Rubik's Cube Simulator

Siddarth Iyer, Geitanksha Tandon

1. Introduction:

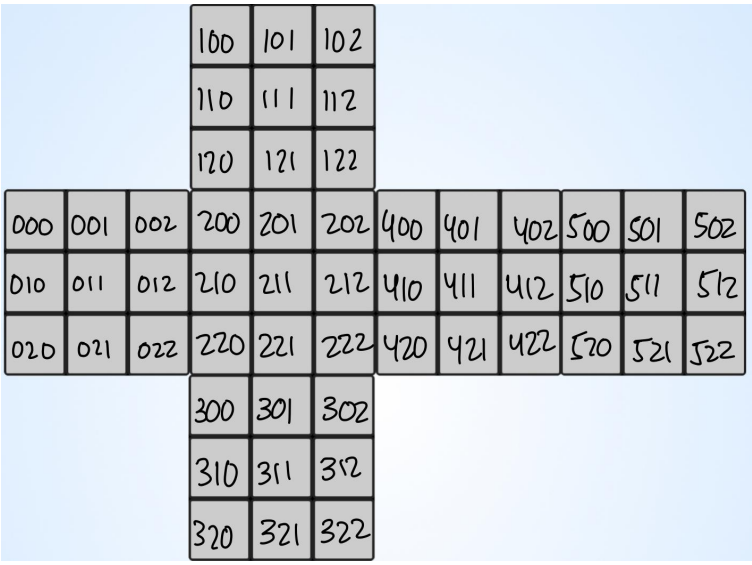
In this project, we have designed and implemented a 3x3 Rubik's Cube Simulator using the FPGA and its associated NIOS-II microcontroller as a System-on-chip. It uses a keyboard peripheral to act as input to the Rubik's cube to rotate the entire cube and perform rotations on specific rows/columns of the tiles on the Cube. Using keyboard clicks, the Rubik's cube replicates various moves such as U, F, D, and R. The arrow keys are used to change the orientation of the cube / rotate the whole cube in the X, Y, Z dimensions, to view a new face of the cube. We employ the use of a VGA to display this Rubik's cube and its simulation in multiple forms. The VGA monitor displays a 3d perspective of the cube, as well as all six faces.
Other features include undo and reset.

---

2. Written Description

i. Rubik's cube storage.

A register represents each tile on the Rubik's cube. Hence, the cube is represented by 54 tiles which translate to registers (6 faces, 9 tiles per face). Each register stores a 3-bit value corresponding to a palette index which describes the color of the tile. The 54 registers are declared as [2:0] Cube[6][3][3]. The first dimension [6] corresponds to one of 6 faces, while the remaining two [3][3] correspond to the 9 tiles on the face.



---

ii. Palette

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

```
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
```

---

iii. Reset cube

The cube is reset/initialized to its solved form via 3 nested for loops. It loops through each face and each of the 9 tiles and sets the register values to the face index, which is equivalent to the palette index.

```
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_;

            end
        end
    end
end
```

---

iv. Cube Rotation

When a valid keycode is pressed, depending on the move to be implemented, the necessary registers are updated. A total of 22 moves are implemented corresponding to various cube manipulations such as rotating the full cube clockwise by 90 degrees, rotating the front face etc. We make use of a copied version of the cube, and update the impacted registers, using the copied cube as a reference.

```
    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];
end
```

---

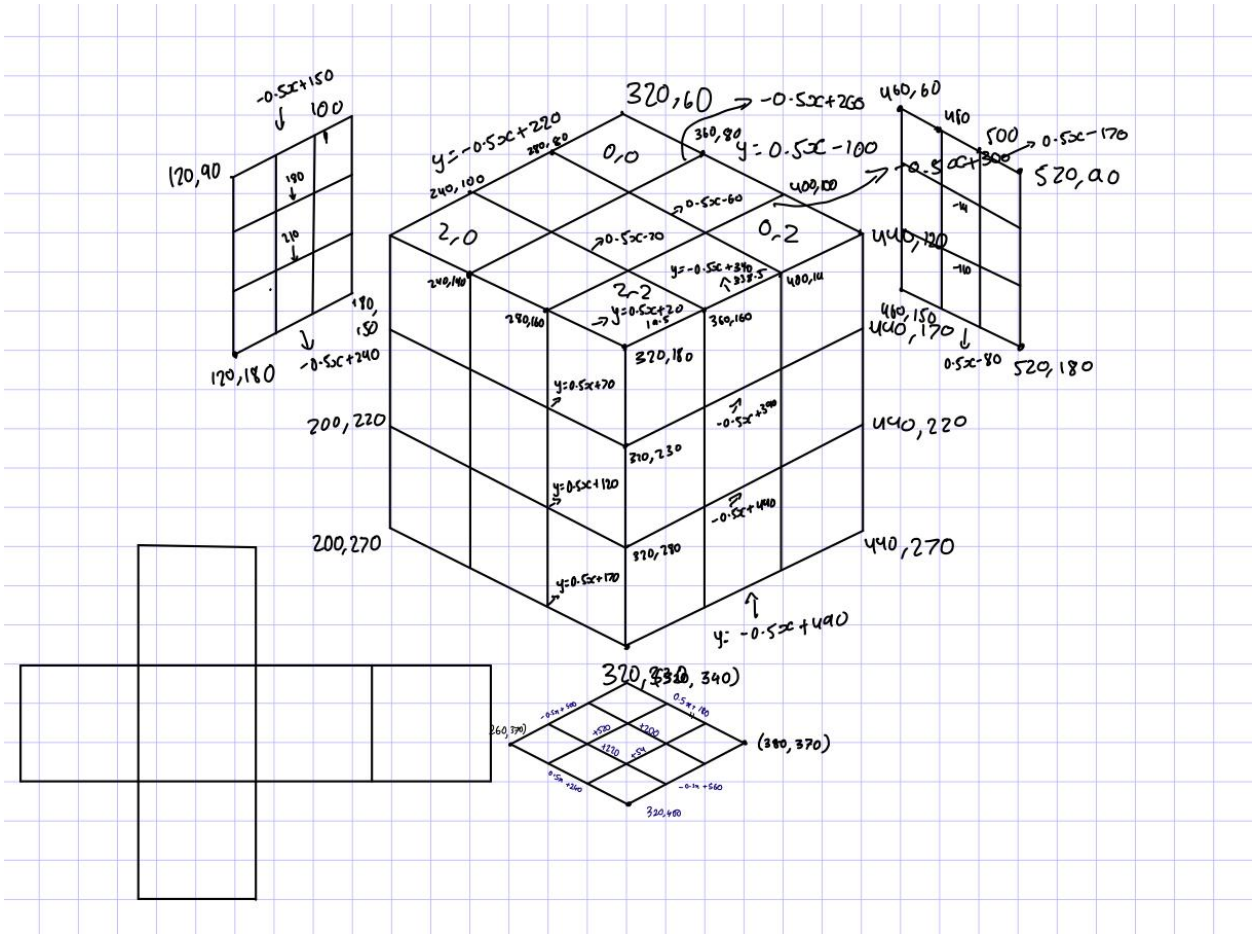
v. Cube Rendering

We have rendered two forms of the cube. A net version of the cube, and an isometric form with the invisible faces rendered away as shown below. Each tile of the cube maps to a certain range of pixels on the VGA monitor. We calculated the various line equations and ranges in order to determine the tile indices using DrawX and DrawY, in order to set the corresponding pixel values.

```
//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
```



---

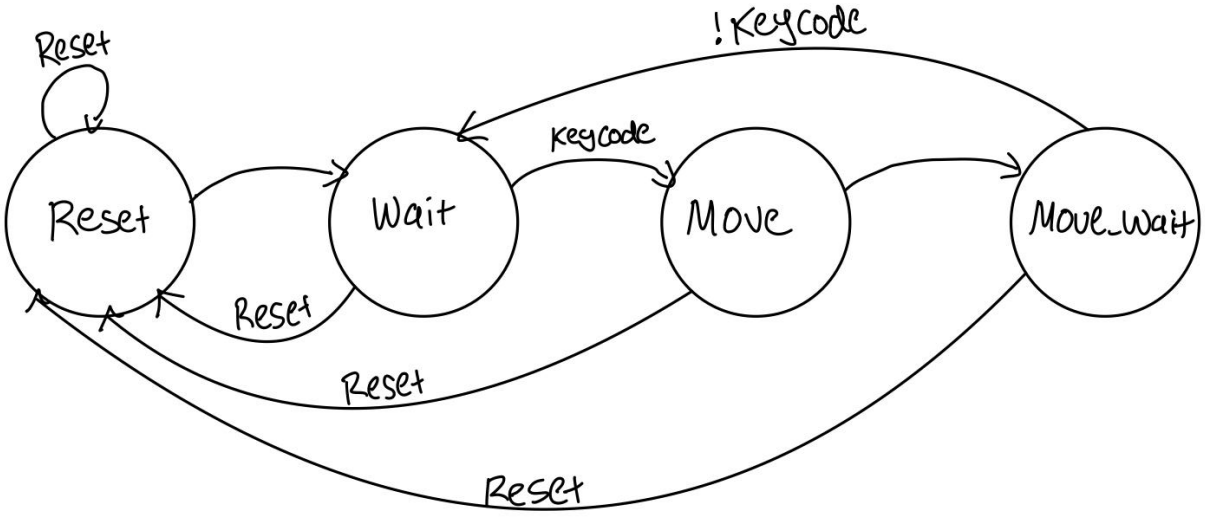
vi. Setting Pixel Values

Firstly, the Pixel value is set to black if blank is not high(active low) or if DrawX, DrawY values correspond to a region where a line is to be drawn. A variable flag is set to 1 if DrawX, DrawY is within the cube region and 0 otherwise. If flag is 0 the pixel value is set to a light blue background. Else, if flag is one, the RGB value is set based on a palette determined by the x,y,z indices decided during the cube rendering.

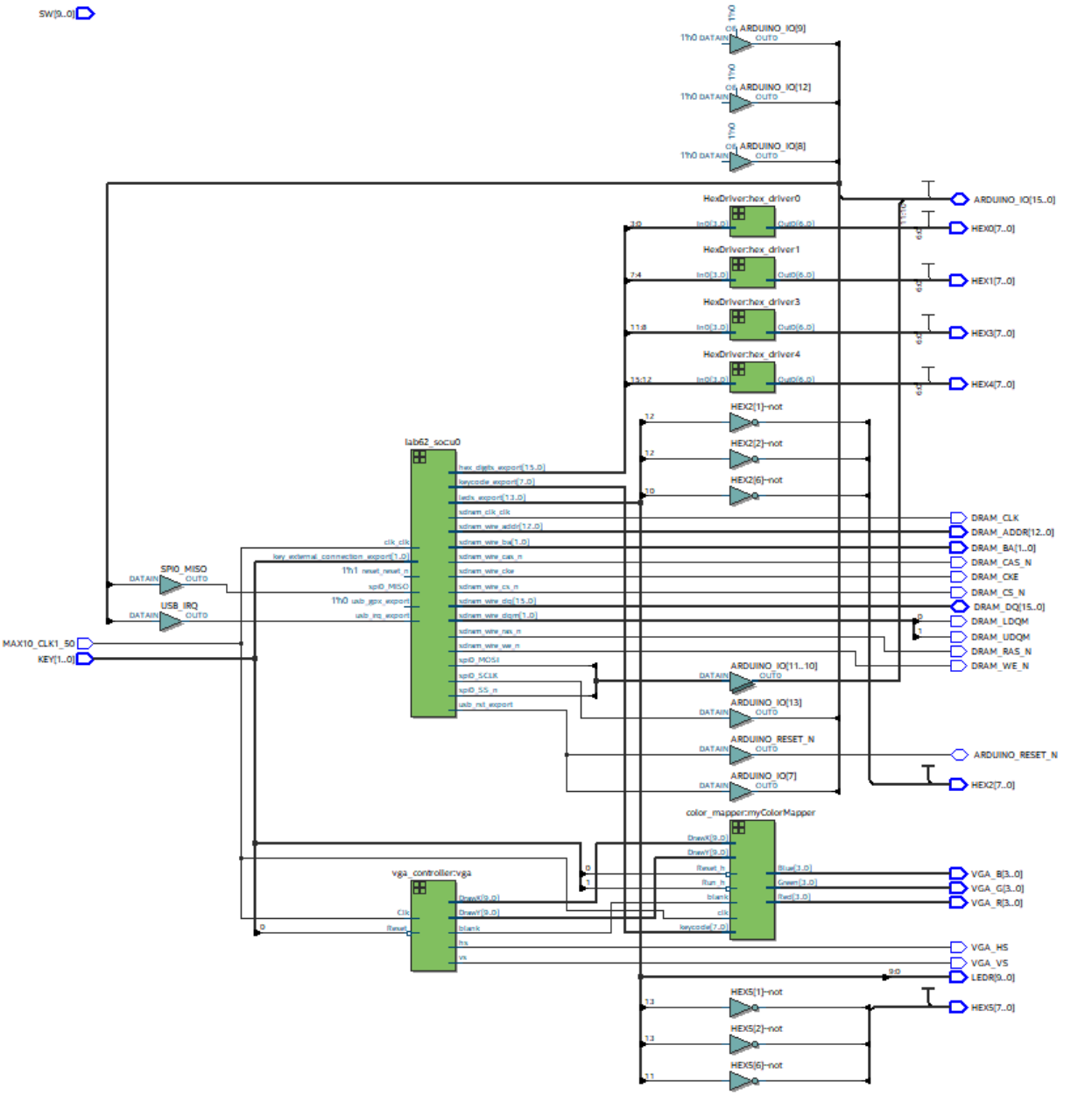
---

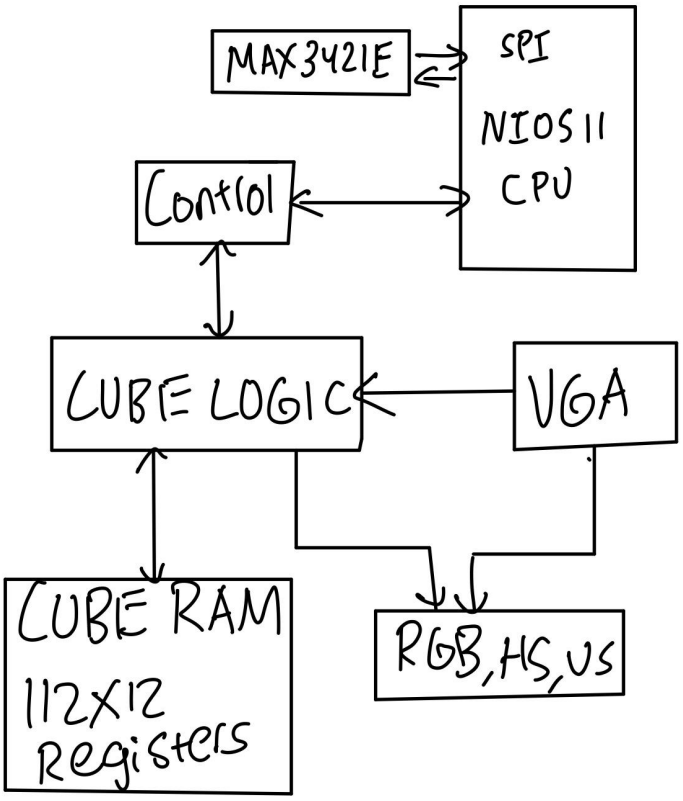
vii. Finite State Machine

Our project uses a simple FSM to reset the cube and implement various moves. When the key corresponding to reset is pressed(enter key), all states transition to reset. This sets the Reset_state input to high, which is then passed to the color_mapper to implement the reset operation. The reset state unconditionally transitions to the wait state. The FSM remains in the Wait state until a key is pressed after which it transitions to the Move state. The Move state sets Move input to high indicating a cube manipulation is to be executed and also passes the keycode as an input. The move state unconditionally transitions to the Move_wait state, which waits until the user releases the key i.e keycode = 0, before transitioning back to the Wait state. This is done to prevent the cube from continually rotating if the key is not de-pressed. The FSM enters the wait state and waits until a new keycode is pressed.



3. Block Diagram




MAX3421E
SPI
NIOS II
CPU
Control
CUBE LOGIC
VGA
CUBE RAM
112X12
Registers
RGB, HS, VS

4. Module Description:

Module: VGA_controller.sv

Inputs: Clk, Reset

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

Description: This module generates the VGA signals by creating a 25MHz clock from the 50MHz clock given to using the always_ff block. If reset is clicked, the ball is moved to the center. Otherwise, it helps keep track of the coordinates of the pixel (X, Y coordinates) and describes the horizontal and vertical sync signals to ensure appropriate movement of the ball. It handles the blanking logic which helps set the value of the black color on the VGA.

Purpose: This module decides the movement and the direction of the ball on the screen using the input given by the keyboard. It decides the movement of the ball when it reaches one end of the screen. It handles the sync of the coordinates, and handles the blanking logic.

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: This module initializes registers corresponding to cube tiles and creates a color palette. It updates the values of the register based on the move to be implemented, and renders the cube by outputting RGB values based on DrawX, DrawY

Purpose: This module creates the rubiks, renders it, and implements various cube manipulations and cube functionalities

---

Module: Control.sv

Inputs: clk, Reset, Run, [7:0] keycode,

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

Description: This module creates an FSM consisting of 4 states. The reset state sends an input to the color mapper indicating a reset is to be executed, the Wait state waits till the user enters a keycode, the Move state sets control signals indicating a move is to be executed, and the move_wait state waits till the user releases the keycode.

Purpose: This module serves as the FSM for our project which outputs various control signals depending 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 file is the resultant .sv file that is generated by the Platform designer tool for the first week. It includes the creation of our Peripherals - NIOS-II, On-Chip Memory, SDRAM, I/O Blocks and additional SPI blocks, USB interface, JTAG_UART, Timer, USB Interrupts.. The NIOS-II is our CPU, the SDRAM is the main memory where our instructions are stored, the SDRAM_pll is the extra module that handles the time delay while instructions transfer from the FPGA to the SDRAM (delay of 1ns), the PIOs are used to display the values on LEDs / Hexes and receive data from the switches.

The newer components are the SPIO block (Serial Port Interface) which enables interaction between the USB Shield and the NIOS-II and hence enables reading and writing to the registers available on the MAX3421E Chip; the JTAG UART component allows debugging by performing the text transfer functionality; the USB Interrupt helps interrupt functionality when the keyboard keys are pressed, and the PIOs are used for displaying content on the HexDrivers.

Purpose: This file is the top-level of our rubiks cube. It executes the descriptions we made via GUI in our Platform Designer.

Components in the execution of the project

- Clk_0: Main clock for the FPGA, MAX3421E, SDRAM.
- Sysid_qsys_0: returns whether the system ID is correct so it correctly executes the software onto the hardware.
- [component]_pio: the address for the PIO displays is given by these which all the files refer to when trying to use them in our software.
- Usb_[fn]: handles the usb functions like the circuit from the keyboard, reset, and the interrupt.

Component	Name	Description
Nios-II Processor	nios2_gen2_0	This is our 32-bit CPU - the primary controller that is controlled by C, a high-level programming language. It is used to control the peripherals that primarily handle data and do not require fast processing times, and only need to transmit data.
Keys	Key_1, key_0	key_1 is the signal to accumulate LEDs and switches. key_0 indicated that the value of the LEDs needed to be reset.
SPI (W2)	spi_0	The SPI (System Peripheral Interface) lets us interact with USB Peripherals like the Keyboard, and mouse if necessary.
JTAG UART (W2)	jtag_uart	This allows character movement between the Computer and the FPGA, enabling transferring of text which avoids the slowing down of the CPU for tasks that do not require high processing powers.

C-code for xUSB/SPI portion

1. `void MAXreg_wr(BYTE reg, BYTE val;`

```
//writes register to MAX3421E via SPI
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");
    }
    //not being done:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //;

}
```

This function writes a 1 BYTE value ‘val’ to a register in the MAX3421E via SPI. This is done by first writing the value of the reg + 2 to specify the address, and the actual value to be written. The SPI function is then called, the writedata is passed and is set to perform a write operation.

2. `BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data);`

```
//multiple-byte write
//returns a pointer to a memory position after last written
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);
    //write reg + 2 via SPI
    //write data[n] via SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0  print an error
    //return (data + nbytes);

    //not being done:
        //select MAX3421E (may not be necessary if you are using SPI peripheral)
        //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
        //;

}
```

This function writes a data array of length nbytes given by the pointer data. First a new array is created consisting of the data to be written. First, the register address reg + 2 is written followed by the N bytes of data. The SPI command is then invoked, the new array is passed, and it is set to write. The entire function returns the pointer to the memory address after the last write function was executed.

3. BYTE MAXreg_rd(BYTE reg);

```
//reads register from MAX3421E via SPI
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");
        }
    //return val
        return val;
    //not being done:
            //select MAX3421E (may not be necessary if you are using SPI peripheral)
            //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
            //;
}
```

This function reads a register and returns the value stored. To do this we invoke the SPI command. The address of the register is written, following which the 1 BYTE value is read and returned.

4. 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);

    //not being done:
            //select MAX3421E (may not be necessary if you are using SPI peripheral)
            //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
            //;
}
```

This function reads nbytes bytes of data and stores it in an array data. It invokes the SPI command. The starting register address is first written and then nbytes of data is read and stored in ‘data’. It then returns the pointer to the memory position where the last data was written.

---

5. Design Resources and Statistics

LUT	7138
DSP	10
Memory (BRAM)	50520
Flip-Flop	2449
Frequency	77.21 MHz
Static Power	96.72 mW
Dynamic Power	87.27 mW
Total Power	221.47 mW

---

6. Conclusion

In conclusion, we have successfully implemented a functional Rubik’s cube on the FPGA that can be rendered on a VGA monitor and that can be controlled with a keyboard. The simulated Rubik’s cube implements all the moves and manipulations of a physical Rubik’s cube, with additional functionalities such as reset and undo. We rendered the image and managed to successfully create a fully functional cube that performs all rotations possible.
