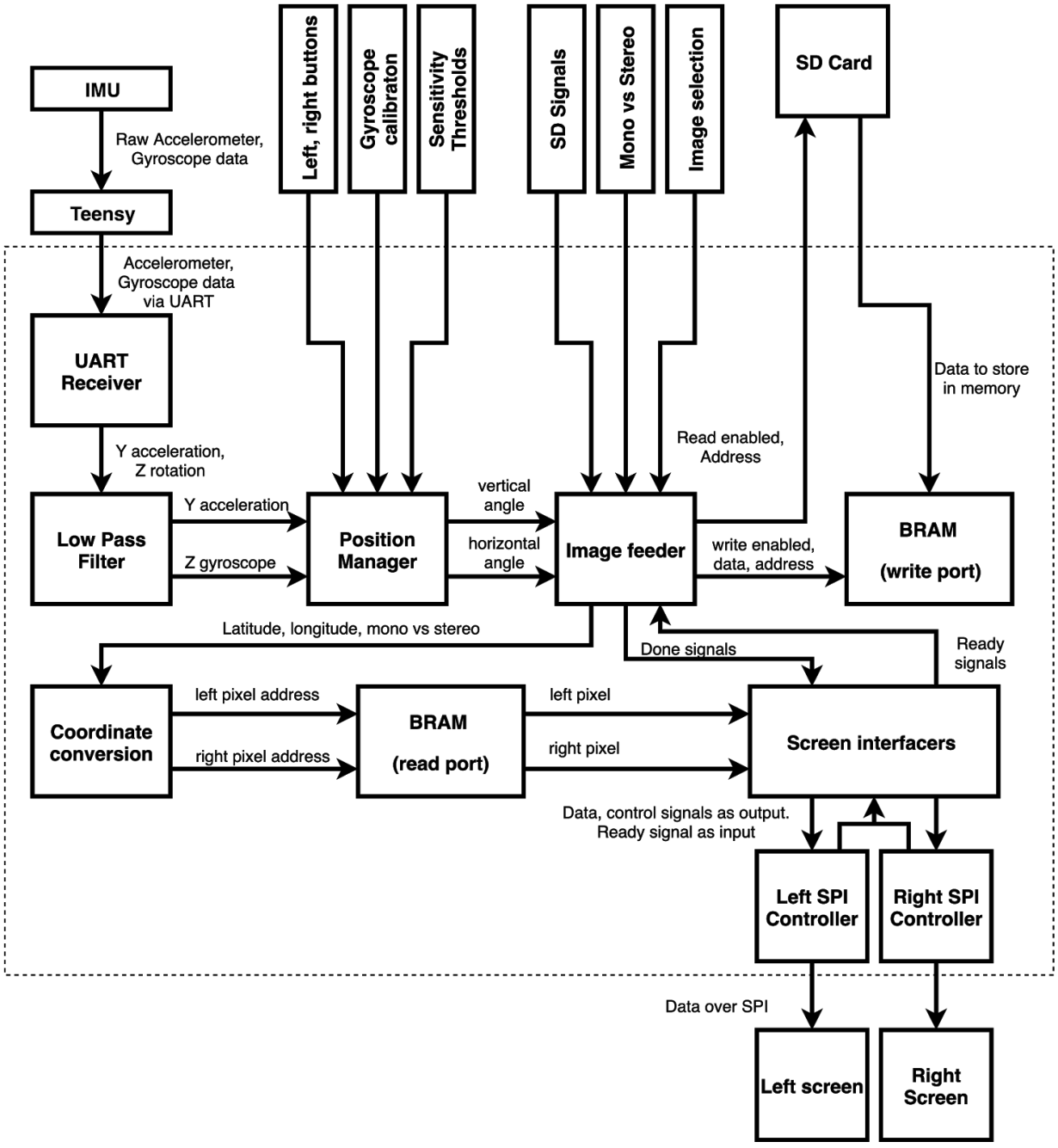

Figure 1: An overview of our system. The section enclosed by the dotted box corresponds to logic performed within the FPGA, whereas components outside the box represent input (buttons, switches, SD card) and output devices (screen)

Modules

Coordinates and interfacing with the IMU (Kendall)

Processing IMU Data

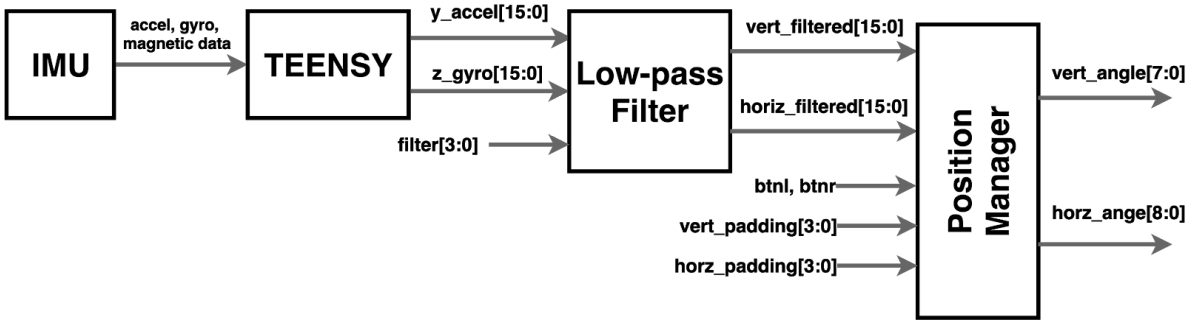

Figure 2: Processing IMU data

We used the MPU-9250 IMU chip to take in accelerometer and gyroscope data. To make reading the data simpler on the chip, we pass in the readings from the IMU to the teensy. The teensy samples y acceleration and z gyroscope every 10 ms (seen here), and then sends the 16 bit signed z gyroscope data, 16 bit signed y acceleration, and 16 zeros. This data is then read by the uart receiver module on the FPGA running at 65 mhz, which accepts and stores the data. To prevent sudden jumps and reduce noise, both inputs are passed through low pass filters. The decay factor is based off of switches 3-6 (zero-indexed), where all the switches in the "off" state corresponds to no decay, and all the switches on the "on" state corresponds to a decay of $\frac{15}{16}$.

Once we have filtered the acceleration and gyroscope data, we convert the data to coordinates: a 9-bit horizontal angle (0-359) representing the head tilting left and right, and an 8-bit vertical angle (30-150), representing how far the head is from being entirely vertical.

Because determining the angle off vertical relies only on knowing the current position of the IMU, there is minimal state associated with this calculation. We start by truncating the 8 lower bits, and only operate on bits [15:8]. Because the max range of the IMU is $\pm 2G$ this gives us a range from 8b'1100_0000 (-64) to 8'b0100_0000 (64) in two's complement. To allow for vertical viewing windows of 60°, we cap the possible range from -60° to 60° and add it to 90°, giving us a total possible range of [30°, 150°].

To prevent the position from shifting between two small values (±2 degrees), we use part of the switches (sw[13:10]), and only update the coordinates if the change in y position is larger than the threshold.

For calculating the horizontal angle, we had two methods: buttons on the fpga, and gyroscopic data. Because calculating the next horizontal angle requires knowing the previous angle, we introduced a register to hold the state.

When in button mode, the user can press the left or right (pressing both does nothing) button on the FPGA, which would move the horizontal position to the left or right based off of the respective input. To prevent the coordinate from changing extremely rapidly at the FPGA's 100MHz clock, we added a clock divider for the horizontal movement which calculated updates thirty times a second. This meant that a user could move up to 30 degrees in one direction every second, which we felt was a reasonable pace.

In gyroscope mode, the user can move their head left and right, which will shift the position relative to the movement. To calculate whether to move, we used some of the switches (sw[7:9]) to calibrate the sensitivity, and if the gyroscope reading is greater than the horizontal sensitivity, the horizontal coordinate will change by 2. Like the buttons, this position only changes at a period of 30 instances per second, meaning at most 60 degrees. To handle initial drift, we also allow for calibrating the base direction.

Converting IMU Data to Coordinates

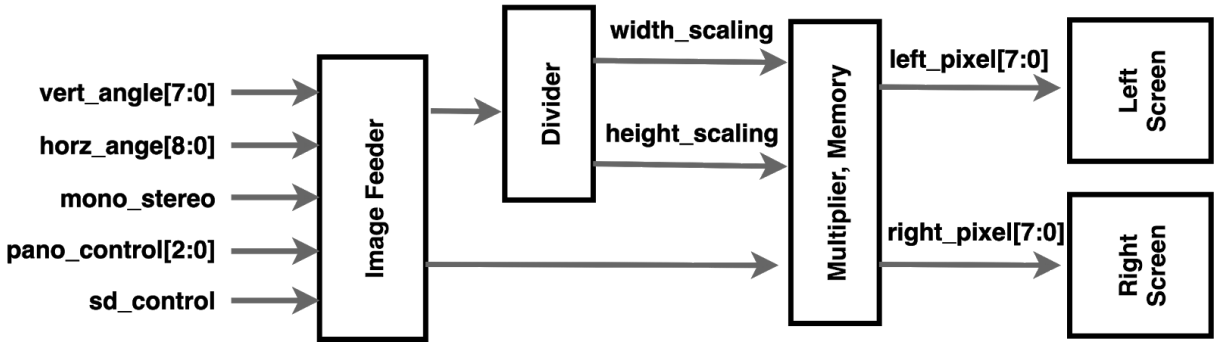

Figure 3: From coordinates to pixels

Once we have calculated the current position using spherical coordinates (horizontal, vertical angle) we need to convert it to rectangular coordinates (x, y) to index into our images stored in RAM and retrieve the current pixel in the [image feeder]

module. When taking in the vertical and horizontal coordinates, we treat the vertical angle as the center of the image vertically, and treat the horizontal angle as the leftmost part of the image. To convert these coordinates, we the general formula:

$$x = R(\lambda - \lambda_0)\cos(\phi_1)$$
$$y = R(\phi - \phi_1)$$

λ = longitude, λ_0 = the central meridian, ϕ =latitude,
ϕ_1 = standard parallels, R = radius of the globe

Because the images we chose to use were stored using equirectangular projection, the central meridian and standard parallels are all zero, our coordinate conversions just scales the latitude and longitude to the width and height of the image. Unfortunately this does have the potential of distorting the image when departing from the equator; however, this does simplify calculations as we only need a single multiplier.

While the scaling math is easy in software, getting sufficient precision in hardware is more difficult. Both the vertical and horizontal angles are positive integers; however, simply using integer multiplication would lack precision. To calculate the width and height scaling factors, we added a divider module which calculates the latitude-to-height and longitude-to-height scaling factors as decimals with eight-bit fractional components. For example, to represent 2.5 using eight-bit values, we would store it as the following:

$$10.10000000$$

The values left to the decimal represent the decimal values, while every bit to the right of the decimal represents 2^{-n}, where n represents the distance away from the decimal. Once we have scaled the coordinates, we drop the fractional bits. Then, we find every pixel in the 240 x 320 screen, mapping both the width and height to a 60 degree view. For the horizontal axis, we use an 8-bit counter, dropping the lower order bits, and for the vertical axis we use the 6 higher bits of an 10-bit signed counter from [-480, 480), which increments by 3 for every iteration of the loop.

Images, SD, Screens, and Hardware (Claire)

SD card

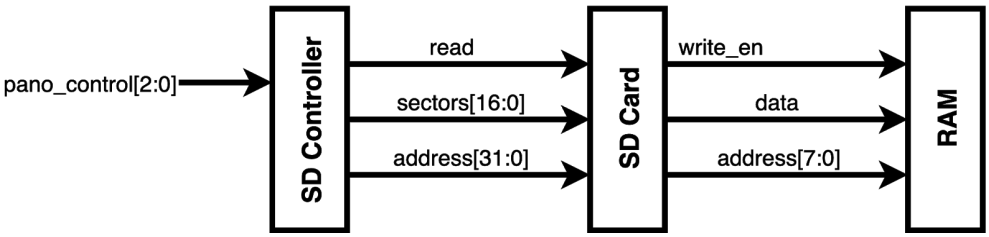

Figure 4: From SD to Memory

The SD card was interfaced with via the sd_controller module provided in the class resources. We used a 2gb microSD that was inserted directly to the Nexys's SD slot, that we pre-loaded with image data using HxD. Interestingly, we found that the entire card had a 0x1FE00 offset from the addresses written to using HxD. We discovered this through an intensive debugging process that involved writing identifiable sequences to the greater part of the card and attempting to read them back. During this process, we developed a light-weight test module whose sole purpose was to read from the card and display its results on the 7 segment displays. After multiple days, we never fully resolved this issue, but instead worked around it.

We kept careful track of the starting locations of our images, and wrote their raw data into the card using HxD. The starting addresses of the images were then hardcoded into the System Verilog. In the future, a more streamlined approach would be to better standardize the image between mono and stereo and have the FPGA determine the start address based off of the image number.

Images

Figure 5: Sample Mono image (the MIT Great Dome) formatted and ready for display. Because the image is mapped to an equirectangular projection, the edges appear distorted. However, because we take this into account when displaying the image on the screens, in the headset everything patches together smoothly.

To save space and increase the size of the panoramas we could display, we decided to only display grayscale images. Because Google provides some sample images that are 830x415, and these images fit easily within the available RAM, we used this as a starting size for our mono images. We later found that we had to reduce the size of the stereo images so we could essentially fit two images, and chose 600x300 pixels per image, for a total of 360,000 bytes per image. We used GIMP to convert the images to grayscale, and then the provided imageRGB_write_file.m Matlab script. We then wrote our own Python script to reformat our data so it could be moved directly into the SD.

We stored stereo images (which are essentially two seperate images) as a single, continuous chunk of data in the SD card. Because all stereo images were the same size, the image feeder could knew where to split up its reads.

We used 8 bits of color depth to make reading and writing from the SD easier, but our display only supports up to 6 bits. If we were concerned about image quality, we could support slightly larger images by storing only the 6 bits that are used, but in this version we both store and send all eight bits. This requires the display to be in color mode “06h” (set by the COL_MODE command), so it expects one 6 bit transmission of color data at a time in slots D2-D7 from the SPI. This means two bits are discarded, and future iterations of our project could speed writes by truncating on the FPGA and then sending in mode 05h. However, because the rate at which changes in RAM reach the screen is still fairly slow, it’s unclear if this effect would be visible.

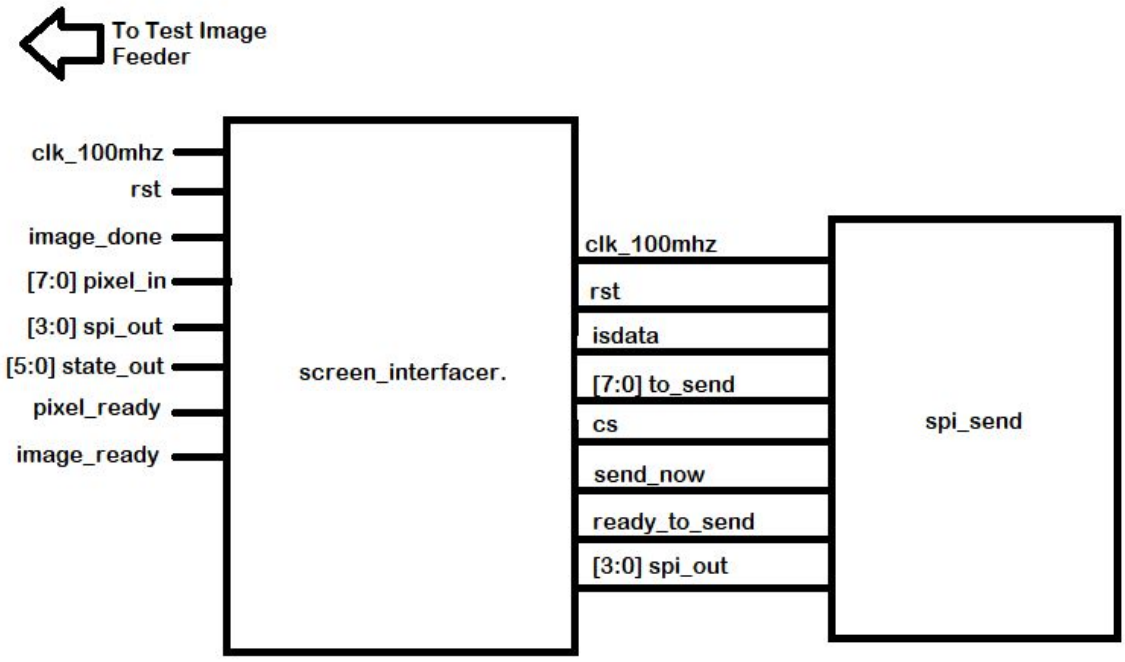

Figure 6: A diagram of how the screen controlling module, screen_interfacer.sv, interacts with spi_send.sv, the module that controls the base level of communication between the FPGA and the screen.

Screens

We used the two ST7788 displays from Adafruit. We structured our display controller off of the "Adafruit ST7735 and ST7789 Library" and the Sitronix datasheet. The display controller, screen_interfacer.sv, used the spi module to send a series of commands that started the display. After that, we paused the spi clock until we were ready to send a RAM write command (to write to the screen's onboard RAM, which is directly displayed) and begin sending an image. Because our images were all intended to be displayed in grayscale, we simply sent each pixel three times. We set the displays to both expect RAM writes from left to write, top to bottom, and display from RAM left to right, top to bottom. An improvement we made over the Adafruit libraries was to continuously stream data after the RAMWR command, instead of interrupting spurts of data with RAM writes to different addresses.

Figure 7: Wiring diagram for the displays. The data ports are connected to jc in the order they are on the display PCB, with SCK connected to jc[3] and D/C connected to jc[0]

Hardware and Focusing

We ordered a cheap “VR headset” intended for use with phones from Amazon, specifically the AOOK SongMI 3D Virtural Reality Glasses. We selected these glasses for their biconvex lenses that allow for focusing at short distances, as well as their useful focus adjust dials. Unfortunately, we learned that the glasses were designed for larger phones, and so our smaller screen would not provide an immersive experience. To compensate, we moved the screens close to the viewer’s nose so they would appear to blend into one. We then uniformly blocked out the rest of the viewing window to intensify the illusion. We faced challenges with wire length, especially with the female wires slipping off of the pins while the headset was in use. To rectify this, we taped the wires at multiple points in an “S” configuration coming off of the headset and placed the Teensy immediately in front of the viewer’s nose.

Timing (Kendall)

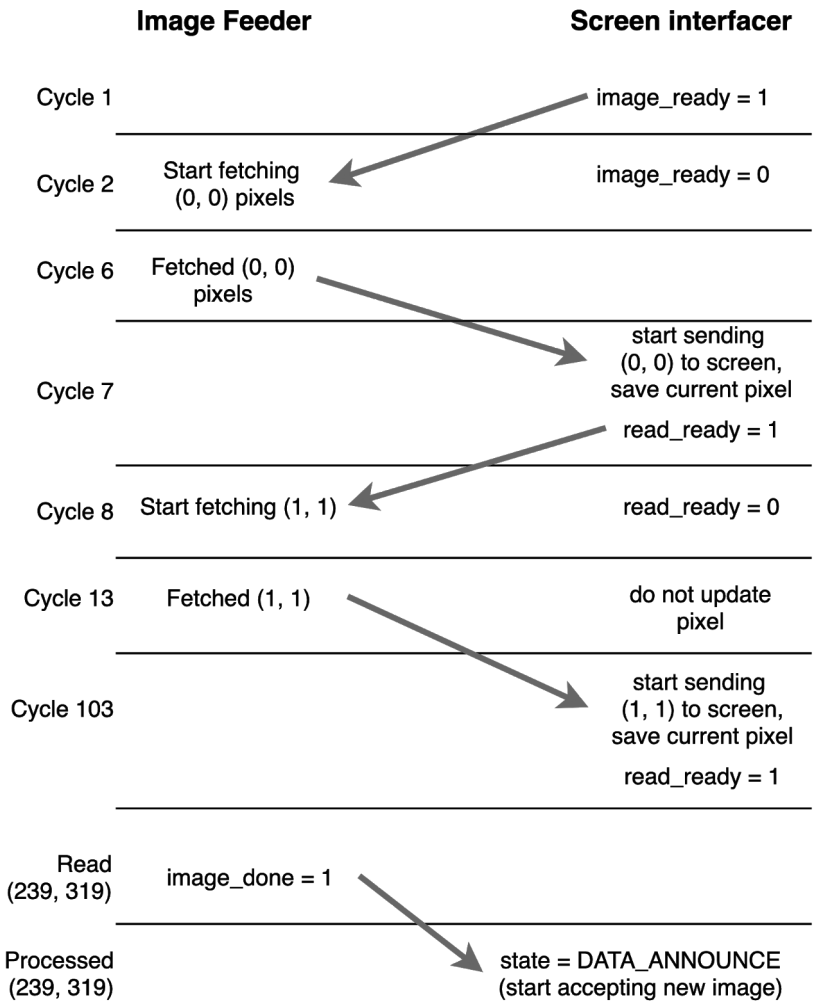

Figure 8: Image sending timing diagram

Once we had working modules for sending content to the screen and getting a position from the gyroscope, a significant challenge we faced was synchronizing timing to allow sending pixels from outside the screen interfacer module. At first, we just send a single color, and then let the interfacer read from a rom. However to accept a pixel as an input to the interfacer, we needed additional timing constraints.

When the image feeder module receives a new coordinate or calculates a new position in the image, it first takes three cycles for the correct left and right screen addresses to be computed: two for calculating the current row and column, and one for

calculating the actual address. Then, accessing a location in memory would take an additional two cycles, resulting in an overall delay of five cycles.

While this calculation is usually fast enough to complete before the screen interfacer has sent the pixel to the screen, we had to be careful about sending the first pixel. If the image feeder started the calculation too early, then the image would be off by one, and we could see weird red pixels rather than the actual image. Consequently, we would have to delay accepting the first pixel by 6 cycles after the image_ready signal is set.

After sending the first pixel, we had to synchronize sending successive pixels. To ensure that the image feeder would not start finding the next pixel until the screen was ready, we added the read_ready pulse, which would signal to the feeder that it should start calculating the next pixel. However, we ran into another issue where occasionally we would miss a pixel and see strange blue squares. This happened because the image feeder had calculated the next pixel, but the current pixel was not done. Because it takes roughly 96 cycles to send a single pixel (24 bits at a 25MHz clock), sometimes the current pixel was corrupted. To alleviate this, we would save the current pixel in the screen interfacer. Finally, to signal that the image was done and that the screen interfacer should start accepting new data, the image feeder sends the image_done flag.

Challenges and Takeaways

Debugging the screen interfacer was among the primary challenges of this project. Because the functionality of the screen is fairly binary, it was difficult for us to pinpoint problems, and took over a week to get the screen to a basic “on” state, let alone display coherent images. The datasheet was long and involved, but reading through the timing expectations and command alternatives was helpful for solving hard to crack problems. Something that was helpful here was writing our code on an Arduino to start, because we were able to determine the order of commands to send without worrying about SPI timing. This intermediary step, proved vital, and was a tool we utilized again with the SD card. In retrospect, if we had a working sd and screen controller coming into the project, we would have probably been able to at least make a good dent in our stretch goals.

Another particularly embarrassing bug that took me (Kendall) many hours to debug was the fact that the size of data that I was sending from the Teensy did not match the size of data I expected on the FPGA. I had thought that I removed two bytes

(we only needed Y accel and Z gyro), but in fact I was still sending an array of 6 ints, only the last values were all zero. Consequently, all the gyro and accel data would skewed.

A third challenge that we faced throughout the project was the fact that our builds took a relatively significant time (15-20 minutes, versus < 5 for some other groups). We initially ascribed this to having relatively large RAM/ROM modules, but the more significant culprit was our reset signal. We had a number of modules with counters (clock dividers, debouncers, filters), as well as modules whose states we would want to reset (position, image), but we had a single wire passed to all the modules. Due to the high fanout, we saw a number of failed routes, and a place and route stage that would take between 10 and 15 minutes. When we started adding reset buffers to each of the modules (registers that stored the input signal), we reduced the route stage to around five minutes, and reduced the number of failed paths by 4-5x.

As mentioned earlier, timing was an especially difficult component. Even reviewing the code now, I can see a few possible timing errors (such as starting at column 1 as opposed to 0). Currently, our screen interfacer expects data to always be sent, even if the image does not change. While the screen does have a buffer and only updates if the buffer changes, a better future design would be to have a “sending signal” that would signal a new update, rather than constant polling. Similarly, it might be better to have a “data ready” input signal which would notify the screen that it can store the current pixel, rather than just assume the value is calculated.

Kendall: A personal takeaway I came away with from this project is the power of enums. Originally in our screen interfacer, every state was given a number, and while there were helpful comments describing each of the states, it was possible to misnumber or have the same number twice. However, when I switched it to enums, I found the code slightly cleaner, and also more difficult to have meaningless transitions.

Claire: This project really drove home the importance of time management. If we hadn’t started the screen so early, we wouldn’t have been able to get as far as we did. I had a lot of fun reading the screen datasheet, and really appreciate whoever designed the display.

Links

- Github: https://github.com/kgarner7/digiteyez
- Projections: https://en.wikipedia.org/wiki/Equirectangular_projection
- Sitronix Datasheet: https://www.rhydolabz.com/documents/33/ST7789.pdf

Appendix

Top level: main.sv

```
`timescale 1ns / 1ps
`default_nettype none

//////////////////////////////////////////////////////////////////////////////////
// controls everything
//////////////////////////////////////////////////////////////////////////////////

module main(
    //sd stuff
    input wire sd_reset,    //  sd reset signal
    input wire sd_dat_0,    //  sd data signal
    //other stuff
    input wire clk_100mhz,  //  clock signal
    input wire btnu,        //  gyroscope calibration
    input wire btnc,        //  to be the reset button
    input wire btnl, btnr,  //  control horizontal scrolling
    input wire jb,          //  uart input
    input wire gyro_enabled,//  switch determining whether to use gyro or buttons
    input wire [3:0] filt,  //  controls low pass filter values
    input wire mono_stereo, //  controls whether to use mono/stereo for images
    input wire [3:0] vert_padding,  //  vertical padding sensitivity (in degrees)
    input wire [2:0] horz_padding,  //  horizontal gyro sensitivity (degrees)
    input wire [2:0] pano_control,  //  selects one of the possible images to
display
    output logic ca, cb, cc, cd, ce, cf, cg, dp,  // segments a-g, dp
    output logic [3:0]  jd,     // sck, mosi,cs, d/c in that order
    output logic [3:0]  jc,     // sck, mosi,cs, d/c in that order
    output logic [7:0]  an,     // Display location 0-7
    output logic sd_cmd,        // sd command dbit
    output logic sd_sck,        // sd sck bit
    output logic sd_dat_1,      // first sd data bit
    output logic sd_dat_2,      // second sd data bit
    output logic sd_dat_3       // third sd data bit
);

    // generate 65mhz clock for uart, 25mhz for sd
    wire clk_65mhz, clk_25mhz;
    clk_wiz_0 clkdivider(
        .clk_in1(clk_100mhz),
        .clk_out1(clk_65mhz),
```

```
    .clk_25mhz(clk_25mhz)
);

// debounce reset, generate buffer
wire reset;
debounce reset_debouncer(
    .reset(1'b0), .clock(clk_100mhz),
    .bounce(btnc), .clean(reset)
);

reg reset_buffer [2:0];

always_ff @(posedge clk_100mhz) begin
    foreach (reset_buffer[i]) begin
        reset_buffer[i] <= reset;
    end
end

// more debouncing
wire calibrate;
debounce calibrate_deouncer(
    .reset(reset_buffer[0]), .clock(clk_100mhz),
    .bounce(btnu), .clean(calibrate)
);

wire gyro;
debounce gyro_debouncer(
    .reset(reset_buffer[0]), .clock(clk_100mhz),
    .bounce(gyro_enabled), .clean(gyro)
);

wire mono_stereo_clean;
debounce mono_stereo_debounce(
    .reset(reset_buffer[0]), .clock(clk_100mhz),
    .bounce(mono_stereo), .clean(mono_stereo_clean)
);

wire [2:0] pano_clean;
debounce #(.SIZE(3)) pano_debounce(
    .reset(reset_buffer[0]), .clock(clk_100mhz),
    .bounce(pano_control), .clean(pano_clean)
);

// calculate vertical and horizontal angles
logic [7:0] vert_angle;
logic [8:0] horz_angle;
```

```
    position_manager manager (
        .clock(clk_65mhz), .reset(reset_buffer[1]),
        .vert_padding(vert_padding), .horz_padding(horz_padding),
.calibrate(calibrate),
        .left_button(btnl), .right_button(btnr),
        .uart_in(jb),
        .filter(filt), .gyro_enabled(gyro),
        .vert_angle(vert_angle), .horz_angle(horz_angle)
    );

    // use coordinates to send an image
    test_image_feeder feeder (
        //sd stuff
        .clk_25mhz(clk_25mhz),
        .sd_cmd(sd_cmd), .sd_sck(sd_sck), .sd_reset(sd_reset),
        .sd_dat_0(sd_dat_0), .sd_dat_1(sd_dat_1), .sd_dat_2(sd_dat_2),
.sd_dat_3(sd_dat_3),
        //other stuff
        .clk_100mhz(clk_100mhz), .rst(reset_buffer[2]),
        .horiz_angle(horz_angle), .vert_angle(vert_angle),
        .mono_stereo(mono_stereo_clean), .pano_control(pano_clean),
        .spi_out_0(jd), .spi_out_1(jc)
    );

    // display debug information on seven segment display
    wire [31:0] data;      //  instantiate 7-segment display; display (8) 4-bit hex
    wire [6:0] segments;

    assign dp = 1'b1;  // turn off the period
    assign data = { filt, vert_padding, 1'b0, horz_padding, 3'b0, horz_angle,
vert_angle };
    assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];

    display_8hex display(
        .clk_in(clk_65mhz),.data_in(data),
        .seg_out(segments),
        .strobe_out(an)
    );
endmodule
```

Coordinates and interfacing with the IMU

bitbang_teensy_serial_mod.imo

```
#include "mpu9255.h"
#include<math.h>
```

```
const int LOOP_SPEED = 10; //milliseconds
int primary_timer = 0;
float x, y, z; //variables for grabbing x,y,and z values (accel)
float xg, yg, zg; //variables for grabbing x,y, and z values (gyro)
MPU9255 imu; //imu object called, appropriately, imu

elapsedMicros comm;
uint8_t comm_pin = 1;
int period = 104; ///period at 9600
//int period = 10; ///period at 9600

uint8_t thresholds[] = {128, 64, 32, 16, 8, 4, 2, 1};



void setup() {
  Serial.begin(115200); //for debugging if needed.
  //Serial1.begin(9600); //for working on Nano will need to convert this to a
software serial (bit-banged UART)
  //or you can also use the fact that the hardware UART used for programming is
broken out onto two pins.  If that
  //is the case, change Serial1.write below to just Serial.write...otherwise,
change it based on the name of your Software serial object
  delay(50); //pause to make sure comms get set up
  Wire.begin();
  delay(50); //pause to make sure comms get set up
  setup_imu();
  pinMode(comm_pin, OUTPUT);

  digitalWrite(comm_pin, 1); //start high like a good serial line should.

  primary_timer = millis();
}

void loop() {
  //read 6 bytes from IMU and store in internal register of imu object
  imu.readAccelData(imu.accelCount);
  x = imu.accelCount[0] * imu.aRes;
  y = imu.accelCount[1] * imu.aRes;
  z = imu.accelCount[2] * imu.aRes;
  imu.readGyroData(imu.gyroCount);//grab gyro stuff and send up after accel stuff.
  xg = imu.gyroCount[0] * imu.gRes;
  yg = imu.gyroCount[1] * imu.gRes;
  zg = imu.gyroCount[2] * imu.gRes;
  uint8_t hey[6] = {};  // the mistake that cost many hours: hey[6]
  hey[0] = (uint8_t)imu.gyroCount[2];
  hey[1] = (uint8_t)(imu.gyroCount[2] >> 8);
```

```
  hey[2] = (uint8_t)imu.accelCount[1];
  hey[3] = (uint8_t)(imu.accelCount[1]>>8);

  joe_write2(hey, 6); //was 6 before. still is in fact
  //Serial.write(hey,6);
  //Serial1.write(hey,6);
  //Serial1.write(imu.accelCount[0],2);
  //Serial1.print(imu.accelCount[0]);
  //Serial1.write(hey[1]);
  //Serial.println(String(hey[0]+20)+","+String(imu.accelCount[0]/256));
  while (millis() - primary_timer < LOOP_SPEED); //wait for primary timer to
increment
  primary_timer = millis();
}

void setup_imu() {
  if (imu.readByte(MPU9255_ADDRESS, WHO_AM_I_MPU9255) == 0x73) {
    imu.initMPU9255();
  } else {
    while (1) Serial.println("NOT FOUND"); // Loop forever if communication doesn't
happen
  }
  imu.getAres(); //call this so the IMU internally knows its range/resolution
}


//MSB first version
void joe_write(uint8_t* invals, int length) {

  for (uint16_t i = 0; i < length; i++) {
    //start bit
    comm = 0;
    digitalWrite(comm_pin, 0);
    while (comm < period);

    comm = 0;
    uint8_t val = (uint8_t)(invals[i]);
    for (uint16_t j = 0; j < 8; j++) {
      if (val >= thresholds[j]) {
        digitalWrite(comm_pin, 1);
        val -= thresholds[j];
      } else {
        digitalWrite(comm_pin, 0);
      }
      while (comm < period);
      comm = 0;
    }
```

```
    digitalWrite(comm_pin, 1);
    while (comm < period);
  }
}

//LSB first version:
void joe_write2(uint8_t* invals, int length) {
  for (uint16_t i = 0; i < length; i++) {
    //start bit
    comm = 0;
    digitalWrite(comm_pin, 0);
    //Serial.print(invals[i]);
    while (comm < period); //start bit

    comm = 0;
    uint16_t val = (uint8_t)(invals[i]);
    //val/=16;
    for (uint16_t j = 0; j < 8; j++) {
      if (val % 2) {
        digitalWrite(comm_pin, 1);
      } else {
        digitalWrite(comm_pin, 0);
      }
      val /= 2;
      while (comm < period);
      comm = 0;
    }

    digitalWrite(comm_pin, 1);
    while (comm < period);
  }
  //Serial.println("");
}
```

---

filter.sv

```
`timescale 1ns / 1ps

module filter#(
    parameter   FILTER_PERIOD = 1_000_00,   // how long to wait before done
                FILTER_SIZE = 16            // size of data to filter
 ) (
    input wire          clock, reset,   // control signal
    input wire [3:0]    filter,         // filter control
    input wire [FILTER_SIZE - 1:0]  data,   // x and y acceleration, calibrated
```

```
    output logic [FILTER_SIZE - 1:0] filtered_data
);

    reg [FILTER_SIZE - 1:0]         previous_data = 0;  // previous states
    reg [2 * FILTER_SIZE - 1:0]     calculated_data;    // calculated values
    reg [$clog2(FILTER_PERIOD) - 1: 0] counter;         // period counter

    logic [25:0] data_signed, data_signed_previous; // stores signed values

    always_comb begin   // sign extend current and previous accelerations
        data_signed = {{FILTER_SIZE {data[FILTER_SIZE - 1]}}, data};

        data_signed_previous = {{FILTER_SIZE {previous_data[FILTER_SIZE - 1]}},
previous_data};
    end

    always_ff @(posedge clock) begin
        if (reset) begin    // reset signal
            counter         <= 0;
            previous_data    <= 0;

            filtered_data   <= 0;
        end else begin
            if (counter == FILTER_PERIOD - 4) begin // calculate filter
                calculated_data <= ((16 - filter) * data_signed + filter *
data_signed_previous) >> 4;
                counter <= FILTER_PERIOD - 3;
            end else if (counter == FILTER_PERIOD - 2) begin // update history and
relative accelerations
                filtered_data   <= calculated_data[FILTER_SIZE - 1:0];
                previous_data   <= calculated_data[FILTER_SIZE - 1:0];
                counter <= 0;
            end else begin
                counter <= counter + 1;
            end
        end
    end
endmodule
```

---

position_manager.sv

```
`timescale 1ns / 1ps
`default_nettype none

module position_manager#(parameter
    FREQUENCY = 65_000_000, // the frequency this module runs at
```

```
    GYRO_BITS = 3,          // how many bits of gyroscope data to use
    GYRO_FREQUENCY = 30     // how often to update gyroscope data
)(
    input wire clock, reset, calibrate,     // control signals
    input wire [3:0] vert_padding,          // vertical sensitivity
    input wire [2:0] horz_padding,          // horizontal sensitivity
    input wire left_button, right_button,   // left and right buttons for manual
scroll
    input wire uart_in,                     // uart data from teensy
    input wire gyro_enabled,                // whether gyro is enabled
    input wire [3:0] filter,                // low pass filter
    output logic [7:0] vert_angle,          // output vertical angle [30, 150]
    output logic [8:0] horz_angle           // output horizontal angle [0, 359]
);

    // synchronize input
    logic uart_sync;
    logic [47:0] uart_data;

    synchronize synchronizer(
        .clock(clock),
        .in(uart_in),
        .out(uart_sync)
    );

    reg reset_buffer [5:0];

    always_ff @(posedge clock) begin
        foreach (reset_buffer[i]) begin
            reset_buffer[i] <= reset;
        end
    end

    // process uart data and pass to low pass filters
    uart_reciever #(.DATA_SIZE(48)) receiver(
        .clock(clock),
        .reset(reset_buffer[0]),
        .data(uart_sync),
        .output_data(uart_data)
    );

    logic [15:0] x_accel_filtered;
    filter x_filter(
        .clock(clock), .reset(reset_buffer[1]),
        .filter(filter), .data(uart_data[15:0]),
        .filtered_data(x_accel_filtered)
    );
```

```
logic [15:0] y_accel_filtered;
filter y_filter(
    .clock(clock), .reset(reset_buffer[2]),
    .filter(filter), .data(uart_data[31:16]),
    .filtered_data(y_accel_filtered)
);

logic left_clean, right_clean;

debounce left_button_debounce(
    .clock(clock), .reset(reset_buffer[3]),
    .bounce(left_button), .clean(left_clean)
);

debounce right_button_debounce(
    .clock(clock), .reset(reset_buffer[3]),
    .bounce(right_button), .clean(right_clean)
);

logic button_enabled;
reg [8:0] current_horz = 180;
logic [8:0] next_horz;

reg [15:0] x_calibrated = 0;
logic [15:0] x_accel_rel;

logic [GYRO_BITS:0]  x_gyro_top;
logic [8:0] x_accel_signed;

// calculate the next horizontal angle positions
always_comb begin
    if (gyro_enabled) begin
        // we look for if the gyroscope reading has exceeded a threshold value
        // controleld by a switch, then move by fixed amount
        x_accel_rel = x_accel_filtered[15:0] - x_calibrated;
        x_gyro_top = x_accel_rel[15:15 - GYRO_BITS - 1];

        if (x_gyro_top[GYRO_BITS]) begin
            if (-x_gyro_top[GYRO_BITS - 1: 0] >= horz_padding) begin
                next_horz = current_horz == 359 ? 0 : current_horz + 2;
            end else begin
                x_accel_signed = 0;
                next_horz = current_horz;
            end
        end else if (x_gyro_top >= horz_padding) begin
```

```
                next_horz = current_horz == 0 ? 359 : current_horz - 2;
            end else begin
                x_accel_signed = 0;
                next_horz = current_horz;
            end
        end else begin
            // gyro disabled; use buttons
            if (left_clean ^ right_clean) begin
                if (left_clean) begin
                    next_horz = current_horz == 0 ? 359 : current_horz - 1;
                end else begin
                    next_horz = current_horz == 360 ? 0 : current_horz + 1;
                end
            end else begin
                next_horz = current_horz;
            end
        end
    end

    logic [15:0] y_shifted, y_unsigned, next_vert;

    // calculate y value. Luckily this can just be truncating the data
    always_comb begin
        y_unsigned = y_accel_filtered[15] ? ~y_accel_filtered + 1 :
y_accel_filtered;
        y_shifted = y_unsigned[15:8];

        if (y_shifted >= 60) begin
            y_shifted = 60;
        end

        if (y_accel_filtered[15]) begin
            next_vert = 90 + y_shifted;
        end else begin
            next_vert = 90 - y_shifted;
        end
    end

    clock_divider #(.FREQUENCY(FREQUENCY), .TARGET_FREQUENCY(GYRO_FREQUENCY))
button_divider(
        .clock(clock), .reset(reset_buffer[4]),
        .divided_clock(button_enabled)
    );

    always_ff @(posedge clock) begin
        if (reset_buffer[5]) begin
            current_horz    <= 180;
```

```
            x_calibrated    <= 0;
        end else begin
            if (calibrate) begin
                x_calibrated    <= uart_data[15:0];
            end

            // only update horizontal every now and then; prevent drift
            if (button_enabled) begin
                horz_angle      <= next_horz;
                current_horz    <= next_horz;
            end

            if (next_vert > vert_angle) begin
                if (next_vert - vert_angle >= vert_padding) begin
                    vert_angle <= next_vert[7:0];
                end
            end else if (vert_angle - next_vert >= vert_padding) begin
                vert_angle  <= next_vert[7:0];
            end
        end
    end
endmodule
```

---

uart_reciever.sv

```
`timescale 1ns / 1ps
`default_nettype none

// uart receiver module
module uart_reciever#(
    parameter   CLOCK_FREQUENCY = 65_000_000,   // the clock frequency in hz
                TARGET_FREQUENCY = 153_600,     // our target frequency in hertz
                WAIT_PERIOD = 428,               // how long to wait at target
frequency for valid data
                DATA_SIZE = 32  // the mistake that cost me many hours
)(
    input wire clock, reset,            // control signals
    input wire data,                    // input data
    output logic [DATA_SIZE - 1:0] output_data     // output data [47:0] { z, y, x
}
);

    logic data_clock;   // the data enable signal

    clock_divider #(
        .FREQUENCY(CLOCK_FREQUENCY),
```

```
        .TARGET_FREQUENCY(TARGET_FREQUENCY)
    ) divider (
        .clock(clock), .reset(reset),
        .divided_clock(data_clock)
    );

    reg [$clog2(WAIT_PERIOD):0] initial_wait = 0;   // used for initial wait
    reg [3:0]   time_counter    = 0;                // counts 16x sample
    reg [2:0]   byte_counter    = 0;                // coutns which byte we are on
    reg [2:0]   bit_counter     = 0;                // counts number of bits per
data
    reg [47:0]  stored_data     = 0;                // stored data; update only
when done

    enum reg [2:0] { WAITING, IDLE, STARTING, READING, STOPPING } state = WAITING;

    always_ff @(posedge clock) begin
        if (reset) begin    // reset?
            bit_counter     <= 0;
            byte_counter    <= 0;
            initial_wait    <= 0;
            output_data     <= 0;
            stored_data     <= 0;
            state           <= WAITING;
            time_counter    <= 0;
        end else if (data_clock) begin
            if (state == WAITING) begin
                output_data     <= 0;

                if (data) begin // waiting for stable high
                    if (initial_wait == WAIT_PERIOD - 1) begin
                        initial_wait    <= 0;
                        state           <= IDLE;
                    end else begin
                        initial_wait    <= initial_wait + 1;
                    end
                end else begin
                    initial_wait    <= 0;
                end
            end else if (state == IDLE) begin

                if (!data) begin // start bit detected
                    bit_counter     <= 0;
                    state           <= STARTING;
                    time_counter    <= 1;
                end
            end else if (state == STARTING) begin
```

```
                time_counter    <= time_counter + 1;

                if (time_counter == 15) begin // 16 samples later, start reading
                    state   <= READING;
                end
            end else if (state == READING) begin
                if (time_counter == 7) begin    // sample in the middle (roughly)
                    stored_data <= { data, stored_data[DATA_SIZE - 1:1] };
                end else if (time_counter == 15) begin  // 16 cycles later
                    bit_counter   <= bit_counter + 1;

                    if (bit_counter == 7) begin
                        state       <= STOPPING;
                    end
                end

                time_counter <= time_counter + 1;
            end else begin  // handle the stop signal, go back to idle
                if (time_counter == 15) begin
                    state        <= IDLE;

                    if (byte_counter == DATA_SIZE / 8 - 1) begin    // if we have
read 6 bytes, done
                        byte_counter    <= 0;
                        output_data     <= stored_data;
                    end else begin
                        byte_counter    <= byte_counter + 1;
                    end
                end

                time_counter <= time_counter + 1;
            end
        end
    end
endmodule
```

Images, SD, and Screens

screen_interfacer.sv

```
`timescale 1ns / 1ps
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
// Written to control the Adafruit 2" color ips TFT display/ST7789,
// requires the spi_send module
//
```

```
//takes standard spi inputs, returns status of the module, more details below
//////////////////////////////////////////////////////////////////////////////////
module screen_interfacer#(parameter
    ms10 = 26'd1000000,  //10 cycle delay for testing without gaps, 1000000 for
real life when we want ms, 200 for testing with gaps
    screen_width = 240,  //replace with the screen width of your hardware
    screen_height = 320, //replace with the screen height of your hardware
    INITIAL_IMAGE_WAIT = 5 //nifty for debugging when you suspect you may be going
too fast, not used in the "non debug" version here
) (
    input wire clk_100mhz, //system clock
    input wire rst, //wired into the system wide reset
    input wire image_done, //high when the image is done and ready to be shipped
off
    input wire [7:0] pixel_in, //byte-wide value to be sent, expects one value per
pixel (so the image will be displayed in grayscale)
    output logic [3:0] spi_out, //four bits wide, connected to jd,
    output logic [5:0] state_out, //for debugging
    output logic pixel_ready, // whether should start read for next chunk
    output logic image_ready, //when we're ready to move on to the next image
    output wire led //largely for debugging, but a nifty indicator in general
);

    logic [30:0] timer; //has to count to 100,000,000 for the second long invert
delay
    enum reg[5:0] {
        RESET, WAKE_FROM_SLEEP,
        SET_COLOR_MODE, SEND_COL_MODE_DATA,
        MEMORY_ACCESS_CONTROL, WRITE_MEMORY_ACCESS_CONTROL,
        CASET_COMMAND, CASET_ZEROS_1, CASET_ZEROS_2, CASET_DATA_1, CASET_DATA_2,
        RASET_COMMAND, RASET_ZEROS_1, RASET_ZEROS_2, RASET_DATA_1, RASET_DATA_2,
        INVERT_ON,
        NORMAL_DISPLAY_ON, DISPLAY_ON,
        DATA_ANNOUNCE, COLOR_SEND, IMAGE_SEND,
        INVERT_LOOP_OFF, INVERT_LOOP_ON
    } state = RESET; //all the states needed for waking the screen

    //registers that help track the general state of the screen
    logic read_ready;
    logic [7:0] to_send_out;
    logic isdata_out;
    logic send_now;
    logic ready_to_send;
    logic cs; //chip select
    logic led_ind;
    logic [1:0] gray_count;
    logic [7:0] pixel_buffer;
```

```
    assign led = led_ind; //for debugging and indication on the finished product
    assign pixel_ready = read_ready && state == IMAGE_SEND && !ready_to_send &&
!image_done; //the combination of events that indicate that we are out of things to
send

    spi_send my_spi( //controls the spi module, which does the actual sending here
        .cs(cs), .clk_100mhz(clk_100mhz), .rst(rst),
        .isdata(isdata_out), .to_send(to_send_out), .send_now(send_now),
        .ready_to_send(ready_to_send),.spi_out(spi_out)
    );

    assign state_out = state; //in case modules above need to be made aware,
largely unnecessary
    //logic [7:0] mosi; //data to be transmitted

    always_ff @(posedge clk_100mhz) begin
        if(rst) begin
            state <= RESET; //reset the state
            send_now <= 1'b0; //don't send stuff yet
            isdata_out <=1'b0; //is a command by default
            timer <= 1'b0;
            cs <= 1'b1;
            led_ind <= 0;
            gray_count <= 0;
            image_ready <= 0;
        end else begin
            case (state)
                RESET: begin //reset to factory settings
                    if(ready_to_send) begin
                        cs <= 1'b0;
                        isdata_out <= 1'b0; //is a command
                        to_send_out <= 8'h01;
                        send_now <= 1'b1;
                        state <= WAKE_FROM_SLEEP;
                    end else begin
                        send_now <= 1'b0;
                    end
                end
                WAKE_FROM_SLEEP: begin //wake up from sleep mode, requires
significant delay before moving on
                    //add a delay 10 ms
                    if(ready_to_send) begin
                        cs <= 1'b1;
                    end
```

```
                    if(ready_to_send && (timer > ms10)) begin
                        isdata_out <= 1'b0; //is a command
                        to_send_out <= 8'h11;
                        cs <= 1'b0;
                        send_now <= 1'b1;
                        timer <= 0;
                        state <= SET_COLOR_MODE;
                    end else begin
                        send_now    <= 1'b0;
                        timer       <= timer + 1;
                    end
                end
                SET_COLOR_MODE: begin //set a color mode, this is just the command
that announces more daya is coming
                    //delay 150ms before proceeding to allow time for the reset
                    if(ready_to_send && (timer > (ms10*15))) begin
                        isdata_out <= 1'b0; //is a command
                        to_send_out <= 8'h3A;
                        send_now <= 1'b1;
                        state <= SEND_COL_MODE_DATA;
                        timer <= 0;
                    end else begin
                        send_now    <= 1'b0;
                        timer       <= timer + 1;
                    end
                end
                SEND_COL_MODE_DATA: begin //send the data that declares which
colmode we'll be using
                    if(ready_to_send && timer > 5) begin //some delay to allow the
screen time to receive data, prevented errors in testing
                        isdata_out <= 1'b1; //is data
                        to_send_out <= 8'h06;
                        send_now <= 1'b1;
                        state <= MEMORY_ACCESS_CONTROL;
                        timer <= 0;
                    end else begin
                        send_now    <= 1'b0;
                        timer       <= timer + 1;
                    end
                end
                MEMORY_ACCESS_CONTROL: begin //memory access control (set a
direction)
                    if(ready_to_send && (timer > ms10)) begin
                        isdata_out <= 1'b0; //is a command
                        to_send_out <= 8'h36;
                        send_now <= 1'b1;
                        state <= WRITE_MEMORY_ACCESS_CONTROL;
```

```
                    timer <= 0;
                end else begin
                    send_now    <= 1'b0;
                    timer       <= timer + 1;
                end
            end
            WRITE_MEMORY_ACCESS_CONTROL: begin //declares that we'll be going
from the top left of the screen to the bottom right
                if(ready_to_send && timer > 5) begin
                    isdata_out <= 1'b0; //is data
                    to_send_out <= 8'h08;
                    send_now <= 1'b1;
                    state <= CASET_COMMAND;
                    timer <= 0;
                end else begin
                    send_now    <= 1'b0;
                    timer       <= timer + 1;
                end
            end
            CASET_COMMAND: begin //set the column address (CA)
                if(ready_to_send && timer > 2) begin
                    isdata_out <= 1'b0; //is a command
                    to_send_out <= 8'h2A;
                    send_now <= 1'b1;
                    state <= CASET_ZEROS_1;
                    timer <= 0;
                end else begin
                    send_now    <= 1'b0;
                    timer       <= timer + 1;
                 end
            end
            CASET_ZEROS_1: begin //We'll be starting at 0, so we send 0
                if(ready_to_send && timer > 2) begin
                    isdata_out <= 1'b1; //is data
                    to_send_out <= 8'h00;
                    send_now <= 1'b1;
                    state <= CASET_ZEROS_2;
                    timer <= 0;
                end else begin
                    send_now    <= 1'b0;
                    timer       <= timer + 1;
                 end
            end
            CASET_ZEROS_2: begin // becuase the CA is 16 bits, we split it into
two parts
                if(ready_to_send && timer > 2) begin
                    isdata_out <= 1'b1; //is data
```

```
                    to_send_out <= 8'h00;
                    send_now <= 1'b1;
                    state <= CASET_DATA_1;
                    timer <= 0;
                end else begin
                    send_now    <= 1'b0;
                    timer       <= timer + 1;
                end
            end
            CASET_DATA_1: begin //here we set the max CA, start with 0 because
the total address needs to be 16 bits
                if(ready_to_send && timer > 2) begin
                    isdata_out <= 1'b1; //is data
                    to_send_out <= 8'h00;
                    send_now <= 1'b1;
                    state <= CASET_DATA_2;
                    timer <= 0;
                end else begin
                    send_now    <= 1'b0;
                    timer       <= timer + 1;
                end
            end
            CASET_DATA_2: begin //sets the max CA
                if(ready_to_send && timer > 3) begin
                    isdata_out <= 1'b1; //is data
                    to_send_out <= 8'd240;
                    send_now <= 1'b1;
                    state <= RASET_COMMAND;
                    timer <= 0;
                end else begin
                    send_now    <= 1'b0;
                    timer       <= timer + 1;
                end
            end
            RASET_COMMAND: begin //Announce that we will be setting the row
address (RA)
                if(ready_to_send && (timer > 3)) begin
                    isdata_out <= 1'b0; //is command
                    to_send_out <= 8'h2B;
                    send_now <= 1'b1;
                    state <= RASET_ZEROS_1;
                    timer <= 0;
                end else begin
                    send_now    <= 1'b0;
                    timer       <= timer + 1;
                end
            end
```

```
RASET_ZEROS_1: begin //start with 0, as with CA
    if(ready_to_send && (timer > 5)) begin
        isdata_out <= 1'b1; //is data
        to_send_out <= 8'd0;
        send_now <= 1'b1;
        state <= RASET_ZEROS_2;
        timer <= 0;
    end else begin
        send_now    <= 1'b0;
        timer       <= timer + 1;
    end
end
RASET_ZEROS_2: begin
    if(ready_to_send && (timer > 5)) begin
        isdata_out <= 1'b1; //is data
        to_send_out <= 8'd0;
        send_now <= 1'b1;
        state <= RASET_DATA_1;
        timer <= 0;
    end else begin
        send_now    <= 1'b0;
        timer       <= timer + 1;
    end
end
RASET_DATA_1: begin //set the first 8 bits of the max row
    if(ready_to_send && (timer > 5)) begin
        isdata_out <= 1'b1; //is data
        to_send_out <= 8'b00000001;
        send_now <= 1'b1;
        state <= RASET_DATA_2;
        timer <= 0;
    end else begin
        send_now    <= 1'b0;
        timer       <= timer + 1;
    end
end
RASET_DATA_2: begin //set the second eight bits
    if(ready_to_send && timer > 2) begin
        isdata_out <= 1'b1; //is data
        to_send_out <= 8'b01000000;
        send_now <= 1'b1;
        state <= INVERT_ON; //
        timer <= 0;
    end else begin
        send_now    <= 1'b0;
        timer       <= timer + 1;
    end
```

```
                end
//                6'd63: begin
//                    if(ready_to_send && timer > 2) begin
//                        isdata_out <= 1'b1; //is data
//                        to_send_out <= 8'b01000000;
//                        send_now <= 1'b1;
//                        state <= 6'd16; //
//                        timer <= 0;
//                    end else begin
//                        send_now <= 1'b0;
//                        timer <= timer +1;
//                    end
//                end
                INVERT_ON: begin //invert the colors, because this screen's scheme
is strange
                   if(ready_to_send && timer > 5) begin
                        isdata_out <= 1'b0; //is command
                        to_send_out <= 8'h21;
                        send_now <= 1'b1;
                        state <= NORMAL_DISPLAY_ON; //
                        timer <= 0;
                    end else begin
                        send_now    <= 1'b0;
                        timer       <= timer + 1;
                    end
                end
                NORMAL_DISPLAY_ON: begin //Turn the display on
                    if(ready_to_send&& timer > ms10) begin
                        isdata_out <= 1'b0; //is command
                        to_send_out <= 8'h13;
                        send_now <= 1'b1;
                        state <= DISPLAY_ON; //
                        timer <= 0;
                    end else begin
                        send_now    <= 1'b0;
                        timer       <= timer + 1;
                    end
                end
                DISPLAY_ON: begin //display on, part 2
                    if(ready_to_send && timer > ms10) begin
                        isdata_out      <= 1'b0; //is command
                        to_send_out     <= 8'h29;
                        send_now        <= 1'b1;
                        state           <= DATA_ANNOUNCE; //
                        timer           <= 0;
                    end else begin
                        send_now    <= 1'b0;
```

```
                    timer       <= timer + 1;
                end
            end
            DATA_ANNOUNCE: begin //announce that we're going to be sending
vals, RAMWR
                if(ready_to_send && timer > 5) begin
                    isdata_out <= 1'b0; //is command
                    to_send_out <= 8'h2C;
                    send_now <= 1'b1;
                    state <= IMAGE_SEND;
                    timer <= 0;
                    image_ready <= 1;
                    read_ready <= 1;
                end else begin
                    send_now    <= 1'b0;
                    timer       <= timer + 1;
                end
            end
//                COLOR_SEND: begin //send a bunch of the same vals
//                    if ((i <230400) && (ready_to_send) && (timer > 2)) begin
//                        isdata_out <= 1'b1;
//                        timer <= 0;
//                        i <= i + 1;
//                        to_send_out <= 8'h55;  //some color
//                        send_now <= 1'b1;
//                        read_ready <= 1;
//                    end else if(i < 230400) begin
//                        timer <= timer + 1;
//                    end else begin
//                        state <= DATA_ANNOUNCE;
//                    end
//                end
            IMAGE_SEND: begin //continuously read an image from memory and
incrementally send it. When we're done, return to the data anounce state to begin
another frame
                image_ready <= 0;

                if (ready_to_send && timer > 5) begin
                    isdata_out <= 1'b1;

                    if (gray_count == 0) begin
                        pixel_buffer <= pixel_in;
                    end

                    read_ready <= gray_count == 0;
                    gray_count <= gray_count == 2 ? 0 : gray_count + 1;
```

```
                    to_send_out <= gray_count == 0 ? pixel_in : pixel_buffer;
                    send_now <= 1'b1;
                end else if (!image_done) begin
                    read_ready  <= 0;
                    send_now    <= 0;

                    if (timer <= 5) begin
                        timer <= timer + 1;
                    end
                end else begin
                    send_now    <= 0;
                    read_ready <= 0;
                    state <= DATA_ANNOUNCE;
                end
            end
            INVERT_LOOP_OFF: begin //invoff, loop back and forth, good for
screen tests, used with invon
                led_ind <= 1;
                if(ready_to_send && (timer >( ms10*100))) begin
                    timer <= 0;
                    isdata_out <= 1'b0; //is command
                    to_send_out <= 8'h20;
                    send_now <= 1'b1;
                    state <= INVERT_LOOP_ON; //

                end else begin
                    timer <= timer + 1;
                    send_now <= 1'b0;
                end
            end
            INVERT_LOOP_ON: begin //invon, loop back and forth
                led_ind <= 0;
                if(ready_to_send && (timer > (ms10*100))) begin
                    timer <= 0;
                    isdata_out <= 1'b0; //is command
                    to_send_out <= 8'h21;
                    send_now <= 1'b1;
                    state <= INVERT_LOOP_OFF; //

                end else begin
                    send_now <= 1'b0;
                    timer <= timer +1;

                    end
            end
            default: begin
                send_now <= 1'b0;
```

```
                end
            endcase
        end
    end
endmodule
```

---

sd_controller.v

```
/* SD Card controller module. Allows reading from and writing to a microSD card
through SPI mode. */

`timescale 1ns / 1ps

module sd_controller(
    output reg cs, // Connect to SD_DAT[3].
    output mosi, // Connect to SD_CMD.
    input miso, // Connect to SD_DAT[0].
    output sclk, // Connect to SD_SCK.
                 // For SPI mode, SD_DAT[2] and SD_DAT[1] should be held HIGH.
                 // SD_RESET should be held LOW.

    input rd,   // Read-enable. When [ready] is HIGH, asseting [rd] will
                // begin a 512-byte READ operation at [address].
                // [byte_available] will transition HIGH as a new byte has been
                // read from the SD card. The byte is presented on [dout].
    output reg [7:0] dout, // Data output for READ operation.
    output reg byte_available, // A new byte has been presented on [dout].

    input wr,   // Write-enable. When [ready] is HIGH, asserting [wr] will
                // begin a 512-byte WRITE operation at [address].
                // [ready_for_next_byte] will transition HIGH to request that
                // the next byte to be written should be presentaed on [din].
    input [7:0] din, // Data input for WRITE operation.
    output reg ready_for_next_byte, // A new byte should be presented on [din].

    input reset, // Resets controller on assertion.
    output ready, // HIGH if the SD card is ready for a read or write operation.
    input [31:0] address,   // Memory address for read/write operation. This MUST
                            // be a multiple of 512 bytes, due to SD sectoring.
    input clk,  // 25 MHz clock.
    output [4:0] status // For debug purposes: Current state of controller.
);

    parameter RST = 0;
    parameter INIT = 1;
    parameter CMD0 = 2;
```

```
parameter CMD55 = 3;
parameter CMD41 = 4;
parameter POLL_CMD = 5;

parameter IDLE = 6;
parameter READ_BLOCK = 7;
parameter READ_BLOCK_WAIT = 8;
parameter READ_BLOCK_DATA = 9;
parameter READ_BLOCK_CRC = 10;
parameter SEND_CMD = 11;
parameter RECEIVE_BYTE_WAIT = 12;
parameter RECEIVE_BYTE = 13;
parameter WRITE_BLOCK_CMD = 14;
parameter WRITE_BLOCK_INIT = 15;
parameter WRITE_BLOCK_DATA = 16;
parameter WRITE_BLOCK_BYTE = 17;
parameter WRITE_BLOCK_WAIT = 18;

parameter WRITE_DATA_SIZE = 515;
parameter BOOT_COUNTER_WAIT = 27'd100_000_000;
reg [4:0] state = RST;
assign status = state;
reg [4:0] return_state;
reg sclk_sig = 0;
reg [55:0] cmd_out;
reg [7:0] recv_data;
reg cmd_mode = 1;
reg [7:0] data_sig = 8'hFF;

reg [9:0] byte_counter;
reg [9:0] bit_counter;

reg [26:0] boot_counter = BOOT_COUNTER_WAIT;
always @(posedge clk) begin
    if(reset == 1) begin
        state <= RST;
        sclk_sig <= 0;
        boot_counter <= BOOT_COUNTER_WAIT; //change back to
    end
    else begin
        case(state)
            RST: begin
                if(boot_counter == 0) begin
                    sclk_sig <= 0;
                    cmd_out <= {56{1'b1}};
                    byte_counter <= 0;
                    byte_available <= 0;
```

```
            ready_for_next_byte <= 0;
            cmd_mode <= 1;
            bit_counter <= 160;
            cs <= 1;
            state <= INIT;
        end
        else begin
            boot_counter <= boot_counter - 1;
        end
    end
    INIT: begin
        if(bit_counter == 0) begin
            cs <= 0;
            state <= CMD0;
        end
        else begin
            bit_counter <= bit_counter - 1;
            sclk_sig <= ~sclk_sig;
        end
    end
    CMD0: begin
        cmd_out <= 56'hFF_40_00_00_00_00_95;
        bit_counter <= 55;
        return_state <= CMD55;
        state <= SEND_CMD;
    end
    CMD55: begin
        cmd_out <= 56'hFF_77_00_00_00_00_01;
        bit_counter <= 55;
        return_state <= CMD41;
        state <= SEND_CMD;
    end
    CMD41: begin
        cmd_out <= 56'hFF_69_00_00_00_00_01;
        bit_counter <= 55;
        return_state <= POLL_CMD;
        state <= SEND_CMD;
    end
    POLL_CMD: begin
        if(recv_data[0] == 0) begin
            state <= IDLE;
        end
        else begin
            state <= CMD55;
        end
    end
    IDLE: begin
```

```
    if(rd == 1) begin
        state <= READ_BLOCK;
    end
    else if(wr == 1) begin
        state <= WRITE_BLOCK_CMD;
    end
    else begin
        state <= IDLE;
    end
end
READ_BLOCK: begin
    cmd_out <= {16'hFF_51, address, 8'hFF};
    bit_counter <= 55;
    return_state <= READ_BLOCK_WAIT;
    state <= SEND_CMD;
end
READ_BLOCK_WAIT: begin
    if(sclk_sig == 1 && miso == 0) begin
        byte_counter <= 511;
        bit_counter <= 7;
        return_state <= READ_BLOCK_DATA;
        state <= RECEIVE_BYTE;
    end
    sclk_sig <= ~sclk_sig;
end
READ_BLOCK_DATA: begin
    dout <= recv_data;
    byte_available <= 1;
    if (byte_counter == 0) begin
        bit_counter <= 7;
        return_state <= READ_BLOCK_CRC;
        state <= RECEIVE_BYTE;
    end
    else begin
        byte_counter <= byte_counter - 1;
        return_state <= READ_BLOCK_DATA;
        bit_counter <= 7;
        state <= RECEIVE_BYTE;
    end
end
READ_BLOCK_CRC: begin
    bit_counter <= 7;
    return_state <= IDLE;
    state <= RECEIVE_BYTE;
end
SEND_CMD: begin
    if (sclk_sig == 1) begin
```

```
            if (bit_counter == 0) begin
                state <= RECEIVE_BYTE_WAIT;
            end
            else begin
                bit_counter <= bit_counter - 1;
                cmd_out <= {cmd_out[54:0], 1'b1};
            end
        end
        sclk_sig <= ~sclk_sig;
    end
    RECEIVE_BYTE_WAIT: begin
        if (sclk_sig == 1) begin
            if (miso == 0) begin
                recv_data <= 0;
                bit_counter <= 6;
                state <= RECEIVE_BYTE;
            end
        end
        sclk_sig <= ~sclk_sig;
    end
    RECEIVE_BYTE: begin
        byte_available <= 0;
        if (sclk_sig == 1) begin
            recv_data <= {recv_data[6:0], miso};
            if (bit_counter == 0) begin
                state <= return_state;
            end
            else begin
                bit_counter <= bit_counter - 1;
            end
        end
        sclk_sig <= ~sclk_sig;
    end
    WRITE_BLOCK_CMD: begin
        cmd_out <= {16'hFF_58, address, 8'hFF};
        bit_counter <= 55;
        return_state <= WRITE_BLOCK_INIT;
        state <= SEND_CMD;
     ready_for_next_byte <= 1;
    end
    WRITE_BLOCK_INIT: begin
        cmd_mode <= 0;
        byte_counter <= WRITE_DATA_SIZE;
        state <= WRITE_BLOCK_DATA;
        ready_for_next_byte <= 0;
    end
    WRITE_BLOCK_DATA: begin
```

```
                if (byte_counter == 0) begin
                    state <= RECEIVE_BYTE_WAIT;
                    return_state <= WRITE_BLOCK_WAIT;
                end
                else begin
                    if ((byte_counter == 2) || (byte_counter == 1)) begin
                        data_sig <= 8'hFF;
                    end
                    else if (byte_counter == WRITE_DATA_SIZE) begin
                        data_sig <= 8'hFE;
                    end
                    else begin
                        data_sig <= din;
                        ready_for_next_byte <= 1;
                    end
                    bit_counter <= 7;
                    state <= WRITE_BLOCK_BYTE;
                    byte_counter <= byte_counter - 1;
                end
            end
            WRITE_BLOCK_BYTE: begin
                if (sclk_sig == 1) begin
                    if (bit_counter == 0) begin
                        state <= WRITE_BLOCK_DATA;
                        ready_for_next_byte <= 0;
                    end
                    else begin
                        data_sig <= {data_sig[6:0], 1'b1};
                        bit_counter <= bit_counter - 1;
                    end;
                end;
                sclk_sig <= ~sclk_sig;
            end
            WRITE_BLOCK_WAIT: begin
                if (sclk_sig == 1) begin
                    if (miso == 1) begin
                        state <= IDLE;
                        cmd_mode <= 1;
                    end
                end
                sclk_sig = ~sclk_sig;
            end
        endcase
    end
end

assign sclk = sclk_sig;
```

```
    assign mosi = cmd_mode ? cmd_out[55] : data_sig[7];
    assign ready = (state == IDLE);
endmodule
```

---

spi_send.sv

```
/*
Written by ctraweek in 2019 for 6.111 to control the Adafruit TFT display breakout
for the ST7789
*/

`default_nettype none
`timescale 1ns / 1ps

module spi_send#(
    parameter SPI_CLOCK_WAIT = 1 ) // This introduces further delays into the SPI
clock. In theory it would be possible to push the SPI clock too fast, and have
communication break down, but with our current clock setup this is not as much of a
problem so this number is quite low
(
    input wire clk_100mhz, //the system clock input
    input wire rst, //a reset signal
    input wire isdata, //A binary value that signifies data or command mode
    input wire [7:0] to_send, //The eight bit value that will be sent over SPI. The
expectation is that this value is held constant until the SPI signals that it's
done
    input wire cs, //control for the chip select pin of the spi. Low means the chip
is selected. Because this module only really controls one piece of hardware at a
time, this is generally low, however, if you wanted to daisy chain devices you
would need multiple pins to selec the device
    input wire send_now, //sends the data on to_send when high
    output logic ready_to_send, //high when the module is finished sending the
current val and is ready to accept another val
    output logic [3:0] spi_out, // should go directly to the output pins on the
FPGA: sck, MOSI, cs, d/c
    output logic [7:0] currently_sending //This is mostly here for legacy reasons
and is a nifty debug tool, signifies the val that is currently being sent
);

    logic mosi;
    logic [3:0] bitcount;
    logic [10:0] clk_count;
    logic spi_clk_out;
    logic ready_to_send_out;
    logic sending; //high if a send is in process
```

```
    assign spi_out = {spi_clk_out, mosi,cs,isdata};
    assign ready_to_send = ready_to_send_out;

    always_ff @(posedge clk_100mhz) begin
        if(rst) begin
            //set to the default vals
            spi_clk_out <= 0;
            mosi <= 1'b0;
            bitcount <= 4'd0;
            clk_count <= 1'b0;
            ready_to_send_out <= 1'b1;
            sending <= 1'b0;
        end else begin //normal operation
            if(ready_to_send_out && send_now)begin //kick off the send
                currently_sending   <= to_send;
                //reset the bit count
                bitcount <=3'd0;
                //reset the clock--most devices read data on the rising edge
                spi_clk_out <= 1'b0; //spi clock starts low
                ready_to_send_out <= 1'b0; //prevents this block from running
multiple times per byte
                clk_count = 0;//give a full spi low to figure things out
                sending <= 1'b1; //announce that we're oging to send
                mosi <= to_send[7]; //get MOSI (data out) set up with the first
value. The hardware we worked with expected things backwards, so that is what this
is written for
                //mosi <= to_send[7]; //load the first value
            end

            //only runs whenever the spi is supposed to be changing
            if ((clk_count == SPI_CLOCK_WAIT) && sending) begin //the logic here
allows you to slow the spi clock in the event that you're going "too fast" for your
device
                spi_clk_out <= !spi_clk_out;
                if(spi_clk_out == 1) begin //means this is the falling edge, and
we're clear to change states so the correct state will be read on the rising edge
                    if(bitcount != 3'd7) begin //iterate through the bits you are
suppoesd to send
                        mosi <= to_send[6-bitcount];
                        bitcount <= bitcount + 1;
                    end else begin
                        //we've finished are are awaiting more data
                        mosi <= to_send[0];
                        ready_to_send_out <= 1'b1;
                        sending <= 1'b0;
                    end
```

```
                end
                clk_count <= 0;
            end else begin
                clk_count <= clk_count +1;
            end
        end

    end
endmodule
```

---

test_image_feeder.sv

```
`default_nettype none
`timescale 1ns / 1ps

module test_image_feeder#(parameter
    MONO_WIDTH = 830,       // max width of a mon image
    MONO_HEIGHT = 415,      // max height of a mono image
    STEREO_WIDTH = 600,     // max width of a stereo image
    STEREO_HEIGHT = 300,    // max height of a stereo image
    screen_width = 240,     // the width of the screen
    screen_height = 320,    // the height of the screen
    ms10 = 26'd1000000      // how many cycles is 10ms
    //sd params
)(
    //sd stuff

    input wire sd_reset,    // sd reset signal
    input wire sd_dat_0,    // sd data signal
    //other stuff
    input wire clk_100mhz,  // clock
    input wire rst,         // reset
    input wire mono_stereo, // toggle mono vs stereo
    input wire [2:0] pano_control,  // select one of eight images
    input wire [7:0] vert_angle,    // current vertical angle
    input wire [8:0] horiz_angle,   // current horizontal angle
    input wire clk_25mhz,           // sd clock
    output logic [3:0] spi_out_0, spi_out_1, //four bits wide, connected to jd,
    output logic sd_cmd,
    output logic sd_sck,
    output logic sd_dat_1,
    output logic sd_dat_2,
    output logic sd_dat_3
);

    // calcualte max width, height, and size for address indexing
```

```
    localparam MAX_WIDTH = MONO_WIDTH > STEREO_WIDTH ? MONO_WIDTH : STEREO_WIDTH;
    localparam MAX_HEIGHT = MONO_HEIGHT > STEREO_HEIGHT ? MONO_HEIGHT :
STEREO_HEIGHT;
    localparam MAX_SIZE = MONO_WIDTH * MONO_HEIGHT > 2 * STEREO_WIDTH *
STEREO_HEIGHT ?
        MONO_WIDTH * MONO_HEIGHT : 2 * STEREO_WIDTH * STEREO_HEIGHT;

    reg reset_buffer [3:0];

    always_ff @(posedge clk_100mhz) begin
        foreach (reset_buffer[i]) begin
            reset_buffer[i] <= rst;
        end
    end

    //------sd vars------//
    logic [15:0] sw_debounce;
    logic check_sw;
    logic sd_initialization_state; //true if the sd card is setting itself up,
basically a hacky way to divert into the sd state machine before the screen does
anything
    enum reg[4:0]{
        RESET, WAIT_FOR_READY, SEND, WAIT_FOR_SENT, UPDATE_ADDRESS, PLAYBACK
    } state = RESET;
    logic byte_available_debounce;
    logic ready; //ready to write, signal from sd card
    logic [31:0] address; //memory address for the read operation
    logic rd; //read enable
    logic [7:0] dout; //data output
    logic byte_available; //a new byte is here
    logic wr;
    logic [10:0] inner_sector_counter; //keeps track of our place in the secotor
    logic [10:0] sector_count; //keeps track of the number of sectors we've run
through, so we know when we're done reading an image overall

    //sd constants
    assign sd_dat_2 = 1;
    assign sd_dat_1 = 1;
    assign sd_reset = 0;
    assign wr = 0;

    //the sd controller iteslf
    sd_controller sd(.reset(reset_buffer[3]),
         .clk(clk_25mhz),
         .cs(sd_dat_3),
         .mosi(sd_cmd),
         .miso(sd_dat_0),
```

```
     .sclk(sd_sck),
     .ready(ready),
     .address(address),
     .rd(rd),
     .wr(wr),
     .dout(dout),
     .byte_available(byte_available)
 );

//-----block mem vars-----//
logic [7:0] data_in;
logic [7:0] data_out_left;
logic [7:0] data_out_right;
logic write_enable_a;
logic [18:0] im_pos_counter; //counts the position of the image
assign data_in = dout; //data in is always whatevers coming off the sd
//block mem itself
image_map_coe memgen(
    .clka(clk_100mhz), .clkb(clk_100mhz),
    .addra(addr_left), .addrb(addr_right),
    .dina(data_in), .dinb(8'b0),
    .douta(pixel_left), .doutb(pixel_right),
    .wea(write_enable_a), .web(1'b0)
);

//-----other vars-----//
enum reg [1:0] {
    MONO = 0,
    STEREO = 1,
    NONE = 2
} current_state = NONE;

// determine current width of image
logic [$clog2(MAX_WIDTH):0] current_width;
assign current_width = current_state == STEREO ? STEREO_WIDTH : MONO_WIDTH;

reg valid_data = 1;
logic [10:0]    dividend;
logic [8:0]     divisor;
logic [17:0]    divider_out;

logic [17:0] height_scaling, width_scaling;

// divide image width by 360, get 8-bit fraction
div_gen_0 divider(
    .s_axis_dividend_tdata(dividend),
    .s_axis_dividend_tvalid(valid_data),
```

```
        .s_axis_divisor_tdata(divisor),
        .s_axis_divisor_tvalid(valid_data),
        .m_axis_dout_tdata(divider_out),
        .aclk(clk_100mhz)
    );

    // when triggered, calculate height and width as values with 8-bit fraction
    reg [4:0] divider_counter = 27;

    always_ff @(posedge clk_100mhz) begin
        if (reset_buffer[2]) begin
            current_state <= NONE;
        end else begin
            if (current_state != mono_stereo) begin
                current_state   <= mono_stereo ? STEREO : MONO;
                divider_counter <= 0;
            end
        end

        if (current_state != NONE) begin
            case (divider_counter)
                0: begin
                    dividend    <= current_state == MONO ? MONO_WIDTH :
STEREO_WIDTH;
                    divisor     <= 360;
                    divider_counter <= 1;
                end
                2: begin
                    dividend    <= current_state == MONO ? MONO_HEIGHT :
STEREO_HEIGHT;
                    divisor     <= 180;
                    divider_counter <= 3;
                end
                22: begin
                    width_scaling   <= divider_out;
                    divider_counter <= 23;
                end
                25: begin
                    height_scaling  <= divider_out;
                    divider_counter <= 26;
                    valid_data      <= 0;
                end
                27: begin end
                default: divider_counter <= divider_counter + 1;
            endcase
        end
    end
```

```
    logic [17:0] horiz_count, vert_count, vert_index;
    logic [$clog2(MAX_WIDTH):0] col_count;
    logic [$clog2(MAX_HEIGHT):0] row_count;

    logic [17:0] next_horiz_angle, scaled_horiz_angle;
    logic [19:0] multiplied_horz_pos;

    // scale horizontal angle to image
    always_ff @(posedge clk_100mhz) begin
        scaled_horiz_angle = { 2'b0, horiz_count } + { horiz_angle, 2'b0 };

        if (scaled_horiz_angle[17:2] >= 360) begin
            next_horiz_angle = scaled_horiz_angle - (360 * 4);
        end else begin
            next_horiz_angle = scaled_horiz_angle;
        end

        multiplied_horz_pos <= next_horiz_angle * { 2'b0, width_scaling };

        if (multiplied_horz_pos[19:10] >= current_width) begin
            col_count <= multiplied_horz_pos[19:10] - current_width;
        end else begin
            col_count <= multiplied_horz_pos[19:10];
        end
    end

    // scale vertical angle to image
    logic [17:0] scaled_vert_angle;
    logic [21:0] multiplied_vert_pos;

    always_ff @(posedge clk_100mhz) begin
        scaled_vert_angle   = {{4{vert_index[17]}}, vert_index } + { vert_angle,
4'b0 };
        multiplied_vert_pos <= scaled_vert_angle * { 4'b0, height_scaling };
        row_count           <= multiplied_vert_pos[21:12];
    end

    // interface with screens
    logic [7:0] pixel_left, pixel_right;
    logic [$clog2(MAX_SIZE):0] addr_left, addr_right;

    reg image_done = 0;
    logic read_ready_left, read_ready_right;
    logic image_ready_left, image_ready_right;

    screen_interfacer#(
```

```
    .ms10(ms10),
    .screen_width(screen_width),
    .screen_height(screen_height)
) left(
    .clk_100mhz(clk_100mhz), .rst(reset_buffer[0]),
    .pixel_in(pixel_left), .image_done(image_done),
    .image_ready(image_ready_left), .pixel_ready(read_ready_left),
    .spi_out(spi_out_0)
);

screen_interfacer#(
    .ms10(ms10),
    .screen_width(screen_width),
    .screen_height(screen_height)
) right (
    .clk_100mhz(clk_100mhz), .rst(reset_buffer[1]),
    .pixel_in(pixel_right), .image_done(image_done),
    .image_ready(image_ready_right), .pixel_ready(read_ready_right),
    .spi_out(spi_out_1)
);

// image control signals
logic image_ready, read_ready;
assign read_ready = read_ready_left && read_ready_right;
assign image_ready = image_ready_left && image_ready_right;

logic [32:0] large_pano_start;//= 32'h77800,//32'h57800,//9 //32'h400
logic [16:0] large_pano_number_of_sectors;//= 10'd673 //10'd672

logic [2:0] pano_control_buffer;

// store static offsets for images in sd
reg [0:7][31:0] pano_start = '{
    32'h77800,
    32'hCBA00,
    32'h11FC00,
    32'h177C00,
    32'h1CBE00,
    32'h223E00,
    32'h27BE00,
    32'h2D3E00
};

reg [0:7][16:0] pano_sectors = '{
    10'd673,
    10'd673,
    10'd704,
```

```
        10'd673,
        10'd704,
        10'd704,
        10'd704,
        10'd673
    };

    always_ff @(posedge clk_100mhz) begin
        if (rst || (pano_control_buffer != pano_control)) begin
            addr_left       <= 0;
            addr_right      <= 0;
            horiz_count     <= 0;
            image_done      <= 0;
            vert_count      <= 0;
            vert_index      <= -480;

            //added for sd: load image into bram
            sd_initialization_state <= 1;
            state <= RESET;
            pano_control_buffer <= pano_control;
            check_sw <= 1;
            large_pano_start <= pano_start[0];
            large_pano_number_of_sectors <= pano_sectors[0];
        end else if (check_sw) begin
            //check for switch changes. If this is the case we'll want to jump back
into sd initialization (after we've made the proper changes)
            check_sw <= 0;
            large_pano_start <= pano_start[pano_control_buffer];
            large_pano_number_of_sectors <= pano_sectors[pano_control_buffer];
            state <= RESET; //go through the whole write to screen process again
        end else if (sd_initialization_state == 1) begin
            //diversion into sd state machine
            case (state)
                RESET: begin
                    im_pos_counter <= 0; //starts at the first pixel in the image
                    address <= large_pano_start;
                    state <= WAIT_FOR_READY;
                    rd <= 0;

                    inner_sector_counter <= 0;
                    addr_left <= 0;
                    addr_right <= 0;
                    sector_count <= 0;
                    byte_available_debounce<=0;
                end
                WAIT_FOR_READY: begin
                    if (ready) begin //ready to read a byte
```

```
                        rd <= 1; //send the thing
                        state <= WAIT_FOR_SENT;
                    end
                end
                WAIT_FOR_SENT: begin
                    write_enable_a <= byte_available;
                    if (inner_sector_counter == 10'd512) begin
                        state <= UPDATE_ADDRESS;
                        inner_sector_counter <= 0;
                    end
                    else if (byte_available)begin
                        if (!byte_available_debounce) begin
                            //save things to ram
                            im_pos_counter <= im_pos_counter + 1;
                            inner_sector_counter <= inner_sector_counter + 1;
                            byte_available_debounce <= 1;
                        end
                    end else if (!byte_available && byte_available_debounce) begin
                        //must happen every time
                        addr_left <= addr_left + 1;
                        byte_available_debounce <= 0;
                    end
                end
                UPDATE_ADDRESS: begin
                    write_enable_a <= 0; //make sure we're not writing anymore
                    rd <= 0; //reset rd so we're ready for next time
                    if (sector_count < large_pano_number_of_sectors-1) begin //flip
to next address
                        address <= address + 32'd512;
                        sector_count <= sector_count + 1;
                        state <= WAIT_FOR_READY;

                    end else begin //otherwise its time to move on
                        state <= PLAYBACK;
                        addr_left <= 0;
                        addr_right <= 0;
                    end
                end
                PLAYBACK: begin
                    //no longer go through the state machine, leave the bram be
                    sd_initialization_state <= 0;
                    write_enable_a <= 0;
                end
            endcase
        end else if (image_ready) begin
            // image is ready, fetch next bit
            addr_left   <= 0;
```

```
            addr_right  <= 0;
            horiz_count <= 1; // should probably be zero
            vert_count  <= 0;
            vert_index  <= -480;
            image_done  <= 0;
        end else if (read_ready && vert_count < screen_height) begin
            addr_left   <= (row_count * current_width) + col_count;

            // calculate the right screen address
            if (current_state == STEREO) begin
                addr_right  <= (row_count * current_width) + col_count +
STEREO_WIDTH * STEREO_HEIGHT;
            end else begin
                addr_right  <= (row_count * current_width) + col_count;
            end

            if (horiz_count == (screen_width-1)) begin //begin a new row
                horiz_count <= 0;
                vert_count  <= vert_count + 1;
                vert_index  <= vert_index + 3;
            end else begin
                horiz_count <= horiz_count + 1;
            end
        end else if (vert_count == screen_height) begin
            image_done <= 1;
        end
    end
endmodule
```

Utility

coe_to_hex.py

```
def converter():
      print("beginning")
      readingline = 3
      res = ""
      F = open("usable_coes/hoverla.coe", "r")
      new_file = open("usable_hex/hoverla_hex.txt", "w")
      for line in F:
            if("0" in line or "1" in line):
                  res+=(str(hex((int(line[0:4],2))))[-1])
                  res+=(str(hex((int(line[4:8],2))))[-1])
```

```
    new_file.write(res)
```

---

clock_divider.sv

```
`timescale 1ns / 1ps
`default_nettype none

module clock_divider#(
    parameter FREQUENCY = 65_000_000,
              TARGET_FREQUENCY = 153600
)(
    input wire clock, reset,    // control signals
    output logic divided_clock  // short pulse only on after target frequency
cycles
);
    localparam COUNTER = FREQUENCY / TARGET_FREQUENCY;

    reg [$clog2(COUNTER): 0] counter = 0;

    always_ff @(posedge clock) begin
        if (reset) begin
            counter <= 0;
        end else begin
            if (counter == COUNTER - 1) begin
                divided_clock <= 1;
                counter <= 0;
            end else begin
                divided_clock <= 0;
                counter <= counter + 1;
            end
        end
    end
endmodule
```

---

debounce.sv

```
`default_nettype none

/**
 * parameterized debouncer module
 * Takes in a bouncy input and only returns the output once deemed
 * stable for 0.01 seconds.
 */
module debounce#(
    parameter COUNT = 1_000_000, // how long to debounce
```

```
            SIZE = 1      // the size of the value to debounce
)(
    input wire                  clock,  //clock in
                                reset,  //reset in
    input wire [SIZE - 1: 0]    bounce, //raw input to the system
    output logic [SIZE - 1: 0]  clean   //debounced output
);

    logic [$clog2(COUNT) - 1:0] count; // is 20 bits enough? yes, but barely
    logic [SIZE - 1: 0] old;    // parameterized old value

    always_ff @(posedge clock) begin
        // reset?
        if (reset) begin
            count <= 16'b0;
            clean <= 1'b0;
        end else begin
            if (old != bounce) begin
                count <= 16'b0;
            end else begin
                //
                if (count >= COUNT - 1) begin
                    clean <= old;
                end else begin
                    count <= count + 1;
                end

            end
            old <= bounce;
        end
    end
endmodule
```

---

display_8hex.sv

```
`timescale 1ns / 1ps


//////////////////////////////////////////////////////////////////////////////////
// Engineer:   g.p.hom
//
// Create Date:    18:18:59 04/21/2013
// Module Name:    display_8hex
// Description:  Display 8 hex numbers on 7 segment display
//
//////////////////////////////////////////////////////////////////////////////////
```

```
module display_8hex(
    input wire          clk_in,     // system clock
    input wire [31:0]   data_in,    // 8 hex numbers, msb first
    output reg [6:0]    seg_out,    // seven segment display output
    output reg [7:0]    strobe_out  // digit strobe
);

    localparam bits = 13;

    reg [bits:0] counter = 0;  // clear on power up

    wire [6:0] segments[15:0]; // 16 7 bit memorys
    assign segments[0]  = 7'b100_0000;  // inverted logic
    assign segments[1]  = 7'b111_1001;  // gfedcba
    assign segments[2]  = 7'b010_0100;
    assign segments[3]  = 7'b011_0000;
    assign segments[4]  = 7'b001_1001;
    assign segments[5]  = 7'b001_0010;
    assign segments[6]  = 7'b000_0010;
    assign segments[7]  = 7'b111_1000;
    assign segments[8]  = 7'b000_0000;
    assign segments[9]  = 7'b001_1000;
    assign segments[10] = 7'b000_1000;
    assign segments[11] = 7'b000_0011;
    assign segments[12] = 7'b010_0111;
    assign segments[13] = 7'b010_0001;
    assign segments[14] = 7'b000_0110;
    assign segments[15] = 7'b000_1110;

    always_ff @(posedge clk_in) begin
      // Here I am using a counter and select 3 bits which provides
      // a reasonable refresh rate starting the left most digit
      // and moving left.
      counter <= counter + 1;
      case (counter[bits:bits-2])
          3'b000: begin  // use the MSB 4 bits
                    seg_out <= segments[data_in[31:28]];
                    strobe_out <= 8'b0111_1111 ;
                end
          3'b001: begin
                    seg_out <= segments[data_in[27:24]];
                    strobe_out <= 8'b1011_1111 ;
                end

          3'b010: begin
```

```
                    seg_out <= segments[data_in[23:20]];
                    strobe_out <= 8'b1101_1111 ;
                   end
           3'b011: begin
                   seg_out <= segments[data_in[19:16]];
                   strobe_out <= 8'b1110_1111;
                  end
           3'b100: begin
                     seg_out <= segments[data_in[15:12]];
                     strobe_out <= 8'b1111_0111;
                  end
           3'b101: begin
                   seg_out <= segments[data_in[11:8]];
                   strobe_out <= 8'b1111_1011;
                  end

           3'b110: begin
                    seg_out <= segments[data_in[7:4]];
                    strobe_out <= 8'b1111_1101;
                   end
           3'b111: begin
                   seg_out <= segments[data_in[3:0]];
                   strobe_out <= 8'b1111_1110;
                  end

       endcase
      end
endmodule
```

---

synchronize.sv

```
`timescale 1ns / 1ps
`default_nettype none

module synchronize #(parameter NSYNC = 3)  // number of sync flops.  must be >= 2
                   (input wire clock, in,
                    output reg out);

  reg [NSYNC-2:0] sync;

  always_ff @ (posedge clock)
  begin
    {out,sync} <= {sync[NSYNC-2:0],in};
  end
endmodule
```