CS 1022 Assignment #2
Sudoku

Report by
Samuel Petit

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

March 2018

Contents

Part 1
Getting and setting digits

1.1 Approach

This first part's focus was accessing a value inside a 2-dimensional array, which was covered during a tutorial session. Therefore, this was quite straightforward. Things to consider were: the size of each numbers (Byte: we then use LDRB/STRB) as well as the method to point at a specific value. This is done using the following formula:

Address = Base Address + (9 * row) + column
Points to
Grid[row][column]

1.2 Testing method

The way I made sure these two methods worked properly was through the 'Memory' window as well as the registers.
The getSquare subroutine:
I tested the program to point at different indexes and compared the final value in the return register (R0) with the value in the grid to make sure the method computed to correct position.
The setsquare subroutine:
I tested this by setting different indexes and checking the memory addresses where the grid is stored. Making sure the value was correctly stored and correctly positioned.

One thing to note is that I tested the different 'ends' of the grid (i.e. if the program will work as well with values positioned at the start/end of a column and/or a row).

Part 2
Validating solutions

2.1 Checking if a square is valid

This subroutine was divided into 3 different subroutines for a cleaner and easier to understand program. The purpose of this subroutine is to return whether there is a conflict when looking at a specific square. Checking its row, column and sub-square he is positioned in.
One thing that caused me a lot of trouble was that I didn't realise the registers used for parameters were not restored after each subroutine call/return. Therefore, one thing I had to be careful about is storing the parameters not once for all subroutine calls but before each subroutine call as these will change some parameters values as they call different subroutine themselves:

Don't do:

```
LDR R1, = 0;
LDR R2, = 1;
BL checkrow
BL checkcol
BL checksquare
```

But:

```
LDR R1, = 0;
LDR R2, =1;
BL checkrow
LDR R1, =0
LDR R2, =1 .........
```

Check Row subroutine:
Is a subroutine that goes through all of the numbers of a given row. It will return true if all the values are unique. Ignore non-filled squares (0) and return false if there is a conflict (if two values are the same).
To loop through the values here is the pseudo code I have decided to use (where row is the given row to checked, passed as a parameter):

```
for(int a = 0; a < 8; a++){
    for(int b = a + 1; b < 9; b++){
        if(getSquare(row, a) == getSquare(row, b)) return false}
} Return true;
```

Check Column subroutine:
Is the same principle as check row but checking for non-unique values of a given column instead of a row.
Check square subroutine:
Is the subroutine that will return if the sub-grid in which the given square is valid or not. This subroutine was divided into three parts.

The first part was getting the position of the square at the top left corner of the sub-grid to make sure we are checking the correct values. The way I have achieved this is using the following pseudo code:

```
If(row == 0 || row == 3 || row == 6) row = row;
Else if(row == 1 || row == 4 || row == 7) row = row -1;
Else row = row – 2;
```

The same pseudo code is used to adjust the column's value as well.

The second part is pushing all of the 9 values of the sub-grid on to the stack pointer for easier comparison (instead of using more than 2 nested for loops).

Finally, the third and final part is to compare the values now stored on the system stack and pushing those values afterwards. We can do this using two nested for loops as done above :

```
for(int a = 0; a < 8; a++){
    value1 = memory[sp + a * 4];
for(int b = a + 1; b < 9; b++){
    value2 = memory[sp + b * 4];
    if(value1== value 2)
        Sp = sp + 4* 9
        return false;}
    } Sp = sp + 4* 9
    Return true;
```

2.2 Testing method

At first this subroutine did not work perfectly : as mentioned above. The way I understood the problem was by setting breakpoints at the return of each of the three subroutines used in the isValid subroutine. I then noticed that the passed row and column parameter passed after the first return from the subroutine were incorrect : the final value was a mix a checking row X with column Y and sub-grid Z.
Once this was fixed, I tried different grid with different scenarios (checking if the algorithm does check for all of the values, for example, doesn't forget about the first or last value)/

Part 3
Solving a sudoku grid

3.1 Translating pseudo-code: the brute force method

This part won't require much explanation as it consists of translating a given pseudo code into assembly language. To do this I adopted a line-by-line method.

3.2 Testing method

After having finished translating the code for this subroutine I ran it and realised after I looked at the memory address of the grid that nothing had changed and a value "false" was returned. This was because I had forgotten about the '!result' condition on the for loop :

for(byte i = 1; i <= 9 && <u>!result</u>; i++)

I figured this was a problem using different breakpoints and eventually fixed it. I tried using different grids and checked if the output grids were correct (using an algorithm which looped through all the values and returned whether all of the values were 'valid' or not, which I did not leave in the final code) .

Part 4
The extra mile

4.1 Displaying the grid

One of the extra features I added to this program is displaying a given grid. The subroutine takes no parameters (the address of the grid is loaded from inside the subroutine) and uses an approach similar to the 'checkrow' subroutine. That is, going to through each of the rows and (in this case displaying instead of checking each of the columns of that specific row). Then, things to consider were : separating each value by a space for better readability, going to the next line for each of the rows, converting each value to ASCII values (done by adding '0' to the given value).
Here is the pseudo code which illustrates my approach :

```
For(int row = 0; row < 9; row++){
    printRow(row);
    print(new Line);
}
Print(new Line);

Void printRow(int displayRow){
Address = baseAddress + 9 * displayRow;
For(int col = 0; col < 9; col++){
Print(getSquare(displayRow, col) + '0');
Print(" "); }
}
```

4.2 Testing method

Testing this was quite straightforward : I compared the output in the console to the input grid (at the bottom of the program).

4.3 Text display

Another thing I added is displaying before and after messages in the console to help the user understand what's happening : at first the original grid is displayed (with a message explaining that the grid being printed is the original one). The program will then execute the subroutine 'sudoku', trying to solve that grid. And produce a solution. Is then printed on the console a message explaining that the final solution is being printed.
To do this I stored two null terminated string and used a while loop to display the text :

```
Address = baseAddress (a parameter)
Char = 1
i=0
While(char != 0){
Char = memory.byte[address + i];
i++;
print(char);
}
```

4.4 Testing method

Again, testing this subroutine was quite simple, I only checked if the algorithm correctly looped through all of the characters and compared the output in the console to the expected output (stored in an array in memory, under the grids declarations).

4.7 Adding functionality to the sudoku subroutine: Finding unique solutions

The final addition to my program was checking if a grid's solution is unique or not. To do this I had to pass two extra parameters (which I did using the system stack) and here is the logic behind it:

- A program will try brute forcing the grid until it finds a solution
- If a solution is found the current grid is copied into a blank grid. The program will then backtrack and try to look for another solution
- If a second solution is found. The program finishes, the copied grid is displayed and a text mentioning the grid has multiple solutions.
- Otherwise, the program will try every single combination of numbers until it returns. The unique solution is displayed via the copied grid. And a text is displayed mentioning the grid's unique solution.

Although this adds a nice functionality to the program, it has to be mentioned that, depending on the grids, the time required to finish computing solutions can be much longer.

4.8 Testing method

The way I tested this code worked was by using grids I found online that have a unique solution and others which don't. Using breakpoints to analyse if my program behaved the way it is supposed to.

Part 5
My testing code (using extra miles)

Here is the code that I have written to test my program's subroutine, to see it in action, copy and paste it into the mainline.
Note : I have also included the provided test lines with the extra parameters required inside of my .s file.

Test ARM code :

```
        LDR R1,= displayTextNonSolved; displayText(nonSolved)
        BL displayText;
        LDR R1,= multiSolutionGrid;          change to uniqueSolutionGrid for a unique grid
        BL printGrid                         ; printGrid(gridAddress)
        MOV    R1, #0
        MOV    R2, #0
        LDR R0,= multiSolutionGrid;  change to uniqueSolutionGrid for a unique grid
        LDR R10,= 0;
        LDR R11,= 0;
        STMFD sp!, {R10, R11}
        BL sudoku    ;sudoku(address, row, column, Boolean : unique, boolean : isNotUnique)
        ADD SP, SP, #8;
        MOV R3, R0                                  ;save (boolean result)
        MOV R4, R1                                  ; save boolean(unique)
        LDR R1,= displayTextFinal    ; displayText(finalGrid)
        BL displayText;
        LDR R1, = finalGrid
        BL printGrid                         ; displayFinalGrid
        LDR R0, =10;                 print(new line)
        BL sendchar

        CMP R4, #1;                  if(unique) display(uniqueSolutionString)
        BNE notUniqueText
        LDR R1, = uniqueSolution
        B uniqueIsFalse
notUniqueText                        ;       else display(multipleSolutionsString);
        LDR R1, = multipleSolutions
uniqueIsFalse
        BL displayText;
```