CS1022 Assignment

Sudoku

Documentation

Kamil Przepiórowski

Student no. 17327895

1 - Getting and Setting Digits

The first part of the program is responsible for getting or setting a value at a given index. The sudoku grids used are represented as a 9 × 9 grid of byte-size values in memory. The value 0 (zero) represents a blank (unfilled) square in the grid.

getSquare

This subroutine takes the input row and column and calculates the necessary index for the 2D array. Then it stores the value at the index in the register R8.

The registers used in this function don't exactly follow the principles of subroutine interface design, and as such I'll give a list of the use of each register for every function to try make things clearer. This is common throughout the whole program and not just this subroutine.

The index (R5) is calculated in the following way; <index> = (<row>x<row size>) + <column>

Purposes of registers in getSquare

R0= start address

R1= row

R2= column

R3= length of row (9)

R5= index

R8= grid[row][column]

e.g.

For a given index [1,1] in the grid:

```
DCB 0,0,0,0,0,5,6,7,0
DCB 0,2,3,0,0,0,0,0,0
DCB 0,4,0,0,0,0,0,0,0
DCB 0,0,0,0,0,0,0,0,0
DCB 0,0,0,0,0,0,0,0,0
DCB 0,0,0,0,0,0,0,0,0
DCB 0,0,0,0,0,0,0,0,0
DCB 8,0,0,0,0,0,0,0,0
DCB 9,0,0,0,0,0,0,0,0
```

The value 2 would be stored in R8.

For an index [3,6], in the grid:

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

The value 4 would be stored in R8.

setSquare

This subroutine is designed to set/reset a value at a given index. It calculates the index the same way as getSquare does, and then loads the value from R10 into the required address in memory.

Purposes of registers in setSquare

R0= start address

R1= row

R2= column

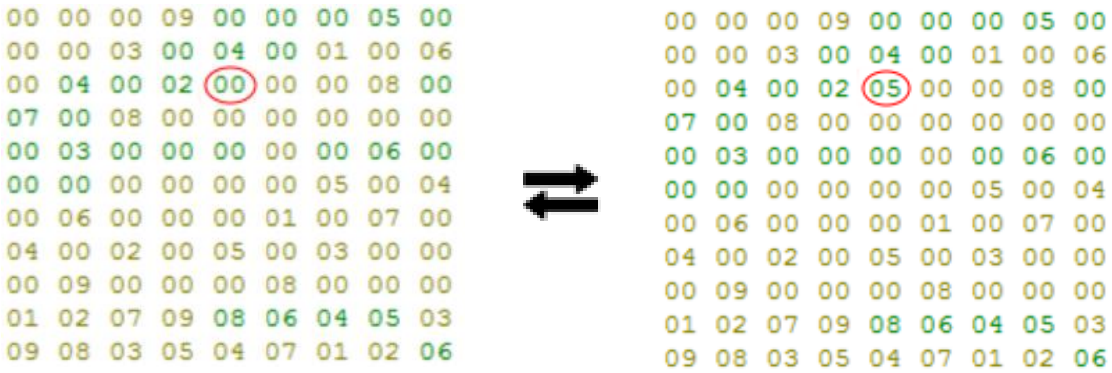
R3= length of row (9)

R5= index

R10= value to be set in memory

e.g.

Given R10=5, and the index as [2,4]. The value in memory at that index is replaced with the value we want to set the square with. This can be both used to set a value e.g. to 5, or to reset it back to 0 in order to backtrack and try another number.



In this case, 5 is stored in the index [2,4], however to replace it back to 0 or any other value the program puts that value into R10 and replaces the 5 if necessary.

2 – Validating Solutions

This part of the program is responsible for determining whether a solution (or a partial solution) is valid according to the rules of Sudoku, ignoring any remaining blank squares in the grid.

These are the rules of Sudoku in a 9x9 grid:

(i) each digit appears exactly once in each row

(ii) each digit appears exactly once in each column

(iii) each digit appears exactly once in each 3 × 3 sub-grid

isValid

The isValid subroutine compares a given value with the rules of Sudoku and returns a boolean value as to whether the value is a valid one or not. It implements the getSquare subroutine but only once, to set the value in R8 to which everything will be compared. I was unable to use getSquare to set the temporary value in R7, as later pointed out.

Purpose of registers in isValid

R0= start address

R1= row

R2= column

R3= length of row

R4= start address of select 3x3 box

R5= index

R6= tmp row

R7= tmp value to be compared against R8

R8= given value to be compared with (grid[row][column])

R9= boolean isValid

R10= encounterCount

R11= tmp column

R12= tmp index

Firstly, the program checks for the validity of rule (i), following this basic pseudo-code:

```
R8= value to be compared
boolean isValid=true;
encounterCount=0;
for(int tmpColumn=0; tmpColumn<MAX_COLUMN;tmpColumn++){
    R7= temporary value to be compared; //(grid[row][tmpColumn])
    if(R7!=0){
        if(R7==R8){
            encounterCount++;
        }
    }
    if(encounterCount==2){
        isValid=false;
    }
}
```

It stores the given value in R8, loops through each value in the row, storing the temporary value in R7. The two are then compared, if there is a duplicate in the row (encounterCount=2), isValid is set to false.

Due to my lack of implementation of the principles of subroutine interface design and improper register use, I was unable to use getSquare to update the tmp value in R7 while looping through row and column and updating the value, and as such had to update it manually without calling the getSquare subroutine each time. However, the subroutine getSquare is used at the start of isValid to get the value in the register R8. This is a problem throughout the isValid subroutine not just the first part.

If rule (i) is satisfied, the program then follows a very similar pseudo-code to check for the validity of rule(ii). If rule (i) is not satisfied, isValid is set to false and the subroutine is ended. The basic principle is the same however this time it's the tmpRow that's updated and column is set put.

The pseudo-code for checking column validity looks like this.

```
R8= value to be compared
boolean isValid=true;
encounterCount=0;
for(int tmpRow=0; tmpRow<MAX_ROW;tmpRow++){
    R7= temporary value to be compared; //(grid[tmpRow][column])
    if(R7!=0){
        if(R7==R8){
            encounterCount++;
        }
    }
    if(encounterCount==2){
        isValid=false;
    }
}
```

Again, R8 is compared against every value in the column and if there is a duplicate isValid is set to false.

If the two rules (i) and (ii) are satisfied (isValid=true) then the program moves on and checks for rule(iii), the 3x3 box. Again, if isValid is false, the subroutine is ended.

First however, the program needs to determine which 3x3 grid is in question. To do this it uses a switch statement using row (R1) and column (R2). Comparing these values against 0,3 and 6, the program finds which 3x3 grid is required, and updates the start address of the array up to that index. R0 (start index of 9x9 grid) is not directly changed, it is stored in R4 as back up and then R4 is updated accordingly.

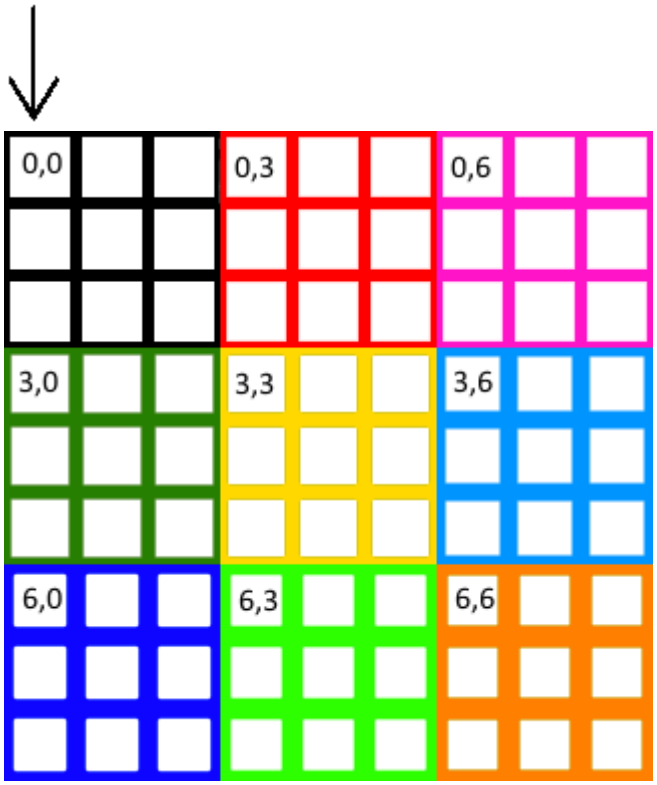
The following pseudo-code is used to determine which 3x3 box is in question:

```
if(row<=2){
        switch(column)
        case (column<=2):
        tmpRow=0;
        tmpColumn=0;
        break;
        case (column<=5):
        tmpRow=0;
        tmpColumn=3;
        startAddress+=3;
        break;
        default:
        tmpRow=0;
        tmpColumn=6;
        startAddress+=6;
        break;
}

else if(row<=5){
        switch(column)
        case (column<=2):
        tmpRow=3;
        tmpColumn=0;
        startAddress+=27;
        break;
        case (column<=5):
        tmpRow=3;
        tmpColumn=3;
        startAddress+=30;
        break;
        default:
        tmpRow=3;
        tmpColumn=6;
        startAddress+=33;
        break;
}

else if(row<=8){
        switch(column)
        case (column<=2):
        tmpRow=6;
        tmpColumn=0;
        startAddress+=54;
        break;
        case (column<=5):
        tmpRow=6;
        tmpColumn=3;
        startAddress+=57;
        break;
        default:
        tmpRow=6;
        tmpColumn=6;
        startAddress+=60;
        break;
}
```

R0-Start address of 9x9 grid (startAddress)



After determining which 3x3 box it is, and the values are updated accordingly, the program checks whether the value in question (R8), appears only once in its 3x3 box. To do this, the program essentially combines the previous loops of checking row and column but sets the limits to end of box length (3) rather than end of grid length (9) as the loops limits. Again, if a duplicate is encountered and encounterCount (R10) is greater than one, the isValid boolean is set to false.

The following pseudo-code shows how the program loops through each value in the box.

```
tmpRowMax=tmpRow+3;
tmpColMax=tmpColumn+3;
encounterCount=0;

for(tmpRow=determined_value; tmpRow<tmpRowMax; tmpRow++){
    for(tmpColumn=determined_value; tmpColumn<tmpColMax; tmpColumn++){
        R7= grid[tmpRow][tmpColumn];
        if(R7!=0){
            if(R7==R8){
                encounterCount++;
            }
        }
        if(encounterCount>=2){
            isValid=false;
        }
    }
}
```

Using the same principles as before, a constant value is set in R8 and compared against the temporary value in R7. As tmpColumn reaches either 3,6 or 9, its reset back to the necessary value, and tmpRow is updated by 1. The comparison ends when tmpRow and tmpColumn are both at their max values, or if isValid is set to false.

If all of the rules are satisfied, and the value is indeed a valid one, then a value of 1 (true) is returned in R9 (isValid). Otherwise R9 is set to 0 (!isValid).

3 – Solving a Sudoku Puzzle

This part of the program is responsible for finding a solution to a given Sudoku grid. It implements all the previously mentioned subroutines.

sudoku

This subroutine adopts a brute force approach to finding the solution to the Sudoku grid.

It does this 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. This is done recursively. If at one point the program cannot find a valid solution to a square, it backtracks, setting the current square to 0, and increasing the value of the previous square to the next valid one.

A solution is found if every square on the 9x9 grid satisfies the previously mentioned rules of Sudoku.

Purpose of registers in sudoku

R0=start address

R1= row

R2= column

R3= length of row (9)

R4= stores original value of row (R1)

R6= tmp row

R7= stores original value of column (R2)

R9= boolean isValid (at the end of subroutine this holds the boolean result)

R10= value to be set in memory (try)

R11= tmp column

R12= tmp boolean result

This pseudo-code was the base design of this subroutine, and was translated into ARM assembly:

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

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

    if(getSquare(grid, row, column)!=0){
        if(row==8 && column==8){
            return true;
        }
        else{
            result=sudoku(grid, nxtrow, nxtcol);
        }
    }
    else{
        for(byte try=1; try <=9; try++){
            setSquare(grid, row, col, try);
            if(isValid(grid, row, col)){
                if(row==8 && col==8){
                    result=true;
                }
                else{
                result=sudoku(grid, nxtrow, nxtcol);
                }
            }
        }

        if(!result){
            //need to backtrack
            setSquare(grid, row, col, 0);
        }
    }
    return result;
}
```

The program goes through each square and tries the values 1-9, using setSquare, checking whether it's a valid value or not, using isValid. If the square is not 0 when it is met, its skipped over and the next square is taken into consideration. If the program cannot find a valid solution, it needs to backtrack by using setSquare with a value of 0, and try again using different values. If it is impossible to backtrack and the grid is an invalid one, the subroutine ends, returning false. As R12 is overwritten during the recursion, the program moves the final value of the boolean result, into the register R9.

The program solves the following grids using brute-force to produce these outputs.

e.g.

```
00 02 07 06 00 00 00 00 03
03 00 00 00 00 09 00 00 00
08 00 00 00 04 00 05 00 00
06 00 00 00 00 02 00 04 00
00 00 02 00 00 00 08 00 00
00 04 00 07 00 00 00 00 01
00 00 03 00 01 00 00 00 07
00 00 00 08 00 00 00 00 09
09 00 00 00 00 06 02 08 00
```

← Original grid

Solved grid →

```
01 02 07 06 05 08 04 09 03
03 05 04 02 07 09 01 06 08
08 09 06 03 04 01 05 07 02
06 03 09 01 08 02 07 04 05
07 01 02 04 09 05 08 03 06
05 04 08 07 06 03 09 02 01
02 08 03 09 01 04 06 05 07
04 06 05 08 02 07 03 01 09
09 07 01 05 03 06 02 08 04
```

e.g.2

```
00 00 00 09 00 00 00 05 00
00 00 03 00 04 00 01 00 06
00 04 00 02 00 00 00 08 00
07 00 08 00 00 00 00 00 00
00 03 00 00 00 00 00 06 00
00 00 00 00 00 00 05 00 04
00 06 00 00 00 01 00 07 00
04 00 02 00 05 00 03 00 00
00 09 00 00 00 08 00 00 00
```

← Original grid

Solved grid →

```
01 02 07 09 08 06 04 05 03
09 08 03 05 04 07 01 02 06
05 04 06 02 01 03 07 08 09
07 05 08 03 06 04 02 09 01
02 03 04 01 09 05 08 06 07
06 01 09 08 07 02 05 03 04
08 06 05 04 03 01 09 07 02
04 07 02 06 05 09 03 01 08
03 09 01 07 02 08 06 04 05
```

An invalid grid such as this one, will prevent the program from finding a solution without altering the pre-inputted values, and so it will return 0 as result and not alter the grid.

```
01 01 01 01 01 01 01 01 01
00 00 03 00 04 00 01 00 06
00 04 00 02 00 00 00 08 00
07 00 08 00 00 00 00 00 00
00 03 00 00 00 00 00 06 00
00 00 00 00 00 00 05 00 04
00 06 00 00 00 01 00 07 00
04 00 02 00 05 00 03 00 00
00 09 00 00 00 08 00 00 00
```