Bangladesh University of Engineering & Technology

Project Title: Implementing Tetris Game using Verilog.

Course No: EEE 304
Course Title: Digital Electronics Laboratory

Department: EEE
Section: C2
Level: 3 **Term:** 2
Students' ID: 1506179
1506180
1506184
1506185

Date of Submission: 09.02.2019

Submitted to: Mohammad Abdullah Al Shohel
Lecturer, Department of EEE, BUET
&
Towsif Taher
Lecturer, Department of EEE, BUET

Implementation of Tetris game Using Verilog

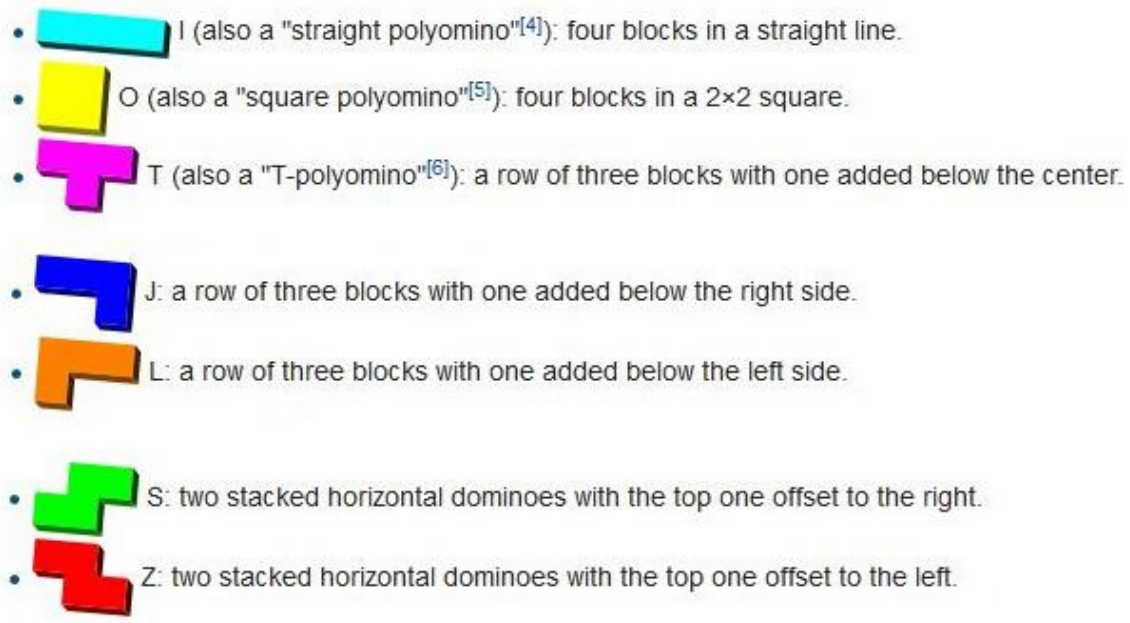
Abstract:

The game of Tetris consists of a 'well' or 'matrix' in which blocks/pieces of fall down gradually with the span of a certain time. Each of the total six kinds of pieces/blocks that fall are basic geometric shapes known as 'tetrominoes'. A random sequence of Tetriminos fall down the playing field. The aim of the player of this game is to manipulate these Tetriminos, by moving each one sideways and/or rotating by quarter-turns, so that they form a solid horizontal line with no gaps.

In this project, simplified version of the Tetris game was implemented using an FPGA (Field Programmable Gate Array) board - EPF10K70 device - a FLEX 10K device. The EPF10K70 device was programmed using Verilog HDL. The game was displayed in a 8 by 8 LED Matrix(each LED) that receiving output from the on-board I/O pins of the mentioned device.

Introduction:

Tetris is a game that can be implemented using basic binary logic. There is either an empty space (Logic state 0) or a block (Logic State 1) existing in the playing field. Tetrominoes - geometric shapes composed of four square blocks - are the basic building block of Tetris game. The six kinds of pieces are shown below:



At a time one of these piece falls down along the playing field at a certain pace.While falling, the user can move the piece left or right by giving necessary input. Another button can make the piece turn one quarter clockwise.

Components:

The components needed were

- FLEX 10K device
- 8 by 8 LED Matrix
- Female to female jumper wires

The Verilog HDL code was written and implemented using the Quartus II software.

Methodology:

In this section we explain how the the game was coded using Verilog HDL.

1. Clock generation:

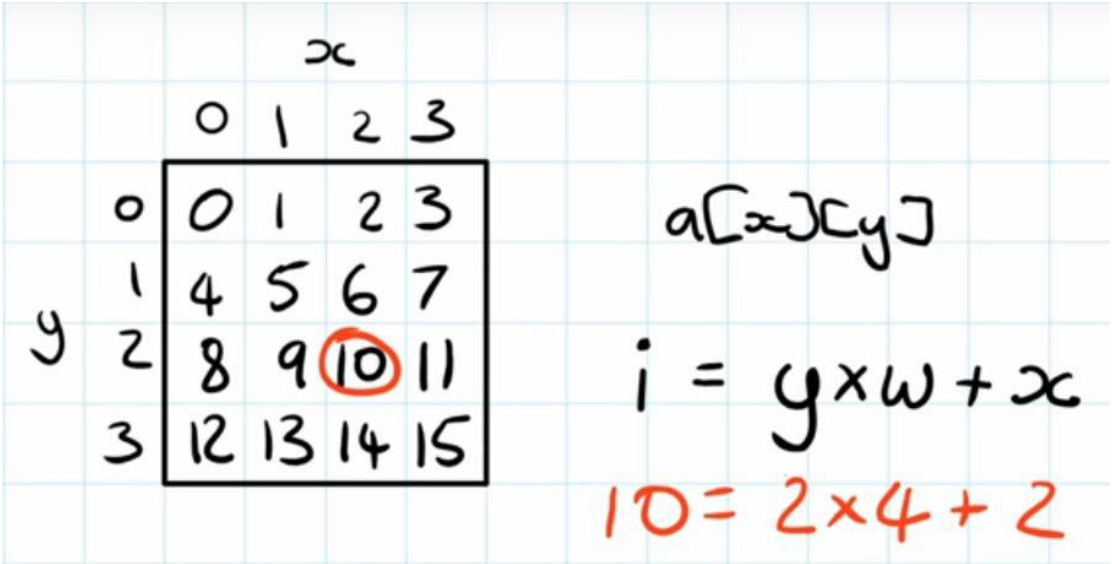
Generating low frequency clock pulse from the high frequency (25.175 MHz) internal clock:

The internal clock was of a very high frequency and hence could not be used as the game pulse, as then the game would become unplayable. So, a separate clock pulse of about 1 Hz (approximately equal to the speed of the falling Tetromino - 1 LED per second) was generated.

[As the game logic contains a lot of logic elements, the game pulse was actually much slower for a 1 Hz pulse because of associated gate delays and propagation delays withing those logic elements required to establish this game].

2. 2-D to 1-D Conversion:

A concept that was repeatedly used while developing the code for this game was converting 2-D matrix data to a 1-D bit sequence that was much simpler to implement and display using the LED matrix.



Here as shown in the above figure, the 1-D index of the bit sequence (represented by i) can be found from the x-coordinate value and y-coordinate value from the following formula:

$$i = y.(width) + x$$

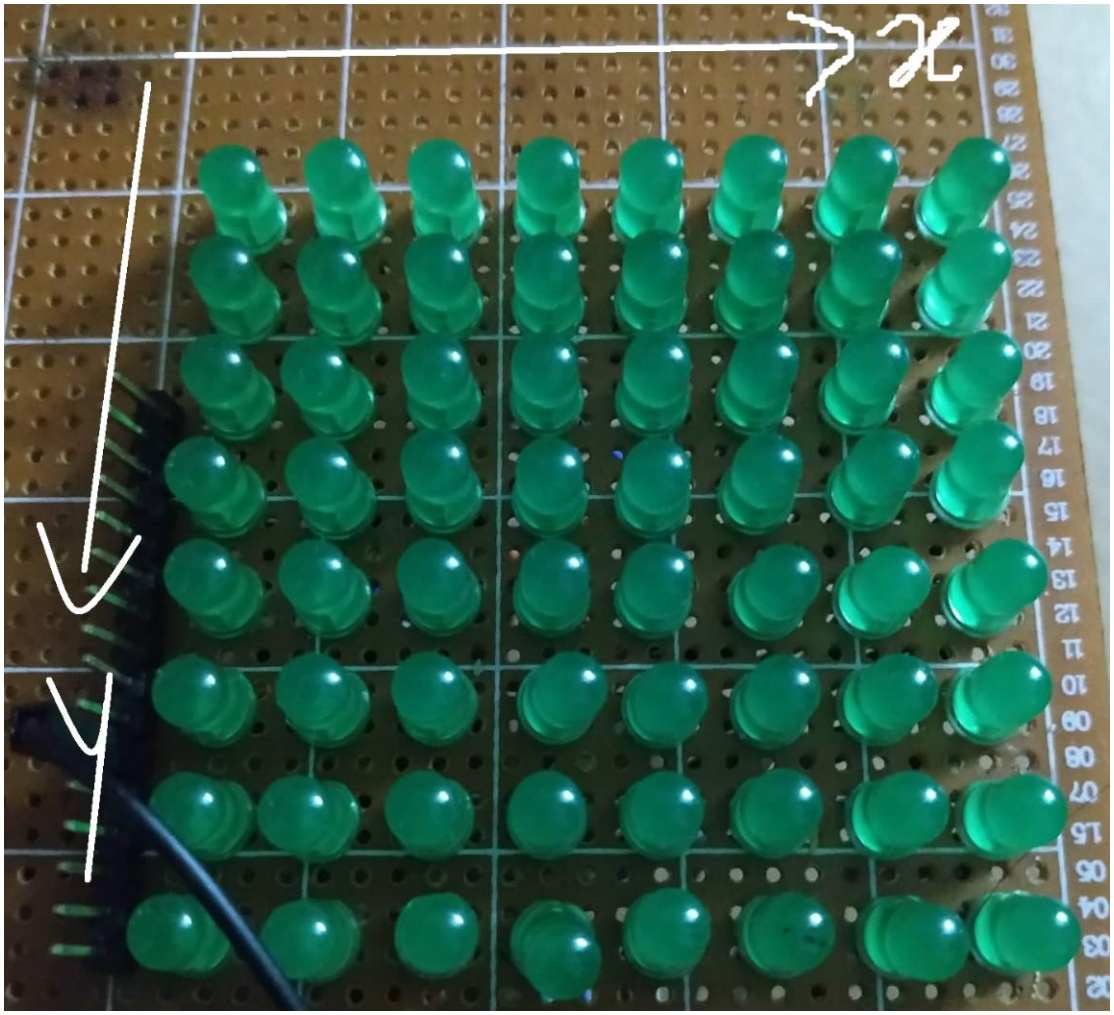
The basic building block of the game was the 7 kinds of Tetrominoes. Each of them were declared as a 16 bit binary data.Utilizing the previous technique, all of the tetrominoes were basically 4 by 4 matrices declared as a 16 bit binary sequence. However, using the above formula we could access the binary data of the block using either its x-y coordinate positional value, or its 1-D bit sequence index value.

For example, the J type block was declared in the 'initial' block of the verilog HDL file as

```
block[3][3:0]=      4'b0100;
block[3][7:4]=      4'b0100;
block[3][11:8]=     4'b0110;
block[3][15:12]=    4'b0000;
```

(Note: As the bit sequence here is written in reverse form - from MSB to LSB the J actually looks like a L in the file. But, when displaying on the LED Matrix, we will see it as a J)

This technique was also used in displaying the required display data in the 64 bit output variable - 'OUT'. For coding the game logic, a 2-D positional denotation was more favourable. But for displaying the data in the 8 by 8 Matrix and for minimizing the number of logic elements used, 1-D indexed notation was more suitable.



3. Game Logic:

In the first layer of task, the idea was to display the Tetrominoes in the output display array - 'OUT' in our desired position - specified by 'nPosX' and 'nPosY'. A nested for loop over the Tetromino piece and the concept of converting 2-D positional data to 1-D index was used to imprint the tetromino bit sequence into the 1-D output display array - 'OUT'.

```
for(k=0;k<=3;k=k+1)
    begin
        for(i=0;i<=3;i=i+1)
            begin

                index=(nPosY+k)*8+(nPosX+i);
//Imprinting the tetromino block into the OUT matrix
                OUT[index]=block[rnd][k*4+i];
```

Thereafter, after every clock pulse the same task of imprinting the tetromino piece to the 'OUT' array was repeated, but with the 'nPosY' variable denoting the next row-new row. (Note: OUT is a 64 bit 1-D data sequence)

```
nPosY = nPosY + 1;
```

Next task was to stack the falling pieces once they reach the last row or if the collide with a fallen piece on its way down.

The next task, consisting of stacking after reaching the last row and the collision detection with an already fallen piece, were done separately.

To detect a newly fallen piece in the last row the following code snippet was added after the imprinting task.

```
if ((|(OUT[63:56]^y_flag[7]))==1)
//Check if last row has at least one LED on
begin
    LastRowFlag=1;
end
```

Here, the y_flag[7] stores the bit sequence of the 8th row (56 to 63)of the 'OUT' display matrix on the previous clock cycle (Initially set to 0 by default). It is compared with the new 8th row (index 56 to 63) of the updated 'OUT' display matrix.
If the two bit sequences are same, then the falling tetromino piece hasn't reached the last row. If not,then the falling piece has successfully reached the last row.This decision is stored in a flag variable 'LastRowFlag'.

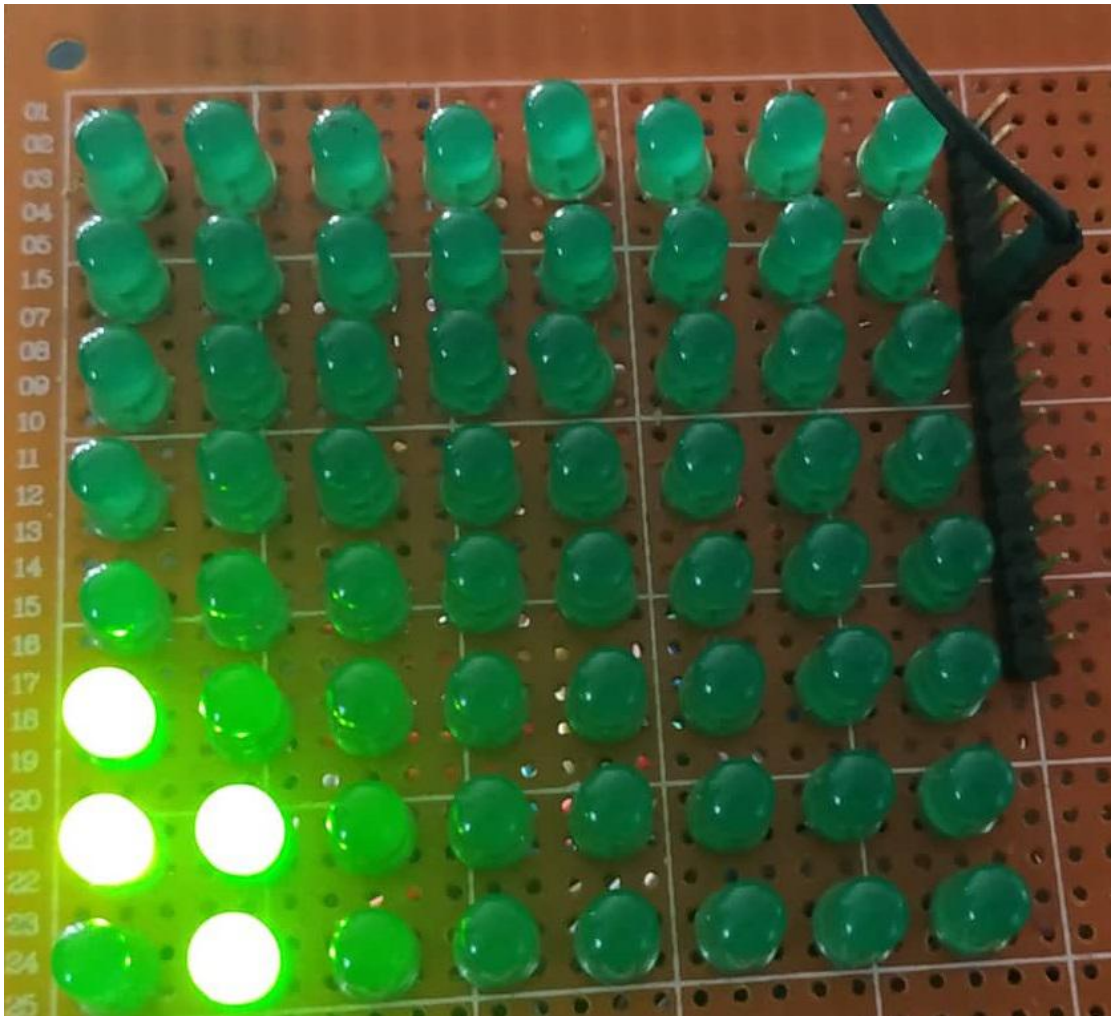


Figure: When one tetromino reaches the bottom

For the collision detection, we have a variable 'DoesNOTFit' to indicate whether the current tetromino piece will fit in its current position in the playing field. It is checked in the same nested loop where the block is imprinted on to the 'OUT' array. If the positions of the tetromino which contains '1' coincide with the '1' of the current playing field (indicated by variable mask) then the DoesNOTFit variable is set to 1 as the piece does not fit in that location.
If either the LastRowFlag or the DoesNOTFit equals 1, then the 'OUT' array is reset to initial state.

Movements of the tetromino pieces are very straightforward. The following code snippet shows it:

```
// Begin Move block
        if (!R)
            nPosX=nPosX+1;
        else if (!L)
            nPosX=nPosX-1;
        else
            nPosX=nPosX;
```

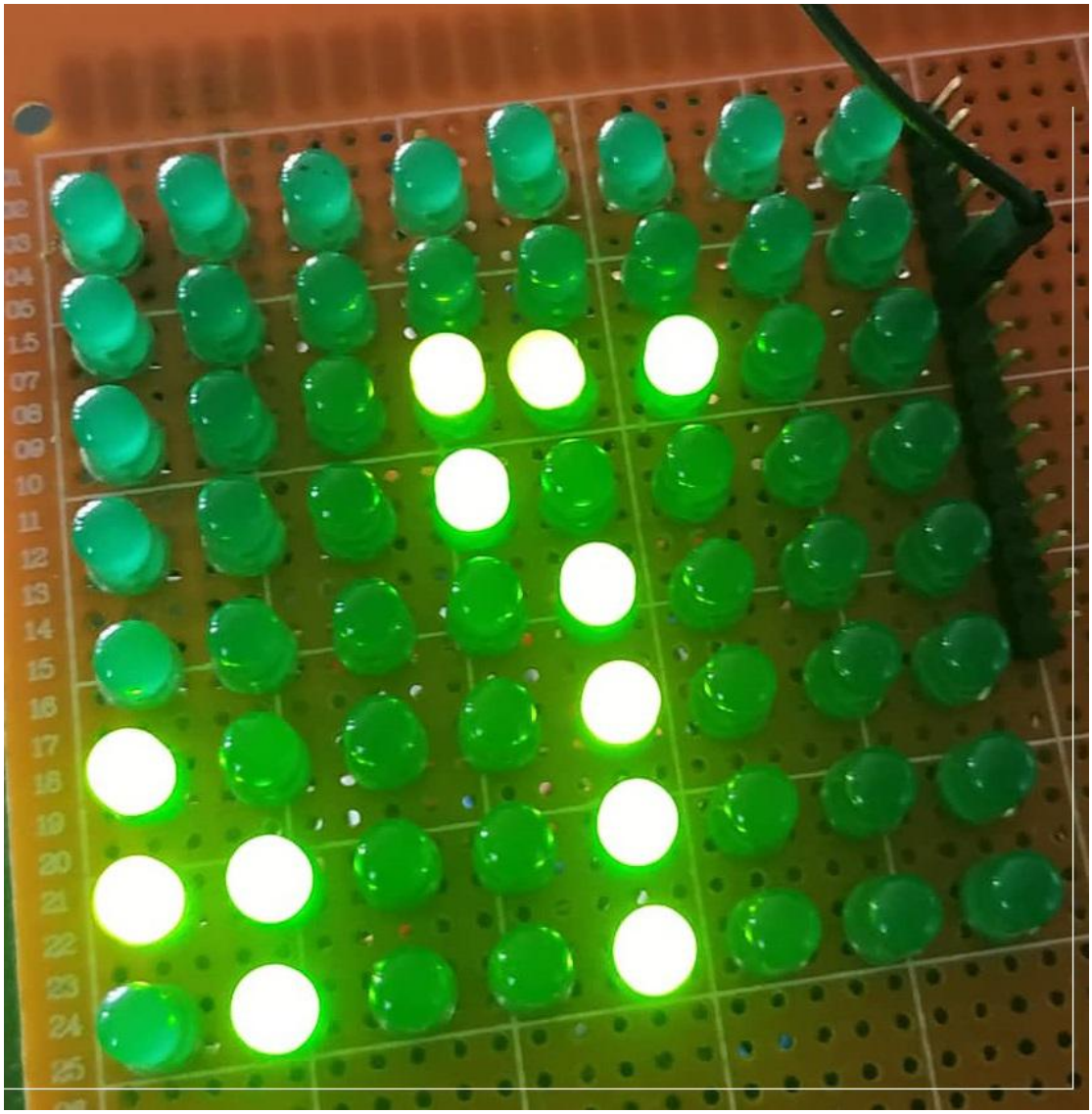
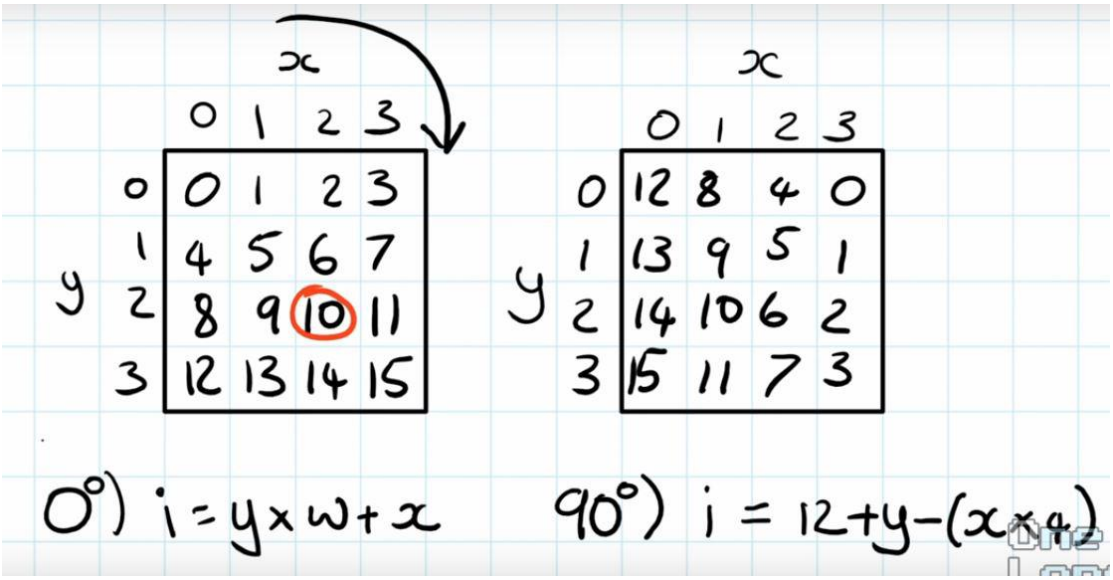


Figure: Step before the J tetromino collides with the I block

4. Rotation:

The index of a rotated tetromino piece (1-D index) can be updated using a similar logic like the one used to convert from 2-D matrix position to 1-D index, mentioned previously.



The rotated index is given by:

$$i = 12 + y - 4x$$

So the tetromino piece is rotated by the following portion of code:

```
//Rotate Block
            if(!z)
                begin
                  for(j=0;j<=3;j=j+1)
                    begin
                        for(i=0;i<=3;i=i+1)
                            begin
                                block[rnd][j*4+i]<=block[rnd][12+j-4*i];
                            end
                    end
                end
```

The rotated block is assigned to the original block if the rotate buttion (z) is pressed.

5. Random number generator:

If a tetromino reaches the last row (indicated by 'LastRowFlag') or if it collides with an existing portion of the playing field (indicated by 'DoesNOTFit'), a new random tetromino piece is to be imprinted on top of the playing field from the top. The random number is generated by a random 1 degree polynomial equation. (Higher order polynomial equation could not be formed because of the limitations in logic elements of the EPF10K70 device).

6. Line detection:

Each of the previous rows of the output display matrix 'OUT' was assigned separately to each of the eight 8 bit binary sequence called 'y_flag'.

```
            for(j=0;j<=7;j=j+1)
            //Storing Rows+Line detection
```

```
                begin
                    for(i=0;i<=7;i=i+1)
                        begin
                            y_flag[j][i]=OUT[j*8+i];
                        end
```

Then it was checked whether each of the 'y_flag'contained all 1's (8'b11111111). If so, then that particular 'y_flag' row was emptied, and the preceding y_flags were shifted downwards. Then, the 'OUT' matrix was updated with the new 'y_flag' values.

```
for(j=0;j<=7;j=j+1)
//Storing Rows+Line detection
    begin
        for(i=0;i<=7;i=i+1)
            begin
                y_flag[j][i]=OUT[j*8+i];
            end
//Line detection portion
            if (y_flag[0]==255)                    //0th row exceptional
                y_flag[0]=0;
            if (y_flag[j]==255)
                begin
                    y_flag[j]=0;
                for(k=j;k>=1;k=k-1)
                    begin
                        y_flag[k]=y_flag[k-1];
                    end
                    y_flag[0]=0;
            end

                            end

                        //Update OUT for line detection
                        for(j=0;j<=7;j=j+1)
                            begin
                                for(i=0;i<=7;i=i+1)
                                    begin
                                        OUT[j*8+i]=y_flag[j][i];
                                    end
                            end
```

Conclusion:

The Tetris game can be implemented by simple boolean logic and variables as we only need two states to indicate all game states and express all game logics. A 8 by 8 LED matrix was used to implement the game, and it was controlled using a Verilog HDL file implemented by an FPGA board. We had to encounter many setbacks and problems while implementing this project because of which this game has certain limitations. Thus

this game may not have all the added features that we wanted to accommodate.The limitations and setbacks are discussed below:

Discussion:

1. Initially, we were interested in implementing the Tetris by only using logic gates and ICs. But the circuit was extremely tedious and difficult to implement. So, we were interested in implementing it by coding it in Verilog using an FPGA board.

2. We, tried to implement Tetris game using a commercially available 8 by 8 LED Matrix. However, It wasn't possible to implement all the logics necessary because we couldn't access all the 64 LEDs separately using the available 16 connection pins of the matrix. We tried to solve the problem by using multiple clocks, to perform different functions of the game - i.e we tried to use separate clocks for display and block movements.But, as the FPGA board only allows for single pin assignment by a single pin, this method did not work.
3. Finally, in order to implement as many features of the game as we could, we soldered separate 64 LEDs on to a veroboard and made a common cathode connection. Thus, by connecting the GND pin from the FPGA board to the common cathode terminal and by using 64 separate pins from the FPGA board, we could access all the 64 LEDs using the FPGA device. The LEDs worked on active high logic.

4. Despite being able to implement most basic functions of the tetris game, the random number generator block was not actually random. It was only pseudo random. Moreover, the degree of predictability is high (we used only a 1 degree polynomial) as we could not accommodate greater randomness (higher degree polynomial) due to lack of storage/logic elements in the FPGA board.

5. We had to skip scoring section as well, as we had reached 97% of total logic elements. Adding any other features other than the ones mentioned above would have required excess logic elements, that would have exceeded device specifications.

We wanted to incorporate scoring, by using a separate variable that would increase in value for every step of the game. Certain excess points would be offered for line detection. As each of these sections have already been implemented, it was only a task of arithmetically manipulating the score variable in each of these sections as per requirement.