Bangladesh University of Engineering and Technology

Department of Electrical and Electronic Engineering

EEE 304 : Digital Electronics Laboratory

A PROJECT ON

SUDOKU SOLVING USING SYSTEM VERILOG AND MICROCONTROLLER

Submitted to,

Rajat Chakraborty

Lecturer

Md Irfan Khan

Lecturer

Dept. of EEE

BUET

Submitted by,

1. Oishy Saha

Std Id : 1606075

2. Jarin Tasnim

Std Id : 1606090

3. Barproda Halder

Std Id : 1606092

Section : B1

Video Link :

https://drive.google.com/file/d/1NgDklTPuhqqkda6hJBh_RhXP29RKANkm/view?fbclid=IwAR0phdoGmmQ70KTy0yrY4YgZAT6A1KWJiRWI_i8wnUTqH4HWFxLVhyJF8pw

Abstract:

The objective of the project is to design a Sudoku Solver using System Verilog and micro-controller. As System Verilog provides a user friendly interface to create an unsolved sudoku and provide the right solution irrespective of the difficulty level, the suitable platform has been incorporated in the project. Along with this implementation, a microcontroller-based Sudoku Solver has also been demonstrated in our project. In the algorithm part, two methods called backtracking and naive approach has been implemented depending upon the difficulty level of the Sudoku. With the increase of difficulty level, the naive approach becomes ineffective and the backtracking method then comes into account. In our project, the SystemVerilog section has been performed in online EDA playground and the microcontroller based part has been simulated in Proteus 8.6 software.

Introduction:

Sudoku is a well-known combinatorial number-placement puzzle which achieved international widespread popularity in 2005 [1]. The most common version of this game consists of 81 cells organized in 9×9 grid (which in turn is divided in nine 3×3 regions) where some cells contain numbers from 1 to 9. The objective is to fill in the remaining empty cells with the numbers 1…9. That

1. every number (1…9) appears in each row only once;

2. every number (1…9) appears in each column only once;

3. every number (1…9) appears in each 3×3 region only once.

The most common version of Sudoku (9×9 grid with 3×3 regions), can be generalized to other sizes. A Sudoku starts with some cells containing numbers (clues) and the goal is to solve the remaining cells. Proper Sudokus have one solution. Players and investigators may use a wide range of computer algorithms to solve Sudokus, study their properties and make new puzzles, including Sudokus with interesting symmetries and other properties.

There are several computer algorithms that will solve most 9×9 puzzles (n=9) in fractions of a second, but combinatorial explosion occurs as n increases, creating limits to the properties of Sudokus that can be constructed, analyzed and solved as

n increases. An example of Sudoku puzzle of order 3 is depicted in ***Figure 1.a*** where 3×3 regions are shown in thick lines. A solution to the puzzle is represented in ***Figure 1.b***

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

Figure 1.a: *A typical sudoku puzzle*

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

Figure 1.b: *A solution to the puzzle*

Available Algorithms:

1. Backtracking :

A backtracking algorithm is a recursive algorithm that attempts to solve a given problem by testing all possible paths towards a solution until a solution is found. Each time a path is tested, if a solution is not found, the algorithm backtracks to test another possible path and so on till a solution is found or all paths have been tested.

Backtracking algorithms rely on the use of a recursive function. A recursive function is a function that calls itself until a condition is met. This can be applied only for the problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution.

Conceptually, the partial candidates are represented as the nodes of a *tree structure*, the *potential search tree*. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that can not be extended any further.

The backtracking algorithms.

The backtracking algorithm traverses this search tree recursively, from the root down, in depth-first order. At each node c, the algorithm checks whether c can be completed to a valid solution. If it cannot, the whole sub-tree rooted at c is skipped (pruned). Otherwise the algorithm checks whether c itself is a valid solution, and if so reports it to the user; and recursively enumerates all sub-trees of c. Therefore, the actual tree that is traversed by the algorithm is only a part of the potential tree. The total cost of the algorithm is the number of the nodes of the actual tree times the cost of obtaining and processing each node. This fact should be considered when choosing the potential search tree and implementing the pruning test.

In order to apply the backtracking to a specific class of problems, one must provide the data *P* for the particular instance of the prolem that is to be solved, and six procedural parameters, *root, reject, accept, first, next and output*. These procedures should take the instance data *P* as a parameter and should do the following :

1. ***root(P) :*** return the partial candidate at the root of the search tree.
2. ***reject(P.c) :*** return true only if the partial candidate c is not worth completing.
3. ***accept(P,c) :*** *return true if c is a solution of P, and false otherwise.*
4. ***first (P,c) :*** generate the first extension of candidate c.
5. ***next (P,s) :*** generate the next alternative extension of a candidate, after the extension s.
6. ***output (P,c) :*** *use the solution c of P, as appropriate to the application.*

The *reject* procedure should be a boolean-valued function that returns true only if it is certain that no possible extension of c is a valid solution for P. If the procedure can not reach definite conclusion, it should return false. An incorrect true result may cause the procedure to miss some valid solutions.

The efficiency of the backtracking algorithm depends on *reject* returning *true* for candidates that are as close to the root as possible.

The *accept* procedure should return true if c is a complete and valid solution for the problem instance P, and *false* otherwise. It may assume that the partial candidate c and all its ancestors in the tree have passed the reject test.

The *first* and *next* procedures are used by the backtracking algorithm to enumerate the children of a node c of the tree, that is the candidates that differ from c by a

single extension step. The call *first (P,c)* should yield the first child of c, in some order, and the *next (P,s)* should return the next sibling of node s, in that order. Both functions should return a distinctive NULL candidate, if the requested child does not exist.

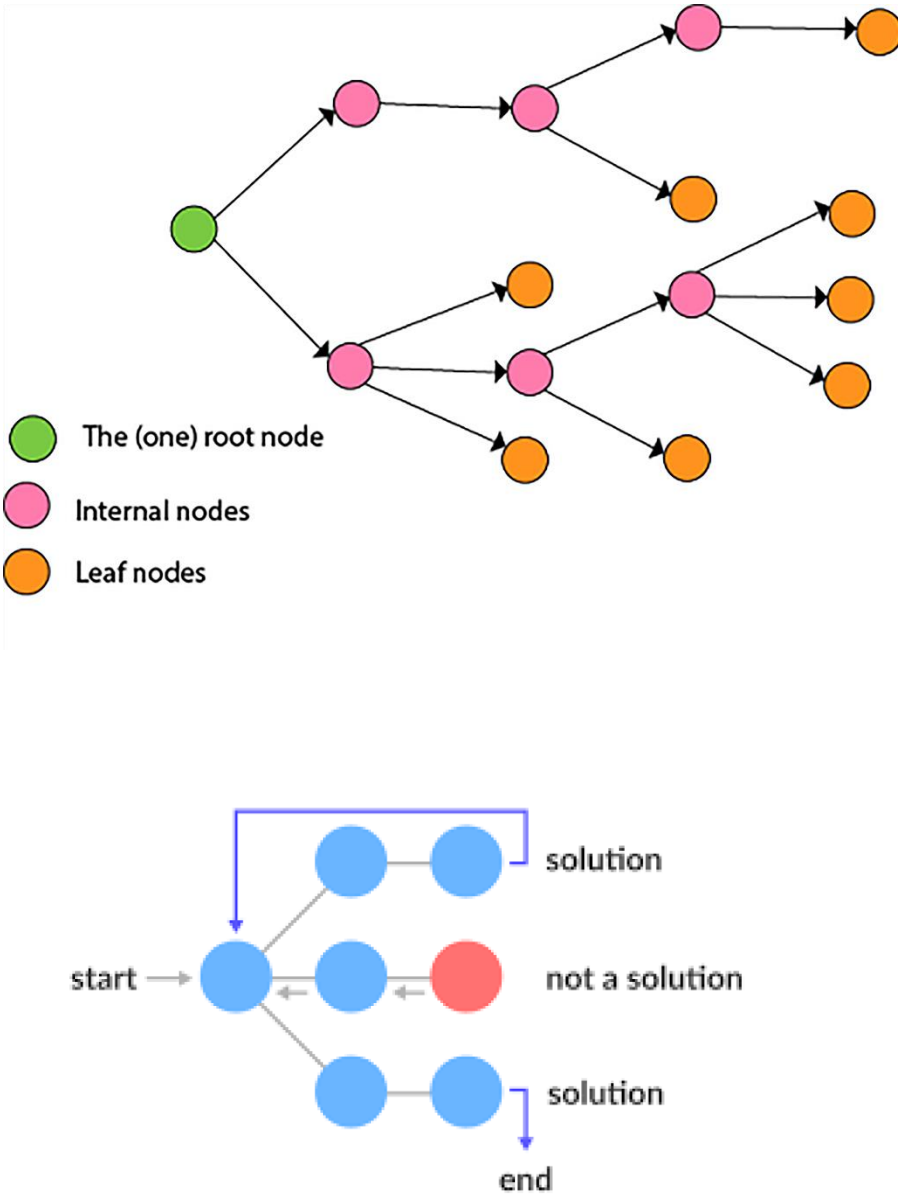

Figure 2 : *Visual illustration of backtracking*

2. Brute – Force Algorithm:

A brute-force algorithm to find the divisors of a natural number *n* would enumerate all integers from 1 to n, and check whether each of them divides n without remainder. A brute-force approach for the eight queens puzzle would examine all possible arrangements of 8 pieces on the 64-square chessboard, and, for each arrangement, check whether each (queen) piece can attack any other.

While a brute-force search is simple to implement, and will always find a solution if it exists, its cost is proportional to the number of candidate solutions – which in many practical problems tend to grow very quickly as the size of the problem increase. Therefore, brute-force search is typically used when the problem size is limited, or when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size. The method is also used when the simplicity of implementation is more important than speed.

3. Stochastic Search / Optimization Methods:

Sudoku can be solved using stochastic (random-based) algorithms. An example of this method is to:

I. Randomly assign numbers to the blank cells in the grid.
II. Calculate the number of errors.
III. "Shuffle" the inserted numbers until the number of mistakes is reduced to zero.

A solution to the puzzle is then found. Approaches for shuffling the numbers include simulated annealing, genetic algorithm and tabu search. Stochastic-based algorithms are known to be fast, through perhaps not as fast as deductive techniques. Unlike the latter however, optimization algorithms do not necessarily require problems to be logic-solvable, giving them the potential to solve a wider range of problems.

4. Constraint Programming:

A sudoku may also be modelled as a constraint satisfaction problem. These techniques are easier to implement than other algorithms. An algorithm combining a constraint-model-based algorithm with backtracking would have the advantage of fast solving time, and the ability to solve all sudokus.

Analysis of Algorithms:

Algorithm in System Verilog:

SystemVerilog is an extension of Verilog with many verification features that allows to verify the design using complex testbench structures and random stimuli in simulation.

A testbench allows us to verify the functionality of the design through simulations. The top level design module is instantiated within the testbench environment and design input/output ports that are connected with the appropriate testbench component signals. Certain values are given as inputs and functionality of the design is observed by comparing the outputs with the expected values.[2]

To solve any Sudoku puzzle in SystemVerilog , the puzzle is first declared in the testbench environment and given as an input to the SystemVerilog design module. The design module solves the Sudoku puzzle and print the solved Sudoku.

Testbench:

```
module top;
  parameter  M = 3;
  localparam N = M * M;
  int unsigned puzzle[N][N];
  sudoku#( M ) s;

  initial begin
    puzzle = '{ '{ 0,3,5, 0,0,0, 2,4,0 },
                '{ 0,0,0, 0,3,2, 0,0,0 },
                '{ 0,0,1, 0,0,0, 9,0,0 },

                '{ 0,7,0, 4,0,8, 0,0,0 },
                '{ 0,1,0, 0,2,0, 0,9,0 },
```

```
                '{ 0,0,0, 6,0,7, 0,8,0 },

                '{ 0,0,3, 0,0,0, 7,0,0 },
                '{ 0,0,0, 3,6,0, 0,0,0 },
                '{ 0,5,2, 0,0,0, 1,6,0 } };
    s = new;
    if ( s.solve_this( puzzle ) ) s.print();
    else $display( "cannot solve the puzzle" );
  end
endmodule: top
```

Analysis of the testbench Code:

The design of our Sudoku solver consists of a class named “sudoku’’. In this testbench, an object ‘s’ of this class is declared. The parameter M can be any value greater than 1 and so any Sudoku of size 4 x 4 / 9 x 9 /16 x 16 etc . can be solved by this solver. The Sudoko puzzle to be solved is declared by the variable ‘puzzle’ . To solve the puzzle , it is sent as a input to respective function of the class object ‘s’ and then the solved puzzle is printed.

Design:

Main design module of our Sudoku Solver consists of the class ‘sudoku’.

```
class sudoku#( int M ); // M >= 1
  localparam N = M * M;
  local int unsigned puzzle[N][N];
  rand  int unsigned box   [N][N];
```

Parameter M is taken as the input of the class where M x M is the size of each region in the puzzle. The class has one local two dimensional array type variable ‘puzzle’ and one random two dimensional array type variable ‘box’. Both the variables are of same size N x N where N = M * M. The values of the solved Sudoku will be kept in the random variable ‘box’.

After declaration of variables , various constraints of Sudoku solving is declared using ‘constraint’ blocks.

```
constraint box_con {
    foreach ( box[row, col] ) {
      box[row][col] inside { [ 1 : N ] };
    }
  }
```

This block sets constraint on each box[row][col] value. The value of each cell (total 81 cells for 9 x 9 box) of box must be between 1 and N.

```
constraint row_con {
    foreach   ( box[row, colA] ) {
      foreach ( box[   , colB] ) {
        if ( colA != colB ) {
          box[row][colA] != box[row][colB];
        }
      }
    }
  }
```

Constraint ‘row_con’ implies that the cells of the same row must have unique values. The cells having the same ‘row’ number but different column number ‘colA’ and ‘colB’ must have different values.

```
 constraint column_con {
    foreach   ( box[rowA, col] ) {
      foreach ( box[rowB,    ] ) {
        if ( rowA != rowB ) {
          box[rowA][col] != box[rowB][col];
        }
      }
    }
```

```
}
```

Constraint ‘column_con’ implies that the cells of the same column must have unique values. The cells having the same ‘col’ number but different row number ‘rowA’ and ‘rowB’ must have different values.

```
constraint block_con {
   foreach   ( box[rowA, colA] ) {
     foreach ( box[rowB, colB] ) {
       if ( rowA / M == rowB / M &&
            colA / M == colB / M &&
            ! ( rowA == rowB && colA == colB ) ) {
         box[rowA][colA] != box[rowB][colB];
       }
     }
   }
 }
```

This constraint ‘block_con’ implies that the cells in the same M x M block of the box must have unique values.

```
 constraint puzzle_con {
   foreach ( puzzle[row, col] ) {
     if ( puzzle[row][col] != 0 ) {
       box[row][col] == puzzle[row][col];
     }
   }
 }
```

‘puzzle_con’ constraint implies that the cells of the variable ‘box’ must have the same value as the variable puzzle's if values of the cells of ‘puzzle’ are not zero.

After declaring all the constraints, a function named ‘solve_this’ is called which will take input from the testbench.

```
function int solve_this( int unsigned puzzle[N][N] );
    this.puzzle = puzzle;
    return this.randomize();
  endfunction: solve_this
```

In this function this keyword refers to the class object ‘s’ and this.puzzle refers to the local variable ‘puzzle’ of the class . So the input N x N puzzle from testbench is kept in this local variable ‘puzzle’. this.randomize() sets the values of ‘rand’ type N x N variable ‘box’ of the class maintaining all the constraints. As the constraints declared previously are the constraints for solving Sudoku, the values set in N x N ‘box’ provides the solved Sudoku.

After solving, the following function prints the Solved Sudoku.

```
function void print();
    for ( int i = 0; i < N; i++ ) begin
      if ( i % M == 0 ) $write( "\n" );
      for ( int j = 0; j < N; j++ ) begin
        if ( j % M == 0 ) $write( " " );
        $write( "%3d", box[i][j] );
      end
      $write( "\n" );
    end
  endfunction: print
endclass: sudoku
```

Analysis of Arduino Code:

In the project, two algorithms have been implemented for solving the 9 × 9 Sudoku in Arduino. The methods are:

- ❖ Simple Approach
- ❖ Backtracking

The solver has been demonstrated in the Proteus 8.6 software through LCD displays. The execution steps are illustrated in the following section:

I. Simple Approach

In this method, all possible entries have been considered for a particular empty cell. This approach is mainly divided in two sections. The first one is preparation of a matrix which contains prediction for each numbers in a cell and next one is to to assign valid number in the empty cells.

```
bool solveGrid (int grid[9][9]) {
  int values[9][9];
  bool numbers[9][9][9];
  int empties = emptyCells(grid);
  int emptiesOld;
  int steps = 0;
  clearGrid(values);
  while (empties>0) {
    emptiesOld = empties;
    steps++;
    prepareCells(grid, numbers);
    checkCells(grid, numbers, values);
    copyValues(grid, values);
    empties = emptyCells(grid);
    if (empties == emptiesOld) {
          return false;
```

```
    }}
  return true;}
```

Here values[][] is the matrix where the predicted number for each step is stored and numbers[][][] stores the prediction for each number in a cell. While empty cells are present in the grid at first the prediction matrix is built using the function prepareCells(grid, numbers).

```
void prepareCells(int grid[9][9], bool numbers[9][9][9]) {
  for (int r=0; r<9; r++) {
    for (int c=0; c<9; c++) { setNumbers(r, c, grid, numbers); }
  }}
void setNumbers(int r, int c, int grid[9][9], bool
numbers[9][9][9]) {
  for (int n = 0; n < 9; n++) {
    numbers[n][r][c] = isSafe(grid, r, c, n+1);
  }}
```

In the above two functions, the prediction matrix is built assigning 1 for a valid entry in a certain cell and 0 for invalid entries. Say in cell (0,0), number 1 and 2 are safe for initial guess. Then only numbers [0][0][0] and numbers [1][0][0] have one. The validation is checked using the following functions.

```
bool isSafe(int grid[9][9], int r, int c, int num) {
  return !UsedInRow(grid, r, num) &&
    !UsedInCol(grid, c, num) &&
    !UsedInBox(grid, r - r % 3, c - c % 3, num) &&
    grid[r][c] == 0;
}
bool UsedInRow(int grid[9][9], int r, int num) {
  for (int c = 0; c < 9; c++) {
    if (grid[r][c] == num) return true;
  }
```

```
  return false;}
```

In this function it is checked whether the number exists in the given row r.

```
bool UsedInCol(int grid[9][9], int c, int num) {
  for (int r = 0; r < 9; r++) {
    if (grid[r][c] == num) return true;
  }
  return false;
}
```

In this function it is checked whether the number exists in the given column c.

```
bool UsedInBox(int grid[9][9], int boxStartRow, int boxStartCol,
int num) {
  for (int r = 0; r < 3; r++) {
    for (int c =0; c < 3; c++)
      if (grid[r + boxStartRow][c + boxStartCol] == num) return
true;
  }
  return false;
}
```

In this function it is checked if the number exists in the given 3×3 box.

If isSafe() returns true value then the prediction matrix has high value at particular cell.

The checkCells(grid, numbers, values) assigns the predicted number in values[ ][ ]. A value is assigned if it is unique in a particular row or column or box or cell and safe to assign.

```
void checkCells(int grid[9][9], bool numbers[9][9][9], int
values[9][9]) {
  for (int r=0; r<9; r++) {
    for (int c=0; c<9; c++) {
      if (grid[r][c]!=0) continue;    // If the cell has a value
then go to next cell
```

```
      if (values[r][c]!=0) continue;  // If the cell has a
prediction then go to next cell


      for (int n=0; n<9; n++) {
        if (values[r][c]==0 && uniqueInBox(numbers, n, r-r%3, c-
c%3) && numbers[n][r][c]) {
          values[r][c]=n+1;
          continue;
        }
        if (values[r][c]==0 && uniqueInRow(numbers, n, r) &&
numbers[n][r][c]) {
          values[r][c]=n+1;
          continue;
        }
        if (values[r][c]==0 && uniqueInCol(numbers, n, c) &&
numbers[n][r][c]) {
          values[r][c]=n+1;
          continue;
        }
        if (values[r][c]==0 && uniqueInCell(numbers, r, c) &&
numbers[n][r][c]) {
          values[r][c]=n+1;
          continue;
        }}}}}
```

After updating the entries of values[ ][ ], it is copied to the grid[ ][ ]. For easy and medium level Sudoku, this approach can be implemented to solve the puzzle.

II. Backtracking

If the 1st approach failed to arrive at a solution for complicated Sudoku puzzle, backtracking algorithm is used. This approach is needed when hard Sudoku needs to be solved.

```
bool solveBacktracking (int grid[9][9], int &moves) {
  int r, c;
  moves++;
  if (!findEmptyCell(grid, r, c)) return true;
  for (int n = 1; n <= 9; n++) {
    if (isSafe(grid, r, c, n)) {
      grid[r][c] = n;
      if (solveBacktracking(grid, moves)) return true;
      grid[r][c] = 0;
    }
  }
return false;
}
```

The Boolean function performs backtracking. At first it is checked if there is any empty cell in the input grid[9][9]. For this purpose findEmptyCell() function is used.

```
bool findEmptyCell(int grid[9][9], int &r, int &c) {
  for (r = 0; r < 9; r++) {
    for (c = 0; c < 9; c++) {
      if (grid[r][c] == 0) return true;
      }}
  return false;
}
```

Here the address of variable r and c are passed. From this function the empty cell position (r,c) has been known. Then a value from 1 to 9 has been assigned in the

empty cell. If this function returns false then the puzzle has been solved otherwise it needs to be solved. To check whether the assigned number is valid or not a function isSafe(grid, r, c, n) is used. The function takes the grid , position of the empty cell and the number(that is supposed to be assigned in the empty cell) as arguments.

This function returns true if the number is valid for the cell. The three constraints have been checked through the functions UsedInRow(), UsedInCol() and UsedInBox().

If the safe number is assigned in a particular cell then the function solveBacktracking() is called again until there is no empty cell and returns true. Once the Sudoku is solved, then the LCD displays are required to display the solution. For this purpose, three liquid crystal objects naming lcd1, lcd2 and lcd3 have been used. The 9×9 solved Sudoku is printed in the 16×2, 20×4 and 20×4 LCD display sequentially. The schematics of our solver is added below:

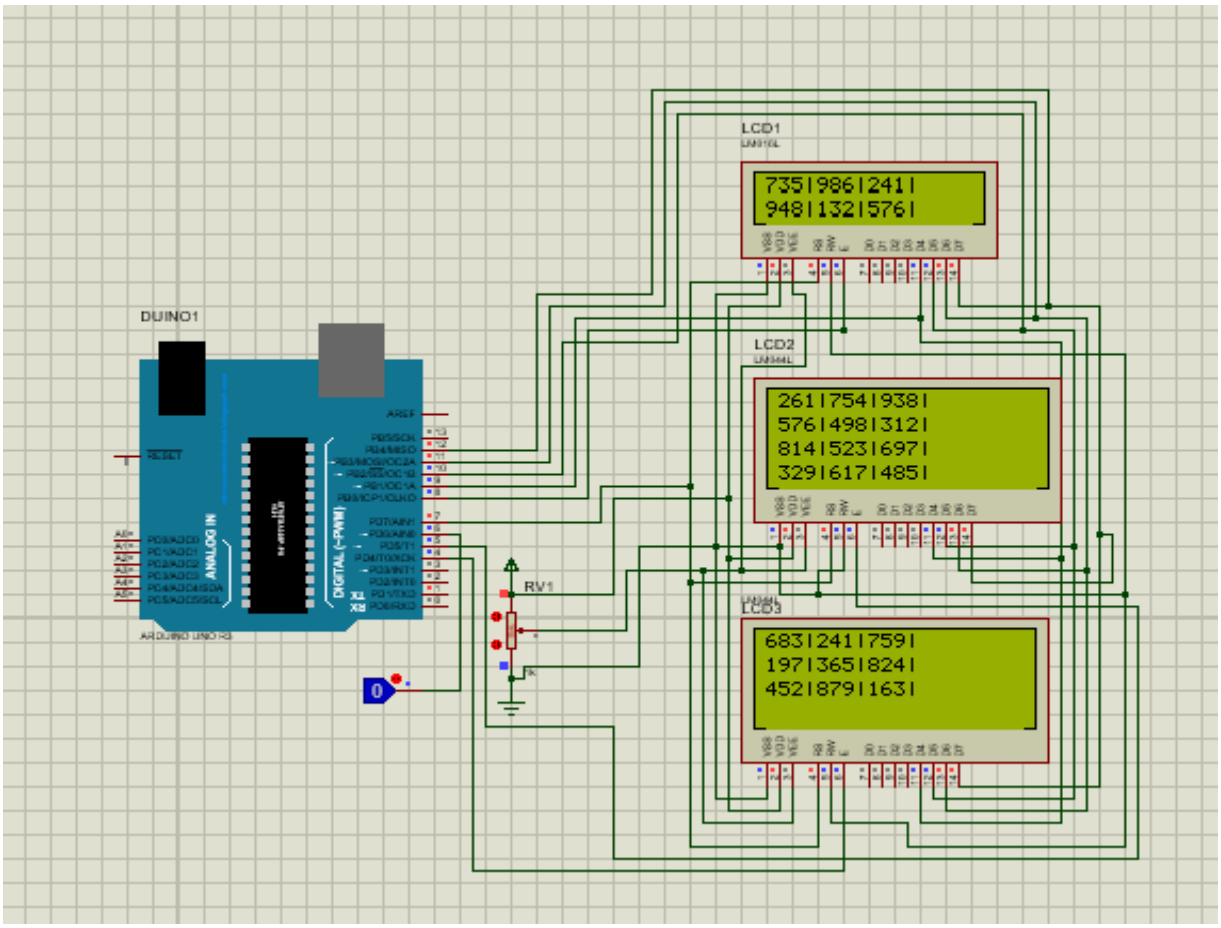

Figure 3: Snapshot of circuit diagram in proteus

Discussion:

The project has been executed in two different platforms. System Verilog based solver has been implemented in online EDA playground. The interface is user-friendly for all types of Verilog language. As the digital circuit implementation is quite complicated, microcontroller has been used in proteus 8.6 for simulation purpose. The algorithm incorporated in Arduino includes both simple and backtracking approaches that has eased the complexity for most Sudoku problems.

We have a plan to explore the problem further and solve the Sudoku with digital logic circuit.

References:

1. https://www.chipverify.com/systemverilog/systemverilog-tutorial?fbclid=IwAR1dkIfBh9NIu7HFZ2GDWgKCMGF1JyNF4r5qlsC_u43RR71JQ0PC_Vs6I4k

2. https://www.chipverify.com/systemverilog/systemverilog-constraint-examples?fbclid=IwAR1sY9sLohc2pSvj9Y16KLEZkcxafZnSyLXMZyurvuqAdEfZ8B6hWTMlvVk

3. https://www.geeksforgeeks.org/sudoku-backtracking-7/

4. https://create.arduino.cc/projecthub/gkentsidis/simple-sudoku-solver-002e52?fbclid=IwAR2FeK3N9GO7dJX4Jry7GHKloQQe3GsxFY_ActybriwJDf8E_tvLHzRKh7o

5. https://en.wikipedia.org/wiki/Sudoku

6. https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

Contribution Part:

1606075- Oishy Saha : Helped to develop the algorithm of Sudoku Solver in System Verilog and backtracking.

1606092- Barproda Halder: Implemented the solver circuit using arduino uno in Proteus and the code in arduino.

1606090- Jarin Tasnim: Designed the Sudoku solver in SystemVerilog, Modified the arduino code to show the input and output of Sudoku solver using multiple lcds.

Attachments:

Codes:

❖ System Verilog:

```
module top;
  parameter  M = 3;
  localparam N = M * M;
  int unsigned puzzle[N][N];
  sudoku#( M ) s;

  initial begin

    puzzle = '{ '{ 0,3,5, 0,0,0, 2,4,0 },
                '{ 0,0,0, 0,3,2, 0,0,0 },
                '{ 0,0,1, 0,0,0, 9,0,0 },

                '{ 0,7,0, 4,0,8, 0,0,0 },
                '{ 0,1,0, 0,2,0, 0,9,0 },
                '{ 0,0,0, 6,0,7, 0,8,0 },
```

```
                '{ 0,0,3, 0,0,0, 7,0,0 },
                '{ 0,0,0, 3,6,0, 0,0,0 },
                '{ 0,5,2, 0,0,0, 1,6,0 } };

    s = new;
    if ( s.solve_this( puzzle ) ) s.print();
    else $display( "cannot solve the puzzle" );
  end
endmodule: top

class sudoku#( int M ); // M >1
  localparam N = M * M;
  local int unsigned puzzle[N][N];
  rand  int unsigned box   [N][N];

  // The value of each cell must be between 1 and N.

  constraint box_con {
    foreach ( box[row, col] ) {
      box[row][col] inside { [ 1 : N ] };
    }
  }

// The cells on the same row must have unique values.
constraint row_con {
    foreach   ( box[row, colA] ) {
      foreach ( box[   , colB] ) {
        if ( colA != colB ) {
          box[row][colA] != box[row][colB];
        }
```

```
    }
  }
}

// The cells on the same column must have unique values.
constraint column_con {
  foreach   ( box[rowA, col] ) {
    foreach ( box[rowB,    ] ) {
      if ( rowA != rowB ) {
        box[rowA][col] != box[rowB][col];
      }
    }
  }
}

// The cells in the same MxM block must have unique values.

constraint block_con {
  foreach   ( box[rowA, colA] ) {
    foreach ( box[rowB, colB] ) {
      if ( rowA / M == rowB / M &&
           colA / M == colB / M &&
           ! ( rowA == rowB && colA == colB ) ) {
        box[rowA][colA] != box[rowB][colB];
      }
    }
  }
}
```

```
  // The box must have the same value as the puzzle's if
specified (!=0).

  constraint puzzle_con {
    foreach ( puzzle[row, col] ) {
      if ( puzzle[row][col] != 0 ) {
        box[row][col] == puzzle[row][col];
      }
    }
  }

  // Sudoku solver

  function int solve_this( int unsigned puzzle[N][N] );
    this.puzzle = puzzle;
    return this.randomize();
  endfunction: solve_this

  // Printing the solution.

  function void print();
    for ( int i = 0; i < N; i++ ) begin
      if ( i % M == 0 ) $write( "\n" );
      for ( int j = 0; j < N; j++ ) begin
        if ( j % M == 0 ) $write( " " );
        $write( "%3d", box[i][j] );
      end
      $write( "\n" );
    end
  endfunction: print
```

```
endclass: Sudoku
```

OUTPUT:

```
   7   3   5    9   8   6    2   4   1
   9   4   8    1   3   2    5   7   6
   2   6   1    7   5   4    9   3   8

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

   6   8   3    2   4   1    7   5   9
   1   9   7    3   6   5    8   2   4
   4   5   2    8   7   9    1   6   3
```

❖ Codes for Microcontroller

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(7, 8, 9, 10, 11, 12);
int receiver = 6;
int val;
#define DEBUG  true
#define SHOW_MOVES true
int col = 0;
int row = 0;
int posX = 3;
int posY = 0;
int grid[9][9]={{0, 3, 5,  0, 0, 0,  2, 4, 0},
                {0, 0, 0,  0, 3, 2,  0, 0, 0},
                {0, 0, 1,  0, 0, 0,  9, 0, 0},
                {0, 7, 0,  4, 0, 8,  0, 0, 0},
                {0, 1, 0,  0, 2, 0,  0, 9, 0},
```

```
                {0, 0, 0,  6, 0, 7,  0, 8, 0},
                {0, 0, 3,  0, 0, 0,  7, 0, 0},
                {0, 0, 0,  3, 6, 0,  0, 0, 0},
                {0, 5, 2,  0, 0, 0,  1, 6, 0}
                };
void setup() {
  Serial.begin(9600);
  pinMode(receiver,INPUT);
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  // Print a message to the LCD.
  printLine(0, "SUDOKU SOLVER");
  delay(2000);
  refreshLCD();
  lcd.blink();



}
void loop() {


  // set the cursor to column 0, line 1
  // (note: line 1 is the second row, since counting begins with
0):
  //lcd.setCursor(posX, posY);
  // print the number of seconds since reset:
  //lcd.print(millis() / 1000);
  val=digitalRead(receiver);
   switch(val) {
    case 1: solve(grid);    break;}
  delay(200);
```

```
}
void refreshLCD() {
  if (posY==0) {
    printLine(posY, gridLine(row));
    printLine(posY+1, gridLine(row+1));
  } else {
    printLine(posY-1, gridLine(row-1));
    printLine(posY, gridLine(row));
  }
  //lcd.setCursor(posX,posY);
}
String getValue(int v) {
  String s = String(v);
  if (v==0) {s=" ";}
  return s;
}
String gridLine(int row) {
  //String s = String(row+1) + " |";
  String s = getValue(grid[row][0]);
  s += getValue(grid[row][1]);
  s += getValue(grid[row][2]);
  s += "|";
  s += getValue(grid[row][3]);
  s += getValue(grid[row][4]);
  s += getValue(grid[row][5]);
  s += "|";
  s += getValue(grid[row][6]);
  s += getValue(grid[row][7]);
  s += getValue(grid[row][8]);
  s += "|";
```

```
  return s;
}
// Clear line r in lcd panel by printing spaces
// then rerurn the cursor to the first char of the line
void clearLine(int r) {
  lcd.setCursor(0,r);
  lcd.print("                ");
  lcd.setCursor(0,r);
}

// Clear line r and then print the text s
void printLine(int r, String s) {
  clearLine(r);
  lcd.print(s);
}
bool solve(int grid[9][9]) {


  if (solveGrid(grid)) {
    refreshLCD();
    return true;
  }

  int moves;
  if (solveBacktracking(grid, moves)) {
      printLine(0, "SUDOKU SOLVED");
      delay(2000);
      //printLine(0,"I had to use backtracking method");
      printLine(1, String(moves));
      delay(2000);
```

```
      refreshLCD();


       delay(2000);
      //printGrid(grid);
    }
  else {
    printLine(0, "NOT SOLVED");
    // SolveWithBruteForce(grid);
  };
  return false;
}
bool solveGrid (int grid[9][9]) {
  int values[9][9];       // Holds the predicted numbers for
each step: values[r][c]
  bool numbers[9][9][9];  // Holds the prediction for each
number in a cell: numbers[n][r][c]
  int empties = emptyCells(grid); // counts the empty cells
  int emptiesOld;
  int steps = 0;


  // clear the values table
  clearGrid(values);


  // loop until there are no empty cells
  while (empties>0) {
    emptiesOld = empties;
    steps++;
    // Fill the "numbers" table with predictions
    prepareCells(grid, numbers);


    checkCells(grid, numbers, values);
```

```
    copyValues(grid, values);
    empties = emptyCells(grid);
    if (empties == emptiesOld) {
      // Cannot solve SUDOKU
      return false;
    }
  }
  return true;
}

bool solveBacktracking (int grid[9][9], int &moves) {
  int r, c;

  moves++;
  // If there is no empty cell then we solved it!
  if (!findEmptyCell(grid, r, c)) return true;

  // test digits 1 to 9
  for (int n = 1; n <= 9; n++) {
    if (isSafe(grid, r, c, n)) {
      grid[r][c] = n;

      //return, if success
      if (solveBacktracking(grid, moves)) return true;

      // failure, unmake & try again
      grid[r][c] = 0;
    }
  }
```

```
  return false;
}
bool findEmptyCell(int grid[9][9], int &r, int &c) {
  for (r = 0; r < 9; r++) {
    for (c = 0; c < 9; c++) {
      if (grid[r][c] == 0) return true;
      }
    }
  return false;
}
void checkCells(int grid[9][9], bool numbers[9][9][9], int
values[9][9]) {
  for (int r=0; r<9; r++) {
    for (int c=0; c<9; c++) {
      if (grid[r][c]!=0) continue;    // If the cell has a value
then go to next cell
      if (values[r][c]!=0) continue;  // If the cell has a
prediction then go to next cell


      for (int n=0; n<9; n++) {
        // check if the prediction is the only one in the box
        if (values[r][c]==0 && uniqueInBox(numbers, n, r-r%3, c-
c%3) && numbers[n][r][c]) {
          values[r][c]=n+1;
          continue;
        }
        // check if the prediction is the only one in the row
        if (values[r][c]==0 && uniqueInRow(numbers, n, r) &&
numbers[n][r][c]) {
          values[r][c]=n+1;
          continue;
        }
```

```
        // check if the prediction is the only one in the column
        if (values[r][c]==0 && uniqueInCol(numbers, n, c) &&
numbers[n][r][c]) {
          values[r][c]=n+1;
          continue;
        }
        if (values[r][c]==0 && uniqueInCell(numbers, r, c) &&
numbers[n][r][c]) {
          values[r][c]=n+1;
          continue;
        }
      }
    }
  }
}
// Check if the given num exists in row
bool UsedInRow(int grid[9][9], int r, int num) {
  for (int c = 0; c < 9; c++) {
    if (grid[r][c] == num) return true;
  }
  return false;
}

// Check if the given num exists in column
bool UsedInCol(int grid[9][9], int c, int num) {
  for (int r = 0; r < 9; r++) {
    if (grid[r][c] == num) return true;
  }
  return false;
}
```

```
// Check if the given num exists in box
bool UsedInBox(int grid[9][9], int boxStartRow, int boxStartCol,
int num) {
  for (int r = 0; r < 3; r++) {
    for (int c =0; c < 3; c++)
      if (grid[r + boxStartRow][c + boxStartCol] == num) return
true;
  }
  return false;
}
// check if the given num is valid for the cell
bool isSafe(int grid[9][9], int r, int c, int num) {
  return !UsedInRow(grid, r, num) &&
    !UsedInCol(grid, c, num) &&
    !UsedInBox(grid, r - r % 3, c - c % 3, num) &&
    grid[r][c] == 0;
}

// check if the possible
bool uniqueInRow(bool numbers[9][9][9], int n, int r) {
  int count=0;
  for (int c = 0; c < 9; c++) {
    if (numbers[n][r][c]) { count++; }
  }
  return count==1;
}

bool uniqueInCol(bool numbers[9][9][9], int n, int c) {
  int count=0;
  for (int r = 0; r < 9; r++) {
    if (numbers[n][r][c]) { count++; }
```

```
  }
  return count==1;
}

bool uniqueInBox(bool numbers[9][9][9], int n, int startRow, int
startCol) {
  int count=0;
  for (int r = 0; r < 3; r++) {
    for (int c =0; c < 3; c++)
      if (numbers[r + startRow][c + startCol]) { count++; }
  }
  return count==1;
}

bool uniqueInCell(bool numbers[9][9][9], int r, int c) {
  int count=0;
  for (int n = 0; n < 9; n++) {
    if (numbers[n][r][c]) { count++; }
  }
  return count==1;
}

void copyValues(int grid[9][9], int values[9][9]) {
  for (int r=0; r<9; r++) {
    for (int c=0; c<9; c++) {
      if (values[r][c] != 0) { grid[r][c] = values[r][c]; }
      values[r][c] = 0;
    }
  }
}
```

```
void clearGrid(int grid[9][9]) {
  for (int r=0; r<9; r++) {
    for (int c=0; c<9; c++) { grid[r][c] = 0; }
  }
}
void prepareCells(int grid[9][9], bool numbers[9][9][9]) {
  for (int r=0; r<9; r++) {
    for (int c=0; c<9; c++) { setNumbers(r, c, grid, numbers); }
  }
}


void setNumbers(int r, int c, int grid[9][9], bool
numbers[9][9][9]) {
  for (int n = 0; n < 9; n++) {
    //if (grid[r][c] != 0) {numbers[n][r][c] = false; continue;}
    numbers[n][r][c] = isSafe(grid, r, c, n+1);
  }
}
int emptyCells(int grid[9][9]) {
  int i=0;
  for (int r=0; r<9; r++) {
    for (int c=0; c<9; c++) {
      if (grid[r][c] == 0) {i++;}
    }
  }
  return i;
}
```