CS1022 SUDOKU ASSIGNMENT DAVY NOLAN

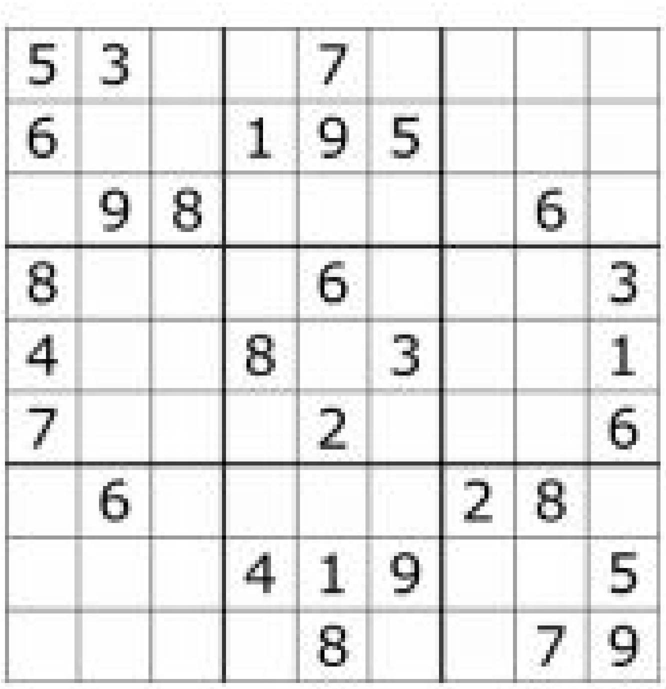
Davy Nolan

Sudoku

Introduction

Sudoku is a popular number puzzle game. You begin with a partially completed 9 x 9 grid of digits in the range from 1…9, as illustrated below. You must complete the grid subject to:

(a) Each digit appearing once in each row.
(b) Each digit appearing exactly once in each column.
(c) Each digit appearing exactly once in each 3x3 sub-grid.



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

Stage 1: Getting and Setting Digits

Aim:

Design and implement ARM Assembly Language subroutines to perform the following operations on a sudoku grid:

1. Get the value of a digit in the given row and column.

```
byte getSquare(word row, word column)
```

2. Set the value of a digit in the given row and column.

```
void setSquare(word row, word column, byte value)
```

Method:

getSquare subroutine:

To retrieve a certain element from a 3D array:

1. Translate 2D array index into 1D array index.

index = (row*rowSize) + column

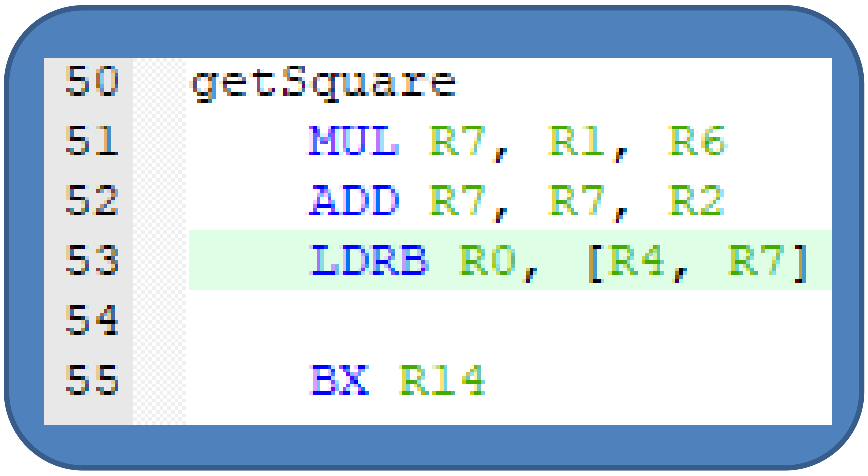
2. Translate 1D array index into byte offset from start address of array in memory.

Byte offset = index*elem size

3. Add byte offset to array base address to access element.

Elem address = array base address + byte offset

<u>Translated pseudocode into Assembly language:</u>

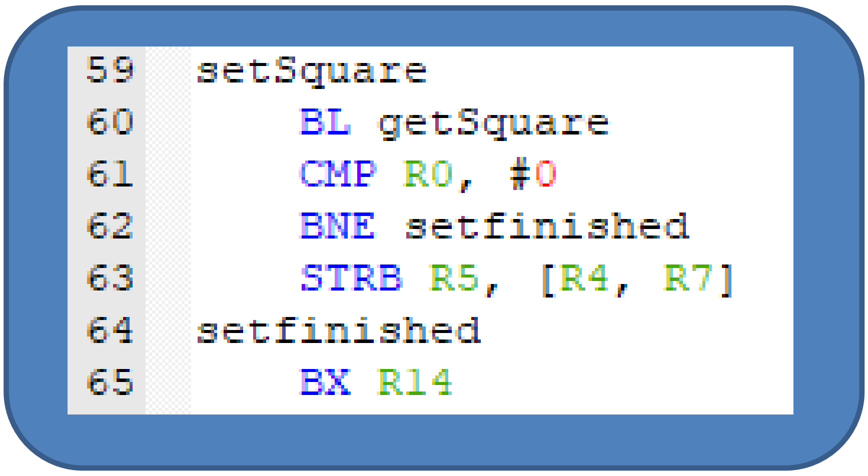


```
getSquare
	MUL R7, R1, R6
	ADD R7, R7, R2
	LDRB R0, [R4, R7]

	BX R14
```

setSquare Subroutine:

This subroutine makes use of the getSquare subroutine in order to find the specified square to set to a specified value. A square can only be set to another value if it contains the value 0.

Assembly Language:



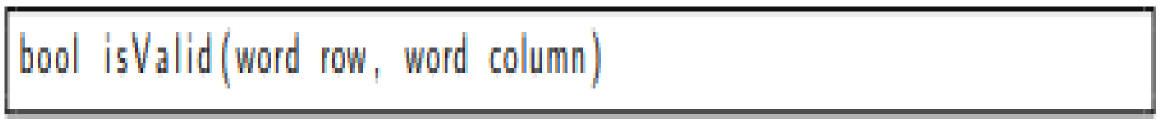
```
setSquare
        BL getSquare
        CMP R0, #0
        BNE setfinished
        STRB R5, [R4, R7]
setfinished
        BX R14
```

The subroutine compares the value that is already in that square to 0. If it is a value other than 0, the program branches to the end of the subroutine doing nothing. However, if the value in the square is 0, then the program stores whatever value is in R5 into the address of that square.

Stage 2: Validating Solutions

Aim:

Design and implement an ARM Assembly Language subroutine to determine whether a Sudoku grid represents a valid solution (or valid partial solution), based on the rules of Sudoku. A partial solution is valid if satisfies the rules of Sudoku, ignoring any remaining blank squares in the grid. The subroutine should return a true/false Boolean result i.e. 1/0.



```
bool isValid(word row, word column)
```

Method:

For a Sudoku grid to be a correct solution, all the values in any row must add up to 45, all the values in any column must add up to 45 and all the values in any 3x3 sub-grid must add up to 45.

Pseudocode:

```
public class SudokuChecker{

static int[][] sMatrix={

            {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}
        };
static int rSum=0;
static int cSum=0;

static int[] rSumArray=new int[9];

static int[] cSumArray=new int[9];

static int[] boxSumArray=new int[9];

static boolean checkArrayStatus(int[] rSumArray,int[] cSumArray,int[] boxSumArray)
{
   int i=0;

   boolean sudukoStatus=true;

   while(i<9){
      if(rSumArray[i]!=45&&cSumArray[i]!=45&&rSumArray[i]!=45)
      {
         sudukoStatus=false;
         break;
      }
      i++;
   }
   return sudukoStatus;
}

   public static void main(String[] args) {
      for(int i=0 ; i<sMatrix.length ; i++){
         for(int j=0 ; j<sMatrix.length ; j++){
            rSum+=sMatrix[i][j];
            cSum+=sMatrix[j][i];
            }
         rSumArray[i]=rSum;
         cSumArray[i]=cSum;
         rSum=0;
         cSum=0;
      }

      for(int i=0 ; i< sMatrix.length ; i++){
         for(int j=0 ; j<sMatrix.length ; j++){
            if(i<=2&&j<=2)
            {
               boxSumArray[0]+=sMatrix[i][j];
            }
            if(i<=2&&(j>=3&&j<=5))
            {
               boxSumArray[1]+=sMatrix[i][j];
            }
            if(i<=2&&(j>=6&&j<=8))
            {
               boxSumArray[2]+=sMatrix[i][j];
            }
            if((i>=3&&i<=5)&&(j<=2))
            {
               boxSumArray[3]+=sMatrix[i][j];
            }
            if((i>=3&&i<=5)&&(j>=3&&j<=5))
            {
               boxSumArray[4]+=sMatrix[i][j];
            }
            if((i>=3&&i<=5)&&(j>=6&&j<=8))
            {
               boxSumArray[5]+=sMatrix[i][j];

            }
            if((i>=6)&&(j<=2))
            {
               boxSumArray[6]+=sMatrix[i][j];
            }
            if((i>=6)&&(j>=3&&j<=5))
            {
               boxSumArray[7]+=sMatrix[i][j];
            }
            if((i>=6)&&(j>=6))
            {
               boxSumArray[8]+=sMatrix[i][j];
            }
         }
      }

   }
}
```

This program adds up all the values in every row, every column and every sub-grid and checks if they are equal to 45. If so, then sudukoStatus = true. This pseudocode was translated into ARM Assembly Language. The Boolean was stored in R11.

Next the subroutine had to compare each and every value in each row, column and sub grid with one another to make sure that there were no duplicates.

Pseudocode:

```
Boolean isValid = 0
while(rowSq1 < 9)
{               get value in square2
                while(colSq1 < 9)
                {
                        get value in square1
                        if(rowSq1 == rowSq2 && colSq1 == colSq2)
                        {
                                colSq1++
                        }
                        if(valueSq1 != valueSq2)
                        {
                                colSq1++
                        }
                }
                colSq2++
                colSq1 = 0
                if(colSq2 == 9)
                {
                        rowSq2++
                colSq2 = 0

                rowSq1++

                colSq1 = 0

                }

}
```

This pseudocode can be applied to both validating rows and validating columns. This pseudocode was then translated to Arm Assembly Language. The Boolean was stored in R12.

For example:
For a Sudoku solution grid that is filled out correctly –

```
R11(Boolean completeBoard) = 0x00000001

R12(Boolean partiallyCorrect) = 0x00000001
```

For a Sudoku solution grid that is incomplete but is correct so far –

```
R11 = 0x00000000

R12 = 0x00000001
```

For a Sudoku solution grid that is incomplete and incorrect so far –

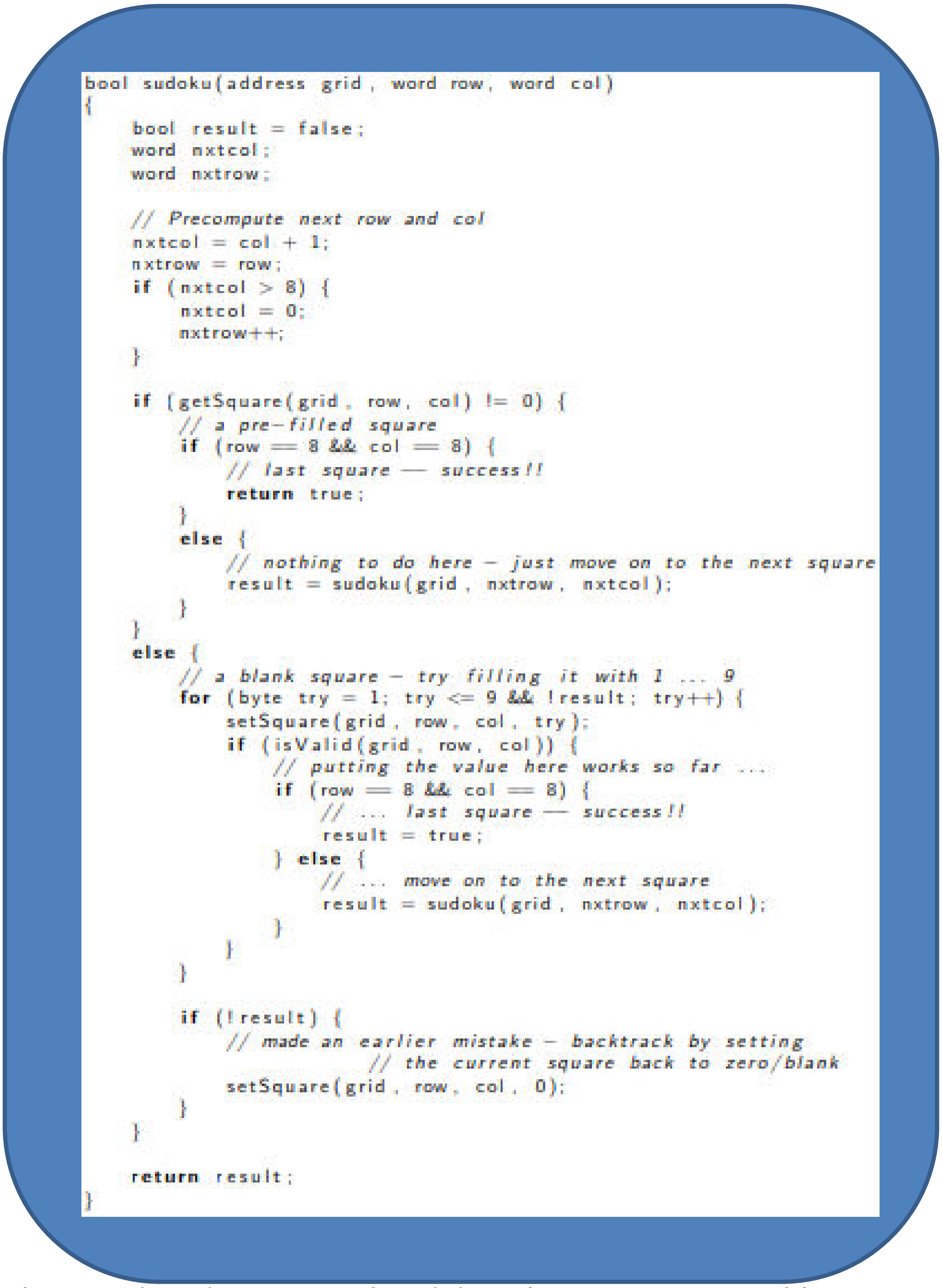
```
R11 = 0x00000000

R12 = 0x00000000
```

Stage 3: Solving a Sudoku Puzzle

Aim:

We can adopt a “brute force” approach to finding a solution to a Sudoku puzzle by iterating through the digits 1 ... 9 in the current blank square and, if a digit is valid, moving on to the next square and repeating. The above approach can be conveniently implemented using recursion, as illustrated in the pseudocode below. We recursively try to solve a Sudoku grid, beginning with a partially complete, valid grid. At some point we might realize that none of the digits 1 ... 9 are valid in the current square, because of some incorrect choice we made in an earlier square. This requires us to “backtrack” Page 2 of 4 School of Computer Science and Statistics CS1022 / Introduction to Computing II and try a different value in a previous square. Again, recursion is ideally suited to implementing a “backtracking” algorithm such as this, where each recursive invocation of the subroutine represents the state for a single square (i.e. which of the digits 1 ... 9) we are currently trying in that square.

Translate the pseudocode on the next page into an ARM subroutine.



```
bool sudoku(address grid, word row, word col)
{
    bool result = false;
    word nxtcol;
    word nxtrow;

    // Precompute next row and col
    nxtcol = col + 1;
    nxtrow = row;
    if (nxtcol > 8) {
        nxtcol = 0;
        nxtrow++;
    }

    if (getSquare(grid, row, col) != 0) {
        // a pre-filled square
        if (row == 8 && col == 8) {
            // last square — success!!
            return true;
        }
        else {
            // nothing to do here - just move on to the next square
            result = sudoku(grid, nxtrow, nxtcol);
        }
    }
    else {
        // a blank square - try filling it with 1 ... 9
        for (byte try = 1; try <= 9 && !result; try++) {
            setSquare(grid, row, col, try);
            if (isValid(grid, row, col)) {
                // putting the value here works so far ...
                if (row == 8 && col == 8) {
                    // ... last square — success!!
                    result = true;
                } else {
                    // ... move on to the next square
                    result = sudoku(grid, nxtrow, nxtcol);
                }
            }
        }

        if (!result) {
            // made an earlier mistake - backtrack by setting
                        // the current square back to zero/blank
            setSquare(grid, row, col, 0);
        }
    }

    return result;
}
```

The pseudocode was translated directly into ARM Assembly Language under the Sudoku subroutine. However, it did not appear to be working as there was a problem with registers overwriting one another as the other subroutines were called upon.

Stage 4: The Extra Mile

Aim:

Improve or extend the Sudoku solver in a manner of your choosing.

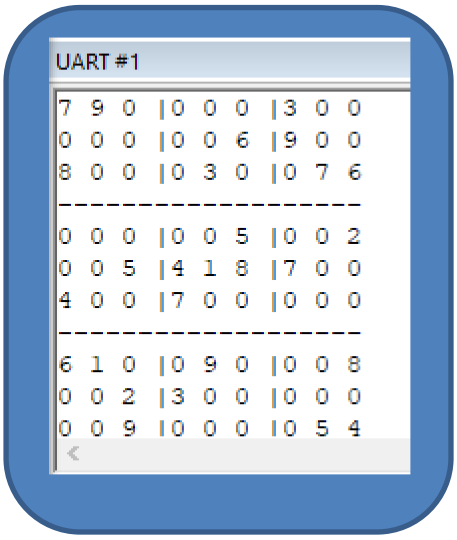
Method:

The extension I decided to make was another subroutine which displays the current Sudoku grid on to the console for the user to easily see and distinguish from the actual code.

The plan of action was to get each value from the Sudoku grid and print them to the console with spaces in between each value and with vertical and horizontal lines to visibly split up the sub-grids.

This was done using the BL sendchar subroutine.

The final result printed the sudoku grid successfully to the console like so.



```
7 9 0 |0 0 0 |3 0 0
0 0 0 |0 0 6 |9 0 0
8 0 0 |0 3 0 |0 7 6
-------------------
0 0 0 |0 0 5 |0 0 2
0 0 5 |4 1 8 |7 0 0
4 0 0 |7 0 0 |0 0 0
-------------------
6 1 0 |0 9 0 |0 0 8
0 0 2 |3 0 0 |0 0 0
0 0 9 |0 0 0 |0 5 4
```