Jack Vincent Nicolas MP

CS 21 Machine Problem: Sudoku Solver

VIDEO DOCUMENTATION:
https://drive.google.com/file/d/15A_XC6rOae4qAulQdPcit0B8XhCsnhKk/view?usp=sharing

Documentation:

1. A summary of how the solver(s) work.
   The solver takes in inputs row by row. So, for the 4x4 solver, it takes a 4-digit integer 4 times. Each number is then dissected and placed inside the data segment. The rows range from 0x10010000 to 0x10010060 and the columns range from Value (+0) to Value(+C).

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0	0	0	0
0x10010020	0	0	0	0
0x10010040	0	0	0	0
0x10010060	0	0	0	0

We then run the sudoku solver, which is denoted as *solve* in the code.

The first thing that happens in the solver is to determine whether there are any 0 in the sudoku problem. The num_unassign variable is the determining factor on whether the sudoku problem still has any 0 or not. At the same time, in this part of the code, the row and col are updated to the position of the 0. This part is done by using nested for loops i and j to check the value of matrix[i][j]. If the value of the matrix is 0, then num_unassign is 1 which means that there is a 0 in the sudoku problem. The row and col of the 0 in the matrix is also saved. On the other hand, if there are no 0 in the problem, then num_unassign is 0 which means that the solver has finished its job and exits.

Next, we have a for loop which starts with 1 until it reaches the value of size. Inside this loop is where the main solver of the sudoku problem happens. However, to get to that point we need to first check if plugging in a number in the sudoku problem is safe. There are 3 parts to the safe checker of the program. Initially, we set-up safe = 1, this means that plugging in the num value to the sudoku problem is safe. Safe here means that it is unique in both its row and column. The first part of the safe checker is a for loop to check the row. If it finds the same value on the row, then safe = 0 which then sends us to loop once again to check another number. The second part is similar to the first part but this time it checks the columns. Finally, it checks the sub matrix of the sudoku problem. In a 4x4 sudoku, it checks the 2x2 and in a 9x9, it checks the 3x3.

If the num is able to get pass through the safe checker, then it's time to move on to the main part of the solver. We change the 0 to the num which passed the safe checker, and then we recurse back to solve. If the solve is finished then we return 1, else we backtrack. We revert the changes made to the matrix and turn it back to 0. We then loop back and increment 1.

This occurs until we have reached a final answer, which is a complete and solved sudoku problem. In the end, we print out the solution to the sudoku problem.

2. A high-level pseudocode of the algorithm you implemented for the solver(s).

This is the high-level pseudocode for the 4x4 sudoku solver.

```
// SIZE = 4
int solve_sudoku()
{
    int row;
    int col;

    int num_unassign = 0;
    int i,j;
    for(i=0;i<SIZE;i++)
    {
        for(j=0;j<SIZE;j++)
        {
            if(matrix[i][j] == 0)
            {
                row = i;
                col = j;
                num_unassign = 1;
                break;
            }
        }
        if (num_unassign == 1){
            break;
        }
    }
    if (num_unassign == 0){
        return 1;
    }
```

```
int n;
for(int num=1;num<=SIZE;num++)
{
    int safe;
    safe = 1;
    for(int i=0;i<SIZE;i++)
    {
        if(matrix[row][i] == num){
            safe = 0;
        }
        if (safe == 0){
            break;
        }
    }
    for(int i=0;i<SIZE;i++)
    {
        if(matrix[i][col] == num){
            safe = 0;
        }
        if (safe == 0){
            break;
        }
    }
```

```
int row_start = (row/2)*2;
int col_start = (col/2)*2;
for(int i=row_start;i<row_start+2;i++)
{
    for(int j=col_start;j<col_start+2;j++)
    {
        if(matrix[i][j]==num)
            safe = 0;
            break;
    }
    if (safe == 0){
        break;
    }
}
```

```
        }
        if(safe == 1)
        {

            matrix[row][col] = num;
            //backtracking
            if(solve_sudoku())
                return 1;
            matrix[row][col]=0;
        }
    }
    return 0;
}
```

3. An in-depth explanation of each function in the 4x4 solver and how they are used to solve the problem.
   We can divide the algorithm to several parts since they were just joined together to make a working sudoku solver algorithm.
   First off, I made some name holders for the registers to make it easier to use in the code.

```
.eqv     row $s0
.eqv     col $s1
.eqv     num $s2
```

   Then, we start taking inputs through the main function.

```
m_loop:
        beq $s0 $t0 end_input

        li $v0 5 # Get nth row
        syscall
        move $t2 $v0

        li $t3 1000 # Get 1st
        div $t2 $t3
        mflo $t2 # X 0 0 0
        mfhi $t3 # 0 0 0

        sw $t2 0($t1)
        move $t2 $t3

        li $t3 100 # Get 2nd
        div $t2 $t3
        mflo $t2 # X 0 0
        mfhi $t3 # 0 0

        sw $t2 4($t1)
        move $t2 $t3

        li $t3 10 # Get 3rd
        div $t2 $t3
        mflo $t2 # X 0
        mfhi $t3 # 0

        sw $t2 8($t1)
        move $t2 $t3

        li $t3 1 # Get 4th
        div $t2 $t3
        mflo $t2 # X

        sw $t2 12($t1)

        addi $s0 $s0 1
        addi $t1 $t1 32
        j m_loop
```

   In this input taker, it takes in 4 digits per row with a maximum of 4 rows. So, what happens in this function is that it takes a row. That row is then divided to 1000 to get both the quotient and remainder. The quotient is saved to the data segment starting in 0x10010000 and increments by 4 for each digit we take. The remainder is then divided to 100 and the process repeats until we store each digit in the row. After that, we add 32 to the base which 0x10010000 + 32 = 0x10010020. We then loop back to take the second-row input and so on, until we complete the 4x4 sudoku.

We now move on to the solve function.

```
end_input:
        li $s6 0x10010000       #.data storage
        li $s7 4        # Size of Grid

        jal solve

        j exit

solve:
        subu $sp $sp 32
        sw $ra 12($sp)
        sw row 8($sp)
        sw col 4($sp)
        sw num 0($sp)

        li row 0
        li col 0
        li num 0
```

First, we initialize $s6 as 0x1001000 which is the base for the matrix and $s7 as 4 which is the size of the solver (4x4). In the solve function, we created a stack to keep track of each recursion. Next, we initialized row, col and num to 0.

Now, we move on to the num_unassigned function of the solver.

```
## na start
num_unassigned:
        li $t0 0 # i = 0
        li $s5 0 # num_unassign = 0

loop_nu: # out loop
        beq $t0 $s7 solve_cont # if i == size: num_unassign = 0
        li $t1 0 # j = 0

loop_nu2: # in loop
        beq $t1 $s7 endloop_nu2 # if j == size: end j loop

        li $s6 0x10010000
        mul $t2 $t0 32 # Get row bytes (i)
        mul $t3 $t1 4 # Get col bytes (j)
        add $t4 $t2 $t3 # Get [i][j]
        add $t4 $t4 $s6 # Get [i][j]
        lw $t3 ($t4) # grid[i][j]
        bne $t3 0 ikot # if grid[i][j] != 0: j loop

        # if grid[i][j] == 0
        move row $t0 # row = i
        move col $t1 # col = j
        li $s5 1 # num_unassign = 1
        j solve_cont
ikot:
        addi $t1 $t1 1 # j++
        j loop_nu2
endloop_nu2: # in loop
        addi $t0 $t0 1 # i++
        j loop_nu

solve_cont:
        beqz $s5 exit # If num_unassign == 0: return 1
        li num 1 # num = 1 (for solve loop)
```

In this function, we make nested loops, loop_nu is the first for loop and loop_nu2 is the second for loop. If the condition in the first loop is satisfied then we jump to solve_cont which is the end of num_unassigned function. If the condition for the second loop is satisfied then we increment $t0 and loop to the first loop. Inside these nested loops, we take the value of matrix[i][j] through several instructions. We take I * 32 to get the row and j * 4 to get the col. Adding them together,

then adding them to 0x10010000, we get the exact location in the data segment whose value is 0. If we find the num_unassign = 1 which is denoted by $s5 = 1, then we go to the solve_cont label. If $s5 is equal to 0 on the other hand, then the program terminates since there is no problem to solve anymore.

Now, before we move on to the main part of the function, we need to check if the number that we are going to replace 0 with is viable.

```
# solve start
loop_s:
        bgt num $s7 ret_0 # num > size -> endloop

# is_safe start
is_safe:
        li $s4 1 # int safe = 1;
        li $t0 0 # i = 0
r_loop:
        beq $t0 $s7 r_loop_end # if i == size: end

        li $s6 0x10010000
        mul $t1 row 32 # Get the row byte
        mul $t2 $t0 4 # Get the col byte (i)
        add $t3 $t1 $t2 # Get [row][i]
        add $t3 $t3 $s6 # Get [row][i]
        lw $t3 ($t3) # grid[row][i]
        beq $t3 num not_safe # if grid[row][i] == num: safe = 0

        addi $t0 $t0 1 # i++
        j r_loop
```

```
r_loop_end:
        li $t0 0 # i = 0
c_loop:
        beq $t0 $s7 c_loop_end # if i == size: end

        li $s6 0x10010000
        mul $t1 $t0 32 # Get the row byte (i)
        mul $t2 col 4 # Get the col byte
        add $t3 $t1 $t2 # Get [i][col]
        add $t3 $t3 $s6 # Get [i][col]
        lw $t3 ($t3) # grid[i][col]
        beq $t3 num not_safe # if grid[i][col] == num; safe = 0

        addi $t0 $t0 1 # i++
        j c_loop

c_loop_end:

        div $t2 row 2 # row_start = row/2
        mul $t2 $t2 2 # row_start = (row/2) * 2

        move $t0 $t2 # i = row_start
        addi $t2 $t2 2 # row_start + 2
```

```
rs_loop: # out loop
        beq $t0 $t2 yes_safe# if r_s == r_s + 2: safe = 1

        div $t3 col 2 # col_start = col/2
        mul $t3 $t3 2 # col_start = (col/2) * 2

        move $t1 $t3 # j = col_start
        addi $t3 $t3 2 # col_start + 2
cs_loop:
        beq $t1 $t3 cs_loop_end # if c_s == c_s + 2: end j loop
        li $s6 0x10010000
        mul $t4 $t0 32 # Get row bytes [i]
        mul $t5 $t1 4 # Get col byte [j]
        add $t6 $t4 $t5 # Get [i][j]
        add $t6 $t6 $s6 # Get [i][j]
        lw $t7 ($t6) # grid[i][j]
        beq $t7 num not_safe# if grid[i][j] == num: safe = 0

        addi $t1 $t1 1
        j cs_loop

cs_loop_end:
        addi $t0 $t0 1
        j rs_loop
not_safe:
        li $s4 0 # safe = 0
        j add1

add1:
        addi num num 1
        j loop_s
```

The is_safe function can be divided into three parts. Before that, we made use of $s4 as the variable for safe. If safe = 0 then we loop again and increase num because that means that they are not unique in that position. When safe = 1, it means that it is safe to place num in that position.

The first part is a for loop which checks the row if there is a cell with the same value as num (what we are going to replace 0 with). In this part, row would remain constant and the column will change for each loop. In each iteration, we check if the matrix[i][j] is equals to num. If so, then we loop again and increment num.

The second part is a for loop which checks the column if there is a cell with the same value as num. In this part, col would remain constant and the row will change for each loop. In each iteration, we check if the matrix[i][j] is equals to num. If so, then we loop again and increment num.

The third part is a nested for loop which checks the 2x2 submatrix of the problem. If in any instance during the check, safe becomes safe = 0, then we loop again and increment num by 1. Otherwise, num is safe to use in the solve function.

Now we move on to the main part of the sudoku solver.

```
yes_safe:
        li $s4 1

        li $s6 0x10010000
        mul $t2 row 32  # Get row bytes
        mul $t3 col 4   # Get col bytes
        add $t4 $t2 $t3 # Get [row][col]
        add $t4 $t4 $s6 # Get [tow][col]
        sw num ($t4) # grid[row][col] = i

        jal solve

        lw $ra 12($sp)
        lw row 8($sp)
        lw col 4($sp)
        lw num 0($sp)
        addu $sp $sp 32

        beq $s3 1 ret_1 # if (solve == 1): return 1

        li $s6 0x10010000
        mul $t2 row 32  # Get row bytes
        mul $t3 col 4   # Get col bytes
        add $t4 $t2 $t3 # Get [row][col]
        add $t4 $t4 $s6 # Get [row][col]
        sw $0 ($t4) # grid[row][col] = 0
```

In this part of the code, we first get the address whose value we are going to replace. We then replace it with the value of num, then we recurse using jal solve. If it is a valid option, then we move on, if not we return back to this part of the code. We restore the previous values and

backtrack. We restore the replaced address with 0 and then we loop and increment num once again.

This process is repeated as long as it takes until we reach the desired output which is a solved sudoku problem. This program would only terminate if there are no longer any 0 in the matrix.

There are also some parts of the code which are mainly used for jal.

```
ret_0:
        li $v0 0
        jr $ra
ret_1:
        li $v0 1
        jr $ra
exit:
        li $s6 0x10010000
        li $t6 0
        li $t7 4

        la $a0 10
        li $v0 11
        syscall
```

```
l_exit:
        beq $t6 $t7 end_exit
        lw $t0 0($s6)
        mul $t0 $t0 1000

        lw $t1 4($s6)
        mul $t1 $t1 100

        lw $t2 8($s6)
        mul $t2 $t2 10

        lw $t3 12($s6)
        mul $t3 $t3 1

        add $t4 $t0 $t1
        add $t4 $t4 $t2
        add $a0 $t4 $t3

        li $v0 1
        syscall

        la $a0 10
        li $v0 11
        syscall

        addi $t6 $t6 1
        addi $s6 $s6 32
        j l_exit
end_exit:
        li $v0 10
        syscall
```

These are used to represent return 0 and return 1.

The exit is used to print out the solution to the sudoku problem and eventually terminate the program. The printing of the solution is similar to taking in the input but instead of dividing, we multiply and instead of storing, we print the values when joined together.

4. Sample test case(s) for each solver aside from the examples given in this document.

Test Case # 1

```
1020
0000
3040
0003
```

Output:

```
1324
2431
3142
4213
```

Test Case # 2

```
1032
0014
4120
2300
```

Output:

```
1432
3214
4123
2341
```

Test Case # 3

```
2001
0000
4010
3102
```

Output:

```
2431
1324
4213
3142
```

Test Case # 4

```
3040
0102
0403
2010
```

Output:

```
3241
4132
1423
2314
```

Test Case # 5

```
0003
3240
0432
2000
```

Output:

```
4123
3241
1432
2314
```