# Project Design: Rubik's Cube Solver

Zongheng Wang zw2223
Ifeoma Okereke iro2103
Yin-Chieh Wang yw2491
Heather Fisher hlf2119

## Design Overview

The aim of this project is to develop a Rubik's Cube Solver using the Thistlethwaite's algorithm. In this project, we intend to solve the Rubik's Cube in the shortest possible time taking advantage of the FPGA's speed. We intend to use pure software to test the time it takes to get a Rubik's Cube solution and compare the execution time with software/hardware hybrid approach. This will help us determine the efficiency of using the FPGA to solve the Rubik's Cube. We also plan to design a user-friendly user interface so that one could input the color configuration of a Rubik's Cube from a PS/2 keyboard and the corresponding solution will be shown on the VGA screen in separate steps. Overall, our project includes these two individual tasks.

## Overall Block Diagram

This project will be implemented using the NIOS II processor with 512K SRAM, a PS/2 keyboard controller, a VGA display controller and three specifically-designed acceleration blocks for the Thistlethwaite's algorithm. The PS/2 keyboard (Enter key, Number Pad and Arrow keys) will be used to obtain input from the user. The VGA screen will display the user's input of the Rubik's Cube color configuration at first. After the Thistlethwaite's algorithm finishes, it will demonstrate the steps to solve the Rubik's Cube using instructions displayed on the VGA Screen and a Static Image of the Rubik's Cube. The instructions will indicate which side the Cube should be rotated and in what direction. The Static Image of the Rubik's Cube will show the results of following the instructions. The software processes the user's input, executes the Thistlethwaite's algorithm and controls the flow of the entire program. The block system of the overall systems is shown in Figure 1.

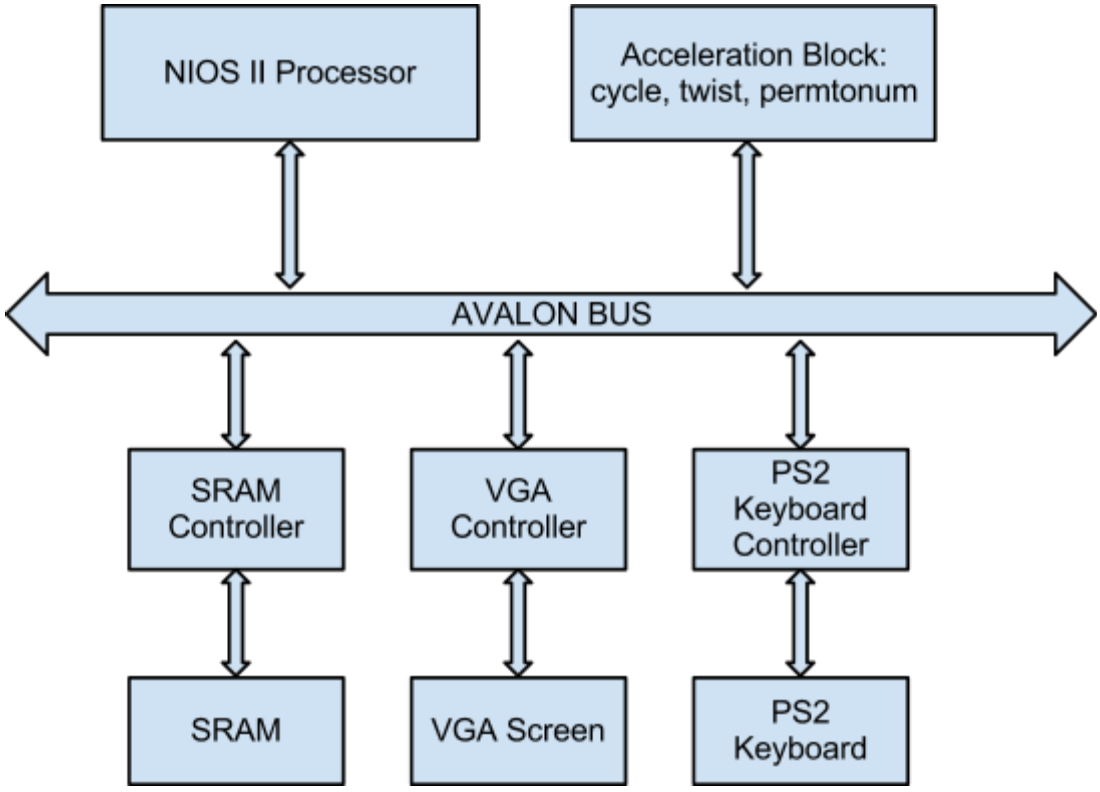


Figure 1: High-Level Block Diagram of RCS

## Memory Analysis

Based on the tests done using *valgrind* on a modern computer, 30946 bytes memory are needed in the Thistlethwaite's algorithm. Since the Altera DE2 board provides a 512 Kbytes SRAM, it is more than enough for this project. The reason why we chose to use the SRAM for the Thistlethwaite's algorithm is straightforward. SRAM guarantees minimal overhead in data transmission and storage and it supports a simple interface to implement.

For most of the hardware components, we decided to store their own data in on-chip block RAMs, since the amount of data they need to store is very limited and we also want to make the entire system faster. When the screen displays the current user's input, the only data it needs to store is the 54 colors from the 6 surfaces of Rubik's cube, based on which it could draw the current user's input. Since for a 3*3*3 Rubik's Cube, there are only six different colors. Then we could encode the color in three bits. Thus, we could store five colors in a half word. Overall, an 11*16 RAM is about enough to store 54 colors for the VGA controller. When displaying the solution demo on the screen, we decide to use software to aid the demo. In fact, when the user enters the stage of showing the solution, the software must have finished the Thistlethwaite's algorithm already. Therefore, this is no computation or memory

burden for the software. Then allocating some new jobs for the software increases the overall utilization of resources. As for the three acceleration blocks, they basically perform data re-wiring, bit manipulations or simple computations, and thus no memory concerns are necessary for them.

## Critical Path

Due to the fact that the Rubik's Cube solver is originally a pure software problem, and we intend to transform it into one that is solved by a hybrid software and hardware system, the critical path of our project lies in the execution of the Thistlethwaite's algorithm. We would ignore the overall critical path of the entire system since the Thistlethwaite's algorithm can only start after the user's input is completed, which could take as long as possible. Similarly, after the Thistlethwaite's algorithm works out a solution, the solution demonstration will start. This only serves as a nice user interface rather than the core task of our accelerating purpose.

The Thistlethwaite's algorithm itself is rather complicated based on a mathematical theory called group theory. Therefore, in our design, we will only discuss the critical path of the acceleration block. The acceleration block includes three individual functions, and the software chooses which function to run. For each function it runs, the software uses the data transmitted from the Avalon bus. The cycle function involves three RAM data swaps, the twist function involves addition and mod operations and the permtonum function involves six comparisons and a look-up table.

## Detailed Design of Each Peripheral

***Acceleration Block***

According to the test result of *gprof*, we discover that three functions in the Thistlethwaite's algorithm take about 75%-80% of the execution time. These three functions are tiny ones which mainly involve holding and modifying the current state of the cube. Each time the Thistlethwaite's algorithm is performed, these functions are executed about two million times. Therefore, it is essential for us to design an acceleration block for them. The three functions are described below:

```
char *perm="AIBJTMROCLDKSNQPEKFIMSPRGJHLNTOQAGCEMTNSBFDHORPQ"

// Use macro to swap items instead of classic routine with pointers for the sole reason of brevity
#define SWAP(a,b) TEMP=a;a=b;b=TEMP;
// number 65='A' is often subtracted to convert char ABC... to number 0,1,2,...
#define CHAROFFSET 65

// Cycles 4 pieces in array p, the piece indices given by a[0..3].
void cycle(char*p, char*a){
```

```
        SWAP(p[*a-CHAROFFSET], p[a[1]-CHAROFFSET]);
        SWAP(p[*a-CHAROFFSET], p[a[2]-CHAROFFSET]);
        SWAP(p[*a-CHAROFFSET], p[a[3]-CHAROFFSET]);
}
```

In this function, three swaps are performed. From the main function of the Thistlethwaite's algorithm, we find that the char*a is always passed with the global character array char *perm, except with different offset. For example, cycle(pos, perm+8) is equal to

```
SWAP(pos[2], pos[11]);
SWAP(pos[2], pos[3]);
SWAP(pos[2], pos[10]);
```

Therefore, the cycle function basically swaps some data of the array char*p. Furthermore, from the main function we can tell as well char*p is only passed with char pos[20] and ori[20]. Therefore, we decide to store these two arrays plus char *perm, which takes 88 bytes in total. The macro CHAROFFSET is purely used for software convenience and is entirely unnecessary for hardware design. Therefore the block diagram is as shown in Figure 2.

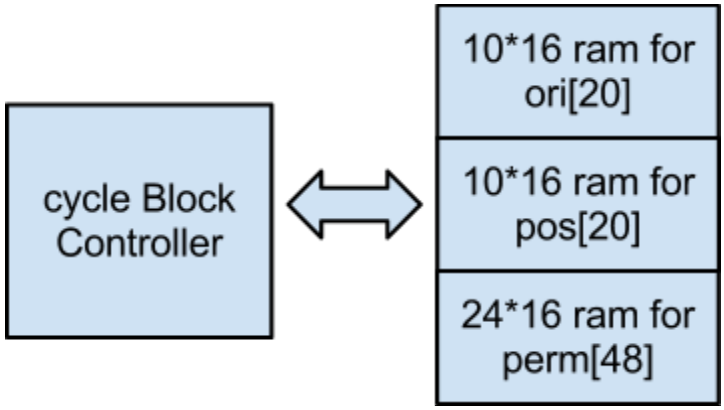


Figure 2: Block Diagram for cycle

In Figure 2, the controller can receive an unsigned integer of 6 bits, indicating the start point of the array perm, plus one bit indicating the target array ori or pos. After fetching the indices, the controller will read data from ori/pos at corresponding indices and swap their data with the help of a 16 bit register.

```
// twists i-th piece a+1 times.
void twist(int i,int a){
        i-=CHAROFFSET;
```

```
        ori[i]=(ori[i]+a+1)%val[i];
}
```

For the twist function, we only apply it to perm array. Note this RAM is shared within the entire acceleration block.

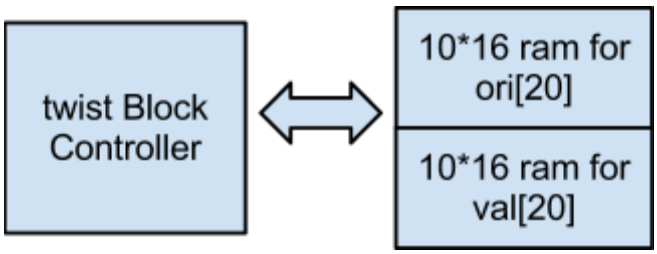


Figure 3: Block Diagram for twist

In Figure 3, the controller can receive two unsigned integer of 8 bits, put in a half word, indicating the start point of the array ori and val. After fetching the data based on the indices, the controller will perform the computation (ori[i]+a+1)%val[i] and write back the result.

```
// convert permutation of 4 chars to a number in range 0..23
int permtonum(char* p){
        int n=0;
        int a, b;
        for (a=0; a<4; a++) {
                n*=4-a;
                for(b=a; ++b<4; )
                        if (p[b]<p[a]) n++;
        }
        return n;
}
```

At last, for the permtonum function, if we unroll the loop, the function will become

```
int permtonum(char* p){
   int n=0;
   if (p[1] < p[0]) n++;
   if (p[2] < p[0]) n++;
   if (p[3] < p[0]) n++;
   n *= 3;
```

```
    if (p[2] < p[1]) n++;
    if (p[3] < p[1]) n++;
    n *= 2;

    if (p[3] < p[2]) n++;

    return n;
}
```

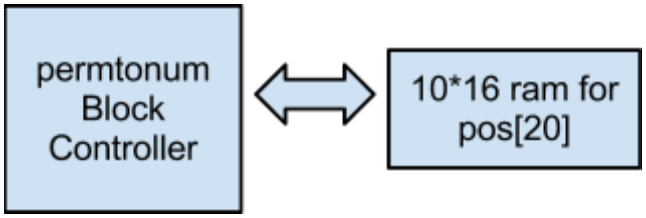


Figure 4: Block Diagram for permtonum

In Figure 4, the controller can receive one unsigned integer of 5 bits, put in a half word, indicating the start point of the array pos that permtonum function is expected to run. Since only four pieces of data need to be compared with each other, we decide to build a look-up table to determine n. For example, if the result of the six comparisons above is 110101, then from the table you will find 15 for n. This is fast, simple and it saves us some shift registers/adders or multipliers. Finally the controller will save the value of n, waiting for the software to read.

***VGA***

The RCS VGA Controller will generate signals that will display the six sides of the Rubik's Cube as six separate squares on the VGA Screen, similar to what Figure 5 illustrates. These squares will enable the user input the current configuration of the Rubik's Cube. After the Rubik's Cube has been solved, the VGA Controller will generate signals that will display the steps to solve the Rubik's cube. Each step will include a static image of the 3D Rubik's Cube, the instructions on how to rotate the cube, and options to move from one step to the other.

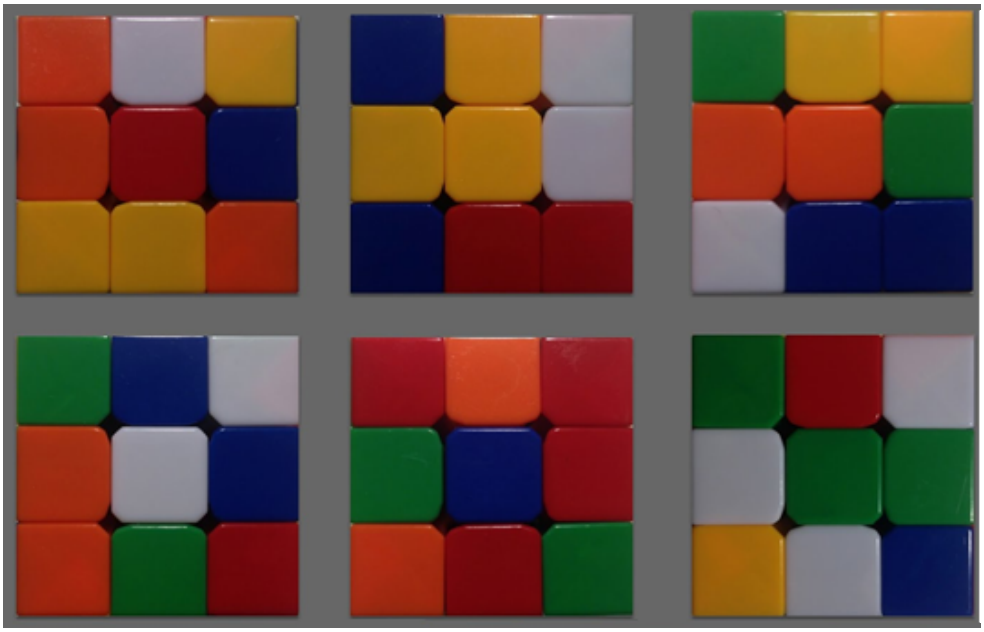

Figure 5: Six different sides of an unsolved Rubik's Cube

***PS/2 Keyboard***

Input from the user will be obtained via the PS/2 Keyboard. Numbers 1-9 on the Number Pad of the Keyboard will be used to select individual boxes of each side of the Rubik's cube. The arrow keys will be used to move from 1 side of the Rubik's Cube to next. Figure 6 shows which number each box represents. When the user presses key 1, the box marked 1 in figure 6 will be selected and the color will change. As long as the user keeps pressing key 1, the color of box 1 will keep changing through the six colors of the Rubik's cube. When the user has inputted all the six sides of the cube, the Enter key is pressed and this starts the computation process.

| | | |
|---|---|---|
| 7 | 8 | 9 |
| 4 | 5 | 6 |
| 1 | 2 | 3 |

Figure 6: Rubik's Cube Side showing each box and the number it represents

We will use the PS/2 Keyboard Controller provided in Lab 2 to handle the keystrokes from the user.

***SRAM***

We will use the 512 Kbytes SRAM on the Altera DE2 board with the Controller provided in the Lab 3.

## Milestones

**Milestone 1:**

**- Hardware:**

Design the layout of the user interface

Implement the VGA controller for the user input part

Integrate the system with the SRAM and PS/2 keyboard controllers

**- Software:**

Write the algorithm for transforming the input from the user into Mike Reid's Cube Solver format

Test the Thistlethwaite's algorithm on Altera DE2 Board with only NIOS II processor and SRAM

**Milestone 2:**

**- Hardware:**

Finish the acceleration block

Implement the Keyboard controller part that receives input color configurations from the user

Integrate the system with all the components so that the modified algorithm can run

**- Software:**

Modify and optimize the Thistlethwaite's algorithm and show the solution through printf

Integrate the algorithm with acceleration blocks

Compare the speed of solving a cube between pure software systems and hybrid software and hardware system

**Milestone 3:**

**- Hardware:**

Implement the VGA controller for solution demonstration part

**- Software:**

Implement the algorithm that coordinates with hardware to show solution demonstration