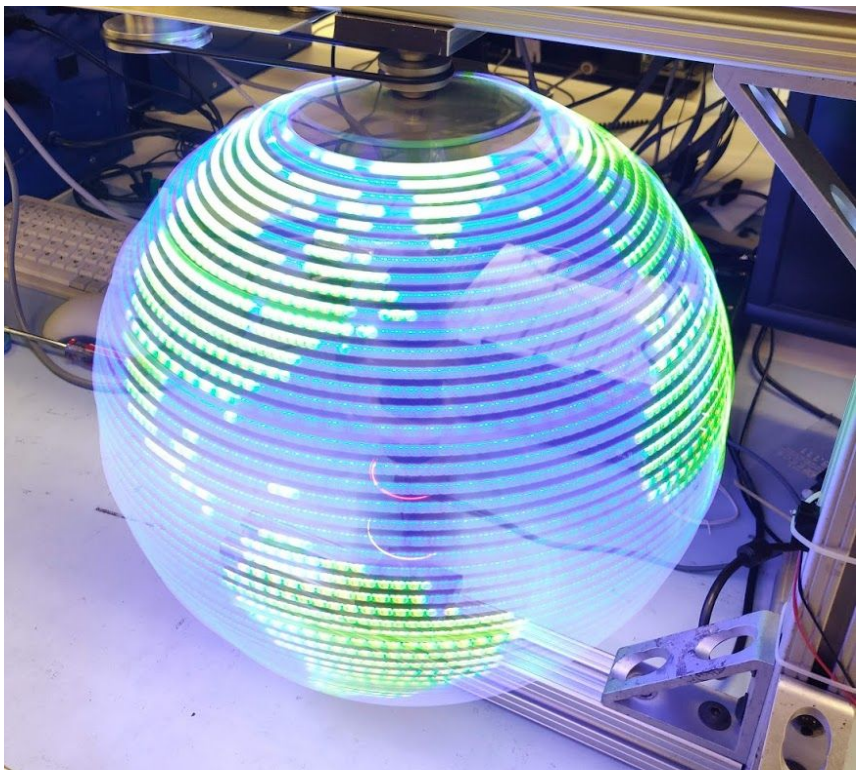


6.111 Final Project: Spherical Persistence of Vision Display

Luis Torres and Noah Moroze

Table of Contents

Introduction

For our final project we developed an FPGA-controlled spherical persistence of vision display. The concept behind the display is that by spinning a single strip of lights at high speed and precisely controlling which LEDs are lit at any given position, we can take advantage of the eye's slow response time to create the illusion of a continuous display.

Our device features a strip of 52 individually controllable RGB LEDs mounted around the edge of a rotating ring. We have a resolution of 52 pixels tall and 256 pixels around, and the device can be programmed to display any arbitrary 256x52 bitmap file.

BOM

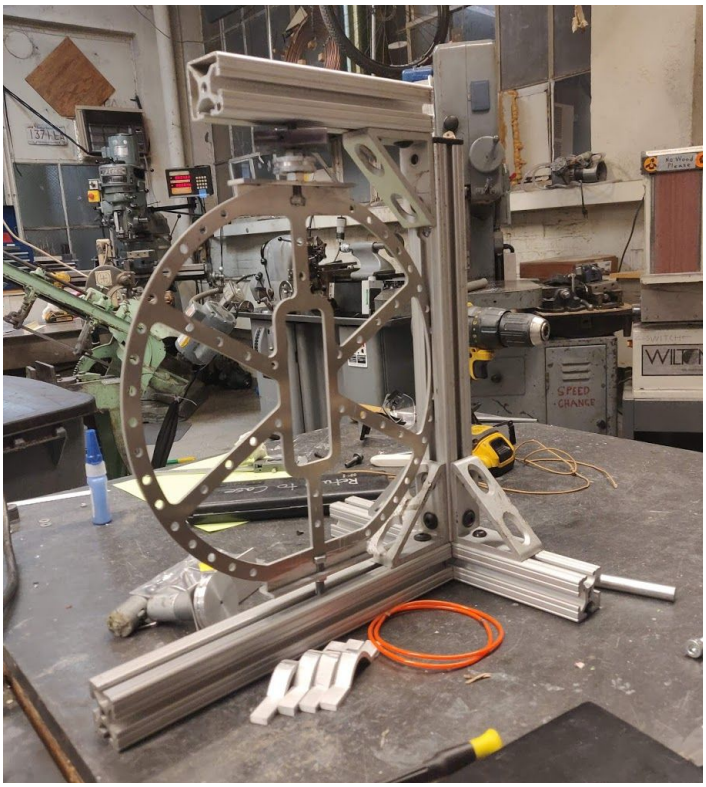
Item	Link/Source	Cost
Frame	Scrap stock from MIT makerspaces	n/a
LED strip	https://www.adafruit.com/product/2241	$50

IR break beam sensor	https://www.adafruit.com/product/2168	$6.50
Motor	Already had	n/a
Cmod A7	Borrow from staff (https://store.digilentinc.com/cmod-a7-breadboardable-artix-7-fpga-module/)	n/a
Battery pack	https://www.amazon.com/gp/product/B01CU1EC6Y	$20

Hardware

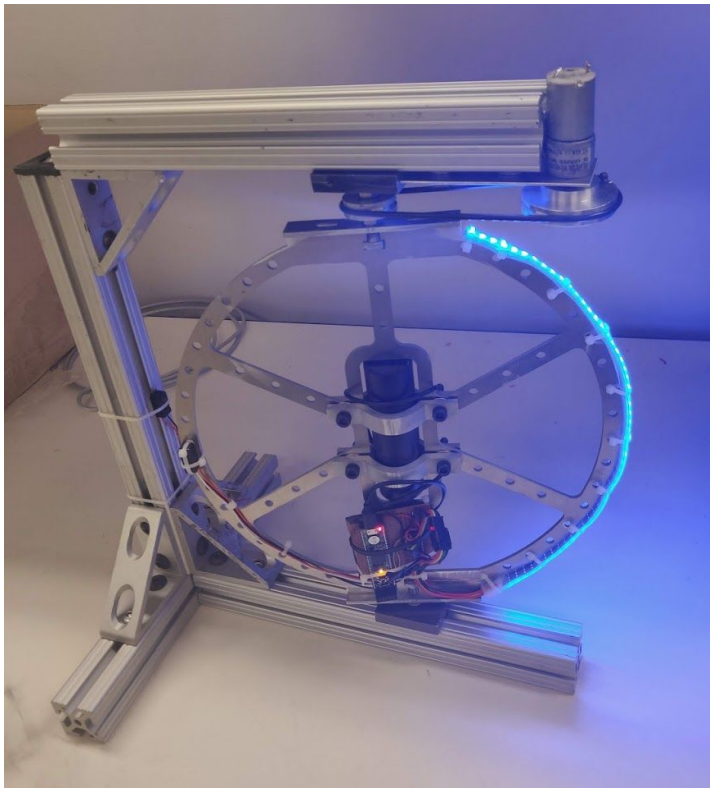
Luis

The hardware design consists of an inner ring that is held in place by an external frame. The inner ring holds all the electronics with the exception of the motor and the infrared emitter for detecting the rotational speed.



All of the hardware was sourced from scrap materials. The frame is constructed with aluminum extrusions and the inner ring and battery clamps were water jetted out of aluminum sheet. We machined bearing plates and pulleys out of aluminum stock as well.

The main challenge with designing the hardware for this project was ensuring that the rotating ring was balanced. To do this, we designed the ring to be symmetrical, and mounted our cylindrical battery pack inline with the axis of rotation of the ring. All other components were mounted as close to the center of the ring as possible, such that the weight was evenly distributed.



The only other challenge we faced was making sure the assembly was rotating fast enough to produce the effect. Our prior work research found that we need a refresh rate somewhere between 13Hz-16Hz. The disk refreshes every time it completes a full revolution. This translates to a rotational speed of 13-16 rotations per second or 780-960 revolutions per minute. We also found that there is no benefit from refresh rates above 60Hz, so there is no point in going faster than 3600 rpm.

Electronics

Our project posed a unique challenge in that every electrical component had to be mounted to the spinning ring. We considered using a slip ring while initially planning the project, which is a component that would allow us to pass electric signals through a rotating joint. However, most commercially available slip rings can support only a limited rate of speed, and even if they can be rotated quickly they often make for fairly noisey connections that can't support high-speed signals.

Since it seemed infeasible to build a mechanism that could spin our Labkit, the course staff loaned us a Cmod A7, a lightweight and minimalist dev board for the Artix 7 FPGA. Although it's much smaller than the Labkit FPGAs, we figured that the Cmod A7 had more than enough space to hold our design and so it was the perfect device for our needs. In addition, it features onboard flash storage to enable configuration on power up, necessary since we would have to turn off the FPGA between programming and starting up the device.

Mounting everything on our spinning disk meant we could have absolutely no wires coming from our FPGA, and therefore we also had to power it via a battery. Since the Cmod board is powered off a standard micro USB connector, we decided to get an off-the-shelf battery pack from Amazon meant for charging cell phones. This ended up being the ideal choice since it was an all-in-one unit that provided a nice regulated 5V over USB and charging capability built in. We searched for a battery that was optimal in terms of low weight, high capacity, and high peak current. We ended up choosing a battery with 5000 mAh capacity, 2A peak current, and a 4.8oz weight. Given this capacity, at continual max current draw (which is worst case) our battery lasts 2.5 hours, which we figured would be more than enough for our purposes.

According to the datasheet, each LED draws 60mA at max current, for 52 * 60 mA = 3.12 A total. In order to fit within the battery's current limit, we used a scheme to light only half of the LEDs at once (described in more detail in the "Image mapper" module section). With max LED current cut in half, we were left with plenty of overhead for the battery to also power the FPGA and sensor.

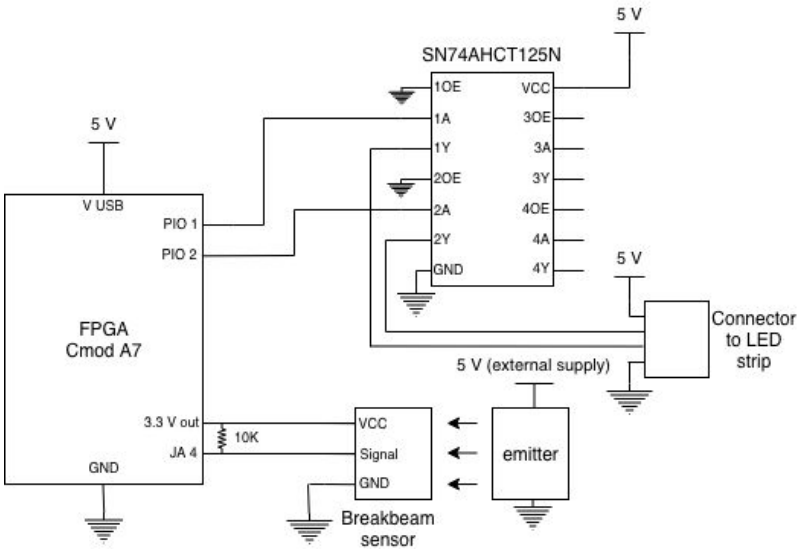
In order to determine our rotation speed, we needed some sort of sensor feedback. We considered a few different options, including hall effect sensors, some sort of glancing touch feedback, or a gyroscope. However, we ultimately settled on using an IR break beam sensor from Adafruit. This two piece sensor includes both an emitter and a detector. The emitter is mounted to the stationary frame of the project (since it just requires power and doesn't need to be connected to the FPGA), and the detector was mounted to the spinning ring opposite the LEDs. Every time this side of the ring passes by the emitter, the detector detects the IR beam coming from the emitter and sends a logic 1 to the FPGA. This proved to be a simple and effective setup.

One challenge we encountered was level shifting the 3.3V digital outputs from the Cmod A7 to the 5V required for the LED strip inputs (we did try without level shifting just in case the LEDs were more tolerant than documented, but this didn't work). We initially tried using a simple level-shifter board from Sparkfun, but unfortunately it was much too slow for our use case. We then tried a SN74LS04 from lab 1, since TTL logic has a very forgiving input high range that will read a 3.3V signal as high but then will output fairly close to 5V. This did work for testing purposes, but we were only able to clock our LED strip somewhere on the order of KHz.

Ultimately, we ended up using a 74AHCT125 IC, which is a buffer chip that Adafruit markets as a level shifter. Using this chip, we were able to clock our LEDs at 12 MHz.

To light all the LEDs on the strip requires a 32 bit start sequence, a 32 bit end sequence, and 32 bits per each of the 52 LEDs for a total of 54 * 32 = 1,728 bits of data. At 12 MHz, you can then update the LED strip approximately $12*10^6/1{,}728$ = 6,944 times per second. Given that we want to rotate at least 800 RPM (or ~13 rotations per second), then we are able to update 6,944/13 = 534 times per rotation. This means we are easily able to meet our desired angular resolution of 256.

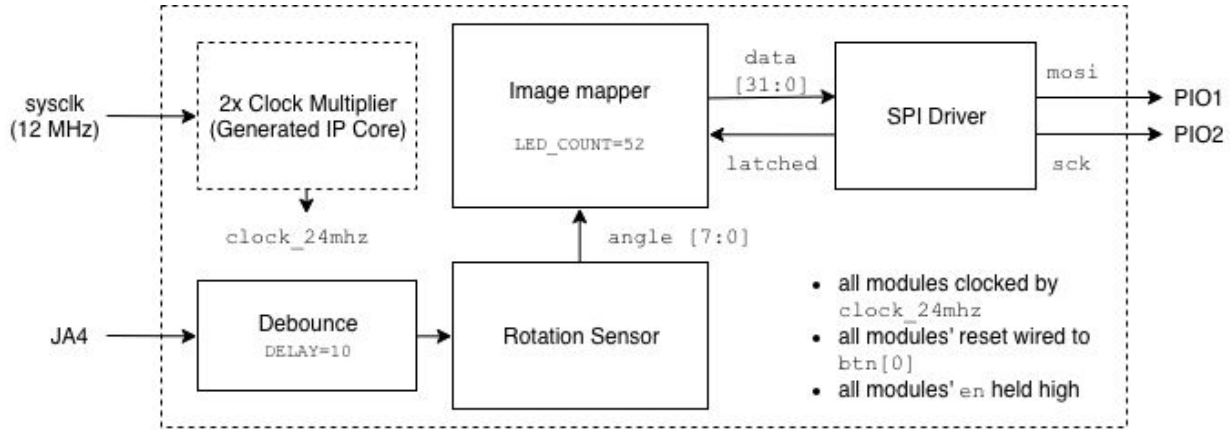
We debugged our electronics on a breadboard, and then soldered them to perfboard for the final prototype.



Schematic of final circuit

Both the IR sensor emitter and motor (which are mounted statically to the frame of the device) were powered by a lab power supply. We powered the motor via the supply's variable output and the sensor via its fixed 5 V output.

Verilog Modules



Overall block diagram.

SPI Driver
Noah

Inputs: `clock, reset, en, data[31:0]`
Outputs: `mosi, sck, latched`

The SPI Driver module is responsible for outputting data over SPI to the LED strips. When `en` is high and `reset` is low, the module will continuously latch the 32 bits presented at its `data` input and output them from MSB to LSB. Each time it finishes outputting its 32 bits of data, the module will latch the next 32 bits presented on `data` and then assert `latched` high for one clock cycle. This tells whichever module is presenting the data that it is okay to present the next 32 bits to be output.

We decided to clock `sck` at one half our system clock speed so that we have one in-between cycle to update the data presented on `mosi` before `sck` goes high. This clocking scheme proved to work out perfectly.

Image mapper
Noah

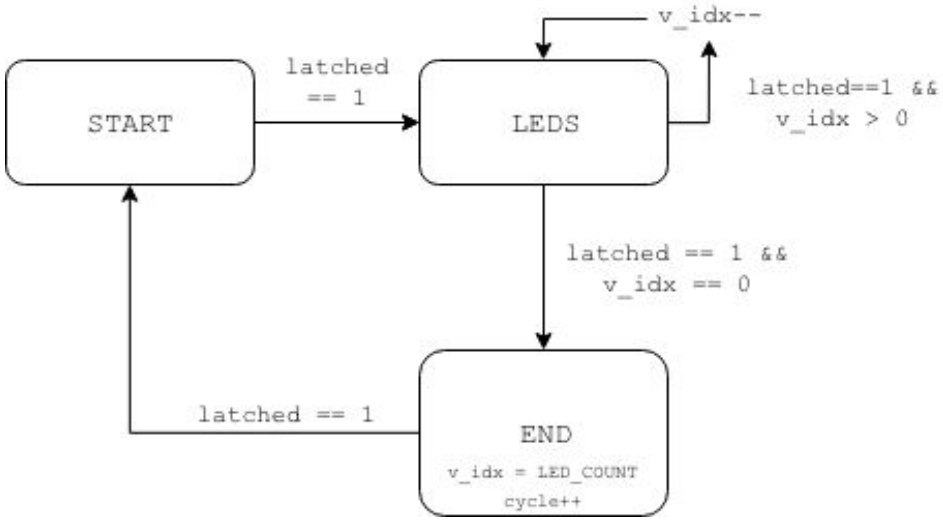
Inputs: `clock, reset, en, angle[7:0], latched`
Outputs: `data_out[31:0]`
Parameters: `LED_COUNT`

The image mapper is responsible for sending the correct data to the SPI driver given an input describing the angle of the spinning ring. We wanted our project to be able to display any

arbitrary bitmap file we provide it, so this module contains the ROM necessary to store the image. We ended up using four generated Block ROM IP cores: `image_rom` to store the main image COE file, and then three more to store the red, green, and blue color tables. The `image_rom` size is 256x52x8 bits (~13,312 KB), and each of the color tables is 256x8 bits (256 bytes).

The x coordinates of the image are mapped linearly with the angle of the ring and each y coordinate is mapped 1:1 to each LED on the arc. This removes the need for any complicated math.

The image mapper is structured as a simple finite state machine with three states. Here's the state transition diagram:

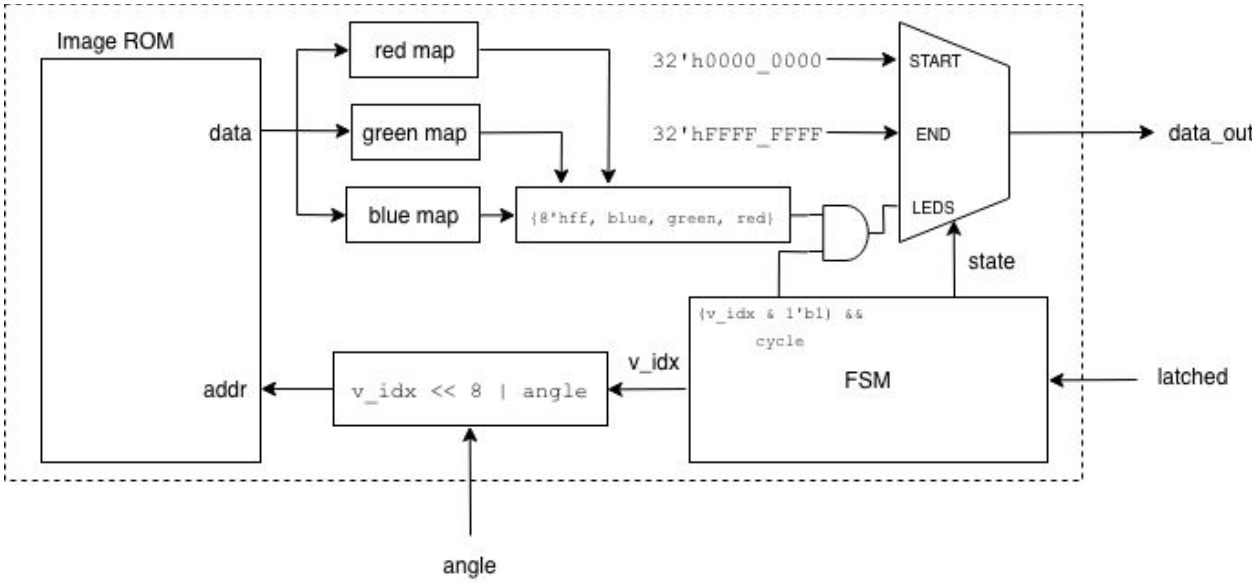


The state transitions are designed to follow the specifications of the LED strip. Before sending out data, the master device must send out a reset packet of 32'b0. Following this, one 32 bit packet is sent out for each LED, which we index using the variable `v_idx`. We originally had `v_idx` counting up from zero, but reversed the direction when we realized our images were appearing upside down on the display. Finally, in the END state we output a single end packet of 32{1'b1}. This actually corresponds to lighting an LED white, but doesn't actually display since we'll have sent colors to every LED in the chain already. Rather, it just gives the strip one

extra LED's worth of clock cycles to finish propagating, which is necessary due to some delays in the spec.

Each transition occurs on an assertion of `latched`, which is wired to the `latched` output of the SPI driver. This tells the image mapper to go ahead and output the next packet of data each time the SPI driver latches the current one. The data to output is determined combinatorially based on the current state, `v_idx`, and `angle`.

In addition, the image mapper actually outputs data to every other LED, cycling between every even LED and every odd LED. This effectively acts as a PWM technique to reduce current draw at any given time (so that we don't max out the current output of our battery). We determined we had plenty of leeway to do this both in terms of timing and brightness. This is controlled by comparing the LSB of `v_idx` with a one bit register called `cycle` that flips with every packet outputted.



Block diagram for image mapper module internals

Rotation sensor
Luis

Inputs: `clock, reset, enable, sensor`
Outputs: `current_angle[7:0]`

The rotation module is responsible for interpreting the synchronized and debounced sensor input and generating the current angular position of the rotating assembly. The strategy for generating the current angle, is to count how many clock cycles it takes for the ring to complete a full revolution and divide that quantity by the total number of degrees. This calculation gives us the number of clock cycles we need to wait between each degree. To simplify the division, we

decided to divide the circle into 256 degrees instead of 360. By doing this, the calculation becomes a simple bit shift instead of a full division.

A design choice that we needed to make was the width of counter register. The smaller it is, the faster the assembly needs to be rotating in order for this module to work. We ended up with a width of 22 bits. This restricts the maximum number of clock cycles per full rotation to 4194304 cycles. With a 24MHz clock, this means that we need to complete a full revolution at least every 0.1747 seconds (~344rpms).

In order to decrease noise and increase the robustness of the module, we keep a running average of the clock cycles per rotation and use that in our calculation instead of the last measurement. We calculate our average using the following equation:

```
counter_average <= 7*(clock_counter >> 3) + (counter_average >> 3);
```

By averaging using this equation, we effectively average over every single counter_average value since the system was turned on. It assigns geometrically decreasing weights to sequentially older values, thereby wighting new values higher than old values. The beauty of this method is that we can consider all of these values without needing to store them.

This module allows the device to rotate at any speed (above 344rpms) while still maintaining a static image. Even when the sensor is completely disconnected, as long as the motor speed is constant, the image remains still. This is because we use the counter_average in calculating the clocks per degree value and don't update theregister until we receive a new pulse from the sensor.

Final Notes

Here are a few lessons we learned that may be helpful for future teams:

- Pair programming and commenting are both helpful for working through tricky modules and making sure that your logic is sound from the very start.
- However, it's also useful to design your system so that you and your partner can work on different components in parallel.
- Simulations are super helpful for testing. Writing test benches was really important for our project so that we could (A) work on the hardware in parallel with the Verilog and (B) debug modules without having to come up with a debugging scheme that works when your system is rotating at 1000 RPM.

Appendices

Verilog

main.v

```
module main(
    input sysclk,
    input [0:0] btn,
    input [7:0] ja,
    output led0_b,
    output led0_g,
    output led0_r,
    output pio1,
    output pio2
    );

    wire reset;
    wire rotation_sensor;
    wire mosi;
    wire sck;
    wire clock_24mhz;
    wire latched;
    wire [7:0] angle;
    wire [31:0] data;


    debounce rotation_sensor_debounce(.reset(reset), .clock(clock_24mhz),
.noisy(ja[3]), .clean(rotation_sensor));
    assign pio1 = mosi;
    assign pio2 = sck;
    assign reset = btn[0];

    clk_wiz_0 clock_multiplier(.reset(reset), .clk_in1(sysclk),
.clk_out1(clock_24mhz));

    rotationSensor sensor(.reset(reset), .clk(clock_24mhz), .enable(1'b1),
.sensor(rotation_sensor), .current_angle(angle));
    image_mapper #(.LED_COUNT(72)) mapper(.reset(reset), .clock(clock_24mhz),
.en(1'b1), .angle(angle), .latched(latched), .data_out(data));
    spi_driver spi(.reset(reset), .clock(clock_24mhz), .en(1'b1), .data(data),
.latched(latched), .mosi(mosi), .sck(sck));

endmodule
```

image_mapper.v

```
module image_mapper(
    input clock,
    input reset,
    input en,
    input [7:0] angle,
    input [7:0] image_data,
    input latched,
    output [15:0] image_addr,
    output reg [31:0] data_out
    );

    parameter LED_COUNT = 72;

    parameter S_START = 2'b00;
    parameter S_LEDS = 2'b01;
    parameter S_END = 2'b11;
    reg [1:0] state = S_START;

    reg [6:0] v_idx = 0;

    // currently harcoded to display LEDS in banks of LED_COUNT/4 to reduce current
draw
    // TODO: parameterize this
    reg cycle = 0;

    // color maps
    wire [7:0] red_mapped, green_mapped, blue_mapped;
    red_coe rcm (.clka(clock), .ena(1'b1), .addra(image_data), .douta(red_mapped));
    green_coe gcm (.clka(clock), .ena(1'b1), .addra(image_data),
.douta(green_mapped));
    blue_coe bcm (.clka(clock), .ena(1'b1), .addra(image_data), .douta(blue_mapped));

    // select data based on vertical index and current angle
    assign image_addr = v_idx << 8 | angle;
    image_rom my_image(.clka(clock), .ena(1'b1), .addra(image_addr),
.douta(image_data));

    // multiplex data output depending on state
    always @(*)
    begin
        case(state)
            S_START: data_out = 32'h00000000;
```

```
            // add in image map for color data
            S_LEDS: data_out = (v_idx & 5'b1) == cycle ? {8'hFF, blue_mapped,
green_mapped, red_mapped} : {3'b111, 29'b0};
            S_END: data_out = 32'hFFFFFFFF;
            default: data_out = 32'h0000;
        endcase
    end

    // state machine to output SPI frame
    always @(posedge clock)
    begin
        if(reset)
        begin
            state <= S_START;
            v_idx <= 0;
            cycle <= 0;
        end
        else if(en)
        begin
            case(state)
                S_START:
                begin
                    if(latched)
                    begin
                        state <= S_LEDS;
                    end
                end
                S_LEDS:
                begin
                    if(latched && v_idx < (LED_COUNT-1))
                    begin
                        v_idx <= v_idx + 1;
                    end
                    else if(latched && v_idx == (LED_COUNT-1))
                    begin
                        state <= S_END;
                        v_idx <= 0;
                        cycle <= cycle + 1;
                    end
                end
                S_END:
                begin
                    if(latched)
                    begin
                        state <= S_START;
```

```
                    end
                end
            endcase
        end
    end


endmodule
```

spi_driver.v

```
module spi_driver(
    input clock,
    input reset,
    input en,
    input [31:0] data,
    output reg mosi,
    output reg sck,
    output latched
    );

    reg [31:0] data_reg = 32'b0;
    reg [4:0] data_idx = 5'd31;

    always @(posedge clock)
    begin
        if(reset)
        begin
            data_idx <= 5'd31;
            data_reg <= data;
            mosi <= 0;
            sck <= 0;
        end
        else if(en)
        begin
            // if sck is currently high, it's about to go low so we should
            // set up the next output bit
            if(sck)
            begin
                mosi <= data_reg[data_idx];
                data_idx <= data_idx - 1;
                if(data_idx == 0)
                begin
                    data_reg <= data;
```

```
                end
            end
            sck <= sck + 1;
        end
    end

    // set latched high every time we load new data (data_idx == 0)
    // since we only want to assert it for one clock cycle, only keep it high during
positive edge of sck
    assign latched = sck && (data_idx == 5'd31);

endmodule
```

rotationModule.v

```
module rotationSensor(
    input clk,
    input reset,
    input enable,
    input sensor,
    output reg [7:0] current_angle = 0
    );

    // Current count of clock cycles from the last
    // pulse from the sensor
    reg [21:0] clock_counter = 0;

    // The running average of clock cycles it takes
    // for a full revolution to complete
    reg [21:0] counter_average = 0;


    // Counter for the number of clock cycles that have
    // passed since the last degree change
    reg [13:0] clock_degree_counter = 0;
    reg [13:0] clocks_per_degree = 0;

    reg previous_sensor = 0;


    always @(posedge clk) begin

        // Keep track of the previous value of sensor in order
        // to detect the rising edge
```

```
previous_sensor <= sensor;

// If reset is high, set all values to 0.
if (reset == 1)begin
    clock_counter <= 0;
    counter_average <= 0;
    clock_degree_counter <= 0;
    clocks_per_degree <= 0;
    current_angle <= 0;
end

else if (enable == 1) begin

    // If the sensor is high, a full revolution has
    // been completed. Current angle is reset to 0.
    // Counter_average is updated. Clock_counter is
    // reset to 0. Clock_per_degree is updated.
    // Clock_degree_counter is reset to 0.
    if (sensor == 1 && previous_sensor == 0) begin
        current_angle <= 0;
        counter_average <= 7*(clock_counter >> 3) + (counter_average >> 3);

        // For testing purposes:
        // Comment the previous line and uncomment the folowing line
        // in order to use the current clock_counter value instead of
        // a weighted average of the past values.
        //counter_average <= clock_counter;

        clock_counter <= 0;
        clock_degree_counter <= 0;
        clocks_per_degree <= counter_average >> 8;
    end

    // Sensor is low; full rotation has not been completed.
    // Increase clock_counter by 1. Calculate current angle
    // based on the clock_counter and counter_average;
    else begin

        // If we have completed the number of clocks per degree,
        // increase the output degree value by 1. Reset clock_degree_counter
        if (clock_degree_counter == clocks_per_degree) begin
            current_angle <= current_angle + 1;
            clock_degree_counter <= 0;
        end
```

```
                // If we haven't completed an extra degree, increase
clock_degree_counter
                else begin
                    clock_degree_counter <= clock_degree_counter + 1;
                end

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

Cmod-A7-Master.xdc

```
## This file is a general .xdc for the CmodA7 rev. B
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level
signal names in the project

## 12 MHz Clock Signal
set_property -dict {PACKAGE_PIN L17 IOSTANDARD LVCMOS33} [get_ports sysclk]
create_clock -period 83.330 -name sys_clk_pin -waveform {0.000 41.660} -add
[get_ports sysclk]

## LEDs
#set_property -dict { PACKAGE_PIN A17   IOSTANDARD LVCMOS33 } [get_ports { led[0] }];
#IO_L12N_T1_MRCC_16 Sch=led[1]
#set_property -dict { PACKAGE_PIN C16   IOSTANDARD LVCMOS33 } [get_ports { led[1] }];
#IO_L13P_T2_MRCC_16 Sch=led[2]

## RGB LED
set_property -dict {PACKAGE_PIN B17 IOSTANDARD LVCMOS33} [get_ports led0_b]
set_property -dict {PACKAGE_PIN B16 IOSTANDARD LVCMOS33} [get_ports led0_g]
set_property -dict {PACKAGE_PIN C17 IOSTANDARD LVCMOS33} [get_ports led0_r]

## Buttons
set_property -dict {PACKAGE_PIN A18 IOSTANDARD LVCMOS33} [get_ports {btn[0]}]
#set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVCMOS33 } [get_ports { btn[1] }];
#IO_L19P_T3_16 Sch=btn[1]

## Pmod Header JA
set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports { ja[0] }];
#IO_L5N_T0_D07_14 Sch=ja[1]
set_property -dict { PACKAGE_PIN G19   IOSTANDARD LVCMOS33 } [get_ports { ja[1] }];
```

```
#IO_L4N_T0_D05_14 Sch=ja[2]
set_property -dict { PACKAGE_PIN N18   IOSTANDARD LVCMOS33 } [get_ports { ja[2] }];
#IO_L9P_T1_DQS_14 Sch=ja[3]
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { ja[3] }];
#IO_L8P_T1_D11_14 Sch=ja[4]
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { ja[4] }];
#IO_L5P_T0_D06_14 Sch=ja[7]
set_property -dict { PACKAGE_PIN H19   IOSTANDARD LVCMOS33 } [get_ports { ja[5] }];
#IO_L4P_T0_D04_14 Sch=ja[8]
set_property -dict { PACKAGE_PIN J19   IOSTANDARD LVCMOS33 } [get_ports { ja[6] }];
#IO_L6N_T0_D08_VREF_14 Sch=ja[9]
set_property -dict { PACKAGE_PIN K18   IOSTANDARD LVCMOS33 } [get_ports { ja[7] }];
#IO_L8N_T1_D12_14 Sch=ja[10]

## Analog XADC Pins
## Only declare these if you want to use pins 15 and 16 as single ended analog
inputs. pin 15 -> vaux4, pin16 -> vaux12
#set_property -dict { PACKAGE_PIN G2    IOSTANDARD LVCMOS33 } [get_ports { xa_n[0]
}]; #IO_L1N_T0_AD4N_35 Sch=ain_n[15]
#set_property -dict { PACKAGE_PIN G3    IOSTANDARD LVCMOS33 } [get_ports { xa_p[0]
}]; #IO_L1P_T0_AD4P_35 Sch=ain_p[15]
#set_property -dict { PACKAGE_PIN J2    IOSTANDARD LVCMOS33 } [get_ports { xa_n[1]
}]; #IO_L2N_T0_AD12N_35 Sch=ain_n[16]
#set_property -dict { PACKAGE_PIN H2    IOSTANDARD LVCMOS33 } [get_ports { xa_p[1]
}]; #IO_L2P_T0_AD12P_35 Sch=ain_p[16]

## GPIO Pins
## Pins 15 and 16 should remain commented if using them as analog inputs
set_property -dict {PACKAGE_PIN M3 IOSTANDARD LVCMOS33} [get_ports pio1]
set_property -dict {PACKAGE_PIN L3 IOSTANDARD LVCMOS33} [get_ports pio2]
#set_property -dict {PACKAGE_PIN A16 IOSTANDARD LVCMOS33} [get_ports pio3]
#set_property -dict { PACKAGE_PIN K3    IOSTANDARD LVCMOS33 } [get_ports { pio4  }];
#IO_L7N_T1_AD6N_35 Sch=pio[04]
#set_property -dict { PACKAGE_PIN C15   IOSTANDARD LVCMOS33 } [get_ports { pio5  }];
#IO_L11P_T1_SRCC_16 Sch=pio[05]
#set_property -dict { PACKAGE_PIN H1    IOSTANDARD LVCMOS33 } [get_ports { pio6  }];
#IO_L3P_T0_DQS_AD5P_35 Sch=pio[06]
#set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVCMOS33 } [get_ports { pio7  }];
#IO_L6N_T0_VREF_16 Sch=pio[07]
#set_property -dict { PACKAGE_PIN B15   IOSTANDARD LVCMOS33 } [get_ports { pio8  }];
#IO_L11N_T1_SRCC_16 Sch=pio[08]
#set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports { pio9  }];
#IO_L6P_T0_16 Sch=pio[09]
#set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 } [get_ports { pio10 }];
#IO_L7P_T1_AD6P_35 Sch=pio[10]
#set_property -dict { PACKAGE_PIN J1    IOSTANDARD LVCMOS33 } [get_ports { pio11 }];
#IO_L3N_T0_DQS_AD5N_35 Sch=pio[11]
#set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { pio12 }];
```

```
#IO_L5P_T0_AD13P_35 Sch=pio[12]
#set_property -dict { PACKAGE_PIN L1    IOSTANDARD LVCMOS33 } [get_ports { pio13 }];
#IO_L6N_T0_VREF_35 Sch=pio[13]
#set_property -dict { PACKAGE_PIN L2    IOSTANDARD LVCMOS33 } [get_ports { pio14 }];
#IO_L5N_T0_AD13N_35 Sch=pio[14]
#set_property -dict { PACKAGE_PIN M1    IOSTANDARD LVCMOS33 } [get_ports { pio17 }];
#IO_L9N_T1_DQS_AD7N_35 Sch=pio[17]
#set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports { pio18 }];
#IO_L12P_T1_MRCC_35 Sch=pio[18]
#set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports { pio19 }];
#IO_L12N_T1_MRCC_35 Sch=pio[19]
#set_property -dict { PACKAGE_PIN M2    IOSTANDARD LVCMOS33 } [get_ports { pio20 }];
#IO_L9P_T1_DQS_AD7P_35 Sch=pio[20]
#set_property -dict { PACKAGE_PIN N1    IOSTANDARD LVCMOS33 } [get_ports { pio21 }];
#IO_L10N_T1_AD15N_35 Sch=pio[21]
#set_property -dict { PACKAGE_PIN N2    IOSTANDARD LVCMOS33 } [get_ports { pio22 }];
#IO_L10P_T1_AD15P_35 Sch=pio[22]
#set_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports { pio23 }];
#IO_L19N_T3_VREF_35 Sch=pio[23]
#set_property -dict { PACKAGE_PIN R3    IOSTANDARD LVCMOS33 } [get_ports { pio26 }];
#IO_L2P_T0_34 Sch=pio[26]
#set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports { pio27 }];
#IO_L2N_T0_34 Sch=pio[27]
#set_property -dict { PACKAGE_PIN R2    IOSTANDARD LVCMOS33 } [get_ports { pio28 }];
#IO_L1P_T0_34 Sch=pio[28]
#set_property -dict { PACKAGE_PIN T1    IOSTANDARD LVCMOS33 } [get_ports { pio29 }];
#IO_L3P_T0_DQS_34 Sch=pio[29]
#set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports { pio30 }];
#IO_L1N_T0_34 Sch=pio[30]
#set_property -dict { PACKAGE_PIN U1    IOSTANDARD LVCMOS33 } [get_ports { pio31 }];
#IO_L3N_T0_DQS_34 Sch=pio[31]
#set_property -dict { PACKAGE_PIN W2    IOSTANDARD LVCMOS33 } [get_ports { pio32 }];
#IO_L5N_T0_34 Sch=pio[32]
#set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports { pio33 }];
#IO_L5P_T0_34 Sch=pio[33]
#set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports { pio34 }];
#IO_L6N_T0_VREF_34 Sch=pio[34]
#set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports { pio35 }];
#IO_L6P_T0_34 Sch=pio[35]
#set_property -dict { PACKAGE_PIN W5    IOSTANDARD LVCMOS33 } [get_ports { pio36 }];
#IO_L12P_T1_MRCC_34 Sch=pio[36]
#set_property -dict { PACKAGE_PIN V4    IOSTANDARD LVCMOS33 } [get_ports { pio37 }];
#IO_L11N_T1_SRCC_34 Sch=pio[37]
#set_property -dict { PACKAGE_PIN U4    IOSTANDARD LVCMOS33 } [get_ports { pio38 }];
#IO_L11P_T1_SRCC_34 Sch=pio[38]
#set_property -dict { PACKAGE_PIN V5    IOSTANDARD LVCMOS33 } [get_ports { pio39 }];
#IO_L16N_T2_34 Sch=pio[39]
#set_property -dict { PACKAGE_PIN W4    IOSTANDARD LVCMOS33 } [get_ports { pio40 }];
```

```
#IO_L12N_T1_MRCC_34 Sch=pio[40]
#set_property -dict { PACKAGE_PIN U5    IOSTANDARD LVCMOS33 } [get_ports { pio41 }];
#IO_L16P_T2_34 Sch=pio[41]
#set_property -dict { PACKAGE_PIN U2    IOSTANDARD LVCMOS33 } [get_ports { pio42 }];
#IO_L9N_T1_DQS_34 Sch=pio[42]
#set_property -dict { PACKAGE_PIN W6    IOSTANDARD LVCMOS33 } [get_ports { pio43 }];
#IO_L13N_T2_MRCC_34 Sch=pio[43]
#set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports { pio44 }];
#IO_L9P_T1_DQS_34 Sch=pio[44]
#set_property -dict { PACKAGE_PIN U7    IOSTANDARD LVCMOS33 } [get_ports { pio45 }];
#IO_L19P_T3_34 Sch=pio[45]
#set_property -dict { PACKAGE_PIN W7    IOSTANDARD LVCMOS33 } [get_ports { pio46 }];
#IO_L13P_T2_MRCC_34 Sch=pio[46]
#set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS33 } [get_ports { pio47 }];
#IO_L14P_T2_SRCC_34 Sch=pio[47]
#set_property -dict { PACKAGE_PIN V8    IOSTANDARD LVCMOS33 } [get_ports { pio48 }];
#IO_L14N_T2_SRCC_34 Sch=pio[48]

## UART
#set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports {
uart_rxd_out }]; #IO_L7N_T1_D10_14 Sch=uart_rxd_out
#set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports {
uart_txd_in  }]; #IO_L7P_T1_D09_14 Sch=uart_txd_in

## Crypto 1 Wire Interface
#set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports { crypto_sda
}]; #IO_0_14 Sch=crypto_sda

## QSPI
#set_property -dict { PACKAGE_PIN K19   IOSTANDARD LVCMOS33 } [get_ports { qspi_cs
}]; #IO_L6P_T0_FCS_B_14 Sch=qspi_cs
#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { qspi_dq[0]
}]; #IO_L1P_T0_D00_MOSI_14 Sch=qspi_dq[0]
#set_property -dict { PACKAGE_PIN D19   IOSTANDARD LVCMOS33 } [get_ports { qspi_dq[1]
}]; #IO_L1N_T0_D01_DIN_14 Sch=qspi_dq[1]
#set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports { qspi_dq[2]
}]; #IO_L2P_T0_D02_14 Sch=qspi_dq[2]
#set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports { qspi_dq[3]
}]; #IO_L2N_T0_D03_14 Sch=qspi_dq[3]

## Cellular RAM
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[0]
}]; #IO_L11P_T1_SRCC_14 Sch=sram- a[0]
#set_property -dict { PACKAGE_PIN M19   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[1]
}]; #IO_L11N_T1_SRCC_14 Sch=sram- a[1]
#set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[2]
}]; #IO_L12N_T1_MRCC_14 Sch=sram- a[2]
#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[3]
```

```
}]; #IO_L13P_T2_MRCC_14 Sch=sram- a[3]
#set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[4]
}]; #IO_L13N_T2_MRCC_14 Sch=sram- a[4]
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[5]
}]; #IO_L14P_T2_SRCC_14 Sch=sram- a[5]
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[6]
}]; #IO_L14N_T2_SRCC_14 Sch=sram- a[6]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[7]
}]; #IO_L16N_T2_A15_D31_14 Sch=sram- a[7]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[8]
}]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sram- a[8]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[9]
}]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=sram- a[9]
#set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[10]
}]; #IO_L16P_T2_CSI_B_14 Sch=sram- a[10]
#set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[11]
}]; #IO_L17P_T2_A14_D30_14 Sch=sram- a[11]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[12]
}]; #IO_L17N_T2_A13_D29_14 Sch=sram- a[12]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[13]
}]; #IO_L18P_T2_A12_D28_14 Sch=sram- a[13]
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[14]
}]; #IO_L18N_T2_A11_D27_14 Sch=sram- a[14]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[15]
}]; #IO_L19P_T3_A10_D26_14 Sch=sram- a[15]
#set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[16]
}]; #IO_L20P_T3_A08_D24_14 Sch=sram- a[16]
#set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[17]
}]; #IO_L20N_T3_A07_D23_14 Sch=sram- a[17]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { MemAdr[18]
}]; #IO_L21P_T3_DQS_14 Sch=sram- a[18]
#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports { MemDB[0]
}]; #IO_L21N_T3_DQS_A06_D22_14 Sch=sram-dq[0]
#set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports { MemDB[1]
}]; #IO_L22P_T3_A05_D21_14 Sch=sram-dq[1]
#set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports { MemDB[2]
}]; #IO_L22N_T3_A04_D20_14 Sch=sram-dq[2]
#set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports { MemDB[3]
}]; #IO_L23P_T3_A03_D19_14 Sch=sram-dq[3]
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { MemDB[4]
}]; #IO_L23N_T3_A02_D18_14 Sch=sram-dq[4]
#set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports { MemDB[5]
}]; #IO_L24P_T3_A01_D17_14 Sch=sram-dq[5]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { MemDB[6]
}]; #IO_L24N_T3_A00_D16_14 Sch=sram-dq[6]
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { MemDB[7]
}]; #IO_25_14 Sch=sram-dq[7]
#set_property -dict { PACKAGE_PIN P19   IOSTANDARD LVCMOS33 } [get_ports { RamOEn
```

```
}]; #IO_L10P_T1_D14_14 Sch=sram-oe
#set_property -dict { PACKAGE_PIN R19   IOSTANDARD LVCMOS33 } [get_ports { RamWEn
}]; #IO_L10N_T1_D15_14 Sch=sram-we
#set_property -dict { PACKAGE_PIN N19   IOSTANDARD LVCMOS33 } [get_ports { RamCEn
}]; #IO_L9N_T1_DQS_D13_14 Sch=sram-ce



set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]
```