Mauricio Mendoza

Xi Yang

11/18/15

Project Report

# Introduction

The topic of our project is a Sudoku puzzle solver. The importance of the project is that it uses in-depth search which allows the program to go back to a previous block if there is an error until the correct answer is in the block. This helps out, since there could be a quite large amount of combinations for the solution and our program enumerate all these combination in a very neat way. The Sudoku puzzle solver uses the UART which is required to input the data for it to solve the puzzle.

# Background

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective of the puzzle is to fill a 9x9 grid with digits so that each column, row, and of the nine 3x3 sub-squared contains the digits from 1 to 9. The research required to be completed before attempting the project is learn how to solve a Sudoku puzzle. The rule to Sudoku is that only one number can be used in each row, column, and 3x3 sub-grid. The difficulty of the Sudoku varies depending on the amount of data given. The smaller the number given in the beginning, the harder it is to solve it. The materials used to complete the project was java, ARM assembly, and websudoku.com. Java was used to insured that the algorithm would work correctly without any errors before using ARM. Afterwards, the code was convert from Java to ARM assembly in order to make use of UART. Lastly, we use websudoku.com to test the answer given by our program.

# Method

To solve an “evil” level Sudoku puzzle may be very hard for human, but it is relatively easy for computers. Since the number of ways to fill out a Sudoku puzzle is limited. For every box in the puzzle, there are nine possible numbers can be filled in. Further, the puzzle contains 81 such boxes. So, if each digit in grid is indepent, then the total number of ways to fill out a puzzle is 9^81. However, the digits in a Sudoku puzzle must satisfy column rule, row rule, and sub-grid rule. Since the available digit for a given box restricted by the state of the grid, the actual number of combination will be much smaller than in the case if the digits are independent. Therefore, we believe it is possible for a computing device to enumerate every possible combination to test the solution in a short time.

Our approach to solve a Sudoku puzzle is the brute force method, in which every possible way to filled out a puzzle is tested until a solution is found. For given “evil” level puzzle as show below, we first index each of the box for indicating them. We choose horizontal axis X and vertical axis Y and index them from 0 to 8 as shown in the picture.

0

1

2

3

4

5

6

7

8

X axis

0 1 2 3 4 5 6 7 8

Y axis

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

We chose to use depth-first search (DFS) to traverse all of the unknown boxes in order of top to bottom and left to right. In each unknown box we try every possible digit for that particular grid state and go to next unknown box to do the same. If all digits from 1 to 9 are tested for an unknown box and putting any of them in the box will violate the rule, we know that the current combination will not work. So we have to backtrack to the previous unknown box to try next possible number for the same state in which we gave a previous try. If we successfully fill a number in the last box, then we found the solution for this Sudoku puzzle. The reason we chose DFS is that it is intuitive for this problem and it can be easily implemented by recursive calls.

For example, we know that for a puzzle to be valid, each column (X axis), each row (Y axis), and each of the nine 3x3 sub-grids that compose the grid must contains all of the digits from 1 to 9. We choose the first unknown box in the order form top to bottom and from left to right. The first unknown box in the above picture is the box at position (0, 0), at the original state the first digit we can try is 3, since 1 and 2 is already in the same sub-grids. After we put 3 in the box(0, 0), the state of the grid changed and we go to next unknown box, which is box(1, 0). The first number we can try is 4, since 3 is just added in the same sub-grids by our first attempt. In the same way we fill out other unknown box in the order. Most probably we will encounter a dead end soon and we need to backtrack. When backtracking, we have to reset unknown to the tried out box where we go to backtrack. In this way, the previous state for the previous unknown box we have tried is brought back, so we can try the next digit for the new current unknown box.

We use an array to represent a Sudoku puzzle, a digit 0 in a box denotes that the box is unknown. We wrote a testing java program to prove our method. Since the program in high-level programming language is easy to debug. So, we can catch other errors that not caused by the algorithm. After the method is proved working, we started to implement it in ARM assembly.

The following is the java code for algorithm check, it is already written without using the features that assembly does not provide, such as 2D array and division.

**import** java**.**io**.**File**;**

**import** java**.**util**.**Scanner**;**

public class Final **{**

public static void main**(**String**[]** args**)** **throws** Exception **{**

Scanner fileScanner **=** **new** Scanner**(new** File**(**"Sudoku.txt"**));**

int**[]** sudoku **=** **new** int**[**81**];**

String line **=** fileScanner**.**nextLine**();**

int x **=** 0**;**

// getting Sudoku puzzle from the a file and

// store the puzzle in sudoku array

// 0 in a box means the box is unknown

**for** **(**int i **=** 0**;** i **<** 81**;** i**++)** **{**

sudoku**[**i**]** **=** Character**.**getNumericValue**(**line**.**charAt**(**x**));**

x**++;**

**if** **(**x **==** 9 **&&** fileScanner**.**hasNextLine**())** **{**

line **=** fileScanner**.**nextLine**();**

x **=** 0**;**

**}**

**}**

// call solve funtion to find the solution

// passing work positon 0, to start the searching

// when solution is found, true is returned and

// the answer is printed out, otherwise print "No solution"

**if** **(**solve**(**sudoku**,** 0**))** **{**

printSudoku**(**sudoku**);**

**}** **else** **{**

System**.**out**.**println**(**"No solution"**);**

**}**

fileScanner**.**close**();**

**}**

// recursive call to find a solution for a puzzle stored in sudoku array

// the current working index is i

// return true is solution is found

private static boolean solve**(**int**[]** sudoku**,** int i**)** **{**

// base case for recursion

// if the working position 81 is being call

// we have been successfully fill all the box from index 0 to 80

// then we found the solution

**if** **(**i **>=** 81**)** **{**

**return** **true;**

**}** **else** **{**

// get the next working postion

int newI **=** i **+** 1**;**

// at the current positon, the value for the box is already give

// skip this position, recursive call to work on the next working position

**if** **(**sudoku**[**i**]** **!=** 0**)** **{**

**return** solve**(**sudoku**,** newI**);**

**}** **else** **{**

// the current positon is unknown

// get the corresponding X and Y coordinate

int cellY **=** getY**(**i**);**

int cellX **=** getX**(**i**,** cellY**);**

// loop to try all digits from 1 to 9

**for** **(**int checkNum **=** 1**;** checkNum **<** 10**;** checkNum**++)** **{**

// if the digits does not violate the rule

**if** **(**checkSquare**(**sudoku**,** cellX**,** cellY**,** checkNum**)** **&&** checkRow**(**sudoku**,** cellY**,** checkNum**)**

**&&** checkCol**(**sudoku**,** cellX**,** checkNum**))** **{**

// fill the tried number in the sudoku array

sudoku**[**i**]** **=** checkNum**;**

// recursive call to working on next position

**if** **(**solve**(**sudoku**,** newI**))** **{**

**return** **true;**

**}**

**}**

**}**

// after the loop there is noway to find a digit without violating the rule

// undo the changes to the current positon by resetting 0 to current position

sudoku**[**i**]** **=** 0**;**

// no solution can be found, goting the back to previous call to try next digit

**return** **false;**

**}**

**}**

**}**

// get X coordinate

private static int getX**(**int i**,** int y**)** **{**

**return** **(**i **-** y **\*** 9**);**

**}**

// get Y coordinate

private static int getY**(**int i**)** **{**

**if** **(**i **<=** 8**)** **{**

**return** 0**;**

**}** **else** **if** **(**i **<=** 17**)** **{**

**return** 1**;**

**}** **else** **if** **(**i **<=** 26**)** **{**

**return** 2**;**

**}** **else** **if** **(**i **<=** 35**)** **{**

**return** 3**;**

**}** **else** **if** **(**i **<=** 44**)** **{**

**return** 4**;**

**}** **else** **if** **(**i **<=** 53**)** **{**

**return** 5**;**

**}** **else** **if** **(**i **<=** 62**)** **{**

**return** 6**;**

**}** **else** **if** **(**i **<=** 71**)** **{**

**return** 7**;**

**}** **else** **{**

**return** 8**;**

**}**

**}**

// is digit toCheck valid at (reqX, reqY) in its sub-grid

private static boolean checkSquare**(**int**[]** sudoku**,** int reqX**,** int reqY**,** int toCheck**)** **{**

// colX and rowY are the coordinate of the top-left corner in the sub-grid

int rowY**;**

int colX**;**

**if** **(**reqX **<** 3**)** **{**

colX **=** 0**;**

**}** **else** **if** **(**reqX **<** 6**)** **{**

colX **=** 3**;**

**}** **else** **{**

colX **=** 6**;**

**}**

**if** **(**reqY **<** 3**)** **{**

rowY **=** 0**;**

**}** **else** **if** **(**reqY **<** 6**)** **{**

rowY **=** 3**;**

**}** **else** **{**

rowY **=** 6**;**

**}**

// the 1D index of the top-left corner in the sub-grid

int i **=** colX **+** rowY **\*** 9**;**

int k **=** 0**;**

// the loop traverse all elements in the sub-grid

**for** **(**int j **=** 0**;** j **<** 9**;** j**++)** **{**

k**++;**

**if** **(**sudoku**[**i**]** **==** toCheck**)** **{**

**return** **false;**

**}**

**if** **(**k **==** 3**)** **{**

k **=** 0**;**

i **=** i **+** 7**;**

**}** **else** **{**

i **=** i **+** 1**;**

**}**

**}**

**return** **true;**

**}**

// is digit toCheck valid in rowY

private static boolean checkRow**(**int**[]** sudoku**,** int rowY**,** int toCheck**)** **{**

// the index of the left most element in rowY

int i **=** rowY **\*** 9**;**

// the loop traverse all elements in rowY

**for** **(**int x **=** 0**;** x **<** 9**;** x**++)** **{**

**if** **(**toCheck **==** sudoku**[**i**])** **{**

**return** **false;**

**}**

i**++;**

**}**

**return** **true;**

**}**

// is digit toCheck valid in colX

private static boolean checkCol**(**int**[]** sudoku**,** int colX**,** int toCheck**)** **{**

// the loop traverse all elements in colX

**for** **(**int y **=** 0**;** y **<** 9**;** y**++)** **{**

**if** **(**toCheck **==** sudoku**[**colX**])** **{**

**return** **false;**

**}**

colX **=** colX **+** 9**;**

**}**

**return** **true;**

**}**

private static void printSudoku**(**int sudoku**[])** **{**

int x **=** 0**;**

**for** **(**int i **=** 0**;** i **<** 81**;** i**++)** **{**

System**.**out**.**print**(**sudoku**[**i**]);**

x**++;**

**if** **(**x **==** 9**)** **{**

System**.**out**.**println**();**

x **=** 0**;**

**}**

**}**

**}**

**}**

# Implementation

The first difficulty we encountered is that there is no 2D array in ARM assembly. So, we have to use 1D array to express the grid. Still we index the grid from 0 in the order of top to bottom and left to right. Since 2D coordinate is still need to determine the column, the row, and the 3x3 sub-grid a box belong to. We added GetX and GetY function to calculate the 2D coordinate from 1D array. The math in this conversion is relatively easy, Y = 1D\_index / 9 (integer division) and X = 1D\_index % 9. However, the division and mod is expensive operation, so we calculate Y by using condition test. If index <= 8, then Y = 0, else if index <=17, then Y = 1, and so on. Then X will be calculated from Y, X = 1D\_index – Y \* 9.

The second difficulty we encountered is that the code becomes extremely hard to read and maintain. The label name started to conflict with each other and the parameter passing becomes unknown. To overcome this, aside from using the ARM application procedure call standard, we use uppercase initial to indicate a subroutine label and lowercase initial to indicate a branch label. Further, we put the lowercase acronym of the subroutine name in the beginning of the branch label to distinguish branches in one subroutine from those in other subroutines. We also comment every subroutine by proving the purpose and parameter passing the subroutine.

The third difficulty we encountered is the hardness to debug assembly code. Although we wrote the code very carefully, a few bugs still made some trouble for us to pin them down. One of the bug is in the loop from ASCII 0 to ASCII 9. We wrote the loop form 0x30 to 0x40, a mistake missed up decimal and hexadecimal. The correct loop should be form 0x30 to 0x3A.

In the next section we will go through the main routine and some of the subroutines. Since the report for basic UART routines, such as initialization, display a string, and save user input to table, are already written in the previous lab report, we will not include them in this project report. We will only show the new written code here.

entry

start

ldr sp, **=** stackstart ; set up stack pointer

bl UARTConfig ; initialize**/**configure UART1

ldr r1, **=** string1 ; starting address of CharData1

bl Display ; display CharData1

ldr r4, **=** ramstart ; r4 stores the address of the sudoku question

mov r1, r4

bl SaveInput

ldr r1, **=** opa ;debug **for** input length

str r2, **[**r1**]** ;debug **for** input length

cmp r2, #81 ; section for checking sudoku question validity

blne WrongLength

mov r1, r4

bl CheckInput

mov r1, r4

mov r2, #0

bl Solve

cmp r0, #1

blne NoSolution

mov r2, r4

bl DisplaySol

done b done

The above code is the main routine. It sets up stack pointer, initialize UART, and display prompt for user input. Then it saves user input as a 0 terminated string stored in address r4. It also verify the user input by checking the length of the input string and making sure each character in the string is in the range form ‘0’ to ‘9’. If a wrong input is found, the program output the error and terminate. Otherwise it calls ‘Solve’ subroutine to find the solution. If the solution is found, it calls ‘DisplaySol’ to output the solution. Otherwise, it calls ‘NoSolution’ to output “No solution!”.

Display1 ; display the string stored at address r1, output a space **in** every 3 character

; parameter r1, the address of the string to be displayed

push **{**lr**}**

mov r2, #-1

dp1start add r2, r2, #1

cmp r2, #3

bne skipspace

mov r2, #0

mov r0, #0x20 ; output a space

bl Transmit

skipspace ldrb r0, **[**r1**]**, #1 ; load character, increment address

cmp r0, #0 ; null terminated?

blne Transmit ; send character to UART

bne dp1start ; continue **if** not a '0'

pop **{**pc**}**

The above code is used to output the Sudoku solution. It is a little different from the normal Display subroutine that is used to display strings. It insert a space in every 3 output character, so that the solution of the Sudoku can be easily read.

SaveInput ; save the input string at address r1 and return the length of the string **in** r2

; parameter r1, the address of the saved input string

; return r2, the length of the string

push **{**lr**}**

mov r2, #0

sistart bl Receive

bl Transmit

cmp r0, #13

addne r2, r2, #1

strbne r0, **[**r1**]**, #1

bne sistart

mov r0, #0

strb r0, **[**r1**]**

pop **{**pc**}**

This code is used to store the input Sudoku problem to a string in memory. It is also a little different from the normal user input save subroutine. In addition to save the string, it also returns the length of the input string. So the main routine use this length to check if the input is valid.

CheckInput ; check the input string at address r1

push **{**lr**}**

ciloop ldrb r2, **[**r1**]**, #1

cmp r2, #0

beq cidone

cmp r2, #0x30

bllt WrongData

cmp r2, #0x39

blgt WrongData

b ciloop

cidone pop **{**pc**}**

The ‘CheckInput’ subroutine tests each element in the string to make sure that they are in range from ‘0’ to ‘9’.

Solve

; recersive call to solve the sudoku puzzle

; parameter r1, the address of the sudoku table

; parameter r2, the current working index of the table

; return r0 **=** 1, found a solution

; return r0 **=** 0, no solution

push **{**r4**-**r9, lr**}**

mov r4, r1 ; the address of the sudoku table

mov r5, r2 ; the current wokring index

cmp r5, #81

beq strue

add r6, r5, #1 ; the next wokring index

ldrb r7, **[**r4, r5**]** ; the digit at current index

cmp r7, #0x30

beq sguess

mov r1, r4 ; preparing call Solve on **next** index

mov r2, r6

bl Solve

b sdone

sguess mov r1, r5 ; preparing call GetY

bl GetY

mov r7, r0 ; the current Y index

mov r1, r5 ; preparing call GetX

mov r2, r7

bl GetX

mov r8, r0 ; the current X index

; check loop

mov r9, #0x31 ; guess number

sloop mov r0, r4 ; preparing call CheckSquare

mov r1, r8

mov r2, r7

mov r3, r9

bl CheckSquare

cmp r0, #0

beq sloopupdate

mov r1, r4 ; preparing call CheckRow

mov r2, r7

mov r3, r9

bl CheckRow

cmp r0, #0

beq sloopupdate

mov r1, r4 ; preparing call CheckCol

mov r2, r8

mov r3, r9

bl CheckCol

cmp r0, #0

beq sloopupdate

strb r9, **[**r4, r5**]**

mov r1, r4 ; preparing call Solve on **next** Index

mov r2, r6

bl Solve

cmp r0, #1

beq strue

sloopupdate add r9, r9, #1

cmp r9, #0x3A

blt sloop

;after loop

mov r1, #0x30

strb r1, **[**r4, r5**]**

mov r0, #0

b sdone

strue mov r0, #1

sdone pop **{**r4**-**r9, pc**}**

The ‘Solve’ subroutine is the central part of the algorithm. It recursively calls itself until a solution is found or every possible combination is tried without finding a solution. The code is in parallel with our java test code.

GetY ; find Y index of the current index

; parameter r1, the current index to be translated

; return r0, the corresponding Y index

push **{**lr**}**

cmp r1, #8

bgt getytest2

mov r0, #0 ; i <= 8

b getydone

getytest2 cmp r1, #17

bgt getytest3

mov r0, #1 ; i <= 17

b getydone

getytest3 cmp r1, #26

bgt getytest4

mov r0, #2 ; i <= 26

b getydone

getytest4 cmp r1, #35

bgt getytest5

mov r0, #3 ; i <= 35

b getydone

getytest5 cmp r1, #44

bgt getytest6

mov r0, #4 ; i <= 44

b getydone

getytest6 cmp r1, #53

bgt getytest7

mov r0, #5 ; i <= 53

b getydone

getytest7 cmp r1, #62

bgt getytest8

mov r0, #6 ; i <= 62

b getydone

getytest8 cmp r1, #71

movle r0, #7 ; i <= 71

ble getydone

mov r0, #8 ; else

getydone pop **{**pc**}**

GetX ; find X index of the current index

; parameter r1, the current index to be translated

; parameter r2, the current Y index

; return r0, the corresponding X index

push **{**lr**}**

add r2, r2, r2, lsl #3 ; Y \* 9

sub r0, r1, r2 ; i **-** Y **\*** 9

pop **{**pc**}**

The ‘GetY’ and ‘GetX’ subroutine is used to get Y and X coordinate, respectively. They are in parallel with java test code.

CheckSquare ; check **if** the current guess number satisfies the little square

; parameter r0, the address of the sudoku table

; parameter r1, the X index

; parameter r2, the Y index

; parameter r3, the current guess number

; return r0 **=** 0, not satisfied

; return r0 **=** 1, satisfied

push **{**r4**-**r7, lr**}**

cmp r1, #2

bgt csxtest1

mov r4, #0 ; X index of the top-left corner of the little square when real X <= 2

b csytest

csxtest1 cmp r1, #5

movle r4, #3 ; real X <= 5

movgt r4, #6 ; real X else

csytest cmp r2, #2

bgt csytest1

mov r5, #0 ; Y index of the top-left corner of the little square when real Y <= 2

b cscalindex

csytest1 cmp r2, #5

movle r5, #3 ; real Y <= 5

movgt r5, #6

cscalindex add r5, r5, r5, lsl #3 ; Y \* 9

add r6, r4, r5 ; X **+** Y **\*** 9 index of the top**-**left corner **in** sudoku table

mov r1, #0 ; index k

mov r2, #0 ; index j

csloop add r1, r1, #1

ldrb r7, **[**r0, r6**]**

cmp r7, r3

beq csfalse

cmp r1, #3

moveq r1, #0

addeq r6, r6, #7

addne r6, r6, #1

add r2, r2, #1

cmp r2, #9

blt csloop

mov r0, #1

b csdone

csfalse mov r0, #0

csdone pop **{**r4**-**r7, pc**}**

CheckRow ; check **if** the current guess number satisfies the row

; parameter r1, the address of the sudoku table

; parameter r2, the Y index

; parameter r3, the current guess number

; return r0 **=** 0, not satisfied

; return r0 **=** 1, satisfied

push **{**r4**-**r5, lr**}**

lsl r4, r2, #3

add r4, r4, r2 ; check index

mov r0, #0

crloop ldrb r5, **[**r1, r4**]**

cmp r5, r3

beq crfalse

add r4, r4, #1

add r0, r0, #1

cmp r0, #9

blt crloop

mov r0, #1

b crdone

crfalse mov r0, #0

crdone pop **{**r4**-**r5, pc**}**

CheckCol ; check **if** the current guess number satisfies the column

; parameter r1, the address of the sudoku table

; parameter r2, the X index

; parameter r3, the current guess number

; return r0 **=** 0, not satisfied

; return r0 **=** 1, satisfied

push **{**r4, lr**}**

mov r0, #0

ccloop ldrb r4, **[**r1, r2**]**

cmp r4, r3

beq ccfalse

add r2, r2, #9

add r0, r0, #1

cmp r0, #9

blt ccloop

mov r0, #1

b ccdone

ccfalse mov r0, #0

ccdone pop **{**r4, pc**}**

The ‘CheckSquare’, ‘CheckRow’, and ‘CheckCol’ are used to test if a digit is valid in given position according to sub-grid rule, row rule, and column rule, respectively. They are also parallel with the java test code.

# Testing and Demo

We tested each new subroutine added to the program to make sure there are bug free. Then we do integrate test to make sure the subroutine is correctly called and the correct value is returned. In this way, if a bug shows up, most probably it is in the newly added code. The final code is tested on many problems from websudoku.com, and the program passed every test.

For the problem in figure under method section, input the puzzle in order of top to bottom and left to right using 0 indicate unknown box as shown below.



Press enter, the program gives the solution showed in the same order.



Fill the solution to the original question:

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

# Conclusion

The assembly language has same power as any high-level imperative programming language. In fact many code written by high-level programming language will be compile to assembly code first. After implementing the algorithm for solving a Sudoku puzzle in ARM assembly, we are proficient in using subroutines. We also gained many experience in writing long assembly code. In the future, we will be able to implement even more complex algorithm in assembly.