ECE 385
Spring 2023
Final Project

3x3 Rubik's Cube Simulator

Siddhant Nanavati (ssn5), Pratham Jain (pnj2)
TA section: HW

Introduction

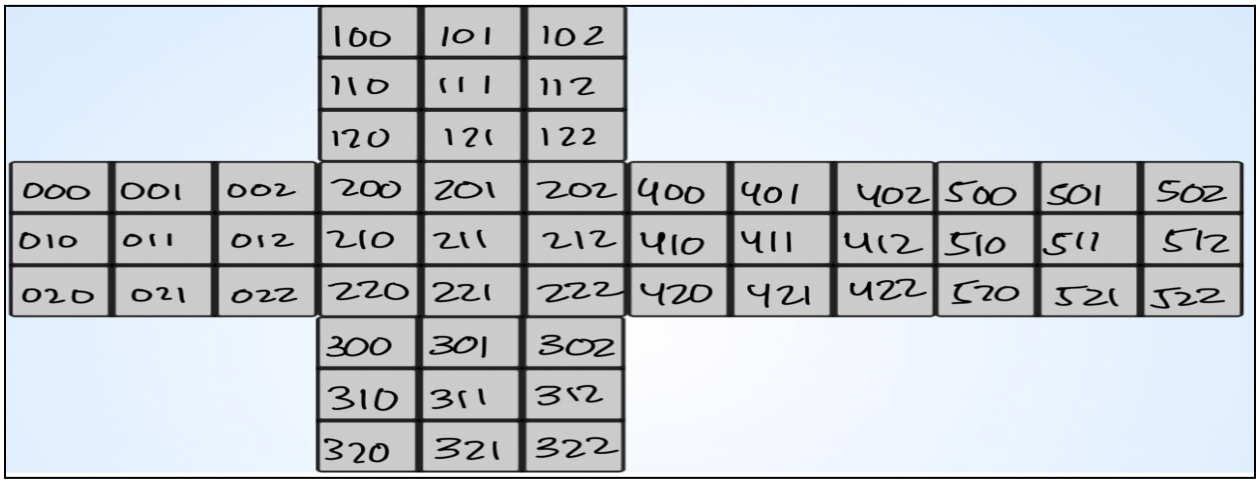
This project involved designing and implementing a 3x3 Rubik's Cube Simulator using the DE 10 lite FPGA board and its associated NIOS-II microcontroller as a System-on-chip. The goal was to create a fully functional Rubik's Cube simulation that could be controlled using a keyboard peripheral. To achieve this, the FPGA board was programmed using SystemVerilog, a hardware description language that allows for the design of digital systems at various levels of abstraction. The NIOS-II processor was used as the main control unit, enabling the Rubik's Cube simulation to be controlled using a keyboard connected to the FPGA board. The Rubik's Cube simulation was designed to rotate the entire cube and perform rotations on specific rows/columns of the tiles on the Cube. The keyboard peripheral was used to replicate various moves such as U, F, D, and R. The arrow keys were used to change the orientation of the cube and rotate the whole cube in the X, Y, Z dimensions, allowing users to view a new face of the cube. The Rubik's Cube simulation was also designed to be displayed on a VGA monitor, displaying a 3D perspective of the cube, as well as all six faces. This was achieved using the VGA driver module, which was written in SystemVerilog and interfaced with the VGA port on the FPGA board.

Overview and Description

Summary

Rubik's Cube Storage

The Rubik's cube can be represented by 54 tiles, with each tile corresponding to a register. The registers store a 3-bit value indicating the color of the tile, and are declared as [2:0] Cube[6][3][3]. The first dimension [6] represents the 6 faces of the cube, while the remaining two dimensions [3][3] correspond to the 9 tiles on each face. The rectangular shape of the cube requires a compact representation of the tiles, making the use of registers an ideal choice. This representation enables the manipulation of the cube's state through software, allowing for the creation of Rubik's cube-solving algorithms.



Colour Palette

Our project involves using six palettes that correspond to the six colors on a Rubik's Cube: orange, white, green, yellow, red, and blue. Each of the six palettes is declared as [11:0] Palette[6] and stores a 12-bit value, which corresponds to the 4-bit RGB value of its corresponding color. The face index determines which palette the tile corresponds to.

Palette Initialization

```
always_comb begin//Palette Initialization
Palette[0] = 12'hf80; //orange
Palette[1] = 12'hfff; //white
Palette[2] = 12'h0c0; //green
Palette[3] = 12'hff0; //yellow
Palette[4] = 12'hf00; //red
Palette[5] = 12'h00f; //blue
end
```

Resetting the Cube

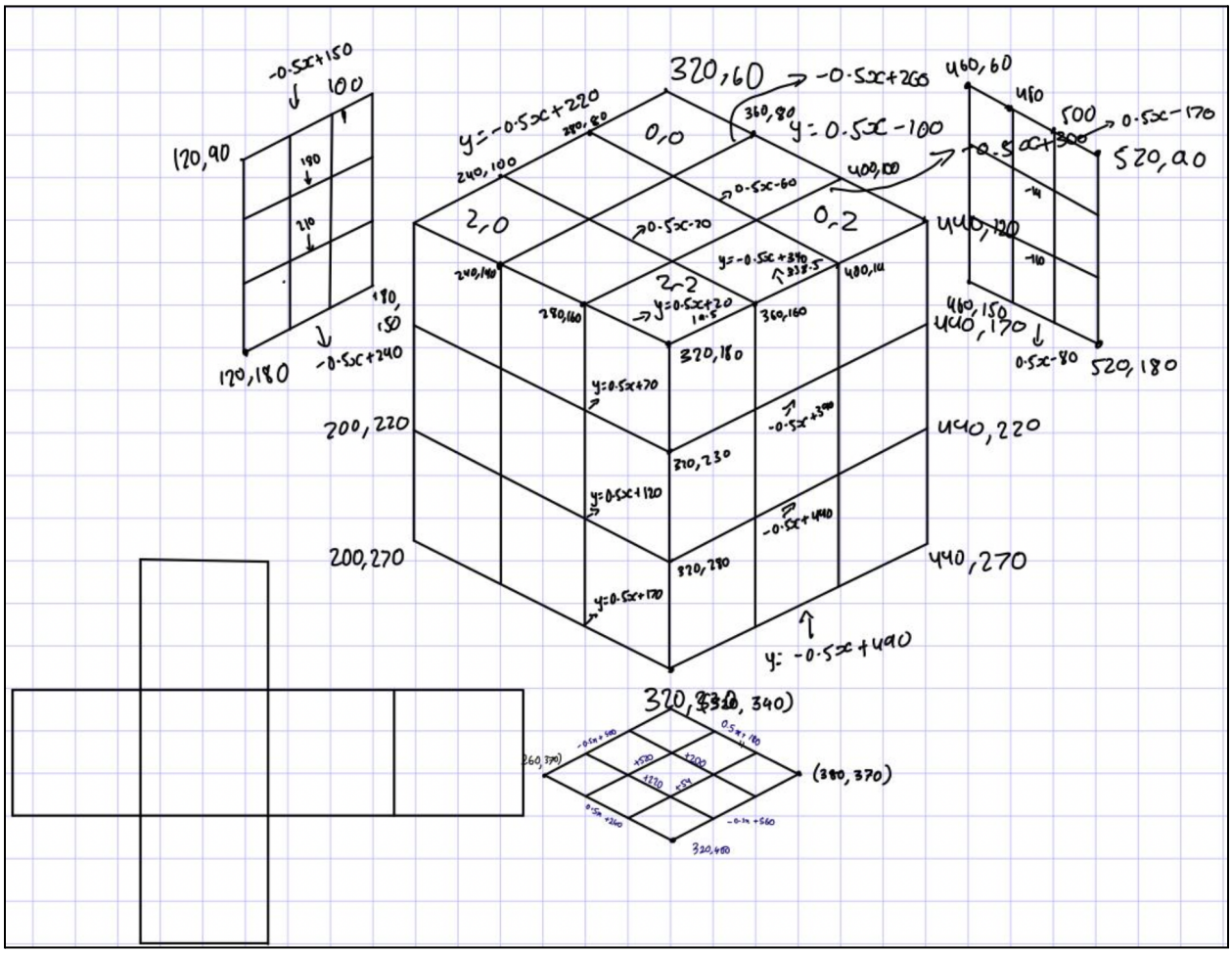
To reset the cube to its solved form, a series of three nested for loops is used. The first loop iterates over each face of the cube, while the second loop iterates over each of the 9 tiles on that face. During this iteration, the register values are set to the corresponding face index value, which is equivalent to the palette index. By setting the register values in this way, the cube is initialized to a state where each face has a distinct color, thus representing a solved state. This process of resetting the cube is an essential step in solving the Rubik's cube, as it establishes a known starting point for the subsequent moves required to solve the cube.

```
if(Reset_state) begin //Cube Initialization
    for(x_ = 0; x_<=5; x_++)begin
        for(y_ = 0; y_<=2; y_++)begin
            for(z_ = 0; z_<=2; z_++)begin
                Cube[x_][y_][z_] <= x_;
                CubeCopy[x_][y_][z_] <= x_;
```

Rendering the Cube

We have generated two visual representations of the cube: a net version and an isometric form with the invisible faces removed. In order to display the cube on a VGA monitor, we mapped each tile of the cube to a specific range of pixels. To achieve this, we determined the line equations and ranges for each tile and calculated the corresponding tile indices using DrawX and DrawY. The rectangular tiles of the net version of the cube allowed for a straightforward mapping of each tile to the VGA monitor pixels. However, the isometric form with the invisible faces rendered away presented a challenge. While the Hamming window has stronger stopband attenuation, its transition band is wider, making it difficult to distinguish between the two peaks. Therefore, we had to carefully calculate the line equations and ranges in order to ensure that the correct tiles were displayed in their corresponding locations on the VGA monitor.

Schematic of Cube Rendering



DrawX and DrawY to Render the Cube

```
//Top Face of main cube
if(DrawY <= (DrawX[9:1]+20) && DrawY>=(-DrawX[9:1]+220)&&DrawY<=(-DrawX[9:1]+340)&&DrawY>=(DrawX[9:1]-100))//Checks within bound of top face
   begin flag = 0;

      if(DrawY == (DrawX[9:1]+20) || DrawY==(-DrawX[9:1]+220)||DrawY==(-DrawX[9:1]+340)||DrawY==(DrawX[9:1]-100)//Checks if line
         || DrawY==(-DrawX[9:1]+260) || DrawY==(-DrawX[9:1]+300) || DrawY == (DrawX[9:1]-20) || DrawY == (DrawX[9:1]-60))
         line = 1;
      else line = 0;
      x = 1;
      if(DrawY < (DrawX[9:1]+20) && DrawY>(-DrawX[9:1]+220)&&DrawY<(-DrawX[9:1]+260)&&DrawY>(DrawX[9:1]-100))
         z=0;
      else if(DrawY < (DrawX[9:1]+20) && DrawY>(-DrawX[9:1]+260)&&DrawY<(-DrawX[9:1]+300)&&DrawY>(DrawX[9:1]-100))
         z=1;
      else z=2;

      if(DrawY < (DrawX[9:1]-60) && DrawY>(-DrawX[9:1]+220)&&DrawY<(-DrawX[9:1]+340)&&DrawY>(DrawX[9:1]-100))
         y=0;
      else if(DrawY < (DrawX[9:1]-20) && DrawY>(-DrawX[9:1]+220)&&DrawY<(-DrawX[9:1]+340)&&DrawY>(DrawX[9:1]-60))
         y=1;
      else y=2;
   end
```

Rotating the Cube

The implemented Rubik's Cube solving algorithm updates the necessary registers depending on the valid keycode pressed for the move to be executed. The algorithm supports a total of 22 moves, each corresponding to different manipulations of the cube such as rotating the entire cube clockwise by 90 degrees or rotating the front face. To ensure that the original cube configuration is not altered during the solving process, a copied version of the cube is used as a reference. Whenever a move is executed, the impacted registers are updated in the copied cube, and the solution is then applied to the original cube. This approach allows for efficient and accurate solving of the cube, while preserving the integrity of the original configuration.

One instance of a Rubik's Cube Move

```
    else if (move_f && keycode_ == 8'h36) begin //M' Push middle tiles downwards
        CubeCopy = Cube;
        //face 2 gets face 1
        Cube[2][0][1] <= CubeCopy[1][0][1];
        Cube[2][1][1] <= CubeCopy[1][1][1];
        Cube[2][2][1] <= CubeCopy[1][2][1];
        //face 3 gets face 2
        Cube[3][0][1] <= CubeCopy[2][0][1];
        Cube[3][1][1] <= CubeCopy[2][1][1];
        Cube[3][2][1] <= CubeCopy[2][2][1];
        //face 5 gets face 3
        Cube[5][0][1] <= CubeCopy[3][2][1];
        Cube[5][1][1] <= CubeCopy[3][1][1];
        Cube[5][2][1] <= CubeCopy[3][0][1];
        //face 1 gets face 5
        Cube[1][0][1] <= CubeCopy[5][2][1];
        Cube[1][1][1] <= CubeCopy[5][1][1];
        Cube[1][2][1] <= CubeCopy[5][0][1];
```

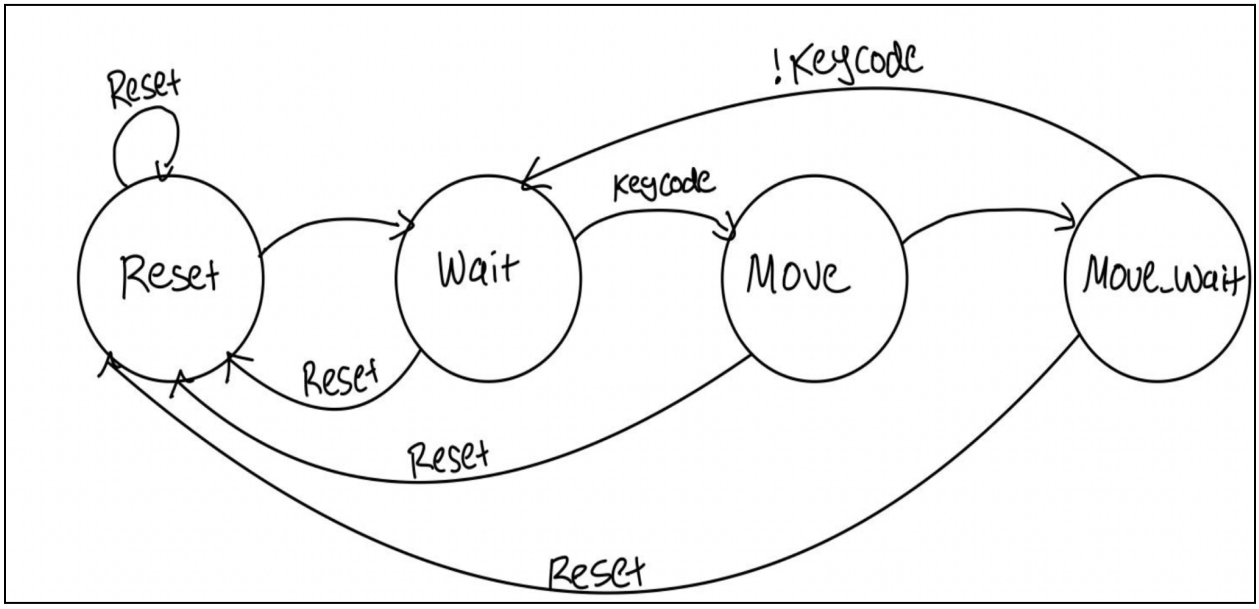
Pixel Values Settings

First, the pixel value is determined based on whether the blank is high (active) or not, and whether the corresponding DrawX and DrawY values indicate a line should be drawn. If a line is to be drawn, the pixel value is set to black. Otherwise, a variable flag is set to 0 if DrawX and DrawY do not fall within the cube region, and 1 if they do. If the flag is 0, the pixel value is set to a light blue background to indicate that no line will be drawn. If the flag is 1, the RGB value of the pixel is determined by a palette that is selected based on the x, y, and z indices that were decided during the cube rendering process. This allows for greater flexibility in color choice and helps to distinguish the pixel from the background.

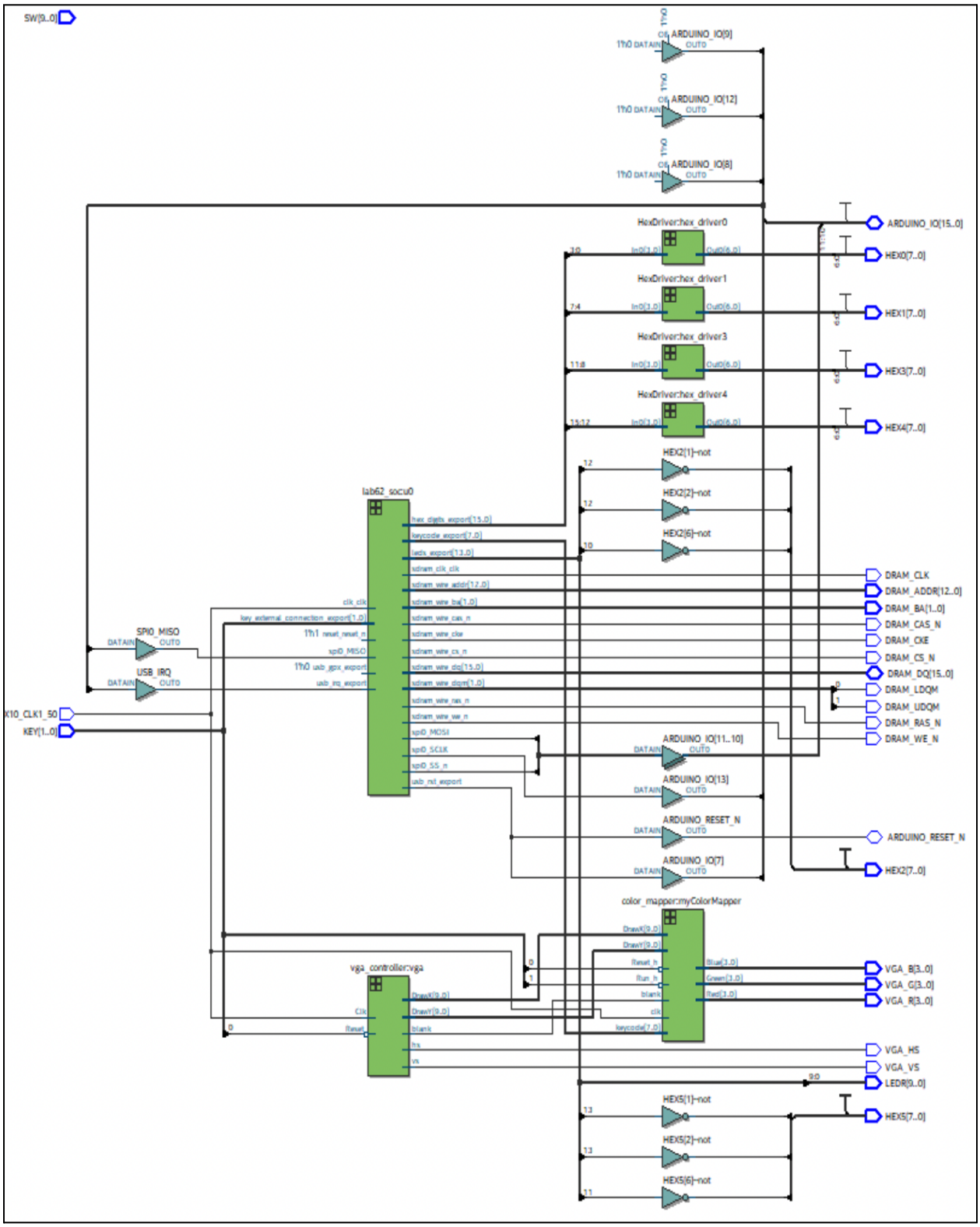
Moore Finite State Machine

The 3x3 Rubik's cube simulator uses a Moore Finite State Machine to execute cube manipulations and reset operations. Pressing the reset key (enter key) transitions all states to reset and sets the Reset_state input to high, which is then passed to the color_mapper for reset operation. The FSM moves unconditionally to the wait state after resetting. It remains in the wait state until a key is pressed, at which point it transitions to the Move state. The Move state signals that a cube manipulation is to be executed by setting the Move input to high and passing the keycode as an input. The move state then transitions unconditionally to the Move_wait state. This state waits until the user releases the key (i.e., keycode = 0) before transitioning back to the Wait state to prevent the cube from continuously rotating if the key is not released. Finally, the FSM enters the wait state again and waits for a new keycode to be pressed.

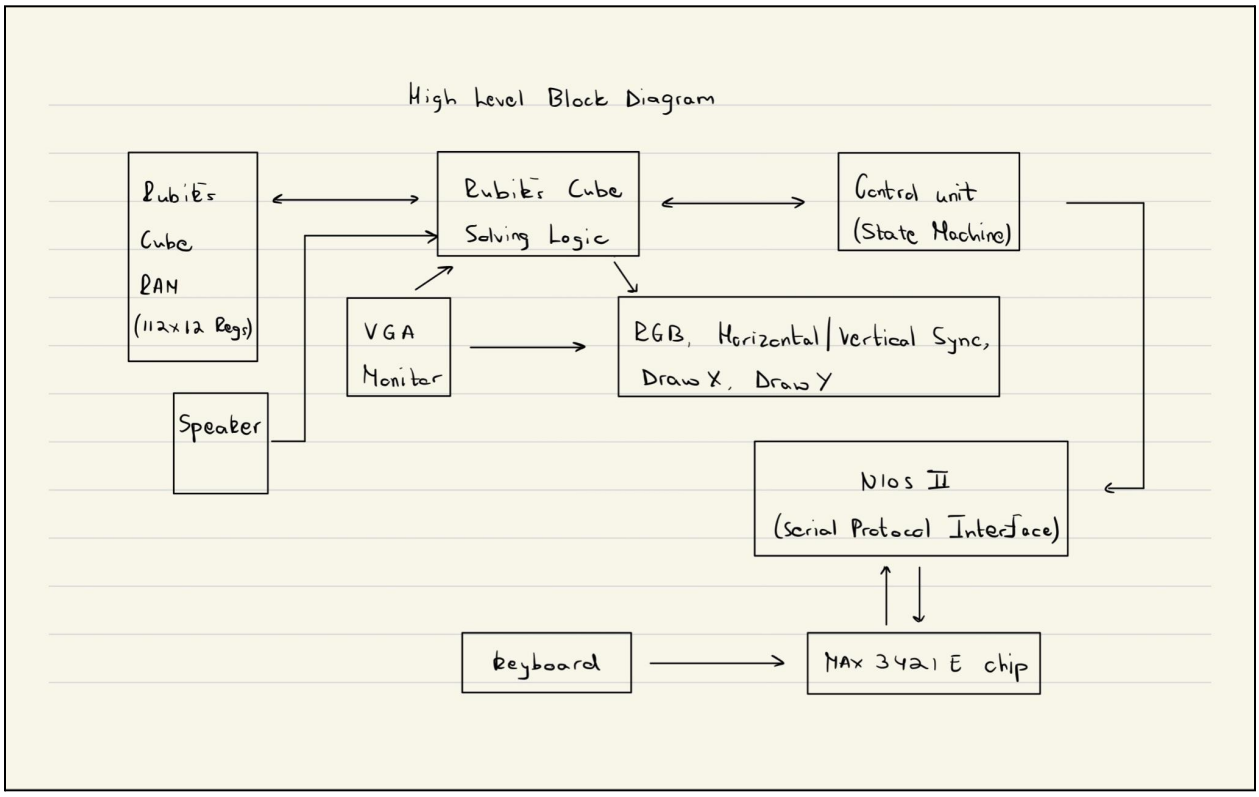
Moore Finite State Machine



Top Level Block Diagram



High Level Block Diagram



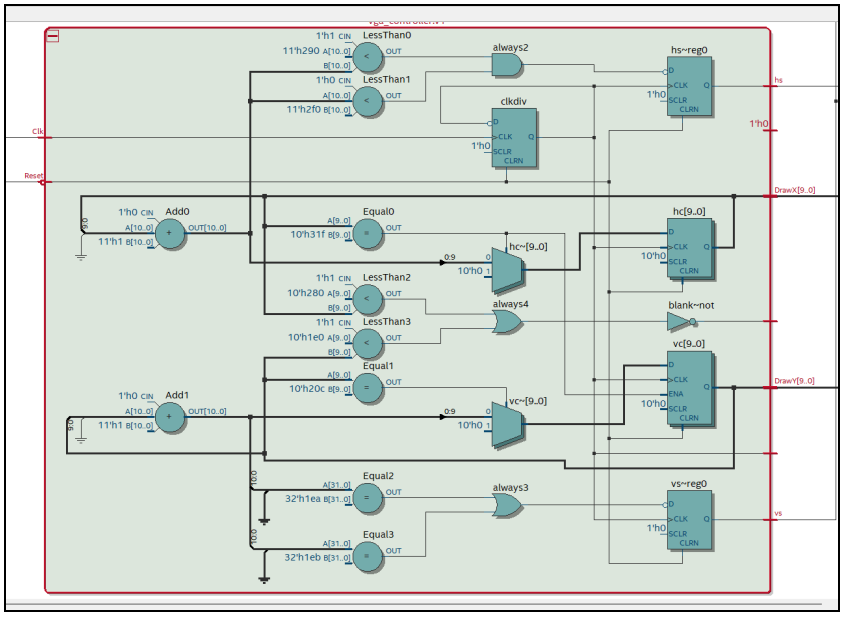
SystemVerilog Module Descriptions:
Module: VGA_controller.sv

Inputs: Clk, Reset

Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, DrawY

Description: This module utilizes the always_ff block to generate VGA signals by producing a 25MHz clock from the supplied 50MHz clock. If reset is activated, the ball is repositioned at the center. Otherwise, it tracks the coordinates of each pixel (X and Y coordinates) and generates horizontal and vertical sync signals to ensure accurate movement of the ball. Additionally, it manages blanking logic to establish the black color value on the VGA.

Purpose: The function of this module is to determine the direction and movement of the ball on the screen based on input received from the keyboard. It also determines the ball's movement when it reaches either end of the screen. This module is responsible for synchronizing the coordinates and managing blanking logic as well.



Module: Rubiks_Cube.sv
Inputs: clk, blank, Reset_h, Run_h, [9:0] DrawX, DrawY, [7:0] keycode

Outputs: [3:0] Red, Green, Blue

Description: The primary objective of this module is to initialize registers that correspond to cube tiles and establish a color palette. It updates the register values according to the intended move and generates RGB values based on DrawX and DrawY to render the cube.

Purpose: The function of this module is to generate the Rubik's cube, render it, and execute various cube manipulations and functionalities.

Module: Control.sv
Inputs: clk, Reset, Run, [7:0] keycode

Outputs: move_f, Reset_state, [7:0] keycode_

Description: This module generates a Finite State Machine (FSM) that includes four distinct states. The first state, known as the reset state, sends an input to the color mapper indicating that a reset operation should be performed. The Wait state waits for the user to input a keycode. The Move state sets control signals to indicate that a move is to be executed, while the move_wait state waits for the user to release the keycode.

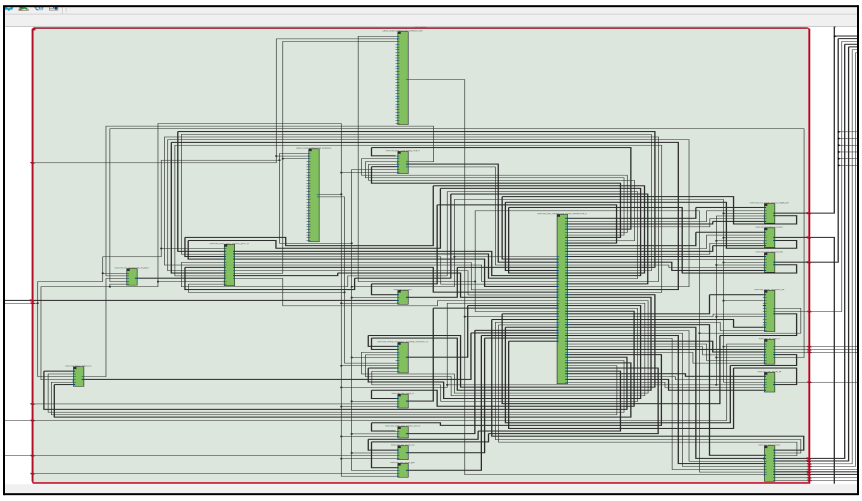
Purpose: The role of this module is to function as the Finite State Machine (FSM) for our project, generating a range of control signals based on the current state.

Module: lab62soc.v:
Inputs: clk_clk, reset_reset_n, usb_irq_export, usb_gpx_export, [1:0] key_external_connection_export

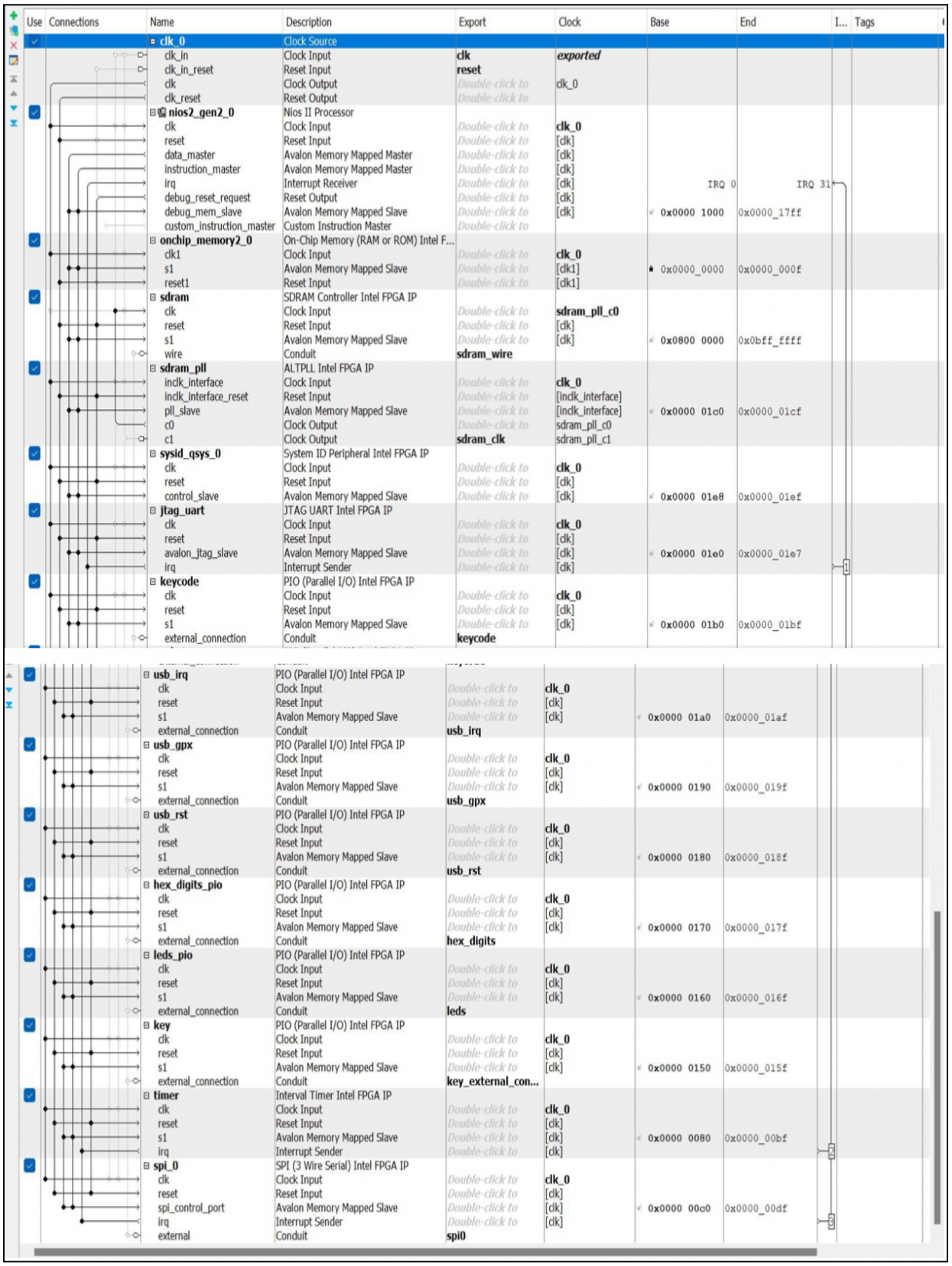
Outputs: [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, [7:0] sdram_clk_clk, [12:0] sdram_wire_addr, [7:0] keycode_export, usb_rst_export, [15:0] hex_digits_export, [13:0] leds_export,spi_0_MISO, spi_0_MOSI, spi_0_SCLK, spi_0_SS_n
Inouts: [15:0] sdram_wire_dq

Description: This is the resulting .sv file that is produced by the Platform Designer tool during the first week. It includes the creation of our peripherals such as the NIOS-II, On-Chip Memory, SDRAM, I/O Blocks, SPI blocks, USB interface, JTAG_UART, Timer, and USB Interrupts. The NIOS-II serves as our CPU, while the SDRAM acts as the primary memory where our instructions are stored. The SDRAM_pll module is an additional component that manages time delays during instruction transfers from the FPGA to the SDRAM, resulting in a delay of 1ns. The PIOs are utilized for displaying values on LEDs or Hexes and receiving data from the switches. Additional components include the SPIO block, which facilitates interaction between the USB Shield and the NIOS-II, allowing for reading and writing to the registers available on the MAX3421E Chip. The JTAG UART component provides debugging capabilities through text transfer functionality. The USB Interrupt supports interrupt functionality when keyboard keys are pressed, and the PIOs are used to display content on the HexDrivers.

Purpose: This file serves as the top-level component for our Rubik's cube, executing the descriptions that we created using the GUI within our Platform Designer.

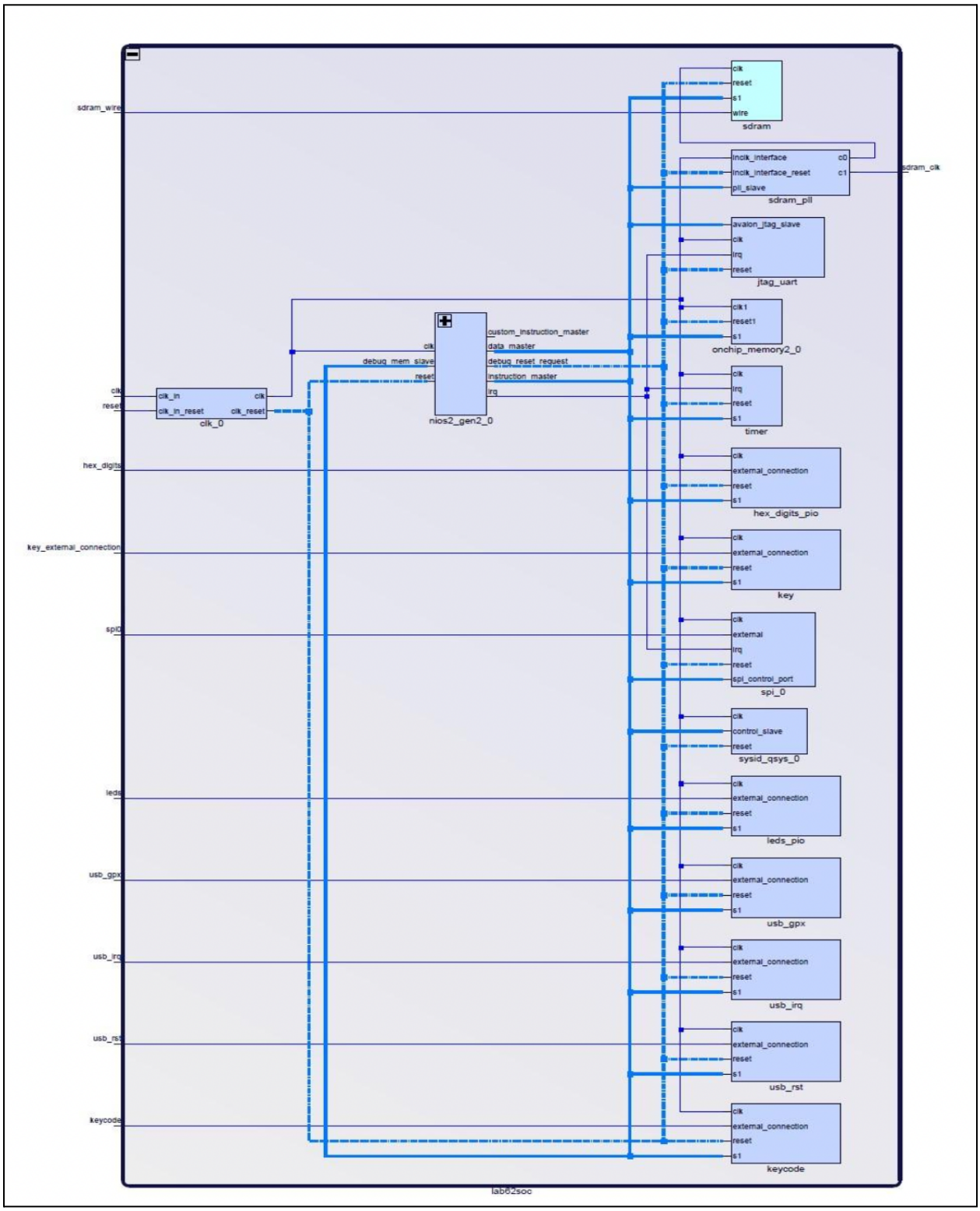


System Level Block Diagram



Use	Connections	Name	Description	Export	Clock	Base	End	I...	Tags
✓		**clk_0**	Clock Source						
		clk_in	Clock Input	**clk**	***exported***				
		clk_in_reset	Reset Input	**reset**					
		clk	Clock Output	*Double-click to*	clk_0				
		clk_reset	Reset Output	*Double-click to*					
✓		**nios2_gen2_0**	Nios II Processor						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		data_master	Avalon Memory Mapped Master	*Double-click to*	[clk]				
		instruction_master	Avalon Memory Mapped Master	*Double-click to*	[clk]				
		irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 31		
		debug_reset_request	Reset Output	*Double-click to*	[clk]				
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 1000	0x0000_17ff		
		custom_instruction_master	Custom Instruction Master	*Double-click to*					
✓		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) Intel F...						
		clk1	Clock Input	*Double-click to*	**clk_0**				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk1]	0x0000_0000	0x0000_000f		
		reset1	Reset Input	*Double-click to*	[clk1]				
✓		**sdram**	SDRAM Controller Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**sdram_pll_c0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0800 0000	0x0bff_ffff		
		wire	Conduit	**sdram_wire**					
✓		**sdram_pll**	ALTPLL Intel FPGA IP						
		inclk_interface	Clock Input	*Double-click to*	**clk_0**				
		inclk_interface_reset	Reset Input	*Double-click to*	[inclk_interface]				
		pll_slave	Avalon Memory Mapped Slave	*Double-click to*	[inclk_interface]	0x0000 01c0	0x0000_01cf		
		c0	Clock Output	*Double-click to*	sdram_pll_c0				
		c1	Clock Output	**sdram_clk**	sdram_pll_c1				
✓		**sysid_qsys_0**	System ID Peripheral Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		control_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 01e8	0x0000_01ef		
✓		**jtag_uart**	JTAG UART Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 01e0	0x0000_01e7		
		irq	Interrupt Sender	*Double-click to*	[clk]			1	
✓		**keycode**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 01b0	0x0000_01bf		
		external_connection	Conduit	**keycode**					
✓		**usb_irq**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 01a0	0x0000_01af		
		external_connection	Conduit	**usb_irq**					
✓		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 0190	0x0000_019f		
		external_connection	Conduit	**usb_gpx**					
✓		**usb_rst**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 0180	0x0000_018f		
		external_connection	Conduit	**usb_rst**					
✓		**hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 0170	0x0000_017f		
		external_connection	Conduit	**hex_digits**					
✓		**leds_pio**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 0160	0x0000_016f		
		external_connection	Conduit	**leds**					
✓		**key**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 0150	0x0000_015f		
		external_connection	Conduit	**key_external_con...**					
✓		**timer**	Interval Timer Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 0080	0x0000_00bf		
		irq	Interrupt Sender	*Double-click to*	[clk]			2	
✓		**spi_0**	SPI (3 Wire Serial) Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		spi_control_port	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x0000 00c0	0x0000_00df		
		irq	Interrupt Sender	*Double-click to*	[clk]			3	
		external	Conduit	**spi0**					

Platform Designer Block Diagram



Component-wise Functionality

General Components

- Clk_0: This serves as the main clock for the FPGA, MAX3421E, SDRAM.
- Sysid_qsys_0: This component verifies the correct system ID to ensure that the software runs accurately on the hardware.
- [component]_pio: These components provide the address for the PIO displays that are utilized by various files in the software.
- Usb_[fn]: These components manage the USB functions, including the keyboard circuit, reset, and interrupt.
- The system includes the Nios-II Processor (nios2_gen2_0), a 32-bit CPU that is primarily controlled by C. The processor is responsible for controlling peripherals that handle data and do not require fast processing times, and only need to transmit data.
- The system has two keys: key_1 is used to accumulate LEDs and switches, while key_0 indicates that the value of the LEDs needs to be reset.
- We use SPI (System Peripheral Interface) through spi_0 to interact with USB peripherals like the keyboard and mouse when needed.
- The system also includes a JTAG UART, which enables character movement between the computer and the FPGA. This allows for text transfer without slowing down the CPU for tasks that do not require high processing power.

Software Components

- BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)

```
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    BYTE newdata[nbytes+1];
    newdata[0] = reg+2;
    for (int i=0; i<nbytes; i++) {
        newdata[i+1] = data[i];
    }
    int  x = alt_avalon_spi_command(SPI_0_BASE, 0, nbytes+1, newdata, 0, 0, 0);
    if (x < 0) {
        printf ("Error \n");
    }
    return (data + nbytes);
```

This particular function aims to write a data array that spans a length of nbytes, which is dictated by the data pointer. The first step taken is to create a fresh array consisting of the data to be written. To begin the actual writing process, the register address reg + 2 is written first, with N bytes of data to follow. Upon completion, the SPI command is executed, and the new array is passed along with a setting to write. Once the execution is complete, the function proceeds to return a pointer to the memory address beyond the last write function executed.

- void MAXreg_wr(BYTE reg, BYTE val)

```
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:

    //write reg + 2 via SPI
    //write val via SPI
    alt_u8 wrdata[2] = {reg+2, val};
    int  x = alt_avalon_spi_command(SPI_0_BASE, 0, 2, wrdata, 0, 0, 0);
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    if (x < 0) {
        printf ("Error \n");
    }
```

This particular function serves the purpose of writing a single-byte value 'val' to a specific register in the MAX3421E device through the Serial Peripheral Interface (SPI) protocol. The SPI protocol is a common synchronous data transfer protocol used for communication between microcontrollers, microprocessors, and other digital devices. To write the value to the designated register, the function first sets the address of the register by writing the value of the register address plus two, which specifies the address of the register in the MAX3421E. After that, the actual value 'val' is written to the same register. The SPI function is then invoked, passing the 'writedata' to perform the write operation. It is important to note that the write operation here is unidirectional, meaning that no data is transferred from the MAX3421E to the master device during this process. This function, therefore, plays a crucial role in the efficient and accurate communication between the MAX3421E and the master device, ensuring that the correct values are written to the appropriate registers, allowing for proper configuration and operation of the device.

- BYTE MAXreg_rd(BYTE reg)

```
BYTE MAXreg_rd(BYTE reg) {
    //psuedocode:

    //write reg via SPI
    //read val via SPI
    alt_u8 val;
    alt_u8 wrdata = reg;
        //read return code from SPI peripheral (see Intel documentation)
    int  x = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &wrdata, 1, &val, 0);
        //if return code < 0 print an error
        if (x < 0) {
            printf ("Error \n");
        }
```

This particular function has been designed to perform the task of reading a register value and returning it. The approach utilized to accomplish this involves invoking the SPI (Serial Peripheral Interface) command, which enables communication between different electronic devices. The first step in the process involves writing the register address, which is where the desired data is stored. Once the register address has been specified, the function proceeds to read a 1 BYTE value that corresponds to the stored data. The data value is then returned to the caller, thereby completing the process of reading and retrieving data from the specified register. This function serves as a vital component in the broader system's architecture, facilitating communication between different components and ensuring the seamless transfer of data.

- BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)

```
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:

    //write reg via SPI
    alt_u8 wrdata = reg;
    //read data[n] from SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    int  x = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &wrdata, nbytes, data, 0);
    //if return code < 0 print an error

    if (x < 0) {
            printf ("Error \n");
        }
    //return (data + nbytes);
    return (data + nbytes);
```

The purpose of this particular function is to read a specified number of bytes of data from a device and store it in an array, which can then be accessed for further processing or analysis. Specifically, it invokes a specific command for communicating with the device, where the starting register address is written first before reading the specified number of bytes and storing them in the designated 'data' array. This function has the ability to return a pointer to the memory location where the last data was written.

Design Statistics for the 3x3 Rubik’s Cube Simulator

Criteria	Statistic
DSP blocks	0
Look Up Tables (LUTs)	3722
Flip-Flops	2501
Memory (BRAM)	50532
Dynamic Power	86.99 mW
Static Power	97.02 mW
Total Power	225.19 mW
Frequency	78.34 MHz

Bugs and Challenges Encountered

Designing the 3x3 Rubik's cube simulator on Quartus Prime with SystemVerilog, C, and the DE-10 Lite FPGA board presented us with a number of challenges. One of the main challenges is dealing with the complexity of the Rubik's cube itself. With its 26 pieces and so many possible configurations, simulating the Rubik's cube was computationally intensive and required careful optimization to ensure that the simulation could run in real-time on an FPGA. Finally, there were a number of potential bugs that arose when designing the Rubik's cube simulator. These can include issues with data transfer and synchronization between different modules, as well as errors in the logic of the simulation itself. To minimize these issues, it was important to carefully test the simulator at each stage of the design process, and to implement thorough error-checking and debugging features to identify and address issues as they arise.

Conclusion

In summary, we have developed a 3x3 Rubik's Cube simulator using Quartus Prime, SystemVerilog, C, the NIOS II processor, and a DE-10 Lite FPGA board. This simulator can be easily controlled through a keyboard and the graphics can be displayed on a VGA monitor. The simulation accurately reflects all the moves and manipulations of a physical Rubik's Cube, and offers additional features such as reset and undo. We have successfully rendered all possible cube rotations, making it a fully functional Rubik's Cube simulator. In addition, we used various design analysis and optimization tools to obtain a complete understanding of the statistics related to our processor's design. Through the use of these tools, we were able to investigate the implementation of digital

hardware design in practical situations. This analysis allowed us to scrutinize the design of the controller and carry out optimization adjustments to achieve optimal performance. Despite many challenges, designing the Rubik's cube simulator was a rewarding experience, providing a unique opportunity to combine hardware design, algorithms, and real-time simulation in a single project.