Lsab 4 - Rubik's Cube Simulator

Vicky Min (604924014), Jorge E. Pineda (204971366), Aseem S. Sane (804915018)
Department of Computer Science
University of California, Los Angeles

INTRODUCTION

The purpose of this lab was to design and implement our own FPGA project using the technical skills that we had learned throughout the duration of the course. For our demo, we chose to implement a Rubik's cube simulator, was run with the Nexys 3 board and displayed on VGA. The VGA display had six different 3x3 sections to represent each of the faces of the Rubik's cube. The Rubik's cube would begin in the solved state), and the user would have the option to scramble it themselves or choose a random scrambling option.

The inputs taken in were 7 of the switches, as well as all 5 of the push-buttons on the board. The first six switches corresponded to each face of the cube, and if one was on, one could press the left or right button to rotate that face clockwise or counterclockwise, respectively. The last switch was used in scramble mode; this meant that when it was on, the cube would scramble using random moves at a rate of 3 moves per second. The middle button was used to reset the cube to the solved default state, and the top and bottom buttons were for shifting the cube either horizontally or vertically, so that the user could control which face was displayed front and centre.

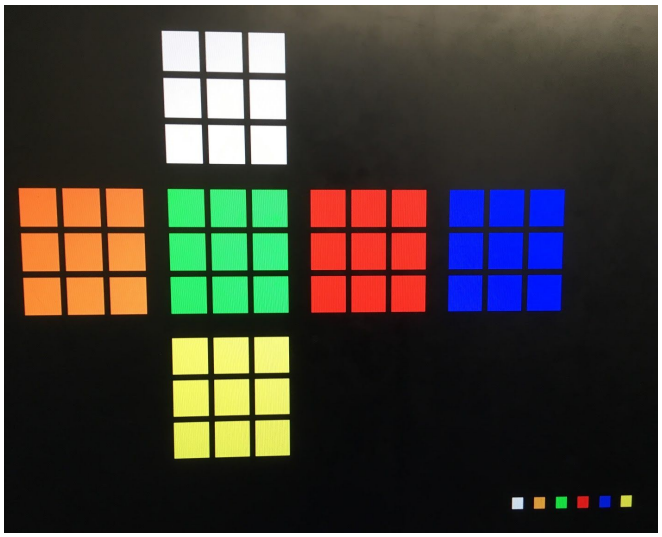

Figure 1: VGA display of Rubik's cube in its default/solved state. The first display state of the cube, from which moves can be made to rotate any face or rotate the cube. The legend in the bottom is representative of which of the first six switches is used to rotate each corresponding face of the cube.

DESIGN DESCRIPTION

Main Module: cube_top

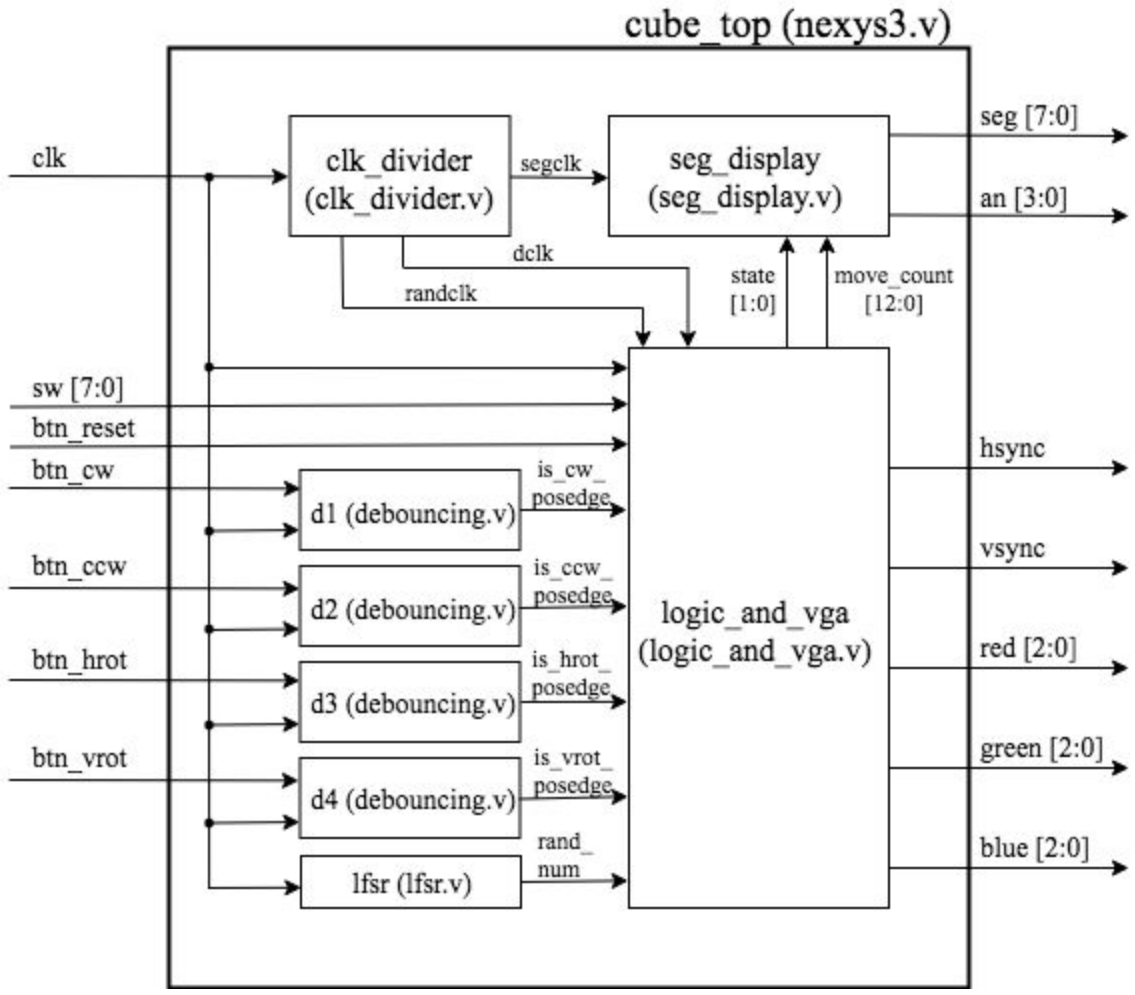

Figure 2: Overall schematic of modular design. The overall module, cube_top, is tasked with declaring all other modules and redirecting different inputs and outputs between the modules as shown. The main module handling the cube movement and logic was dealt with in logic_and_vga, which took in many inputs from other modules, such as the clock divider, debouncing modules, and the lfsr module.

The overall module takes in all the inputs necessary found in nexys3.ucf, as described in the design diagram above. The clk input corresponds to the 100 MHz internal clock; sw[7:0] corresponds to the board's switches, btn_reset corresponds to BTNS, the middle button on the board; btn_cw corresponds to BTNR, the right button on the board; btn_ccw corresponds to BTNL, the left button on the board; btn_hrot corresponds to BTNU, the up button; and btn_vrot corresponds to BTND, the down button on the board.

The cube_top module calls clk_divider.v using the clk input, which returns segclk, used for the seven segment display, dclk, used for the VGA display, and randclk, used for the random shuffle. The debouncing module is called four times as different modules - d1 for debouncing the btn_cw input and

returns is_cw_posedge, d2 for debouncing btn_ccw for is_ccw_posedge, d3 for debouncing btn_hrot for is_hrot_posedge, and d4, for debouncing btn_vrot into is_vrot_posedge. The lfsr module is a linear shift feedback register used for producing rand_num for the random shuffling, and uses the clk input. Finally, the module calls logic_and_vga using most of the other modules' outputs as described above to perform the main VGA display logic, and seg_display for the seven segment display.

Sub-Module #1: clk_divider

This sub-module uses a 24-bit register, q, that increments every time the posedge of the input clk is received. The outputs are each assigned to a certain bit of q: randclk is assigned to the 23rd bit, segclk assigned to the 17th bit, and dclk assigned to the first bit. This makes randclk 5.96 Hz, segclk 381.47Hz, and dclk 25 MHz, each operating at a different frequency for their intended purposes in logic_and_vga. These output clocks are used as inputs to the main logic module, so that different actions could be performed at different frequencies.

Sub-Module #2: seg_display

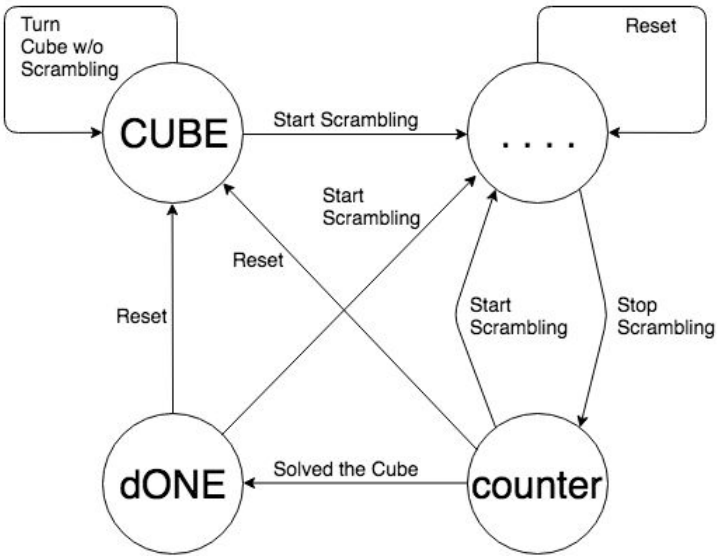

Figure 3: Seven-segment display state diagram. Seg displays based on state[1:0] input, and each functionality (i.e. start scrambling, reset, etc.) updates state[1:0] accordingly. 0 is CUBE, 1 is …., 2 is dONE, 3 is counter.

This module uses various parameters to represent constant values to be represented by the seven segment display, as well as registers digit[1:0], to select between each of the four digits on the display, and seg_values[31:0], to store the values to be displayed by each of the digits. Each time a posedge clk is triggered, the module sets the seg and an Nexys3 outputs to the values stored in digit and corresponding part of seg_values based on the input state as shown in the diagram. The counter is implemented by mapping the number of moves to the corresponding digit on the display using the function seg_val.

Sub-Module #3: debouncing

The debouncing module uses a 22-bit register, step, that increments by 1 during each posedge clk if the input, btn, is detected to be 1. The output, is_posedge, is assigned to be 1 only if step reaches a value of

2^22 - 1 (its maximum value), ensuring that only sufficiently long button presses are detected. These output values are used in the main cube logic module to check if switches were appropriately activated.

Sub-Module #4: lfsr

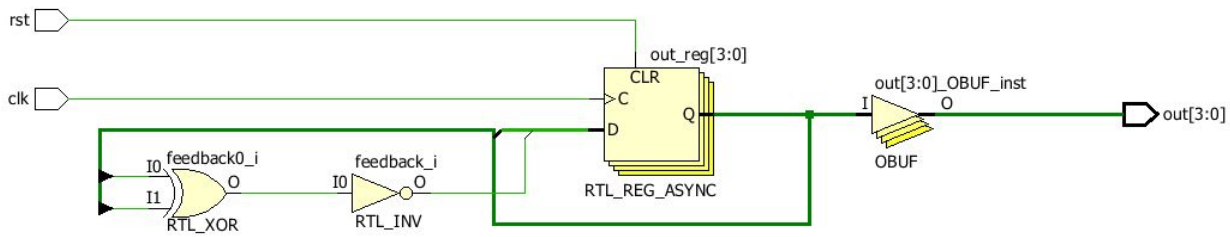

Figure 4: 4-bit Linear Feedback Shift Register (LFSR) register transfer level design. This is the standard design for a 4-bit LFSR module. Our module made slight modifications, in order to only output values between 0 and 11, rather than up to 16, as was necessary for the random scrambler.[2]

The lfrr module, as shown in the diagram, used the clk input to generate a random output from 0 to 11 to store in out[3:0]. On a high level, the lfsr would just generate sequences of random numbers by XNOR'ing bits 3 and 2 of the register, then appending this bit to the end of the register, effectively shifting all the bits left one (so the most significant bit disappears with each run). Unlike figure 4, we did not make use of rst as an input because we didn't have the need to ever reset the output register back to 4'b0. Because the above lfsr design picks an output between 0 and 15 (inclusive), we added an if statement to check that the number chosen was between the range of 0 to 11 (inclusive), since we only had 12 moves to choose from. If the number chosen was larger than 11, we moved on to the next iteration and chose a new number until it was within the correct range.

Sub-Module #5: logic_and_vga

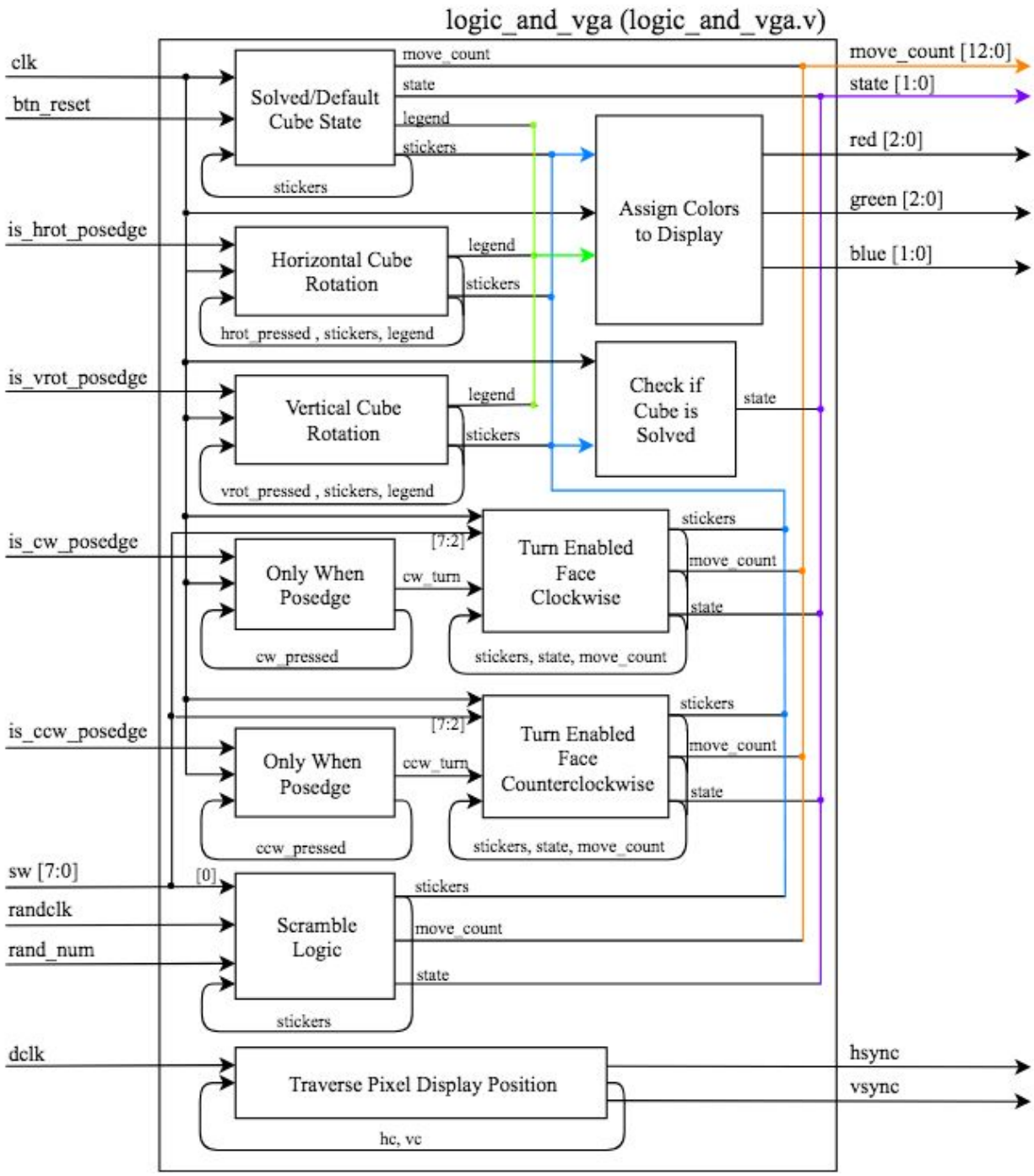

Figure 5: Schematic of logic_and_vga module design. For simplicity, temporary registers within the module are not shown with their size. Note that each clk posedge will only go through a small number of the components in the schematic. The values of certain registers (legend, stickers, move_count and state) are thereby determined by only one of the components that are eventually connected to each respective colored line: legend - green, stickers - blue, move_count - orange, state - purple.

Component #1: Traverse Pixel Display Position

This component activated during each posedge of dclk, the display clock, and incremented hc if hc was within the preset hpixels - 1 value, and if not, set hc to 0, and to increment vc, checked if vc was in the preset vlines - 1 value, and if not, set vc to 0. The hsync and vsync outputs were assigned to the values (hc < hpulse) ? 0:1 and (vc < vpulse) ? 0:1 respectively.

Component #2: Only When Posedge (x2)

The cw_turn and ccw_turn registers were used as output to perform one turn per button press from is_cw_posedge and is_ccw_posedge, and is_hrot_posedge, is_vrot_posedge, and is_rand_posedge were used for one cube rotation per up and down button press.

Component #3/4: Horizontal/Vertical Cube Rotation

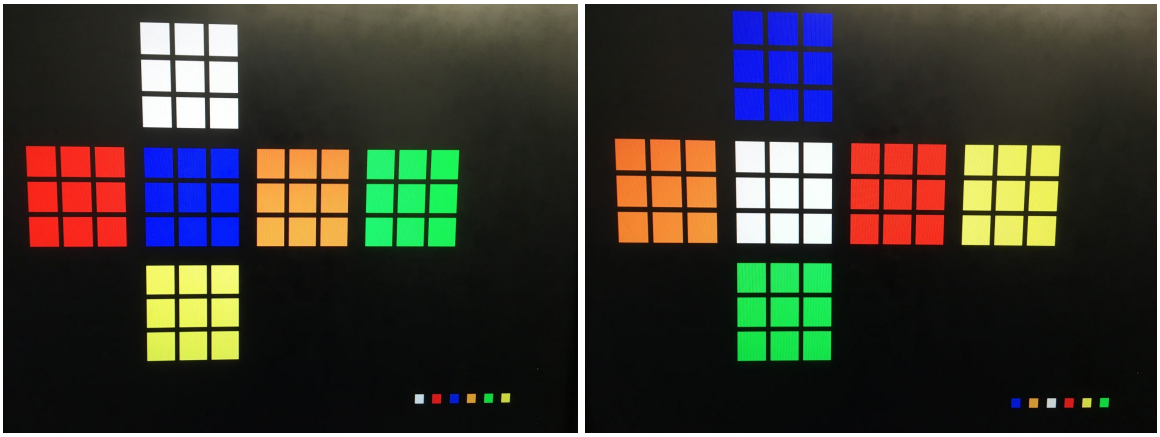

Figure 6: Cube state after horizontal and vertical rotations. Two horizontal rotations on left, one vertical rotation on right.

Cube rotation was implemented by shifting the position of bits as needed in the stickers and legend registers, which were used to determine the VGA display of all the squares. Four faces would be moved for each cube rotation in the stickers and legend registers, and two faces would be rotated clockwise and counterclockwise accordingly to accurately emulate a cube rotation in three dimensions.

Component #5: Solved/Default Cube State

As shown in Figure 1, the btn_reset input, whenever it held the value of 1, would result in setting the stickers and legend registers to their default values, which would be checked for during each posedge clk.

Component #6: Scramble Logic

The moveseq and randclk inputs were both detected for nonzero values to trigger the random mode. The case statement for moveseq[3:0] checked for values between 1 and 12, and each case corresponded to a rotation of a face clockwise or counterclockwise, with the 12 different possibilities accounting for every possible move of a face.

Component #7/8: Turn Enabled Face Clockwise/Counterclockwise

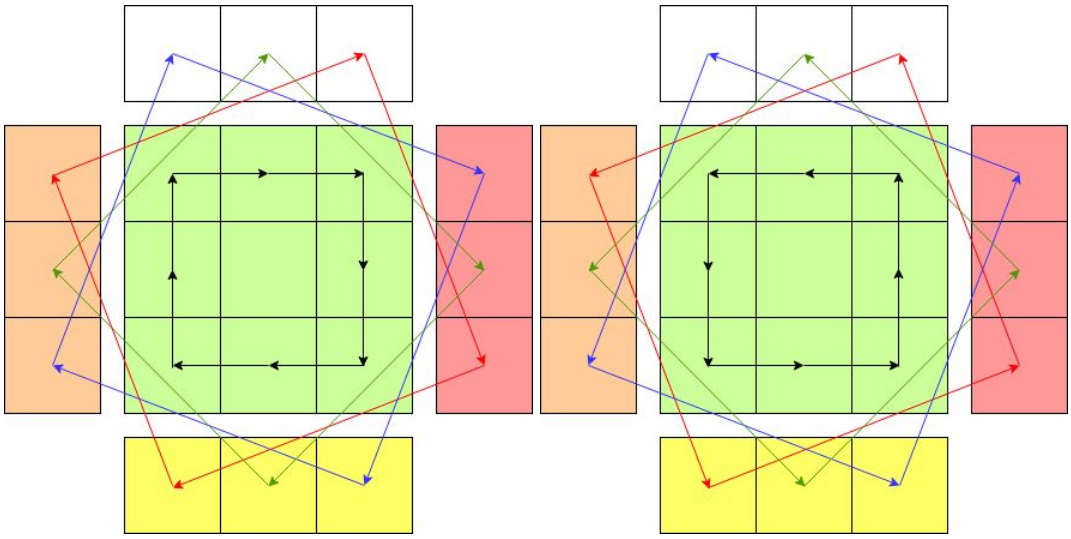

Figure 7: Clockwise and counterclockwise turn logic. The face rotation logic involved the rotation of the given face, along with the movement of the nearby groups of three squares for a total of 21 affected squares as shown.

The cw_turn and ccw_turn registers were checked for activation, and for each, a case statement was used to check for the value of the sw input, and each sw case corresponded to the rotation of a particular face depending on the activation of a single switch out of the first six. For each case, the part of the stickers register corresponding to the face to be rotated was passed to the face_cw or face_ccw function, which performed the rotation as shown by the green face in the above diagram, by returning a new array which was a rotated copy of the input array. Then, a few temporary registers were used to perform the movement of the surrounding groups of 3 squares as described above, using the adjacent squares for each face.

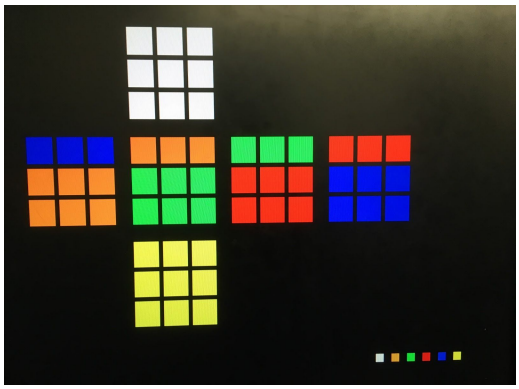

Figure 8: Cube state after clockwise/counterclockwise rotation. After one counterclockwise rotation of white face, or equivalently three clockwise rotations of the white face.

Component #9: Check if Cube is Solved

The state_temp register was used to assign to the state output in order to cycle through the different states for the seven segment display. A state of 1 meant the cube was being scrambled, a state of 2 meant the cube underwent scrambling, and if the state was 2, the cube was checked to see if it matched the default

value for the stickers register, and if so, the state was changed to 3, which would result in “dONE” being displayed on the seven segment display.

Component #10: Assign Colors to Display

The values of both the hc and vc registers are checked to see if they match the range for any square to be displayed on the VGA, and each square on the cube, represented by a part of the stickers array, is assigned to the register curr_color[2:0] if the hc and vc registers are within that square’s range. Then curr_color[2:0] is used for a switch statement, where each value of the curr_color register corresponds to a given color. Each case corresponds to a different color, and so the red_temp[2:0], green_temp[2:0], and blue_temp[1:0] registers are given custom values to display either white, orange, red, green, blue, or yellow by assigning these temp registers to their respective red, green, and blue outputs.

TESTING DOCUMENTATION

Due to the great number of configurations that the cube could take on, most of the testing for this project was done by using the inputs manually and checking the VGA output for the correct behavior. This meant having to check all the possible moves that could be performed on the cube, and checking whether each move resulted in the correct output on the VGA display. As a result, we did not use any simulations for testing in this lab.

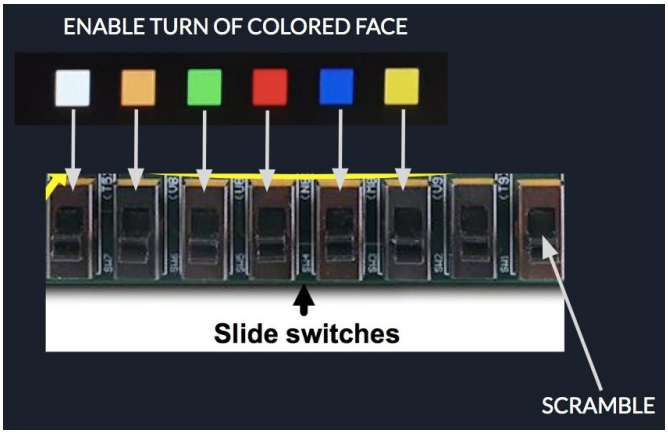

Figure 9: Slide switches functionality. Each of the first six switches corresponds to a face of the cube which has a center color as described by this image. The corresponding face to each switch changes as the cube is shifted on the VGA screen. The last switch is used to randomly scramble the cube.

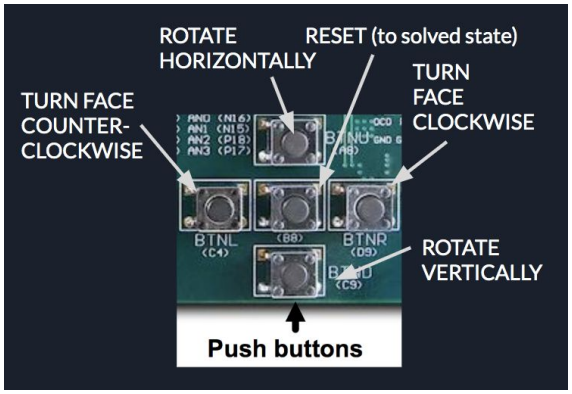

Figure 10: Push buttons functionality. Each push button corresponded to an action that would change the state of the stickers register, and thus change the display of the cube on screen. Two of them for turning, two for shifting entire faces and one for reset.

Together, the switches and buttons allowed us to test the movement and display functionality for the cube. The initial state of the cube was checked every time to see if it was the same configuration as the state that would be reached using the reset button. Upon this confirmation, a few initial moves were tested to check for correct movement of the cube. Each face was rotated clockwise using the left and right buttons, as this would check for all possible moves that could be performed the cube.

Each move was performed subsequently compared to the last, and each move was checked for correctness by checking the face for correct rotation, as well as all the four side faces' nearby squares for correct movement. For example, a rotation of the center face would result in the rotation of that face, along with a shift in the nearest groups of three squares of the top, left, bottom, and center-right faces. There were a total of twelve potential moves, so we tested using this process for all twelve possibilities.

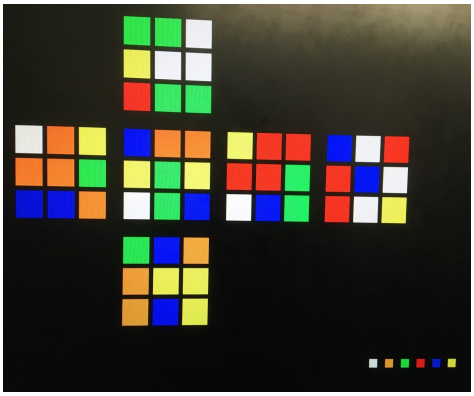

Figure 11: Rubik's cube state following random scramble. Each move in the random scramble was automatically performed using preset move functions, and were checked for correctness.

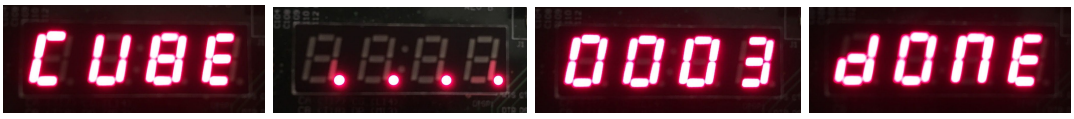

Figure 12: Seven-segment display through its possible states. The seven segment display was made to automatically detect cube state and number of moves. Displays "CUBE" in most cases, "...." when

currently scrambling the cube, the current number of moves made when a random scramble sequence has been ended, and finally, the word “dONE” when a cube is solved after being randomly scrambled.

The random scrambler automatically selected a random possible move to perform, and this functionality was used to check the correctness of moves one at a time by allowing the scrambler to perform a single random move, and checking if the move was legally performed by comparing the state of the cube to the previous state. In addition, the seven segment display would display “dONE” if the state of the cube was detected to be solved, which was used to test the reset button’s functionality.

The cube rotation feature was tested by checking both the top and bottom button presses to result in correct horizontal and vertical rotation, respectively. The horizontal rotation was tested for correct movement of each of the four moved faces in the rotation and the two rotated but stationary faces at the top and bottom, and the vertical rotation was similarly checked for the four moved faces and the two rotated faces on the left and center-right. The legend was also checked for correct updating along with the changed face locations.

CONCLUSION

Since we had complete creative control over this lab, we chose to modularize our code as much as possible, without making our logic flow too difficult to transfer from module to module. We chose to separate some of the more intuitive functionalities, such as having a separate module for generating the different clocks we used, a linear feedback shift register module for picking a random move, a debouncing module for the switches, and a display module for actually translating pixels to a VGA screen. However, we encased most of the main cube logic in one module, which we felt was more efficient, because we stored our cube representation in an array that was 162 bits, and the hardware would have been much slower if this array was being used as inputs and outputs to different modules.

Since this lab was so open-ended, we didn’t make much use of the specifications except when designing our own rubric. However, it was clear what the basic requirements for our demo was, so we had no issue working these into our project. For such a flexible lab, the lab specifications were not lacking.

Throughout the process of designing our Rubik’s cube simulator, we ran into two major hardships. One was dealing with the logic of the Rubik’s cube. It was difficult to translate the three-dimensionality of the cube to a two-dimensional surface, because we had to go through each potential turn and calculate how each “sticker” of the cube would be affected. Additionally, there was just a multitude of cases to handle, which meant that we had to be extremely careful and double check our logic multiple times, because one issue in our cube simulation could cause multiple future wrongdoings for moves made later on.

The other major difficulty we faced was when implementation the random scrambling functionality of the cube. At first, we were using the $random function to try and pick a random move at every clock uptick, but this was giving us issues. The solution ended up being switching to using a linear-feedback shift register, which was efficient because it didn’t use many resources but could run at a pace that didn’t slow our scrambling down at all.

REFERENCES

NERP_demo:
https://www.element14.com/community/thread/23394/l/draw-vga-color-bars-with-fpga-in-verilog
LFSR:
https://vlsicoding.blogspot.com/2014/07/verilog-code-for-4-bit-linear-feedback-shift-register.html
https://www.nandland.com/vhdl/modules/lfsr-linear-feedback-shift-register.html