LED Rubik’s Cube

Final Project Report
December 11, 2019
E155

Josephine King and Aaron Trujillo

Abstract:
The Rubik’s Cube is a fun and educational toy that can teach people of all ages valuable skills of spatial visualization, concentration, and problem solving. This project prototypes a digital Rubik’s Cube, which could be enhanced to add entertaining and educational features that traditional cubes do not have. The digital cube is composed of six LED matrices, and receives input from a user control panel of seven buttons and a knob. The user presses a button to choose a face to rotate, and then rotates in the desired direction by twisting the knob. The user input is read by a microcontroller, which processes the rotation to obtain the new color orientation. Then, the orientation is sent over SPI to an FPGA, which drives all six LED matrices through one output pin.

1 Introduction

The Rubik's Cube was invented in 1974 and today remains one of the world's most popular toys. Its value reaches beyond entertainment and into education; the Rubik's Cube helps users across all age groups with "VSI (Visual-Spatial Intelligence), [strategy development], memorization and logistics, concentration, and persistence in problem solving" [1]. However, as entertainment becomes increasingly digitized, how can the Rubik's Cube, with all of its educational value, continue to stay relevant?

For our final project, we made a digital Rubik's Cube out of LED arrays. While our cube's functionality is very similar to the traditional Rubik's Cube, further implementations could add fun features, such as Internet of Things, competitions, timers, hints, and other teaching features. We think that these features could improve the appeal of the Rubik's Cube for a new generation of users who are accustomed to digital forms of entertainment.

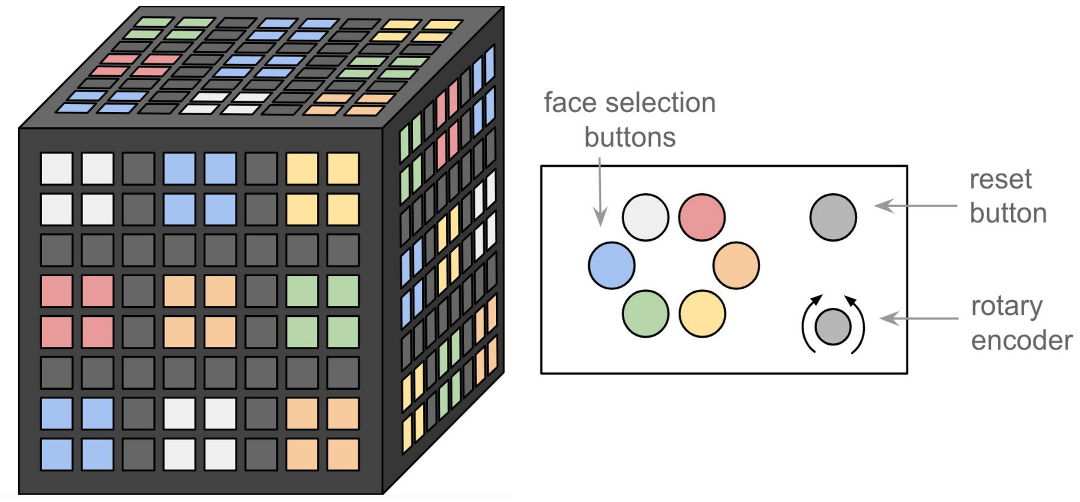


Figure 1: Diagram of LED Rubik's cube and user control panel.

Unlike the original Rubik's Cube, the LED Rubik's Cube does not use mechanical rotations. Instead, the user manipulates the LED cube with an array of seven buttons and a knob. The user begins by pressing the reset button, which scrambles the cube if it is solved, and solves it if it is scrambled. Then, the user selects a face using one of six colored buttons, where each color corresponds to the center square of a face. Lastly, they rotate the selected face either clockwise or counterclockwise by twisting the knob.

The cube is comprised of six 8-by-8 LED matrices and a user control panel, which has seven buttons and a rotary encoder for the knob (Figure 1). The user input is read by an ATSAM4S4B microcontroller. Based on this input, the microcontroller performs the rotation and determines the new color configuration of the cube. This color configuration is sent over SPI to a Cyclone

IV FPGA, which drives all six LED matrices with one pin. A block diagram of the system is shown in Figure 2.

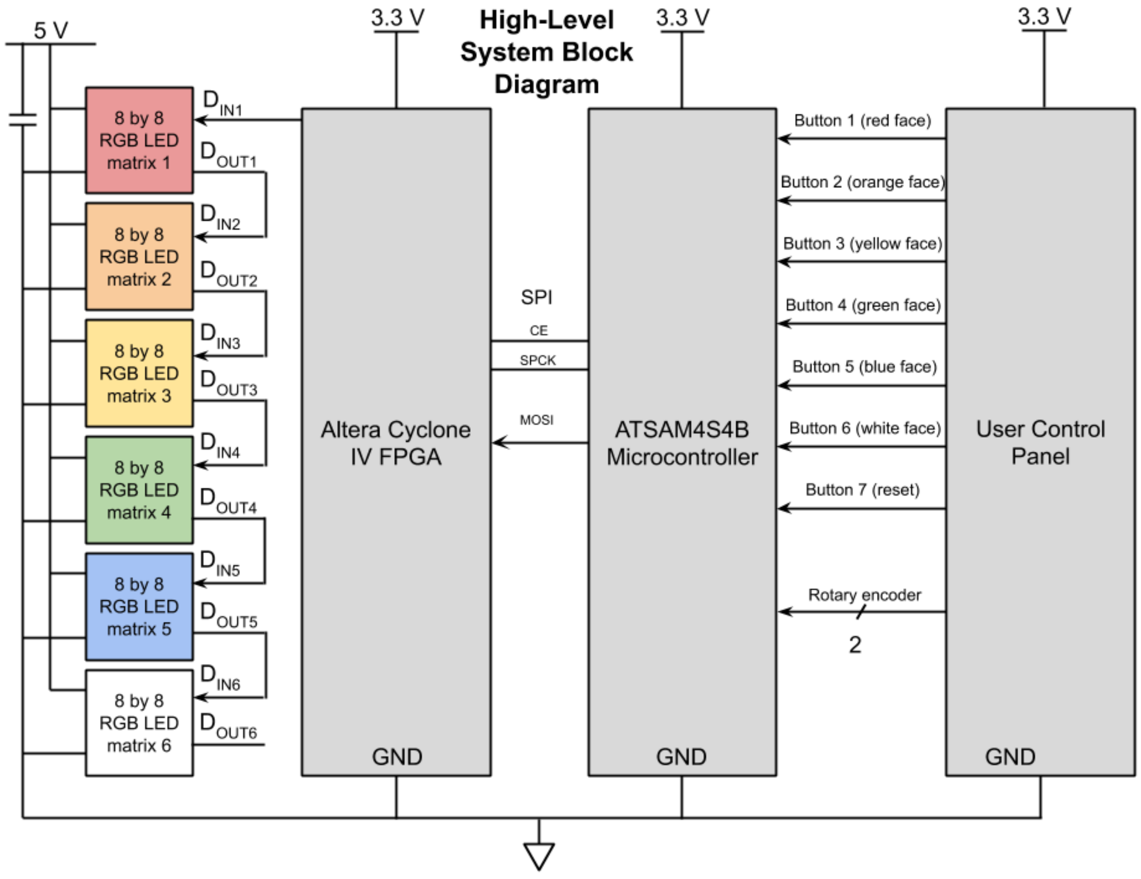


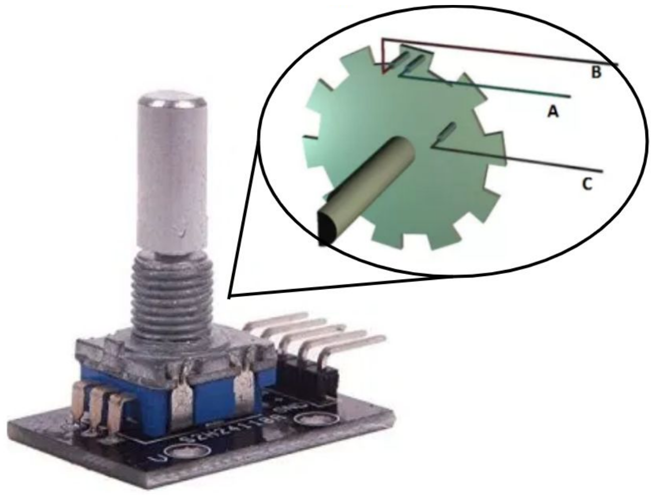
Figure 2: High-level block diagram of system.

2 New Hardware:

For this project, we used two pieces of hardware which we had not used before in E155: a rotary encoder and an 8-by-8 RGB LED matrix.

2.1 Rotary Encoder

We used the Keyes KY-040 rotary encoder. This piece of hardware is a rotary input device that can determine both degree and direction of rotation. The device has two powered switches that come into contact with a gear-shaped, rotating, grounded plate. The direction of rotation is determined by examining the phase changes of these two switches to see which one goes high or low first. Illustrations of the switch design and phase changes are shown in Figures 3 and 4.



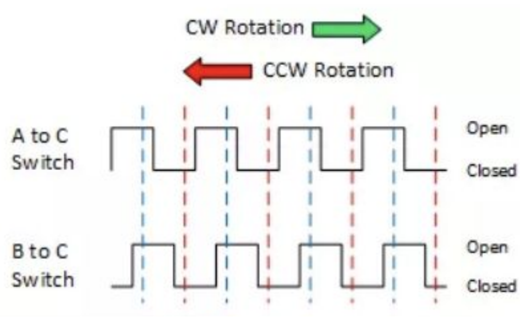


Figure 3: Illustration showing the internal and external design of the rotary encoder, with output pins A and B, and rotating node C. [2]

Figure 4: The direction of rotation can be determined from the phase shift between pins A and B. [2]

The rotary encoder that we chose (KY-040) is well-documented, with many tutorials on the internet for use with microcontrollers [2][3]. However, we found that the implementations demonstrated in these tutorials had to be modified for our purposes. The rotary encoder uses haptic feedback (clicks) so that the user can rotate the knob with more ease and precision. Specifically, the KY-040 rotary encoder clicks into place when both output pins are low. This means that each click causes the output pins to change value twice. However, most of the tutorial implementations that we found only look at one change in value of the output pins. Thus, we had to adjust these implementations so that our rotary encoder's haptic feedback would actually be helpful for the user. This implementation is explained in more detail in Section 4.1.

2.2 LED Matrix:

To make the cube, we used six 8-by-8 RGB WS2812B LED arrays from Kuman. These LED matrices came with power, ground, DIN, and DOUT pins on them, and with soldered male and female headers that could be used to connect multiple matrices in series (Figure 5). Adjacent LEDs are connected together with shift registers according to the zig-zag green path shown in Figure 5. Because values are shifted in and out of the LEDs, each LED is individually addressable, and all 64 LEDs in a matrix can be controlled with the DIN pin.

Each LED is controlled by 24 bits of color data, with 8 bits each for green, red, and blue light. The bits are encoded in the pattern shown in Figure 6. A 0 is encoded by a short high pulse and a long low pulse, while a 1 is encoded by a long high pulse and a short low pulse. All of these pulses are less than a microsecond, and their specific values are shown in the WS2812B datasheet [4]. The LEDs are reset to be written to a new configuration with a low pulse of at least 50 µs. In our SystemVerilog (Appendix A), the module `make_data_stream` translates 24 bits of color data into the correct datastream to interface with the LEDs.

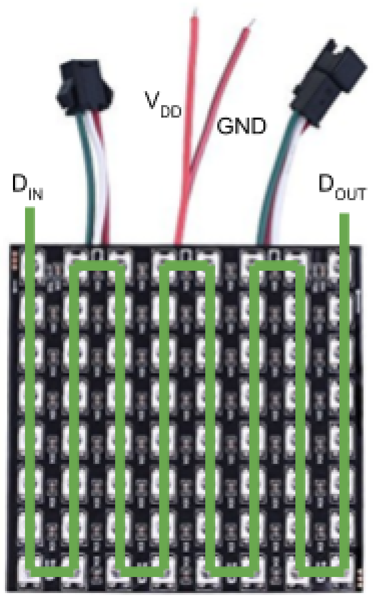


Figure 5: Kuman's 8-by-8 WS2812B LED matrix. The LEDs are connected with shift registers according to the zig-zag green path. [5]

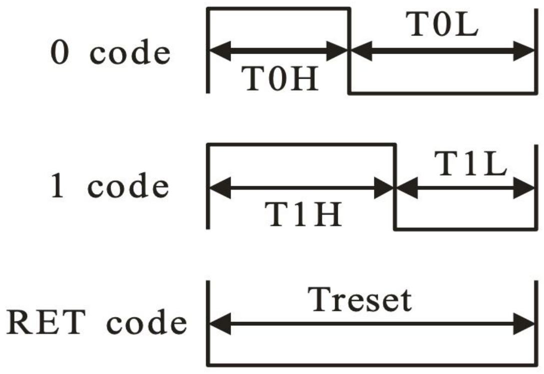


Figure 6: The encodings for 0, 1, and reset for WS2812B LEDs. [4]

3 Schematics

Figure 7 shows the schematic of our entire system, which is built on the E155 protoboard with a uMudd board. The three major subsystems—LED Rubik's Cube, user control panel, and SPI—are outlined by dotted boxes. The microcontroller IO pins need 3.3V, while the LED matrices need 5V, so the LED cube and user control panel are powered with different sources. The 1 mF bypass capacitor between 5V and ground ensures that the current through the LEDs does not increase too quickly and damage them when the power is turned on. All of the SPI wiring is internal to the uMudd board.

4 Microcontroller Design

Our microcontroller has three main functions: read user inputs, execute cube rotation logic, and send cube orientation to the FPGA using SPI.

4.1 Reading User Input

The user input is represented in our C code by an array of two bytes. The first byte indicates the user's button press, which can be a value between 0 and 7. Values of 0-5 indicate that the user has pressed one of the face-selection buttons. A value of 6 indicates that the user has pressed the reset button, and a value of 7 indicates that the user has not pressed a button since resetting the cube. The second byte indicates the user's direction of rotation, and can be a value of 0, 1, or 3. A value of 0 indicates a clockwise rotation and 1 indicates a counterclockwise rotation. A value of 3 indicates that the user has not yet rotated the knob since the last rotation.

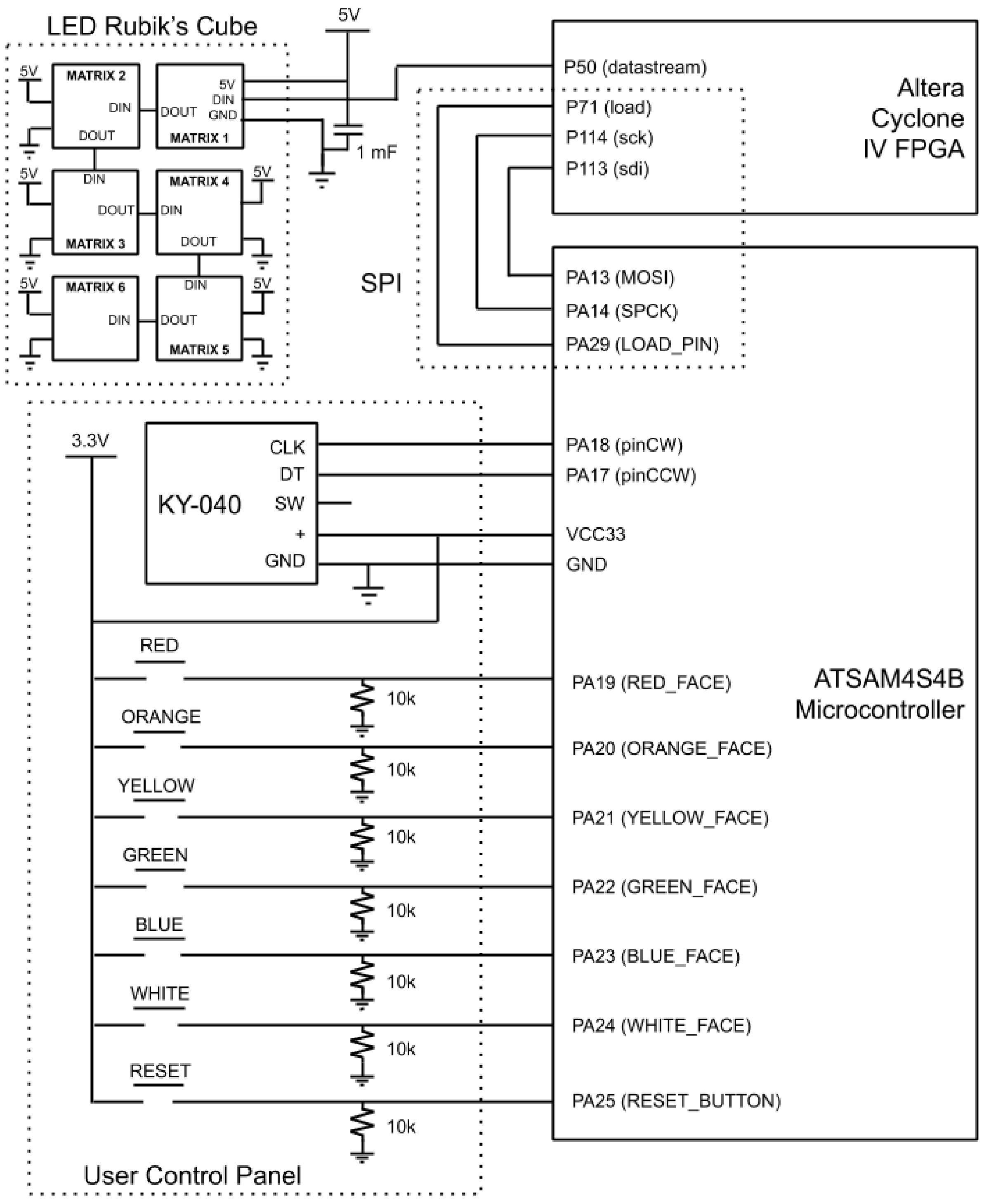


Figure 7: Schematic of entire system on breadboard.

The function `read_user_input()` reads the seven button presses from the user using the `pioDigitalRead()` function in the PIO peripheral. The user must select a face before the rotary encoder's input is read and used to perform rotations. In order to scramble the cube to a different random orientation each time, a counter is incremented while the reset button is pressed. The number of the counter is then used as a random seed for the random number generation in the `scramble_cube()` function.

After the user presses a button, the microcontroller waits for either another button press or a rotation of the rotary encoder. Another button press simply overwrites the previous button press. A rotation of the rotary encoder is read in two phases, as one "click" of a rotation actually corresponds to two changes in value of the rotary encoder output pins. If the last two changes in value of the rotary encoder pins correspond to a clockwise rotation, then the rotation value is set to 0. Similarly, if the last two changes correspond to a counterclockwise rotation, then the rotation value is set to 1. If the last two changes correspond to a clockwise and a counterclockwise rotation, then the user has changed rotation directions, and the most recent value is used to determine the rotation direction. After reading the rotary encoder and rotating the cube's orientation, the variables that store the most recent values read by the rotary encoder are reset to 3 to indicate that the knob has not yet been rotated.

4.2 Rotations

In order to perform rotations, it is necessary to encode the cube's orientation, which is the color and location of each square. There are six faces with nine squares each, so to represent the cube we use an array of 54 bytes. The values in the array represent the square colors, with 0 for red, 1 for orange, and so on. The indices in the array represent the locations of the square on the cube. Each square belongs to a face, where each face is labelled according to its central square color, with 0 for white, 1 for blue, and so on. The center square color remains constant throughout the game. The square's location is also defined by its position on the face. The square location numbers and their orientations on the cube are shown in Figure 8. The location of the square is translated to an index in the array by the equation:

```
index = 9 x (face value) + square value
```

Thus, the first 9 bytes of the array represent squares 0-8 on the white face, the next 9 bytes represent squares 0-8 on the blue face, and so on.

The `rotate_cube()` function takes in the two-byte user input array and the current 54-byte color orientation of the cube. To determine the new color orientation, we use a series of 12 if statements for the 12 possible moves that the user can perform. Inside these if statements, the cube is rotated through a series of translations from the old orientation to the new orientation. When the `scramble_cube()` function is called, the cube begins in a solved orientation and is scrambled through a series of random rotations. This method ensures that the resulting scrambled cube is solvable.

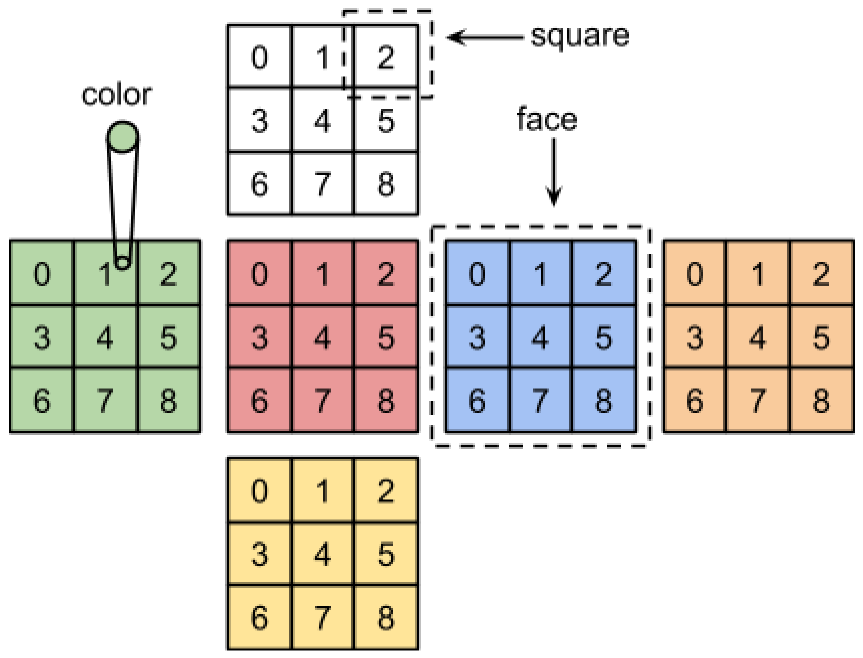


Figure 8: Representation of Rubik's cube used for rotations.

4.3 SPI

To send the color orientation from the microcontroller to the FPGA, we use SPI, where the microcontroller is the master and the FPGA is the slave. We encountered signal integrity issues with SPI when we sent the entire 54-byte array. Thus, instead of sending the 54-byte orientation, we condensed the orientation to 21 bytes by representing each color as 3 bits instead of an entire byte. The function `shift_orientation()` performs this compression. Then, the `send_orientation()` function uses the `spiSendReceive()` function in the SPI peripheral to send the 21-byte array to the FPGA one byte at a time.

5 FPGA Design

Our top-level SystemVerilog module has three inputs: `load`, `sdi`, `sck`, and `clk`. The former three are inputs for SPI communication, while the latter is the 40 MHz clock from the oscillator on the uMudd board. The only output of the top-level module is `datastream`, which is the input signal to the DIN pin of the Rubik's Cube. Our top-level module contains two high-level modules: `rubiks_spi` and `rubiks_core`. The module `rubiks_spi` reads in the SDI/MOSI value sent by the microcontroller and shifts it one bit at a time into a 162-bit bus, `orientation`. The cube's entire orientation can be represented with 162 bits, as there are 6 faces, 9 squares per face, and 3 bits per color, which results in 162 total bits.

Our other high level module, `rubiks_core`, takes in `orientation` as an input. This module breaks up orientation into six separate faces of 27 bits each, and sends one face at a time to the cube. `Rubiks_core` contains two submodules, `make_squares` and `make_data_stream`. The module `make_squares` takes in a 27-bit face orientation and has row and column state machines which are incremented to follow the zig-zag order of the LEDs shown in Figure 5. Then, according to the row and column of the state machines, the color of the next LED to write is determined with combinational logic and the helper modules `convert_orientation` and `colormux`. Together, these two modules translate a face of 3-bit color encodings into 24-bit color encodings, and allow a 24-bit color encoding to be accessed according to its 0-8 location value described in Section 4.2. This 24-bit color is output from `make_squares` and input to `make_data_stream`, which translates these 24 bits into a single-bit datastream with the encoding shown in Figure 6. Figure 9 shows a block diagram of the `rubiks_core` module.

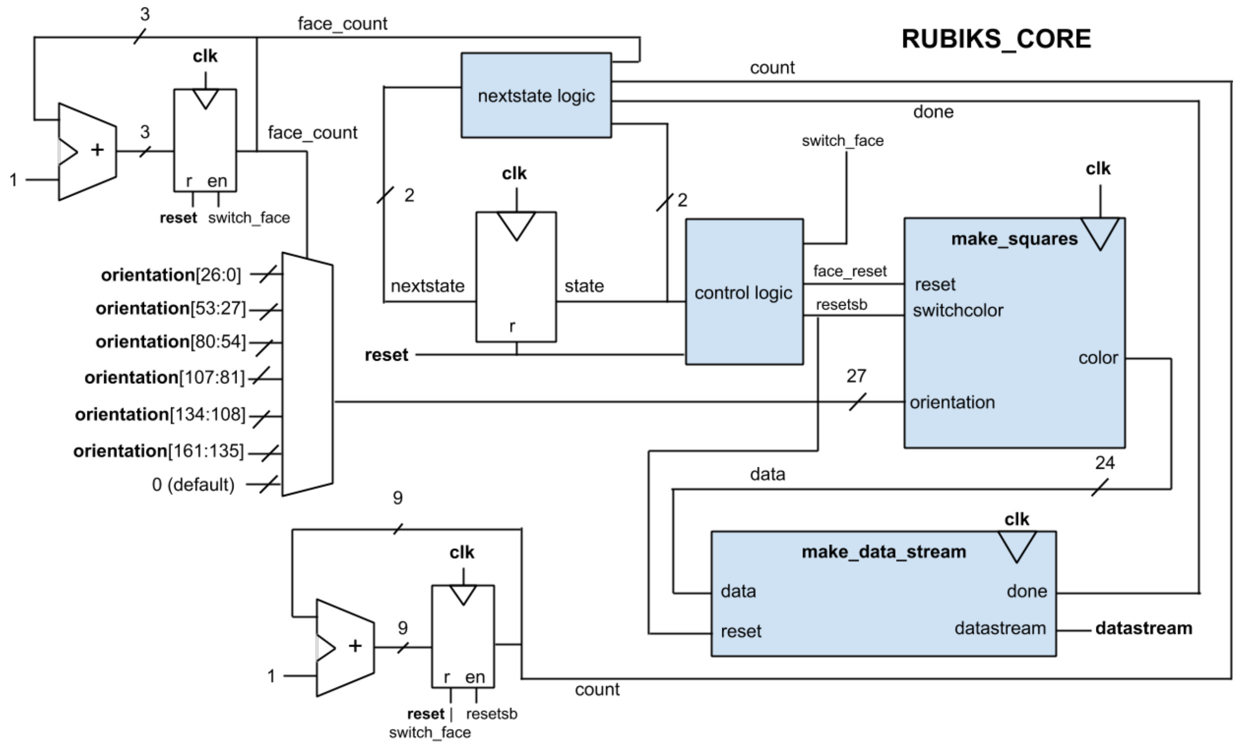


Figure 9: Block diagram of `rubiks_core` module.

6 Results

We successfully implemented a fully-functional LED Rubik's Cube that is controlled by a user control panel with seven buttons and a rotary encoder. Our microcontroller correctly rotates the cube orientation according to the user input and sends this orientation to the FPGA over SPI. The FPGA then drives all six LED matrices with one output pin.

There were three main challenges that we faced as we implemented this design. As mentioned in Section 4.3, we experienced problems with signal integrity when we sent 54 bytes over SPI. To fix this, we wrote a method in our microcontroller code that condensed the 54-byte orientation down to 21 bytes, as we were using only 3 bits in each byte. Another challenge was driving all six LED arrays using the data sent over SPI. For a while, we were able to light up the arrays using a value hard-coded into our SystemVerilog. But, once we started using SPI, the entire orientation was off by a single cycle. We used ModelSim to try to debug this issue, and ultimately found that we needed to be updating the current face that was being illuminated a clock cycle earlier than we were. The final big challenge that we faced was aligning the haptic feedback of the rotary encoder with discrete cube rotations, as explained in Sections 2.1 and 4.1.

We met all of the deliverables outlined in our proposal, with a few minor exceptions. We were initially concerned about power consumption, and decided to replace the white face with a purple face for this reason. However, at our desired brightness, the cube only used about 0.5 A of current, so we switched back to the traditional Rubik's Cube colors. Another difference is that we used just one wire to drive all six faces, rather than one wire per face. We chose this implementation because although it is slightly slower to write all six faces with one wire, it also uses less hardware and fewer wires. We decided that a single input wire to the cube was ultimately more elegant and user-friendly, and the latency to write to the cube was negligible either way. These two changes from our proposal were minor and ultimately improved our cube beyond what we had initially outlined.

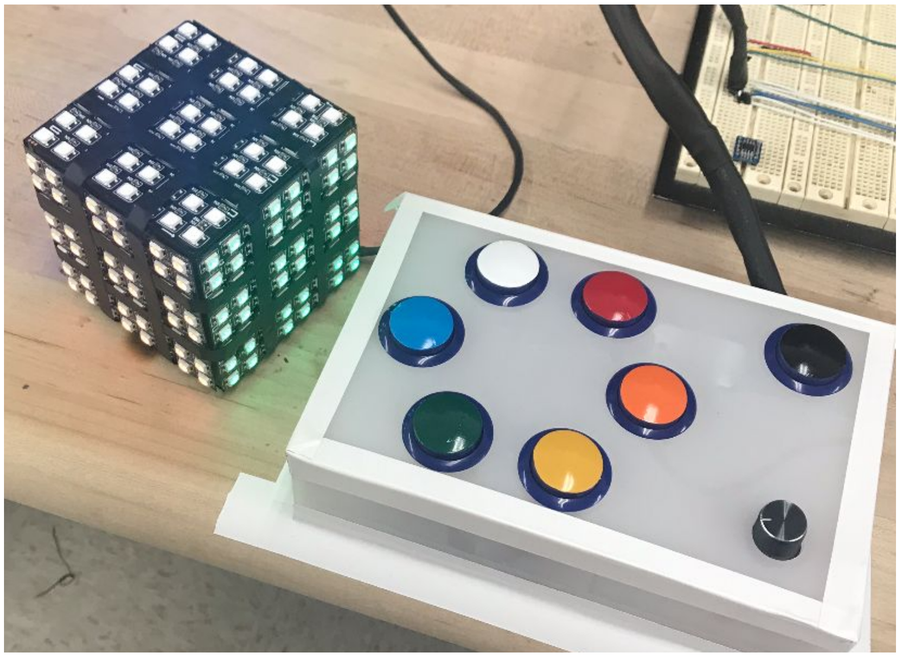


Figure 10: Our finished LED Rubik's Cube and user control panel.

We are happy with our final prototype, which is shown in Figure 10. Many people who used the cube were confused at first by the user interface, but understood it after a brief explanation. Even people who have experience solving Rubik's Cubes struggled to solve the LED cube, as they could not make use of muscle memory and were required to think about the cube in a new way. One user said that these features helped him understand his moves, rather than just rely on muscle memory. This user was able to solve the cube in about 30 minutes on the first try, and 10 minutes on the second try. Another advantage of our cube over a traditional cube is the self-scrambling feature.

In the future, we would like to add a solve feature which shows each step to solve the cube. We would also like to add flashing faces to show the user which face is currently selected.

7 References

[1] Kiss, Sandor. “Educational Challenges of Rubik's Cube.” 2014. X Presented at Workshop on Particle Correlations and Femtoscopy (WPCF 2014), Gyongyos, Hungary. https://arxiv.org/abs/1505.00750.
[2] Henry’s Bench. “KEYES KY-040 ARDUINO ROTARY ENCODER USER MANUAL.” 2015. http://henrysbench.capnfatz.com/henrys-bench/arduino-sensors-and-input/keyes-ky-040-arduino-rotary-encoder-user-manual/.
[3] How To Mechatronics. “How Rotary Encoder Works and How To Use It with Arduino.” Youtube, 2016. https://www.youtube.com/watch?v=v4BbSzJ-hz4.
[4] Worldsemi. “WS2812B Intelligent Control LED Light Source.” https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf.
[5] Kuman. “Kuman for Arduino 8x8 RGB LED Flexible WS2812B 5050 Matrix Dream Color Individually Addressable LED Programmable Pixel Display Screen led Panel WS01.” Amazon. https://www.amazon.com/Kuman-Flexible-Individually-Addressable-Programmable/dp/B01MCUOD8N.

8 Parts List

Table 1 shows the parts that we bought to create the project. In addition to these parts, we used standard pushbuttons, resistors, and capacitors found in the digital lab. To create the boxes for the user control panel and Rubik’s Cube, we laser cut ⅛” acrylic and assembled them with epoxy and tape.

Table 1: Parts list

Part	*Quantity*	*Source*	*Vendor Part #*	*Total Price*
8 by 8 LED Matrix	6	Kuman	WS2812B 8*8 LED Matrix panel	$101.94
Rotary Encoder	1	Keyes	KY-040	$8.99
12” x 12” x ⅛” White Acrylic	2	Falken Design	WT2447	$20.98

9 Appendices

Appendix A: SystemVerilog

```
/*
Aaron Trujillo & Pinky King
atrujillo@g.hmc.edu pking@g.hmc.edu
Fall 2019
*/

// top level module, contains spi and core modules
module send_bytes(input  logic clk,
                  input  logic sck,
                  input  logic sdi,
                  input  logic load,
                  output logic datastream);

        logic [161:0] orientation;
        logic reset;
        rubiks_spi spi(sck, clk, sdi, load, reset, orientation);
        rubiks_core core(clk, reset, orientation, datastream);

endmodule

// SPI module used to retrieve the current cube orientation from the MC and return
// a DONE signal to the MC once the display has been illuminated
module rubiks_spi(input  logic sck,
                  input  logic clk,
                  input  logic sdi,
                  input  logic load,
                  output logic reset,
                  output logic [161:0] orientation);

   logic [7:0] counter;
   logic [161:0] value;
   // assert load
   // apply 162 sclks to shift orientation starting with orientation[0]
   always_ff @(posedge sck)
        if (!load) begin
                counter <= 0;
        end
        else if (load & counter != 8'd162) begin
                {value} = {value[160:0], sdi};
                counter = counter+1;
        end

        // assert a reset signal once the orientation is loaded in
        level_to_pulse rst(clk, (counter == 8'd162), reset);
        assign orientation = value[161:0];
```

```
endmodule

// programs a rubiks face with colors given by orientation
module rubiks_core(input  logic clk, reset,
                   input  logic [162:0] orientation,
                   output logic datastream);

        typedef enum logic [2:0] {switching, sending, new_face, load, over} statetype;
        statetype state, nextstate;

        logic resetsb, done, face_reset;
        logic [26:0] current_face_orientation, blank_face;
        logic [23:0] data;
        logic [8:0] count;
        logic [2:0] face_count;

        // register for finite state machine and counter
        always_ff @(posedge clk)
           if (reset) begin
                       state <= switching;
              count <= 0;
                       face_count <= 0;
           end
           else if (nextstate == new_face) begin
                       face_count <= face_count + 1;
                       state <= nextstate;
                       count <= 0;
                end
                else if (state == switching) count <= count+1;
           else state <= nextstate;


        // next state logic for finite state machine
        // sending: waits for make_data_stream to send 24 bits
        // switching: updates data to be a new 24 color bits
        // over and finish: show that this face is done writing so that we can write the next one
        always_comb
           case (state)

              switching: nextstate = sending;
              sending:  if      (count == 9'd65)       nextstate = new_face;
                        else if (done)                 nextstate = switching;
                        else                           nextstate = sending;
              new_face: if      (face_count == 3'b110) nextstate = over;
                        else                           nextstate = switching;
              over:                                    nextstate = over;
              default:                                 nextstate = over;

           endcase
        // chooses which face to be making a datastream for
        always_comb
           case (face_count)
                   3'b000:    current_face_orientation = orientation[26:0];
                        3'b001:    current_face_orientation = orientation[53:27];
                        3'b010:    current_face_orientation = orientation[80:54];
                        3'b011:    current_face_orientation = orientation[107:81];
                        3'b100:    current_face_orientation = orientation[134:108];
                        3'b101:    current_face_orientation = orientation[161:135];
                        default:   current_face_orientation = 27'b0;
                endcase

        // control logic
        assign resetsb = (nextstate == new_face) | (state == switching);
        assign face_reset = (state == new_face) | reset;

        // get the 24-bit color data from make squares
        makesquares ms(clk, face_reset, resetsb, current_face_orientation, data);

        // make the datastream based on the 24 bits of color data
        make_data_stream mds(clk, resetsb, reset, data, datastream, done);

endmodule

// outputs colors in correct order to display squares for rubiks cube
module makesquares(input  logic clk, reset, switchcolor,
                   input  logic [26:0] orientation,
                   output logic [23:0] color);

        logic [3:0] column, nextcolumn;
        logic [3:0] row, nextrow;
        logic switchcolumn, oddcol;

        // control logic for choosing correct color
        logic color1, color2, color3, color4, color5, color6, color7, color8, color9, blank;
        logic [9:0] controlcolors;

        // register for updating the row and column of the current LED
        always_ff @(posedge clk)
           if (reset) begin
              row <= 4'd9;
              column <= 4'd0;
           end
           else if (switchcolumn & switchcolor) column <= nextcolumn;
           else if (switchcolor)                   row <= nextrow;

        // two finite state machines, one switches cols, one switches rows
        // goes through in a snakelike order, order in which LEDs are written
        always_comb
           case (column)
```

```
      4'd0:    nextcolumn = 4'd1;
      4'd1:    nextcolumn = 4'd2;
      4'd2:    nextcolumn = 4'd3;
      4'd3:    nextcolumn = 4'd4;
      4'd4:    nextcolumn = 4'd5;
      4'd5:    nextcolumn = 4'd6;
      4'd6:    nextcolumn = 4'd7;
      4'd7:    nextcolumn = 4'd8;
      4'd8:    nextcolumn = 4'd8;
      default: nextcolumn = 4'd8;
endcase

always_comb
   case (row)
      4'd9: nextrow = 4'd0; // buffer state to handle reset case
      4'd0: if      (column == 4'd7) nextrow = 4'd8;
            else if (oddcol)         nextrow = 4'd0;
            else                     nextrow = 4'd1;
      4'd1: if      (oddcol)         nextrow = 4'd0;
            else                     nextrow = 4'd2;
      4'd2: if      (oddcol)         nextrow = 4'd1;
            else                     nextrow = 4'd3;
      4'd3: if      (oddcol)         nextrow = 4'd2;
            else                     nextrow = 4'd4;
      4'd4: if      (oddcol)         nextrow = 4'd3;
            else                     nextrow = 4'd5;
      4'd5: if      (oddcol)         nextrow = 4'd4;
            else                     nextrow = 4'd6;
      4'd6: if      (oddcol)         nextrow = 4'd5;
            else                     nextrow = 4'd7;
      4'd7: if      (oddcol)         nextrow = 4'd6;
            else                           nextrow = 4'd7;
      4'd8:                          nextrow = 4'd8;
      default:                       nextrow = 4'd8;
   endcase

// control logic
assign switchcolumn = (oddcol & (row == 4'd0)) | ((~oddcol) & (row == 4'd7));
assign oddcol = (column == 4'd1)|(column == 4'd3)|(column == 4'd5)|(column == 4'd7);

// color bit logic to determine which LEDs correspond to each square
assign blank = (row == 4'd2)|(row == 4'd5)|(column== 4'd5)|(column== 4'd2);
assign color1 = ((row == 4'd0)|(row == 4'd1))&((column== 4'd0)|(column== 4'd1));
assign color2 = ((row == 4'd3)|(row == 4'd4))&((column== 4'd0)|(column== 4'd1));
assign color3 = ((row == 4'd6)|(row == 4'd7))&((column== 4'd0)|(column== 4'd1));
assign color4 = ((row == 4'd6)|(row == 4'd7))&((column== 4'd3)|(column== 4'd4));
assign color5 = ((row == 4'd3)|(row == 4'd4))&((column== 4'd3)|(column== 4'd4));
assign color6 = ((row == 4'd0)|(row == 4'd1))&((column== 4'd3)|(column== 4'd4));
assign color7 = ((row == 4'd0)|(row == 4'd1))&((column== 4'd6)|(column== 4'd7));
assign color8 = ((row == 4'd3)|(row == 4'd4))&((column== 4'd6)|(column== 4'd7));
```

```
        assign color9 = ((row == 4'd6)|(row == 4'd7))&((column== 4'd6)|(column== 4'd7));

        // choose the color data based on the color mux
        assign controlcolors = {blank, color9, color8, color7, color6, color5, color4, color3, color2, 
color1};
        colormux cm(controlcolors, orientation, color);

endmodule

// takes in 3 bits of current orientation and converts them to the
// corresponding HEX values that we need to illuminate the matrix
module convert_orientation(input  logic [2:0] bit_value,
                           output logic [23:0] hex_value);

        always_comb
        casex (bit_value)
                3'b000:  hex_value = 24'h00b000; // red
                3'b001:  hex_value = 24'h00f060; // orange
                3'b010:  hex_value = 24'h00b0b0; // yellow
                3'b011:  hex_value = 24'h0000b0; // green
                3'b100:  hex_value = 24'hb00000; // blue
                3'b101:  hex_value = 24'h909090; // white
                default: hex_value = 24'h000000; // blank
        endcase

endmodule

// takes in the current orientation as well as a one-hot encoding that
// allows us to illuminate the matrix properly
module colormux(input  logic [9:0] colorcontrol,
                input  logic [26:0] orientation,
                output logic [23:0] color);

        logic [23:0] sqr1color, sqr2color, sqr3color, sqr4color, sqr5color, sqr6color, sqr7color, 
sqr8color, sqr9color;

        // convert each necessary piece of the orientation into the proper
        // HEX value for the square that the color corresponds to
        convert_orientation color1(orientation[8:6], sqr1color);
        convert_orientation color2(orientation[5:3], sqr2color);
        convert_orientation color3(orientation[2:0], sqr3color);
        convert_orientation color4(orientation[11:9], sqr4color);
        convert_orientation color5(orientation[14:12], sqr5color);
        convert_orientation color6(orientation[17:15], sqr6color);
        convert_orientation color7(orientation[26:24], sqr7color);
        convert_orientation color8(orientation[23:21], sqr8color);
        convert_orientation color9(orientation[20:18], sqr9color);

        // use 1-Hot encoding to determine which square is currently being lit
        always_comb
```

```
         case (colorcontrol)
            10'b0000000001: color = sqr1color;
            10'b0000000010: color = sqr2color;
            10'b0000000100: color = sqr3color;
            10'b0000001000: color = sqr4color;
            10'b0000010000: color = sqr5color;
            10'b0000100000: color = sqr6color;
            10'b0001000000: color = sqr7color;
            10'b0010000000: color = sqr8color;
            10'b0100000000: color = sqr9color;
            default:        color = 24'h000000;
         endcase

endmodule

//////////////////////////////////////////////////////////////
// Takes in 24-bit color data, outputs one bit that follows
// the pattern detailed here:
// https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf
// for 1s and 0s. Assumes a 40 MHz clock.
//////////////////////////////////////////////////////////////
module make_data_stream(input  logic clk, reset, globalreset,
                        input  logic [23:0] data,
                        output logic datastream, done);

   // counter logic
   logic [10:0] counterval;
   logic [4:0] bitcounter;
   logic [10:0] count;

   // control logic
   logic currentbit, nextbit;
   logic [2:0] s;
   logic reset_counter, incbitcounter;

   // state type
   typedef enum logic [2:0] {T0H, T1H, T0L, T1L, R} statetype;
   statetype state, nextstate;

   // register for main counter
   always_ff @(posedge clk)
      if      (reset | reset_counter | globalreset) count <= 0;
      else if (~done)                               count <= count+1;

   // register for bitcounter
   always_ff @(posedge clk)
      if      (reset | globalreset)                            bitcounter <= 0;
      else if (reset_counter & incbitcounter & ~done) bitcounter <= bitcounter+1;

   // register for FSM
   always_ff @(posedge clk)
      if (reset) begin
         if (currentbit) state <= T1H;
         else            state <= T0H;
      end
      else               state <= nextstate;

   // nextstate logic
   // T0H: high pulse for writing a 0
   // T1H: high pulse for writing a 1
   // T0L: low pulse for writing a 0
   // T1L: low pulse for writing a 1
   // R: reset to indicate shift
   always_comb
      case (state)
         T0H: if      (~reset_counter)       nextstate = T0H;
              else                           nextstate = T0L;
         T1H: if      (~reset_counter)       nextstate = T1H;
              else                           nextstate = T1L;
         T0L: if      (~reset_counter)       nextstate = T0L;
              else if (nextbit)              nextstate = T1H;
              else if (bitcounter == 5'd23)  nextstate = R;
              else                           nextstate = T0H;
         T1L: if      (~reset_counter)       nextstate = T1L;
              else if (nextbit)              nextstate = T1H;
              else if (bitcounter == 5'd23)  nextstate = R;
              else                           nextstate = T0H;
         R:   if      (~reset_counter)       nextstate = R;
              else if (bitcounter == 5'd24)  nextstate = R;
              else if (nextbit)              nextstate = T1H;
              else                           nextstate = T0H;
         default:                            nextstate = R;
      endcase

   // control signal logic
   assign currentbit = data[bitcounter]; // current bit that is being written
   assign nextbit = data[bitcounter+1]; // get the next bit to write
   assign done = (bitcounter == 5'd24 & reset_counter); // when all 24 bits have been written, single
pulse
   assign reset_counter = (count == counterval); // switch to next state, and reset counter
   assign incbitcounter = (state == T0L)|(state==T1L); // when the bit counter should be updated, which
is low pulse
   assign datastream = ((state == T1H)|(state == T0H))&(~reset); // data stream is high when we are in
high pulse states
   assign s = {(state==R),(state==T0L)|(state==T1L), (state==T1H)|(state==T1L)}; // input to mux to
choose constants
   countervalmux cntrvalmux(s, counterval); // mux chooses constants, depending on how long the pulse
should be

endmodule
```

```
// mux for choosing counter value, depending on state
module countervalmux(input logic [2:0] s,
                     output logic [10:0] out);

   always_comb
      case (s)
         3'b000:  out = 11'd16; // T0H
         3'b001:  out = 11'd32; // T1H
         3'b010:  out = 11'd34; // T0L
         3'b011:  out = 11'd18; // T1L
         3'b100:  out = 11'd1;  // R
         default: out = 11'd1;  // R
      endcase

endmodule

// this module is used for when a button is held for longer than
// one clock cycle
module level_to_pulse(input logic clk,
                      input logic load,
                      output logic reset);

        typedef enum logic [1:0] {s0, s1, s2, s3} statetype;
        statetype state, nextstate;

        // state register
        always_ff@(posedge clk)
                state <= nextstate;

        // nextstate logic
        always_comb
                case(state)
                        s0:     if (load) nextstate = s1;
                                else      nextstate = s0;
                        s1:     if (load) nextstate = s2;
                                else      nextstate = s0;
                        s2:     if (load) nextstate = s2;
                                else      nextstate = s0;
                        default:          nextstate = s0;
                endcase
        // output logic
        assign reset = (state == s1);
Endmodule
```

Appendix B: Microcontroller Code

```
/*
Aaron Trujillo & Pinky King
atrujillo@g.hmc.edu pking@g.hmc.edu
E155 Final Project: LED Rubik's Cube
Fall 2019
*/

#include "SAM4S4B.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Rot Encoder Pins
#define pinCCW PIO_PA17 // Connected to DT on KY040
#define pinCW PIO_PA18 // Connected to CLK on KY040
// SPI Pins
#define LOAD_PIN PIO_PA29
// Button Pins
#define RED_FACE PIO_PA19
#define ORANGE_FACE PIO_PA20
#define YELLOW_FACE PIO_PA21
#define GREEN_FACE PIO_PA22
#define BLUE_FACE PIO_PA23
#define WHITE_FACE PIO_PA24
#define RESET_BUTTON PIO_PA25

#define SCRAMBLE_NUM 15

char starting_orientation[54] = {0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5,
                                 0x4, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4,
                                 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3,
                                 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2,
                                 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1,
                                 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
char shifted[21];


///////////////////////////
// function prototypes
///////////////////////////

// set up PIO pins for buttons and rot encoder
void user_interface_setup(void);

// reads in button presses from user
// if reset button is pressed, returns random seed depending
// on length of button press
```

```
int read_input(char*);

// spi communication for microcontroller
// takes in an array of bytes that are then sent over
// using spiSendRecieve and then waits for a DONE signal
void send_orientation(char*);

// these functions are used to change the orientation of the cube
void scramble_cube(char*, int);
void rotate_cube(char*, char*);

// these functions are used to determine the new orientation
// of the cube based off the rotation of the rotary encoder
void counter_clockwise_turn(char*, char);
void clockwise_turn(char*, char);

// these functions are used to perform the shifting necessary to
// be able to send our orientation over spi
void shift_helper(char*, int, int, int, int, int, int, int, int, int, int, int);
void final_shift_helper(char*);
void shift_orientation(char*);

//////////////////////////
// main
//////////////////////////
int main(void) {

        // initialize peripherals
        samInit();
        pioInit();
        tcDelayInit();
        // set up SPI and load pin
        spiInit(MCK_FREQ/244000, 0, 1); // 244000
        pioPinMode(LOAD_PIN, PIO_OUTPUT);

        // starting orientation is solved cube
        char orientation[54];
        for (int i = 0; i < 54; i++) {
                        orientation[i] = starting_orientation[i];
        }
        shift_orientation(orientation);
        send_orientation(shifted);

        // set up variables for reading user input
        int pinCWLast; // used to track the most recent rot. encoder position
        int rot; // used to read a current rot position
        int rand_seed;
        char scramble = 1;
        char user_input[2];
        char rotations[2];
        rotations[0] = 0x3;  // initialize rotary encoder
        rotations[1] = 0x3;
        user_input[0] = 0x7; // a button press of 7 indicates that the user has not yet pressed a
button
        int reset = 0;
        // set up user interface
        user_interface_setup();
        // perform initial read from rot encoder
        pinCWLast = pioDigitalRead(pinCW);
        while (1) {

                // read user input and get random seed
                rand_seed = read_input(user_input);

                if (user_input[0] == 0x7) { // user has never picked a face
                        reset = 0; // keeps track of whether cube has been reset
                        continue; // wait for them to choose a face

                } else if (user_input[0] == 0x6) { // user has reset cube
                          if (reset == 0) {
                                        // if cube is solved, scramble
                                        // otherwise, reset to solved cube
                                        scramble = 1;
                                        for (int i = 0; i < 54; i++) {
                                                if (orientation[i] != starting_orientation[i]) {
                                                        scramble = 0;
                                                        break;
                                                }
                                        }
                                        if (scramble) { // cube is solved, so scramble
                                                scramble_cube(orientation, rand_seed);
                                                user_input[0] = 0x7;
                                        }
                                        else { // cube is scrambled, so solve
                                                for (int i = 0; i < 54; i++) orientation[i] =
starting_orientation[i];
                                        }
                                        shift_orientation(orientation);
                                        send_orientation(shifted);
                                        // indicate that cube was just reset so that it isn’t
immediately reset again
                                        reset = 1;
                                }
                                // change out of reset command
                                user_input[0] = 0x7;

                } else { // user has chosen a face
                        reset = 0;
                        // read the rotary encoder
                        rot = pioDigitalRead(pinCW);
```

```
if (rot != pinCWLast) { // rotation has occurred
        if (pioDigitalRead(pinCCW) != rot) {
                // a "3" indicates that the last rotation has not been read
                // check that two rotations (indicated by click) have occured
                if (rotations[0] != 0x3) rotations[1] = 0x0;
                else rotations[0] = 0x0;
        } else {
                // check that two rotations (indicated by click) have occured
                if (rotations[0] != 0x3) rotations[1] = 0x1;
                else rotations[0] = 0x1;
        }
        // user is rotating clockwise
        if ((rotations[0] == 0x0) & (rotations[1] == 0x0)) {
                user_input[1] = 0x0;
                rotate_cube(orientation, user_input); // ROTATE
                shift_orientation(orientation);
                send_orientation(shifted);
                rotations[1] = 0x3; // reset rotary encoder reading
                rotations[0] = 0x3;
                pinCWLast = pioDigitalRead(pinCW);
        }
        // user is rotating counterclockwise
        else if ((rotations[0] == 0x1) & (rotations[1] == 0x1)) {
                user_input[1] = 0x1;
                rotate_cube(orientation, user_input); // ROTATE
                shift_orientation(orientation);
                send_orientation(shifted);
                rotations[1] = 0x3; // reset rotary encoder reading
                rotations[0] = 0x3;
                pinCWLast = pioDigitalRead(pinCW);
        }
        // user has switched from clockwise to counterclockwise
        else if ((rotations[0] == 0x0) & (rotations[1] == 0x1)) {
                user_input[1] = 0x1;
                rotate_cube(orientation, user_input); // ROTATE
                shift_orientation(orientation);
                send_orientation(shifted);
                rotations[1] = 0x3; // partially rotary encoder reading
                rotations[0] = 0x1;
                pinCWLast = pioDigitalRead(pinCW);
        }
        // user has switched from counterclockwise to clockwise
        else if ((rotations[0] == 0x1) & (rotations[1] == 0x0)) {
                user_input[1] = 0x0;
                rotate_cube(orientation, user_input); // ROTATE
                shift_orientation(orientation);
                send_orientation(shifted);
                rotations[1] = 0x3; // partially rotary encoder reading
                rotations[0] = 0x0;
                pinCWLast = pioDigitalRead(pinCW);
```

```
                                }
                        }
                        pinCWLast = rot;
                }
        }
}
///////////////////////////
// helper functions
///////////////////////////

void user_interface_setup() {
        // Setup Buttons
        pioPinMode(RED_FACE, PIO_INPUT);
        pioPinMode(ORANGE_FACE, PIO_INPUT);
        pioPinMode(YELLOW_FACE, PIO_INPUT);
        pioPinMode(GREEN_FACE, PIO_INPUT);
        pioPinMode(BLUE_FACE, PIO_INPUT);
        pioPinMode(WHITE_FACE, PIO_INPUT);
        pioPinMode(RESET_BUTTON, PIO_INPUT);

        // Rotary Encoder Setup
        pioPinMode(pinCW, PIO_INPUT);
        pioPinMode(pinCCW, PIO_INPUT);
}

// scrambles the cube
void scramble_cube(char* orientation, int rand_seed) {
        char user_input[2];
        char range = rand_seed % 10; // determines how many moves will be performed
        // first, set to starting orientation so that the resulting orientation is
        // actually solvable
        for (int i = 0; i < 54; i++) {
                orientation[i] = starting_orientation[i];
        }
        // creates a number of random moves, then performs those rotations
        for (int i = 0; i < SCRAMBLE_NUM + range; i++) {
                // gets random moves with rand_seed
                user_input[0] = rand()%6;
                user_input[0] = (user_input[0]+rand_seed)%6;
                user_input[1] = rand()%2;
                user_input[1] = (user_input[1]+rand_seed)%2;
                rotate_cube(orientation, user_input);
                rand_seed = rand_seed*13;
        }
}

// sends a new orientation over SPI
void send_orientation(char* current_orientation){
        int i;
        // assert load pin to begin SPI
        pioDigitalWrite(LOAD_PIN, 1);
        for (i = 0; i < 21; i++){
                spiSendReceive(current_orientation[i]);
        }
        // deassert
        pioDigitalWrite(LOAD_PIN, 0);
        // give FPGA enough time to program cube
        tcDelayMillis(13);
        // get response from FPGA
        spiSendReceive(0);
}

// reads the user input buttons and modifies the global array of user input
int read_input(char* user_input) {

        char red, orange, yellow, green, blue, white, reset;
        int rand_seed = 0;
        // read buttons
        red = pioDigitalRead(RED_FACE);
        orange = pioDigitalRead(ORANGE_FACE);
        yellow = pioDigitalRead(YELLOW_FACE);
        green = pioDigitalRead(GREEN_FACE);
        blue = pioDigitalRead(BLUE_FACE);
        white = pioDigitalRead(WHITE_FACE);
        reset = pioDigitalRead(RESET_BUTTON);
        if      (red)     user_input[0] = 0x0;
        else if (orange)  user_input[0] = 0x1;
        else if (yellow)  user_input[0] = 0x2;
        else if (green)   user_input[0] = 0x3;
        else if (blue)    user_input[0] = 0x4;
        else if (white)   user_input[0] = 0x5;
        else if (reset)   user_input[0] = 0x6;

        // creates random seed depending on how long
        // the reset button was pressed for
        while (pioDigitalRead(RESET_BUTTON)) {
                rand_seed++;
        }
        return rand_seed;

}

// rotates the cube based on the face and direction selected by user
// user_input[0] is face selection: red-orange-yellow-green-blue-white
// user_input[1] is direction: clockwise-counterclockwise
void rotate_cube(char* current_orientation, char* user_input) {
        if (user_input[1] == 0) { // clockwise
                clockwise_turn(current_orientation, user_input[0]);
        }
        else { // counter clockwise
```

```
                counter_clockwise_turn(current_orientation, user_input[0]);
        }
}

// takes in 54 char color orientation, and a face to rotate
// modifies the orientation to the clockwise rotated orientation
void clockwise_turn(char* current_orientation, char color){
        char temp[54];
        for (int i = 0; i < 54; i++) {
          temp[i] = current_orientation[i];
        }
        // white face starts at 0
        // blue face starts at 9
        // green face starts at 18
        // yellow face starts at 27
        // orange face starts at 36
        // red face starts at 45
        if (color == 0x0) { // red face rotates
                // fix red face
                temp[45] = current_orientation[45+6];
                temp[46] = current_orientation[45+3];
                temp[47] = current_orientation[45];
                temp[48] = current_orientation[45+7];
                temp[50] = current_orientation[45+1];
                temp[51] = current_orientation[45+8];
                temp[52] = current_orientation[45+5];
                temp[53] = current_orientation[45+2];
                // fix white face
                temp[0+6] = current_orientation[18+8];
                temp[0+7] = current_orientation[18+5];
                temp[0+8] = current_orientation[18+2];
                // fix blue face
                temp[9] = current_orientation[0+6];
                temp[9+3] = current_orientation[0+7];
                temp[9+6] = current_orientation[0+8];
                // fix yellow face
                temp[27] = current_orientation[9+6];
                temp[27+1] = current_orientation[9+3];
                temp[27+2] = current_orientation[9];
                // fix green face
                temp[18+2] = current_orientation[27];
                temp[18+5] = current_orientation[27+1];
                temp[18+8] = current_orientation[27+2];
        }
        else if (color == 0x1) { // orange
                // fix orange face
                temp[36] = current_orientation[36+6];
                temp[37] = current_orientation[36+3];
                temp[38] = current_orientation[36];
                temp[39] = current_orientation[36+7];
                temp[41] = current_orientation[36+1];
                temp[42] = current_orientation[36+8];
                temp[43] = current_orientation[36+5];
                temp[44] = current_orientation[36+2];
                // fix white face
                temp[0] = current_orientation[9+2];
                temp[1] = current_orientation[9+5];
                temp[2] = current_orientation[9+8];
                // fix blue face
                temp[9+2] = current_orientation[27+8];
                temp[9+5] = current_orientation[27+7];
                temp[9+8] = current_orientation[27+6];
                // fix yellow face
                temp[27+6] = current_orientation[18];
                temp[27+7] = current_orientation[18+3];
                temp[27+8] = current_orientation[18+6];
                // fix green face
                temp[18+0] = current_orientation[2];
                temp[18+3] = current_orientation[1];
                temp[18+6] = current_orientation[0];
        }
        else if (color == 0x2) { // yellow
                // fix yellow face
                temp[27] = current_orientation[27+6];
                temp[28] = current_orientation[27+3];
                temp[29] = current_orientation[27];
                temp[30] = current_orientation[27+7];
                temp[32] = current_orientation[27+1];
                temp[33] = current_orientation[27+8];
                temp[34] = current_orientation[27+5];
                temp[35] = current_orientation[27+2];
                // fix red face
                temp[45+6] = current_orientation[18+6];
                temp[45+7] = current_orientation[18+7];
                temp[45+8] = current_orientation[18+8];
                // fix blue face
                temp[9+6] = current_orientation[45+6];
                temp[9+7] = current_orientation[45+7];
                temp[9+8] = current_orientation[45+8];
                // fix orange face
                temp[36+6] = current_orientation[9+6];
                temp[36+7] = current_orientation[9+7];
                temp[36+8] = current_orientation[9+8];
                // fix green face
                temp[18+6] = current_orientation[36+6];
                temp[18+7] = current_orientation[36+7];
                temp[18+8] = current_orientation[36+8];
        }
        else if (color == 0x3) { // green
                // fix green face
                temp[18] = current_orientation[18+6];
                temp[19] = current_orientation[18+3];
                temp[20] = current_orientation[18];
                temp[21] = current_orientation[18+7];
                temp[23] = current_orientation[18+1];
                temp[24] = current_orientation[18+8];
                temp[25] = current_orientation[18+5];
                temp[26] = current_orientation[18+2];
                // fix yellow face
                temp[27+0] = current_orientation[45+0];
                temp[27+3] = current_orientation[45+3];
                temp[27+6] = current_orientation[45+6];
                // fix red face
                temp[45+0] = current_orientation[0];
                temp[45+3] = current_orientation[3];
                temp[45+6] = current_orientation[6];
                // fix white face
                temp[0] = current_orientation[36+8];
                temp[3] = current_orientation[36+5];
                temp[6] = current_orientation[36+2];
                // fix orange face
                temp[36+2] = current_orientation[27+6];
                temp[36+5] = current_orientation[27+3];
                temp[36+8] = current_orientation[27+0];
        }
        else if (color == 0x4) { // blue
                // fix blue face
                temp[9] = current_orientation[9+6];
                temp[10] = current_orientation[9+3];
                temp[11] = current_orientation[9];
                temp[12] = current_orientation[9+7];
                temp[14] = current_orientation[9+1];
                temp[15] = current_orientation[9+8];
                temp[16] = current_orientation[9+5];
                temp[17] = current_orientation[9+2];
                // fix yellow face
                temp[27+2] = current_orientation[36+6];
                temp[27+5] = current_orientation[36+3];
                temp[27+8] = current_orientation[36+0];
                // fix red face
                temp[45+2] = current_orientation[27+2];
                temp[45+5] = current_orientation[27+5];
                temp[45+8] = current_orientation[27+8];
                // fix white face
                temp[2] = current_orientation[45+2];
                temp[5] = current_orientation[45+5];
                temp[8] = current_orientation[45+8];
                // fix orange face
                temp[36+0] = current_orientation[8];
                temp[36+3] = current_orientation[5];
```

```
                temp[36+6] = current_orientation[2];

        }
        else if (color == 0x5) { // white
                // fix white face
                temp[0] = current_orientation[6];
                temp[1] = current_orientation[3];
                temp[2] = current_orientation[0];
                temp[3] = current_orientation[7];
                temp[5] = current_orientation[1];
                temp[6] = current_orientation[8];
                temp[7] = current_orientation[5];
                temp[8] = current_orientation[2];
                // fix red face
                temp[45+0] = current_orientation[9+0];
                temp[45+1] = current_orientation[9+1];
                temp[45+2] = current_orientation[9+2];
                // fix blue face
                temp[9+0] = current_orientation[36+0];
                temp[9+1] = current_orientation[36+1];
                temp[9+2] = current_orientation[36+2];
                // fix orange face
                temp[36+0] = current_orientation[18+0];
                temp[36+1] = current_orientation[18+1];
                temp[36+2] = current_orientation[18+2];
                // fix green face
                temp[18+0] = current_orientation[45+0];
                temp[18+1] = current_orientation[45+1];
                temp[18+2] = current_orientation[45+2];
        }
        for (int i = 0; i < 54; i++) {
          current_orientation[i] = temp[i];
        }
}

// takes in 54 char color orientation, and a face to rotate
// modifies the orientation to the counterclockwise rotated orientation
void counter_clockwise_turn(char* current_orientation, char color){
        char temp[54];
        for (int i = 0; i < 54; i++) {
          temp[i] = current_orientation[i];
        }
        if (color == 0x0) { // red
                // fix red face
                temp[45] = current_orientation[45+2];
                temp[46] = current_orientation[45+5];
                temp[47] = current_orientation[45+8];
                temp[48] = current_orientation[45+1];
                temp[50] = current_orientation[45+7];
                temp[51] = current_orientation[45+0];
                temp[52] = current_orientation[45+3];
                temp[53] = current_orientation[45+6];
                // fix white face
                temp[0+6] = current_orientation[9+0];
                temp[0+7] = current_orientation[9+3];
                temp[0+8] = current_orientation[9+6];
                // fix blue face
                temp[9] = current_orientation[27+2];
                temp[9+3] = current_orientation[27+1];
                temp[9+6] = current_orientation[27+0];
                // fix yellow face
                temp[27] = current_orientation[18+2];
                temp[27+1] = current_orientation[18+5];
                temp[27+2] = current_orientation[18+8];
                // fix green face
                temp[18+2] = current_orientation[8];
                temp[18+5] = current_orientation[7];
                temp[18+8] = current_orientation[6];
        }
        else if (color == 0x1) { // orange
                // fix orange face
                temp[36] = current_orientation[36+2];
                temp[37] = current_orientation[36+5];
                temp[38] = current_orientation[36+8];
                temp[39] = current_orientation[36+1];
                temp[41] = current_orientation[36+7];
                temp[42] = current_orientation[36+0];
                temp[43] = current_orientation[36+3];
                temp[44] = current_orientation[36+6];
                // fix white face
                temp[0] = current_orientation[18+6];
                temp[1] = current_orientation[18+3];
                temp[2] = current_orientation[18+0];
                // fix blue face
                temp[9+2] = current_orientation[0];
                temp[9+5] = current_orientation[1];
                temp[9+8] = current_orientation[2];
                // fix yellow face
                temp[27+6] = current_orientation[9+8];
                temp[27+7] = current_orientation[9+5];
                temp[27+8] = current_orientation[9+2];
                // fix green face
                temp[18+0] = current_orientation[27+6];
                temp[18+3] = current_orientation[27+7];
                temp[18+6] = current_orientation[27+8];
        }
        else if (color == 0x2) { // yellow
                // fix yellow face
                temp[27] = current_orientation[27+2];
                temp[28] = current_orientation[27+5];
                temp[29] = current_orientation[27+8];
                temp[30] = current_orientation[27+1];
                temp[32] = current_orientation[27+7];
                temp[33] = current_orientation[27+0];
                temp[34] = current_orientation[27+3];
                temp[35] = current_orientation[27+6];
                // fix red face
                temp[45+6] = current_orientation[9+6];
                temp[45+7] = current_orientation[9+7];
                temp[45+8] = current_orientation[9+8];
                // fix blue face
                temp[9+6] = current_orientation[36+6];
                temp[9+7] = current_orientation[36+7];
                temp[9+8] = current_orientation[36+8];
                // fix orange face
                temp[36+6] = current_orientation[18+6];
                temp[36+7] = current_orientation[18+7];
                temp[36+8] = current_orientation[18+8];
                // fix green face
                temp[18+6] = current_orientation[45+6];
                temp[18+7] = current_orientation[45+7];
                temp[18+8] = current_orientation[45+8];
        }
        else if (color == 0x3) { // green
                // fix green face
                temp[18] = current_orientation[18+2];
                temp[19] = current_orientation[18+5];
                temp[20] = current_orientation[18+8];
                temp[21] = current_orientation[18+1];
                temp[23] = current_orientation[18+7];
                temp[24] = current_orientation[18+0];
                temp[25] = current_orientation[18+3];
                temp[26] = current_orientation[18+6];
                // fix yellow face
                temp[27+0] = current_orientation[36+8];
                temp[27+3] = current_orientation[36+5];
                temp[27+6] = current_orientation[36+2];
                // fix red face
                temp[45+0] = current_orientation[27+0];
                temp[45+3] = current_orientation[27+3];
                temp[45+6] = current_orientation[27+6];
                // fix white face
                temp[0] = current_orientation[45+0];
                temp[3] = current_orientation[45+3];
                temp[6] = current_orientation[45+6];
                // fix orange face
                temp[36+2] = current_orientation[6];
                temp[36+5] = current_orientation[3];
                temp[36+8] = current_orientation[0];
        }
```

```
else if (color == 0x4) { // blue
        // fix blue face
        temp[9] = current_orientation[9+2];
        temp[10] = current_orientation[9+5];
        temp[11] = current_orientation[9+8];
        temp[12] = current_orientation[9+1];
        temp[14] = current_orientation[9+7];
        temp[15] = current_orientation[9+0];
        temp[16] = current_orientation[9+3];
        temp[17] = current_orientation[9+6];
        // fix yellow face
        temp[27+2] = current_orientation[45+2];
        temp[27+5] = current_orientation[45+5];
        temp[27+8] = current_orientation[45+8];
        // fix red face
        temp[45+2] = current_orientation[2];
        temp[45+5] = current_orientation[5];
        temp[45+8] = current_orientation[8];
        // fix white face
        temp[2] = current_orientation[36+6];
        temp[5] = current_orientation[36+3];
        temp[8] = current_orientation[36+0];
        // fix orange face
        temp[36+0] = current_orientation[27+8];
        temp[36+3] = current_orientation[27+5];
        temp[36+6] = current_orientation[27+2];
}
else if (color == 0x5) { // white
        // fix white face
        temp[0] = current_orientation[2];
        temp[1] = current_orientation[5];
        temp[2] = current_orientation[8];
        temp[3] = current_orientation[1];
        temp[5] = current_orientation[7];
        temp[6] = current_orientation[0];
        temp[7] = current_orientation[3];
        temp[8] = current_orientation[6];
        // fix red face
        temp[45+0] = current_orientation[18+0];
        temp[45+1] = current_orientation[18+1];
        temp[45+2] = current_orientation[18+2];
        // fix blue face
        temp[9+0] = current_orientation[45+0];
        temp[9+1] = current_orientation[45+1];
        temp[9+2] = current_orientation[45+2];
        // fix orange face
        temp[36+0] = current_orientation[9+0];
        temp[36+1] = current_orientation[9+1];
        temp[36+2] = current_orientation[9+2];
        // fix green face
```

```
                temp[18+0] = current_orientation[36+0];
                temp[18+1] = current_orientation[36+1];
                temp[18+2] = current_orientation[36+2];
        }
        for (int i = 0; i < 54; i++) {
          current_orientation[i] = temp[i];
        }
}

// takes in a 54-byte array and condenses it down to 21 bytes
void shift_orientation(char* current_orientation){
        shift_helper(current_orientation, 15, 16, 17, 47,46,45,44,43,42,41,40);
        shift_helper(current_orientation, 12, 13, 14 ,39,38,37,36,35,34,33,32);
        shift_helper(current_orientation, 9, 10, 11, 31,30,29,28,27,26,25,24);
        shift_helper(current_orientation, 6, 7, 8, 23,22,21,20,19,18,17,16);
        shift_helper(current_orientation, 3, 4, 5, 15,14,13,12,11,10,9,8);
        shift_helper(current_orientation, 0, 1, 2, 7,6,5,4,3,2,1,0);
        final_shift_helper(current_orientation);
}

// takes in 8 bytes and condenses them into 3 bytes by removing the 5 most significant bits and
// shifting and or'ing the remaining bits together.
void shift_helper(char* current_orientation, int shift_byte0, int shift_byte1, int shift_byte2, int
ori_byte0, int ori_byte1, int ori_byte2, int ori_byte3, int ori_byte4, int ori_byte5, int ori_byte6,
int ori_byte7) {
    char byte0;
    char byte1;
    char byte2;

    byte2 = (current_orientation[ori_byte2] << 6) | (current_orientation[ori_byte1] << 3) |
current_orientation[ori_byte0];
    byte1 = (current_orientation[ori_byte5] << 7) | (current_orientation[ori_byte4] << 4) |
(current_orientation[ori_byte3] << 1) | (current_orientation[ori_byte2] >> 2);
    byte0 = (current_orientation[ori_byte7] << 5) | (current_orientation[ori_byte6] << 2) |
(current_orientation[ori_byte5] >> 1);

    shifted[shift_byte0] = byte0;
    shifted[shift_byte1] = byte1;
    shifted[shift_byte2] = byte2;
}

// Since 54 is not divisible by 8, we needed a special helper function for the final elements of the
array
void final_shift_helper(char* current_orientation) {
    shifted[18] = (current_orientation[48] << 5) | (current_orientation[49] << 2) |
(current_orientation[50] >> 1);
    shifted[19] = (current_orientation[50] << 7) | (current_orientation[51] << 4) |
(current_orientation[52] << 1) | (current_orientation[53] >> 2);
    shifted[20] = (current_orientation[53] << 6) | (0 << 3) | 0;
}
```