CS1022 Computing Assignment #1 Report.

Part (1). [10%] Getting and Setting digits.

This part of the assignment needed two subroutines to be written, getSquare(), and setSquare().

* byte getSquare(grid, row, col)

**int** getSquare(**int**[][] grid,**int** row, **int** col) {

**return** grid[row][col];

}

I wrote my subroutine by this pseudocode, the inputs would be the start address of the sudoku grid, a desired col and row, the return would be the digit stored in that square of the 2D array that is the sudoku grid.



I used the system stack to save save, and then restore certain registers in this case most importantly R1, and R2 which are the row and col registers, being used everywhere in the program, to find the desired square I used the formula

* (GRID\_SIZE \* row ) + col

This found the position in the sudoku grid, then I stored it by using LDRB, and using a scaled offset of LSL #0 (1) multiplied by the position. I used r9 as my output register for getSquare();

* void setSquare(grid, row, col, value)

**public** **static** **void** setSquare(**int**[][] board, **int** row, **int** col,**int** value) {

board[row][col] = value;

}

setSquare() doesn't return anything it, only changes memory on the arm, it is quite similar to getSquare(), it uses the mechanic described above, however instead of a LDR instruction it STRs a byte at the position found by the row and col, storing a value passed in.



Part (2). [25%] Validating Solutions.

The final point of this section was to write a subroutine that checked whether a particular point in the grid was valid.

* boolean isValid(grid, row, col);

I decided to split this questions into 4 subroutines, as I feared that I would run out of registers to use, I split up the main isValid into 3 subroutines:

* boolean checkRow(grid, row, col)
* boolean checkCol(grid, row, col)
* boolean check3x3(grid, row, col)

And each of those would use a:

* boolean isUnique(array);

Together these subroutines break down isValid() into a managable form for me.

**public** **static** **boolean** isValid(**int**[][] board, **int** row, **int** col) {

**if**(*checkRow*(row, board) && *checkCol*(col,board) && *threeCheck*(row,col,board)) {

**return** **true**;

}

**return** **false**;

}

This is how the isValid subroutine was implemented by translating the pseudocode into assembly.



I used r12 as the final result for each of the three checker subroutines, and made a triple && condition with each of them, if all three functions return true (1), then isValid() itself returns a one, and else it returns false (0)

Each checker subroutine had a similar principle of operation, I read in the 9 values into an array and checked by the isUnique() whether the array was a unique array.

For the checkRow subroutine this meant, reading all the values in the row, disregarding the col value passed into isValid(), and for checkCol, col values stayed constant as the row was variable.

For check3x3, I first found the relevent 3x3 square in the 9x9 grid, by using ifs and elseif statements. Then I set the upper and lower limits for the row and col of this 3x3 and using a nested loop read all the values into the array to be checked for Uniqueness.

checkRow:



getSquare was used in this subrouine as asked by the specification, as I had to get the values at positions in the grid.

checkCol:



This function is mirrored in reference to row and col compared to the checkRow() function. it's operation aside from that is no different.

Check3x3:



This was the mechanism that was used to pick which square I needed to go to. where R1 is the row and R2 is the col.



Each branch 1-9 had a block like this, where the upper and lower limits for row and col would be set and then right after used in the loop that wrote the values to the array.



The nested loop is used to store the values from all 9 positions in the 3x3 grid, after the array is done writing the isUnique() function is invoked, the result of that is checked and the final result for check3x3 is stored in r12, as for all three of the checker functions and for isValid() itself.

Part (3). [25%] Sudoku Solver.

I was given the pseudo-code for this part of the assignment, I rewrote this code in java for debugging purposes.



From my experience trying to implement this code into arm assembly form, the main points of the program, ie the most important where the two set square calls, they told me a lot about the operation of the program. eg when a branch was wrong and the program decided to go backwards reverting incorrect values back to zero, and the first setsquare in the for loop, which iterated through all the possible values searching for a version which was valid.

Another observation that I made was that it is very difficult to debug a recursive program in any language whether java or especially arm due to it's nature of being hard to read, The program finishes when the double AND condition is met ie row == 8 and col == 8 , and the program goes back through it's 80 levels of recurison to reach the most outside program.

Part(x) Debugging and What I would do next time.

I found this program quite hard to debug, due it's recursive nature as mentioned above, what I found very helpful white debugging was using breakpoints at the setSquare() function calls, and at the point where the sudoku solver reaches a new row, this allowed me to skip large parts of time, because stepping individual steps or even sudoku() function calls would take an extremely long amount of time.

I found it very useful to create a java program which mimicked the question and I could use it as pseudo code if I was writing this again, I found it helpful to place the same breakpoints in the java program and assembly, and look at the memory and respective outputs from a printBoard() function in the java, I used this to find out where 2 of my programs where going in different directions and helped me diagnose the problem.

I found that writing tests for the isValid() functions and getter and setter functions was quite helpful, however I found the memory window to be the most useful debugging tool that I used.



Part (4). Extra mile.

I decided to write a printBoard() functions for the sudokuSolver

* void printBoard(grid);

**public** **static** **void** printBoard(**int**[][] board) {

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

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

System.***out***.print(board[i][j]+" ");

}

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

}

}

This is my pseudo code for the function, to write this in arm assembly I used the BL sendchar function to send characters found by getSquare() to the console and so I could print newLine characters to make the appearance of the grid be realistic in the console