Felix Q. Bueno IV

2020-05609

CS 21 MP 1 Project Documentation

Summary of how the Solver(s) work

First, we note that we used entirely the same concept and code for the 4x4 and 9x9 sudoku solvers. They just differ in some values and ranges that were used in the registers and instructions that will be explained more later.

We used several simple macros (mostly for system calls) to avoid confusion and make our code easier to understand. Here are the following macros that we used in our program:

```
.macro do_syscall(%n)
        li $v0, %n
        syscall
.end_macro

.macro read_int
        do_syscall(5)
.end_macro

.macro print_int
        do_syscall(1)
.end_macro

.macro allocate_str(%label, %str)
        %label: .asciiz %str
.end_macro

.macro print_str(%label)
        la $a0, %label
        do_syscall(4)
.end_macro

.macro exit
        do_syscall(10)
.end_macro
```

Figure 1. Macros Used

We also created four main functions other than the main function in our program. The four main functions are named as: *take_input, check_vacant, check_rowcol,* and *print_grid.* Each of these functions will also be explained more later.

For the *.data* portion of my code, we allocated space for our grid depending on whether what type of solver we are using. We allocated 64 bytes for our 4x4 solver and 324 bytes for 9x9 solver. This is because, we decided to store each cell value of our grid as a word (4 bytes). And there are 16 cells in a 4x4 grid (Hence, 16 x 4 = 64 bytes), while there are 81 cells in a 9x9 grid (Hence, 81 x 4 = 324 bytes). We also allocated space for a new line string to be used in printing our output.

```
.data
        grid: .space 64
        allocate_str(new_line, "\n")

```

```
.data
        grid: .space 324
        allocate_str(new_line, "\n")

```

Figure 2. .data portion of code

Now, for an overview of how the Solver(s) work, first, we read the integer input per line and process it in a way arithmetically (*div, mfhi, mflo*) such that we can get each individual digit per line and store them in the memory. Secondly, after we have processed the input and we have a virtual/imaginary grid in memory, we can begin solving the sudoku problem. Our first function, *check_vacant* is a recursive function that checks for vacant cells in the grid and presents these vacant cells with some values (1-4 or 1-9 depending on the grid) and tries whether placing such values in the vacant cell will be a valid move or safe move according to the rules of Sudoku. This validating happens in our *check_rowcol* function. Lastly, after a number of recursive calls and checking, if the program has finished solving the grid, it will then proceed to the *print_grid* function that prints the grid from memory in the same format as the input. More in-depth explanation for these functions shall be seen later.

High level Pseudocode of the algorithm

We created a pseudocode with the usage of mostly python syntax for us to understand it easier. Do note that the pseudocode made will still not work if compiled as a python program since it is still a pseudocode that just presents the overall flow of the program and how each function is implemented and how these functions interact with each other. Note also that for the program, we made use of the Backtracking method.

4x4 Sudoku Solver Pseudocode:

```
def check_rowcol(test_num, row_index, col_index, grid):

    #check_row

    for i in range(0, 4):                       #check other cells in the same row
        if grid[row_index][i] == test_num:
            return False

    #check_col

    for i in range(0, 4):                       #check other cells in the same column
        if grid[i][col_index] == test_num:
            return False

    #check_box & check_box_col
    #we used two for loops for this to check whether there is already the same value
    #in the corresponding box for the cell
    #the process of these loops will be explained more later but for now, let box[element] be
    #one of the values in the corresponding box that needs to be checked

    if box[element] == test_num:
        return False


    #if the code has reached this point, then it has not yet returned False
    #meaning the test_num is a valid move, thus,

    return True
```

```
def check_vacant(grid):

    for i in range(0, 4):
        for j in range(0, 4):
            if grid[i][j] == 0:                          #check if vacant cell
                for k in range(1, 5):                    #consider values 1-4
                    if check_rowcol(k, i, j, grid) == True:   #test value according to sudoku rules
                        grid[i][j] = k                   #if safe so far, assign it to the vacant cell
                        if check_vacant(grid) == True:   #recursively call check_vacant
                            return True                  #if success, return True
                        grid[i][j] = 0                   #if failure, set the vacant cell back to zero
                return False                             #return False to trigger backtracking

    return True                                          #if there's no vacant, return True
```

```python
def print_grid(grid):

    #prints the grid

    for i in range(0, 4):
        for j in range(0, 4):
            print(grid[i][j])
        print("\n")
```

```python
def take_input():

    #takes an integer per line of input and proccesses it to create a grid
    grid = [[], [], [], []]
    for i in range(0, 4):
        a = input()
        [grid[i].append(int(b)) for b in a]

    return grid


grid = take_input
check_vacant(grid)
#since it is assumed that the input test cases are always valid,
#we can just print the grid after check_vacant grid
print_grid(grid)
```

9x9 Sudoku Solver Pseudocode:

```
def check_rowcol(test_num, row_index, col_index, grid):

    #check_row

    for i in range(0, 9):                          #check other cells in the same row
        if grid[row_index][i] == test_num:
            return False

    #check_col

    for i in range(0, 9):                          #check other cells in the same column
        if grid[i][col_index] == test_num:
            return False

    #check_box & check_box_col
    #we used two for loops for this to check whether there is already the same value
    #in the corresponding box for the cell
    #the process of these loops will be explained more later but for now, let box[element] be
    #one of the values in the corresponding box that needs to be checked

    if box[element] == test_num:
        return False


    #if the code has reached this point, then it has not yet returned False
    #meaning the test_num is a valid move, thus,

    return True
```

```python
def check_vacant(grid):

    for i in range(0, 9):
        for j in range(0, 9):
            if grid[i][j] == 0:                                 #check if vacant cell
                for k in range(1, 10):                           #consider values 1-4
                    if check_rowcol(k, i, j, grid) == True:     #test value according to sudoku rules
                        grid[i][j] = k                          #if safe so far, assign it to the vacant cell
                        if check_vacant(grid) == True:          #recursively call check_vacant
                            return True                         #if success, return True
                        grid[i][j] = 0                          #if failure, set the vacant cell back to zero
                return False                                    #return False to trigger backtracking

    return True                                                 #if there's no vacant, return True
```

```python
def print_grid(grid):

    #prints the grid

    for i in range(0, 9):
        for j in range(0, 9):
            print(grid[i][j])
        print("\n")
```

```python
def take_input():

    #takes an integer per line of input and proccesses it to create a grid
    grid = [[], [], [], []]
    for i in range(0, 9):
        a = input()
        [grid[i].append(int(b)) for b in a]

    return grid
```

```python
grid = take_input
check_vacant(grid)
#since it is assumed that the input test cases are always valid,
#we can just print the grid after check_vacant grid
print_grid(grid)
```

In depth explanation of the 4x4 Sudoku Solver

As mentioned before, we made use of the *take_input*, *check_vacant*, *check_rowcol*, and *print_grid* functions for our solver. In this section, we will look at how these functions work and what MIPS instructions are involved in these functions.

Before that, we consider the *main* portion of our code, first, it sets up the necessary registers ($t0, $t2) for the *take_input* function and then calls it. Next, it sets up $a0 for the *check_vacant* function. And lastly, it sets up other temporary registers again ($t0, $t1) for the *print_grid* function. Note that as these registers are initialized, they are loaded with values of zero at first.

```
main:
        li $t0, 0                          #set $t0 = 0, let $t0 be the number of input lines taken
        li $t2, 0                          #set $t2 = 0, let $t0 be the offset to store each digit to the grid
        jal take_input                     #jal to take_input
        li $a0, 0                          #set $a0 = 0, $a0 is the offset to be used to check each element
        jal check_vacant                   #jal to check_vacant

#set up registers to be used in printing the grid

        li $t0, 0                          #set $t0 = 0 , let $t0 be the register used to keep track of the offsets used to access each cell
        li $t1, 0                          #set $t1 = 0, let $t1 be the register used to keep track of the number of cells printed in a row
        print_str(new_line)                #print new line
        j print_grid                       #jump to print_grid
```

Now, we start with our first function (*take_input)*. Note the two registers that were initialized before the *jal take_input* line in the *main* portion: $t0 = 0 which is used to monitor the number of input lines taken, $t2 = 0 which is used to determine the offset that we will need for the storing of each digit to memory/grid. Since we are given an integer value per line, in order to separate each digit and store them to memory, we made use of arithmetic operations. First, we load $t1 = 1000, since we would be dealing with 4-digit integer inputs. Next, we branch to *return* if $t0 >= 4 since that would mean that we have processed at least 4 lines of inputs already. Otherwise, we then make use of one of the macros mentioned earlier which is *read_int* that stores the integer input to $v0. Lastly, we use *move $a0, $v0* to copy the contents of $v0 (integer input) to $a0.

```
#take input function processes each line of input and stores them to memory

take_input:                                #take_input reads a 4-digit integer, and stores each digit of this integer to the gri
        li $t1, 1000                       #set $t1 = 1000
        bge $t0, 4, return                 #branch to return if $t0 >= 4
        read_int                           #take integer as input (macro)
        move $a0, $v0                      #move input to $a0
```

We then move on to the *loop_input* label, we shall branch to *next_input* if $t1 < 1, which means that we have done processing one line of input and should move to the next line. Otherwise, we are not yet done separating each digit and storing them to the grid. Thus, we make use of *div* and divide $a0 (integer input) by $t1 (initial value of 1000). We then use *mflo* to store the quotient to $t3 and then used *sw $t3, grid + 0($t2)* to store the single digit to the grid. We then use *mfhi* to store the remainder of the division process earlier to $a0. Next, we divide $t1 by 10 and add 4 to

$t2 (add 4 to offset because we stored a single digit/value as a word). And then, we jump back to loop_input to continue processing the line of input.

```
loop_input:
        blt $t1, 1, next_input                  #branch to next_input if $t1 <= 1
        div $a0, $t1                            #divide $a0 by $t1
        mflo $t3                                #move the quotient to $t3 (digit to store to grid)
        sw $t3, grid + 0($t2)                   #store the content of $t3 to the grid ($t2 as offset)
        mfhi $a0                                #move the remainder to $a0
        div $t1, $t1, 10                        #set $t1 to the quotient of $t1 / 10
        addi $t2, $t2, 4                        #set $t2 = $t2 + 4
        j loop_input                            #jump to loop_input
```

A visualization of the *loop_input* label would look like this:

```
Example first line of integer input: 4321

First iteration: $t1 = 1000, $t2 = 0 (offset), $a0 = 4321
        => 4321 / 1000
        => mflo, $t3 = 4; mfhi, $a0 = 321
        => sw $t3, grid + 0($t2)
        => $t1 / 10 = 100
        => $t2 + 4 = 4

Second iteration: $t1 = 100, $t2 = 4 (offset), $a0 = 321
        => 321 / 100
        => mflo, $t3 = 3; mfhi, $a0 = 21
        => sw $t3, grid + 0($t2)
        => $t1 / 10 = 10
        => $t2 + 4 = 8

Third iteration: $t1 = 10. $t2 = 8 (offset), $a0 = 21
        => 21 / 10
        => mflo, $t3 = 2; mfhi, $a0 = 1
        => sw $t3, grid + 0($t2)
        => $t1 / 10 = 1
        => $t2 + 4 = 12

Fourth iteration: $t1 = 1, $t2 = 12 (offset), $a0 = 1
        => 1/1
        => mflo, $t3 = 1; mfhi, $a0 = 0
        => sw $t3, grid + 0($t2)
        => $t1 / 10 = 0.1 ~ 0
        => $t2 + 4 = 16

We then branch to next_input in the next iteration since $t1 = 0 is less than 1.
```

Now, for the *next_input* label, we just add 1 to $t0 to indicate that a line of input has been processed successfully. Then, we jump again to take_input to process the next line of input. In *take_input* if $t0 reaches the value of 4, we jump to *return* and we would just go back to the return address (*jr $ra)*.

```
next_input:                                 #next_input ackowledges when the first line of integer input is done being processed
        addi $t0, $t0, 1                    #add 1 to $t0
        j take_input                        #jump to take_input

return:
        jr $ra                              #return control to main
```

After we are done processing the input, the allocated memory/data segment for our grid would look like this: Note that the numbers in each cell indicate the offset we would need to add to the grid label to access each value in the sudoku grid (this is because we stored it as a word and a word in MIPS is composed of 4 bytes).

0	4	8	12
16	20	24	28
32	36	40	44
48	52	56	60

Figure 3. Offsets of the Sudoku Grid (4x4)

We now move on to the *check_vacant* function of our program. The *check_vacant* function is a recursive function that checks the grid for vacant cells (value = 0) and tries to solve the grid by presenting numbers from 1-4 and testing/validating them according to the rules of sudoku.

First, we allocate the stack frame for the function, and for this function, we allocated 24 bytes to save the return address, and 5 saved registers namely: $s0 = offset to be used to access each element of the grid, $s1 = the number of bytes in a row/column/box, $s2 = the row index of a cell, $s3= the column index of the cell, $s4 = possible values to test for vacant cells.

```
#check_vacant is a recursive function that checks the grid for empty cells (0) and stores the appropriate digit for each empt

check_vacant:
        #####preamble######
        subu $sp, $sp, 24                       #allocate stack frame for check_vacant function
        sw $ra, 0($sp)                          #save the return address
        sw $s0, 4($sp)                          #save the s0 register - the offset to be used to access each element
        sw $s1, 8($sp)                          #save the s1 register - the number of bytes in a row/column/box
        sw $s2, 12($sp)                         #save the $s2 register - the row index of a cell
        sw $s3, 16($sp)                         #save the $s3 register - the column index of a cell
        sw $s4, 20($sp)                         #save the $s4 register - this register will be used to test the possi
        #####preamble######
```

Remember that in *main*, we initialized \$a0 = 0 for the *check_vacant* function. Now, we *move \$s0, \$a0*. Next, we branch to *return_success* if \$s0 > 60 because as seen in Figure 3, the very last element of our grid has an offset of 60, hence, the *check_vacant* is done testing all the cells of the grid. The next step would be to *li \$s1, 16* which is the number of bytes in a row which can be computed again in Figure 3. We then divide \$s0 (offset of a cell) by \$s1 and then use *mflo* to store the quotient/row index of a cell to \$s2, while using *mfhi* to store the remainder to \$s3. We then divide \$s3 by 4 and store the quotient again to \$s3 which is now the column index of a cell. This process can be tested and confirmed through Figure 3. Next, we *li \$s4, 1*, we start with 1 since we will use this register to try the possible values from (1-4).

```
        move $s0, $a0                           #set $s0 to the contents of $a0
        bgt $s0, 60, return_success             #branch to return_success if $s0 > 60 (the offset has passed the en

        li $s1, 16                              #set $s1 = 16
        div $s0, $s1                            #divide $s0 by $s1
        mflo $s2                                #set the quotient to $s2; cell row index
        mfhi $s3                                #set the remainder to $s3;
        div $s3, $s3, 4                         #set $s3 to the quotient of $s3 divided by 4; cell column index
        li $s4, 1                               #set $s4 = 1
```

Now, we begin searching the grid for vacant cells. We first use *lw \$t0, grid + 0(\$s0)* which loads a value of a cell to \$t0 depending on the offset in \$s0. Since the starting value of \$s0 is zero, we start checking the very first element in the grid. We then branch to *if_vacant* if \$t0 = 0, meaning the specific cell is vacant. Otherwise, we add 4 to \$s0 and store it to \$a0 and call *check_vacant again*. We then jump to *return_vacant* to restore the used registers for the function.

```
        lw $t0, grid + 0($s0)                   #load the current cell's value to $t0
        beqz $t0, if_vacant                     #branch to if_vacant if $t0 = 0 (empty cell)
        addi $a0, $s0, 4                        #else, set $a0 = $s0 + 4; (offset + 4)
        jal check_vacant                        #jal to check_vacant
        j return_vacant                         #jump to return_vacant
```

For the *if_vacant* label, we first setup the registers/parameters we would need for the *check_rowcol* function. We setup $a1 which is the contents of $s4 (values to try for vacant cell), set $a2 to the contents of $s2 (cell row index), set $a3 to the contents of $s3 (cell column index). Now, we are ready to call the *check_rowcol* function. Before we explain the *check_rowcol* function, let's just consider what happens to the rest of the *check_vacant* function after we receive the return value ($v0) of the *check_rowcol* function (0 or 1). If $v0 = 1, we shall branch to *next_guess*, this means that the number we are testing has violated a rule and is an invalid move. Thus, we add 1 to $s4 which becomes the next number that we should test. Now, if $s4 is less than or equal to 4 we shall branch back to *if_vacant* in order to test this number. Otherwise, having a an $s4 value of greater than 4, means that there is no possible solution yet and thus we set the value of the cell back to zero. We then load 1 to $v0 and jump to *return_vacant*. However, if $v0 = 0, then it means that the number we are trying to test is a valid move and thus we should tentatively store the value in $s4 to the vacant cell. The next step would be to add 4 to the value of $s0 and store it in $a0 which means that we should add 4 to the offset. Next, we shall call *check_vacant* again to check whether the next cell is empty or not. Finally, if $v0 = 0, then we jump to *return_vacant*.

```
if_vacant:
        move $a1, $s4                   #set $a1 to the contents of $s4; used to try values for the empty cell
        move $a2, $s2                   #set $a2 to the contents of $s2; cell row index
        move $a3, $s3                   #set $a3 to the contents of $s3; cell column index
        jal check_rowcol                #jal to check_rowcol
        beq $v0, 1, next_guess          #branch to next_guess if $v0 = 1 (return value of check_rowcol)
        sw $s4, grid + 0($s0)           #else, store the value in $s4 to the empty cell being processed

        addi $a0, $s0, 4                #set $a0 to $s0 + 4; (offset + 4)
        jal check_vacant                #jal to check_vacant
        beqz $v0, return_vacant         #branch to return_vacant if $v0 = 0

next_guess:
        addi $s4, $s4, 1                #set $s4 = $s4 + 1; try another value for the empty cell
        ble $s4, 4, if_vacant           #branch to if_vacant if $s4 is less than or equal to 4
        sw $zero, grid + 0($s0)         #set the value of the cell back to zero
        li $v0, 1                       #set $v0 = 1
        j return_vacant                 #else, jump to return_vacant


return_success:
        li $v0, 0                       #set $v0 to zero; indicates a successful solve for the sudoku

return_vacant:
        #destroys stack frame and restores the used registers for the function
        #####end######
        lw $ra, 0($sp)
        lw $s0, 4($sp)
        lw $s1, 8($sp)
        lw $s2, 12($sp)
        lw $s3, 16($sp)
        lw $s4, 20($sp)
        addi $sp, $sp, 24
        #####end######
        jr $ra
```

We now move on to the *check_rowcol* function. Recall that this function checks whether the number being tested is a valid move according to the rules of sudoku, meaning we have to check the row, column, and box it belongs in to see if there are any duplicates that would make it an invalid move. We have divided this function into three main parts: checking the row, checking the column, and checking the box.

```
#check row_col is a function to check whether a specific value that is tested is valid in terms of the rules for sudoku
#$a1: value to be checked, $a2: cell row index, $a3: cell column index

check_rowcol:
        li $t1, 0                       #set $t1 = 0; $t1 is the number of cells in the row that has been checked
        mul $t2, $a2, 16                #set $t2 = $a2 * 16; $t2 is the offset of the starting cell in the row
```

Firstly, we setup the registers we would need for checking the row. We initialize $t1 = 0 which is the number of cells in the row that has been checked. Recall that we also have setup the ff: $a1 which is the contents of $s4 (values to try for vacant cell), $a2 which is the content of $s2 (cell row index), $a3 which is the content of $s3 (cell column index). We then set $t2 to the product of $a2 and 16. This would give us the offset of the very first cell in the specific row. This can be confirmed by:

```
Cell Row Indeces: 0, 1, 2, 3, 4

Multiply each by 16:

index 0 * 16 = 0  (offset of first coluimn in grid)
index 1 * 16 = 16 (offset of second row in grid)
index 2 * 16 = 32 (offset of third row in grid)
index 3 * 16 = 48 (offset of the fourth row in grid)

We can confirm these offsets in Figure 3.
```

We now reach the *check_row* label, first, we load the value of the specific cell in the corresponding row to $t3. We then branch to *invalid* label (more on this later) if $t3 = $a1, meaning there is already a duplicate and this move would be invalid. Otherwise, we set $t2 = $t2 + 4 (add 4 to offset), then we also add 1 to $t1. We will also branch back to *check_row* if $t1 is still less than 4, which means we have not yet compared the test value to all the cells in the corresponding row.

```
check_row:
        lw $t3, grid + 0($t2)           #load the value of the specific cell in the row to $t3
        beq $t3, $a1, invalid           #branch to invalid if $t3 = $a1; there is a duplicate in the row
        addi $t2, $t2, 4                #else, set $t2 = $t2 + 4; add 4 to the offset
        addi $t1, $t1, 1                #add 1 to $t1
        blt $t1, 4, check_row           #branch to check_row if $t1 < 4
```

Else, if we do not branch to invalid and $t1 is greater than 4, then it means the test value does not create a conflict with the other elements in their row, and thus, we should setup the registers we would need for *check_col*. First, we *move $t2, $a3* which is the cell column index. We then multiply it by 4, and thus, obtaining the offset of the first cell in the corresponding column. Next, we also setup $t3 = $t2 + 48, we will use this as a bound to check the number of elements we have checked/compared in the column.

```
    #else, we set the needed registers for column checking

    move $t2, $a3                    #set $t2 to the contents of $a3; (cell column index)
    mul $t2, $t2, 4                  #multiply $t2 by 4 to get the offset of the first element in the said column
    addi $t3, $t2, 48                #set $t3 = $t2 + 48; we will use this as a bound to check the number of elem

```

We can confirm this by:

```
Cell Column Indeces: 0, 1, 2, 3

Multiply each by 4:

index 0 * 4 = 0 (offset of first column in grid)
index 1 * 4 = 4 (offset of second column in grid)
index 2 * 4 = 8 (offset of third column in grid)
index 3 * 4 = 12 (offset of fourth column in grid)

We can confirm these offsets in Figure 3.

if $t2 = 0, add 48, we would have = 48, the offset of the last element in the same column
if $t2 = 4, add 48, we would have = 52, the offset of the last element in the same column
if $t2 = 8, add 48, we would have = 56, the offset of the last element in the same column
if $t2 = 12, add 48, we would have = 60, the offset of the last element in the same column
```

We then reach the *check_col* label. We first load the value of the specific cell in the column to $t4. We then branch to *invalid* if $t4 = $a1 (same reasoning as earlier). Otherwise, we add 16 to $t2 to access the next cell in the same column (again, see Figure 3 to confirm). We then branch to *check_col* again if $t2 <= $t3, meaning we have not yet exceeded the bound we have set for the column. If we do not branch to *invalid* or reached a value in $t2 greater than $t3 then it means that the value is tentatively correct row-wise and column-wise. Thus, we have one final condition to check which is the box the vacant cell is a part of.

```
check_col:
    lw $t4, grid + 0($t2)            #load the value of the specific cell in the column to $t4
    beq $t4, $a1, invalid            #branch to invalid if $t4 = $a1; there is a duplicate in the row
    addi $t2, $t2, 16                #else, set $t2 = $t2 + 16; add 16 to the offset to access the next cell
    ble $t2, $t3, check_col          #branch to check_col if $t2 is less than or equal to $t3

```

We now setup the needed registers for *check_box*. We first set $t2 to the quotient of $a2 and 2. We also set $t3 to the quotient of $a3 and 2. We now multiply $t2 by 32 and multiply $t3 by 8. We add the values in these two registers, and we obtain the offset of the first element in the corresponding box. We can understand this more clearly by considering an example and looking at Figure 3:

Consider Figure 3 once again:

0	4	8	12
16	20	24	28
32	36	40	44
48	52	56	60

Figure 4. Offsets of the Sudoku Grid (4x4)

```
For example, the vacant cell is the cell with the offset of 36. Thus, from Figure 3, we can
see that the first element of the box it belongs to is the cell with the offset of 32. Testing
our formula, we have,

Note the following values:

$t2 = $a2 / 2 = row index / 2, the row index of the cell with offset of 36 is 2, 2/2 = 1 (integer div)
$t3 = $a3 / 2 = column index / 2, the column index of the cell with offset of 36 is 1, 1/2 = 0 (integer div)
$t2 = $t2 x 32 = 1 x 32 = 32
$t3 = $t3 x 8 = 0 x 8 = 0
$t3 = $t2 + $t3 = 32 + 0 = 32

We obtain 32 as the value in $t3 which we can confirm is the offset of the first element in
the box it belongs to.
```

After this, we also setup $t1 = 0 which will be used to count the number of times we went down a level in the box and $t2 = 0 which will be used to count the number of elements checked in a row of the box.

```
    #else, we set the needed registers for box checking

    div $t2, $a2, 2                 #set $t2 = $a2 / 2
    div $t3, $a3, 2                 #set $t3 = $a3 / 2
    mul $t2, $t2, 32                #set $t2 = $t2 * 32
    mul $t3, $t3, 8                 #set $t3 = $t3 * 8
    add $t3, $t2, $t3               #set $t3 = $t2 + $t3; this is the offset of the first element

    li $t1, 0                       #set $t1 = 0; this register will be used to count the number
    li $t2, 0                       #set $t2 = 0; this register will be used to count the number
```

We now reach the *check_box* label. Similar to the checking of the row and column, we first load the value of the specific cell in the box to $t4. We then compare this to the value in $a1 and branch to *invalid* if they are equal. Else, we add 1 to $t2 and then check whether $t2 is greater than or equal to 2 which makes the program branch to *check_box_col* label. This means that in a 2x2 box, we are done checking the upper two elements in the box and we should move to the lower row. Otherwise, we should just add 4 to $t3 to access the next upper element and jump back to *check_box*.

For the *check_box_col* portion, we first add 12 to $t3 to access the first cell in the lower part of the 2x2 box. Again, this can be confirmed through Figure 3. An example of this would be:

```
Consider the 2x2 box with offsets

0    4
16   20

After checking the upper two elements, the value of $t3 is 4. To access the next element
in the box, we subtract 16 - 4, which gives us 12. So we need to add 12 to $t3 to get the
offset of the first element in the lower part of the box.
```

Next, we set the value of $t2 back to zero since we have to check 2 elements in the lower part of the box also. We add 1 to $t1 since we have moved down in our box once. We then branch to *check_box* again if $t1 is less than 2. Otherwise, we are done checking for the rules in the row, column, and box, and the value being checked is tentatively correct. Thus, we set $v0 to zero and jump to *return_check.* As for the *invalid* label, we just set $v0 to 1 and continue on to *return_check.*

```
check_box:
        lb $t4, grid + 0($t3)                 #load the value of the specific cell in the box to $t4
        beq $t4, $a1, invalid                 #branch to invalid if $t4 = $a1; there is a duplicate in the row
        addi $t2, $t2, 1                      #else, set $t2 = $t2 + 1
        bge $t2, 2, check_box_col             #branch to check_box_col if $t2 is greater than or equal to 2
        addi $t3, $t3, 4                      #else, set $t3 = $t3 + 4; get offset for the next element in the row ir
        j check_box                           #jump to check_box

check_box_col:
        addi $t3, $t3, 12                     #set $t3 = $t3 + 12; get the offset of the element in the bottom left c
        li $t2, 0                             #set $t2 = 0
        addi $t1, $t1, 1                      #set $t1 = $t1 + 1
        blt $t1, 2, check_box                 #branch to check_box if $t1 is less than 2

        #else, we are done checking the rules of sudoku

        li $v0, 0                             #set return value $v0 = 0 (TRUE)
        j return_check                        #jump to return_check
invalid:
        li $v0, 1                             #set return value $v0 = 1 (FALSE)

return_check:
        jr $ra                                #jump to return address
```

With that, we are done explaining the *check_vacant* and *check_rowcol* functions. We only have the *print_grid* function left. Recall that in the *main* portion of our code, we have setup $t0 = 0 (used to monitor the offsets used to access each cell, and $t1 = 0 (used to monitor the number cells printed in a row). The first step in the *print_grid* function is to setup the conditions for our nested loop. We branch to *terminate_program* if $t0 > 60, which means that the offset has exceeded the offset of the last element in the grid (60, check Figure 3.) Otherwise, we check if $t1 = 4, which means we have printed four elements in a row and thus, we should branch to *print_new_line*. Else, we just load to $a0 the value in the grid with the use of the offset in $t0. We then use the macro *print_int* to print the value. Next, we add 4 to $t0 to the offset to access the next element, and we also add 1 to $t1 to keep track of how many elements are printed in a row.

In the *print_new_line* portion, we use the macro *print_str(new_line)* to print a new line. We then set the value of $t1 back to zero since we are now about to start printing elements in a new row. We then jump back to *print_grid*. For the *terminate_program* label, it simply uses a macro named as *exit* that terminates the whole program.

```
print_grid:
        bgt $t0, 60, terminate_program        #branch to terminate_program if $t0 > 60, this means
        beq $t1, 4, print_new_line            #else, branch to print_new_line if $t1 = 4, this mear
        lw $a0, grid + 0($t0)                 #else, load the value of the cell to $a0
        print_int                             #print the integer in $a0
        addi $t0, $t0, 4                      #set $t0 = $t0 + 4; offset + 4; access the next eleme
        addi $t1, $t1, 1                      #set $t1 = $t1 + 1
        j print_grid                          #jump back to print_grid

print_new_line:
        print_str(new_line)                   #print new line to indicate new row
        li $t1, 0                             #set $t1 = 0
        j print_grid                          #jump back to print_grid

terminate_program:
        exit                                  #terminate the program
```

In depth explanation of the 9x9 Sudoku Solver

As mentioned earlier, the code for the 4x4 and 9x9 sudoku solver are essentially the same. They just differ in several of the values used for the registers and bounds used for our conditionals.

We made use of the *take_input*, *check_vacant*, *check_rowcol*, and *print_grid* functions for our solver. In this section, we will look at how these functions work and what MIPS instructions are involved in these functions.

Before that, we consider the *main* portion of our code, first, it sets up the necessary registers ($t0, $t2) for the *take_input* function and then calls it. Next, it sets up $a0 for the *check_vacant* function. And lastly, it sets up other temporary registers again ($t0, $t1) for the *print_grid* function. Note that as these registers are initialized, they are loaded with values of zero at first.

```
main:
        li $t0, 0                         #set $t0 = 0, let $t0 be the number of input lines taken
        li $t2, 0                         #set $t2 = 0, let $t0 be the offset to store each digit to the grid
        jal take_input                    #jal to take_input
        li $a0, 0                         #set $a0 = 0, $a0 is the offset to be used to check each element
        jal check_vacant                  #jal to check_vacant

#set up registers to be used in printing the grid

        li $t0, 0                         #set $t0 = 0 , let $t0 be the register used to keep track of the offsets used to access each cell
        li $t1, 0                         #set $t1 = 0, let $t1 be the register used to keep track of the number of cells printed in a row
        print_str(new_line)               #print new line
        j print_grid                      #jump to print_grid
```

Now, we start with our first function (*take_input)*. Note the two registers that were initialized before the *jal take_input* line in the *main* portion: $t0 = 0 which is used to monitor the number of input lines taken, $t2 = 0 which is used to determine the offset that we will need for the storing of each digit to memory/grid. Since we are given an integer value per line, in order to separate each digit and store them to memory, we made use of arithmetic operations. First, we load $t1 = 100000000, since we would be dealing with 9-digit integer inputs. Next, we branch to *return* if $t0 >= 9 since that would mean that we have processed at least 9 lines of inputs already. Otherwise, we then make use of one of the macros mentioned earlier which is *read_int* that stores the integer input to $v0. Lastly, we use *move $a0, $v0* to copy the contents of $v0 (integer input) to $a0.

```
#take input function processes each line of input and stores them to memory

take_input:                          #take_input reads a 4-digit integer, and stores each digit of
        li $t1, 100000000            #set $t1 = 100000000
        bge $t0, 9, return           #branch to return if $t0 >= 9
        read_int                     #take integer as input (macro)
        move $a0, $v0                #move input to $a0
```

We then move on to the *loop_input* label, we shall branch to *next_input* if $t1 < 1, which means that we have done processing one line of input and should move to the next line. Otherwise, we are not yet done separating each digit and storing them to the grid. Thus, we make use of *div* and divide $a0 (integer input) by $t1 (initial value of 100000000). We then use *mflo* to store the quotient to $t3 and then used *sw $t3, grid + 0($t2)* to store the single digit to the grid. We then use *mfhi* to store the remainder of the division process earlier to $a0. Next, we divide $t1 by 10 and add 4 to $t2 (add 4 to offset because we stored a single digit/value as a word). And then, we jump back to loop_input to continue processing the line of input.

```
loop_input:
        blt $t1, 1, next_input                      #branch to next_input if $t1 <= 1
        div $a0, $t1                                #divide $a0 by $t1
        mflo $t3                                    #move the quotient to $t3 (digit to store to grid)
        sw $t3, grid + 0($t2)                       #store the content of $t3 to the grid ($t2 as offset)
        mfhi $a0                                    #move the remainder to $a0
        div $t1, $t1, 10                            #set $t1 to the quotient of $t1 / 10
        addi $t2, $t2, 4                            #set $t2 = $t2 + 4
        j loop_input                                #jump to loop_input
```

A visualization of the *loop_input* label would look like this:

```
Example first line of integer input: 987654321

First iteration: $t1 = 100000000, $t2 = 0 (offset), $a0 = 987654321
        => 987654321 / 100000000
        => mflo, $t3 = 9; mfhi, $a0 = 87654321
        => sw $t3, grid + 0($t2)
        => $t1 / 10 = 10000000
        => $t2 + 4 = 4

Second iteration: $t1 = 10000000, $t2 = 4 (offset), $a0 = 87654321
        => 87654321 / 10000000
        => mflo, $t3 = 8; mfhi, $a0 = 7654321
        => sw $t3, grid + 0($t2)
        => $t1 / 10 = 1000000
        => $t2 + 4 = 8

Third iteration: $t1 = 1000000. $t2 = 8 (offset), $a0 = 7654321
        => 7654321 / 1000000
        => mflo, $t3 = 7; mfhi, $a0 = 654321
        => sw $t3, grid + 0($t2)
        => $t1 / 10 = 100000
        => $t2 + 4 = 12

Fourth iteration: $t1 = 100000, $t2 = 12 (offset), $a0 = 654321
        => 654321/100000
        => mflo, $t3 = 6; mfhi, $a0 = 54321
        => sw $t3, grid + 0($t2)
        => $t1 / 10 = 10000
        => $t2 + 4 = 16

This process continues until the value of $t1 < 1. When we reach this point, we then
branch to next_input.
```

Now, for the *next_input* label, we just add 1 to $t0 to indicate that a line of input has been processed successfully. Then, we jump again to take_input to process the next line of input. In *take_input* if $t0 reaches the value of 9, we jump to *return* and we would just go back to the return address (*jr $ra)*.

```
next_input:                                     #next_input ackowledges when the first line of integer input is done being processed
        addi $t0, $t0, 1                        #add 1 to $t0
        j take_input                            #jump to take_input

return:
        jr $ra                                  #return control to main
```

After we are done processing the input, the allocated memory/data segment for our grid would look like this: Note that the numbers in each cell indicate the offset we would need to add to the grid label to access each value in the sudoku grid (this is because we stored it as a word and a word in MIPS is composed of 4 bytes).

0	4	8	12	16	20	24	28	32
36	40	44	48	52	56	60	64	68
72	76	80	84	88	92	96	100	104
108	112	116	120	124	128	132	136	140
144	148	152	156	160	164	168	172	176
180	184	188	192	196	200	204	208	212
216	220	224	228	232	236	240	244	248
252	256	260	264	268	272	276	280	284
288	292	296	300	304	308	312	316	320

Figure 4. Offsets of the Sudoku Grid (9x9)

We now move on to the *check_vacant* function of our program. The *check_vacant* function is a recursive function that checks the grid for vacant cells (value = 0) and tries to solve the grid by presenting numbers from 1-9 and testing/validating them according to the rules of sudoku.

First, we allocate the stack frame for the function, and for this function, we allocated 24 bytes to save the return address, and 5 saved registers namely: $s0 = offset to be used to access each element of the grid, $s1 = the number of bytes in a row/column/box, $s2 = the row index of a cell, $s3= the column index of the cell, $s4 = possible values to test for vacant cells.

```
#check_vacant is a recursive function that checks the grid for empty cells (0) and stores the appropriate digit for each empt

check_vacant:
        #####preamble######
        subu $sp, $sp, 24                       #allocate stack frame for check_vacant function
        sw $ra, 0($sp)                          #save the return address
        sw $s0, 4($sp)                          #save the s0 register - the offset to be used to access each element
        sw $s1, 8($sp)                          #save the s1 register - the number of bytes in a row/column/box
        sw $s2, 12($sp)                         #save the $s2 register - the row index of a cell
        sw $s3, 16($sp)                         #save the $s3 register - the column index of a cell
        sw $s4, 20($sp)                         #save the $s4 register - this register will be used to test the possi
        #####preamble######
```

Remember that in *main*, we initialized $a0 = 0 for the *check_vacant* function. Now, we *move $s0, $a0*. Next, we branch to *return_success* if $s0 > 320 because as seen in Figure 3, the very last element of our grid has an offset of 320, hence, the *check_vacant* is done testing all the cells of the grid. The next step would be to *li $s1, 36* which is the number of bytes in a row which can be computed again in Figure 4. We then divide $s0 (offset of a cell) by $s1 and then use *mflo* to store the quotient/row index of a cell to $s2, while using *mfhi* to store the remainder to $s3. We then divide $s3 by 4 and store the quotient again to $s3 which is now the column index of a cell. This process can be tested and confirmed through Figure 3. Next, we *li $s4, 1*, we start with 1 since we will use this register to try the possible values from (1-9).

```
	move $s0, $a0				#set $s0 to the contents of $a0
	bgt $s0, 320, return_success		#branch to return_success if $s0 > 320 (the offset has passed the

	li $s1, 36				#set $s1 = 36
	div $s0, $s1				#divide $s0 by $s1
	mflo $s2				#set the quotient to $s2; cell row index
	mfhi $s3				#set the remainder to $s3;
	div $s3, $s3, 4				#set $s3 to the quotient of $s3 divided by 4; cell column index
	li $s4, 1				#set $s4 = 1
```

Now, we begin searching the grid for vacant cells. We first use *lw $t0, grid + 0($s0)* which loads a value of a cell to $t0 depending on the offset in $s0. Since the starting value of $s0 is zero, we start checking the very first element in the grid. We then branch to *if_vacant* if $t0 = 0, meaning the specific cell is vacant. Otherwise, we add 4 to $s0 and store it to $a0 and call *check_vacant again*. We then jump to *return_vacant* to restore the used registers for the function.

```
	lw $t0, grid + 0($s0)			#load the current cell's value to $t0
	beqz $t0, if_vacant			#branch to if_vacant if $t0 = 0 (empty cell)
	addi $a0, $s0, 4			#else, set $a0 = $s0 + 4; (offset + 4)
	jal check_vacant			#jal to check_vacant
	j return_vacant				#jump to return_vacant
```

For the *if_vacant* label, we first setup the registers/parameters we would need for the *check_rowcol* function. We setup $a1 which is the contents of $s4 (values to try for vacant cell), set $a2 to the contents of $s2 (cell row index), set $a3 to the contents of $s3 (cell column index). Now, we are ready to call the *check_rowcol* function. Before we explain the *check_rowcol* function, let's just consider what happens to the rest of the *check_vacant* function after we receive the return value ($v0) of the *check_rowcol* function (0 or 1). If $v0 = 1, we shall branch to *next_guess*, this means that the number we are testing has violated a rule and is an invalid move. Thus, we add 1 to $s4 which becomes the next number that we should test. Now, if $s4 is less than or equal to 9 we shall branch back to *if_vacant* in order to test this number. Otherwise, having a an $s4 value of greater than 9, means that there is no possible solution yet and thus we set the value of the cell back to zero. We then load 1 to $v0 and jump to *return_vacant*. However, if $v0 = 0, then it means that the number we are trying to test is a valid move and thus we should tentatively store the value in $s4 to the vacant cell. The next step would be to add 4 to the value of $s0 and store it in $a0 which means that we should add 4 to the offset. Next, we shall call *check_vacant* again to check whether the next cell is empty or not. Finally, if $v0 = 0, then we jump to *return_vacant*.

```
if_vacant:
        move $a1, $s4                           #set $a1 to the contents of $s4; used to try values for the empty cell
        move $a2, $s2                           #set $a2 to the contents of $s2; cell row index
        move $a3, $s3                           #set $a3 to the contents of $s3; cell column index
        jal check_rowcol                        #jal to check_rowcol
        beq $v0, 1, next_guess                  #branch to next_guess if $v0 = 1 (return value of check_rowcol)
        sw $s4, grid + 0($s0)                   #else, store the value in $s4 to the empty cell being processed

        addi $a0, $s0, 4                        #set $a0 to $s0 + 4; (offset + 4)
        jal check_vacant                        #jal to check_vacant
        beqz $v0, return_vacant                 #branch to return_vacant if $v0 = 0

next_guess:
        addi $s4, $s4, 1                        #set $s4 = $s4 + 1; try another value for the empty cell
        ble $s4, 9, if_vacant                   #branch to if_vacant if $s4 is less than or equal to 4
        sw $zero, grid + 0($s0)                 #if it's invalid, set the value of the cell back to zero
        li $v0, 1                               #set $v0 = 1
        j return_vacant                         #else, jump to return_vacant


return_success:
        li $v0, 0                               #set $v0 to zero; indicates a successful solve for the sudoku

return_vacant:
        #destroys stack frame and restores the used registers for the function
        ######end######
        lw $ra, 0($sp)
        lw $s0, 4($sp)
        lw $s1, 8($sp)
        lw $s2, 12($sp)
        lw $s3, 16($sp)
        lw $s4, 20($sp)
        addi $sp, $sp, 24
        ######end######
        jr $ra
```

We now move on to the *check_rowcol* function. Recall that this function checks whether the number being tested is a valid move according to the rules of sudoku, meaning we have to check the row, column, and box it belongs in to see if there are any duplicates that would make it an invalid move. We have divided this function into three main parts: checking the row, checking the column, and checking the box.

```
#check row_col is a function to check whether a specific value that is tested is valid in terms of the rules for sudoku
#$a1: value to be checked, $a2: cell row index, $a3: cell column index

check_rowcol:
        li $t1, 0                       #set $t1 = 0; $t1 is the number of cells in the row that has been checked
        mul $t2, $a2, 36                #set $t2 = $a2 * 36; $t2 is the offset of the starting cell in the row
```

Firstly, we setup the registers we would need for checking the row. We initialize $t1 = 0 which is the number of cells in the row that has been checked. Recall that we also have setup the ff: $a1 which is the contents of $s4 (values to try for vacant cell), $a2 which is the content of $s2 (cell row index), $a3 which is the content of $s3 (cell column index). We then set $t2 to the product of $a2 and 36. This would give us the offset of the very first cell in the specific row. This can be confirmed by:

```
Cell Row Indeces: 0, 1, 2, 3, 4, 5, 6, 7, 8

Multiply each by 36:

index 0 * 36 = 0 (offset of first row in grid)
index 1 * 36 = 36 (offset of second row in grid)
index 2 * 36 = 72 (offset of third row in grid)
....
index 8 * 36 = 288 (offset of the ninth row in grid)

We can confirm these offsets in Figure 4.
```

We now reach the *check_row* label, first, we load the value of the specific cell in the corresponding row to $t3. We then branch to *invalid* label (more on this later) if $t3 = $a1, meaning there is already a duplicate and this move would be invalid. Otherwise, we set $t2 = $t2 + 4 (add 4 to offset), then we also add 1 to $t1. We will also branch back to *check_row* if $t1 is still less than 9, which means we have not yet compared the test value to all the cells in the corresponding row.

```
check_row:
        lw $t3, grid + 0($t2)           #load the value of the specific cell in the row to $t3
        beq $t3, $a1, invalid           #branch to invalid if $t3 = $a1; there is a duplicate in the row
        addi $t2, $t2, 4                #else, set $t2 = $t2 + 4; add 4 to the offset
        addi $t1, $t1, 1                #add 1 to $t1
        blt $t1, 9, check_row           #branch to check_row if $t1 < 9
```

Else, if we do not branch to invalid and $t1 is greater than 9, then it means the test value does not create a conflict with the other elements in their row, and thus, we should setup the registers we would need for *check_col*. First, we *move $t2, $a3* which is the cell column index. We then multiply it by 4, and thus, obtaining the offset of the first cell in the corresponding column. Next, we also setup $t3 = $t2 + 288, we will use this as a bound to check the number of elements we have checked/compared in the column.

```
    #else, we set the needed registers for column checking

    move $t2, $a3                    #set $t2 to the contents of $a3; (cell column index)
    mul $t2, $t2, 4                  #multiply $t2 by 4 to get the offset of the first element in the said colu
    addi $t3, $t2, 288               #set $t3 = $t2 + 288; we will use this as a bound to check the number of e
```

We can confirm this by:

```
Cell Column Indeces: 0, 1, 2, 3, 4, 5, 6, 7, 8

Multiply each by 4:

index 0 * 4 = 0  (offset of first column in grid)
index 1 * 4 = 4 (offset of second column in grid)
index 2 * 4 = 8 (offset of third column in grid)
...
index 8 * 4 = 32 (offset of the ninth column in grid)

We can confirm these offsets in Figure 3.

if $t2 = 0, add 288, we would have = 288, the offset of the last element in the same column
if $t2 = 4, add 288, we would have = 292, the offset of the last element in the same column
if $t2 = 8, add 288, we would have = 296, the offset of the last element in the same column
....
if $t2 = 32, add 288, we would have = 320, the offset of the last element in the same column
```

We then reach the *check_col* label. We first load the value of the specific cell in the column to $t4. We then branch to *invalid* if $t4 = $a1 (same reasoning as earlier). Otherwise, we add 36 to $t2 to access the next cell in the same column (again, see Figure 3 to confirm). We then branch to *check_col* again if $t2 <= $t3, meaning we have not yet exceeded the bound we have set for the column. If we do not branch to *invalid* or reached a value in $t2 greater than $t3 then it means that the value is tentatively correct row-wise and column-wise. Thus, we have one final condition to check which is the box the vacant cell is a part of.

```
check_col:
        lw $t4, grid + 0($t2)            #load the value of the specific cell in the column to $t4
        beq $t4, $a1, invalid            #branch to invalid if $t4 = $a1; there is a duplicate in the row
        addi $t2, $t2, 36                #else, set $t2 = $t2 + 36; add 36 to the offset to access the next
        ble $t2, $t3, check_col          #branch to check_col if $t2 is less than or equal to $t3
```

We now setup the needed registers for *check_box*. We first set $t2 to the quotient of $a2 and 3. We also set $t3 to the quotient of $a3 and 3. We now multiply $t2 by 108 and multiply $t3 by 12. We add the values in these two registers, and we obtain the offset of the first element in the corresponding box. We can understand this more clearly by considering an example and looking at Figure 4:

Consider Figure 4 once again:

0	4	8	12	16	20	24	28	32
36	40	44	48	52	56	60	64	68
72	76	80	84	88	92	96	100	104
108	112	116	120	124	128	132	136	140
144	148	152	156	160	164	168	172	176
180	184	188	192	196	200	204	208	212
216	220	224	228	232	236	240	244	248
252	256	260	264	268	272	276	280	284
288	292	296	300	304	308	312	316	320

Figure 4. Offsets of the Sudoku Grid (9x9)

```
For example, the vacant cell is the cell with the offset of 304. Thus, from Figure 4, we can
see that the first element of the box it belongs to is the cell with the offset of 228. Testing
our formula, we have,

Note the following values:

$t2 = $a2 / 3 = row index / 3, the row index of the cell with offset of 304 is 8, 8/3 = 2 (integer div)
$t3 = $a3 / 3 = column index / 3, the column index of the cell with offset of 304 is 4, 4/3 = 1 (integer div)
$t2 = $t2 x 108 = 2 x 108 = 216
$t3 = $t3 x 12 = 1 x 12 = 12
$t3 = $t2 + $t3 = 216 + 12 = 288

We obtain 288 as the value in $t3 which we can confirm is the offset of the first element in
the box it belongs to.
```

After this, we also setup $t1 = 0 which will be used to count the number of times we went down a level in the box and $t2 = 0 which will be used to count the number of elements checked in a row of the box.

```
        #else, we set the needed registers for box checking

        div $t2, $a2, 3                         #set $t2 = $a2 / 3
        div $t3, $a3, 3                         #set $t3 = $a3 / 3
        mul $t2, $t2, 108                       #set $t2 = $t2 * 106
        mul $t3, $t3, 12                        #set $t3 = $t3 * 12
        add $t3, $t2, $t3                       #set $t3 = $t2 + $t3; this is the offset of the first

        li $t1, 0                               #set $t1 = 0; this register will be used to count the
        li $t2, 0                               #set $t2 = 0; this register will be used to count the
```

We now reach the *check_box* label. Similar to the checking of the row and column, we first load the value of the specific cell in the box to $t4. We then compare this to the value in $a1 and branch to *invalid* if they are equal. Else, we add 1 to $t2 and then check whether $t2 is greater than or equal to 3 which makes the program branch to *check_box_col* label. This means that in a 3x3 box, we are done checking the upper two elements in the box and we should move one row lower. Otherwise, we should just add 4 to $t3 to access the next upper element and jump back to *check_box*.

For the *check_box_col* portion, we first add 28 to $t3 to access the first cell in the lower row of the 3x3 box. Again, this can be confirmed through Figure 4. An example of this would be:

```
Consider the 3x3 box with offsets

0     4     8
36    40    44
72    76    80

After checking the upper three elements, the value of $t3 is 8. To access the next element
in the box, we subtract 36 - 8, which gives us 28. So we need to add 28 to $t3 to get the
offset of the first element in the lower part of the box.
```

Next, we set the value of $t2 back to zero since we have to check 3 elements in the other rows of the box also. We add 1 to $t1 since we have moved down in our box once. We then branch to *check_box* again if $t1 is less than 3. Otherwise, we are done checking for the rules in the row, column, and box, and the value being checked is tentatively correct. Thus, we set $v0 to zero and jump to *return_check.* As for the *invalid* label, we just set $v0 to 1 and continue on to *return_check.*

```
check_box:
        lb $t4, grid + 0($t3)                   #load the value of the specific cell in the box to $t4
        beq $t4, $a1, invalid                   #branch to invalid if $t4 = $a1; there is a duplicate in the row
        addi $t2, $t2, 1                        #else, set $t2 = $t2 + 1
        bge $t2, 3, check_box_col               #branch to check_box_col if $t2 is greater than or equal to 3
        addi $t3, $t3, 4                        #else, set $t3 = $t3 + 4; get offset for the next element in the row included in the box
        j check_box                             #jump to check_box

check_box_col:
        addi $t3, $t3, 28                       #set $t3 = $t3 + 12; get the offset of the element in the bottom left of the box
        li $t2, 0                               #set $t2 = 0
        addi $t1, $t1, 1                        #set $t1 = $t1 + 1
        blt $t1, 3, check_box                   #branch to check_box if $t1 is less than 3

        #else, we are done checking the rules of sudoku

        li $v0, 0                               #set return value $v0 = 0 (TRUE)
        j return_check                          #jump to return_check
invalid:
        li $v0, 1                               #set return value $v0 = 1 (FALSE)

return_check:
        jr $ra                                  #jump to return address
```

With that, we are done explaining the *check_vacant* and *check_rowcol* functions. We only have the *print_grid* function left. Recall that in the *main* portion of our code, we have setup $t0 = 0 (used to monitor the offsets used to access each cell, and $t1 = 0 (used to monitor the number cells printed in a row). The first step in the *print_grid* function is to setup the conditions for our nested loop. We branch to *terminate_program* if $t0 > 320, which means that the offset has exceeded the offset of the last element in the grid (320, check Figure 4.) Otherwise, we check if $t1 = 9, which means we have printed nine elements in a row and thus, we should branch to *print_new_line*. Else, we just load to $a0 the value in the grid with the use of the offset in $t0. We then use the macro *print_int* to print the value. Next, we add 4 to $t0 to the offset to access the next element, and we also add 1 to $t1 to keep track of how many elements are printed in a row.

In the *print_new_line* portion, we use the macro *print_str(new_line*) to print a new line. We then set the value of $t1 back to zero since we are now about to start printing elements in a new row. We then jump back to *print_grid*. For the *terminate_program* label, it simply uses a macro named as *exit* that terminates the whole program.

```
print_grid:
        bgt $t0, 320, terminate_program         #branch to terminate_program if $t0 > 320, this means that the offset has ex
        beq $t1, 9, print_new_line              #else, branch to print_new_line if $t1 = 9, this means that 4 elements in a
        lw $a0, grid + 0($t0)                   #else, load the value of the cell to $a0
        print_int                               #print the integer in $a0
        addi $t0, $t0, 4                        #set $t0 = $t0 + 4; offset + 4; access the next element
        addi $t1, $t1, 1                        #set $t1 = $t1 + 1
        j print_grid                            #jump back to print_grid

print_new_line:
        print_str(new_line)                     #print new line to indicate new row
        li $t1, 0                               #set $t1 = 0
        j print_grid                            #jump back to print_grid

terminate_program:
        exit                                    #terminate the program
```

Other Sample Test Cases for 4x4

Here, we try three different test cases for the 4x4 Sudoku Solver program. Note that these test puzzles were taken from http://www.sudoku-download.net/sudoku_4x4.php

Test Case 1:

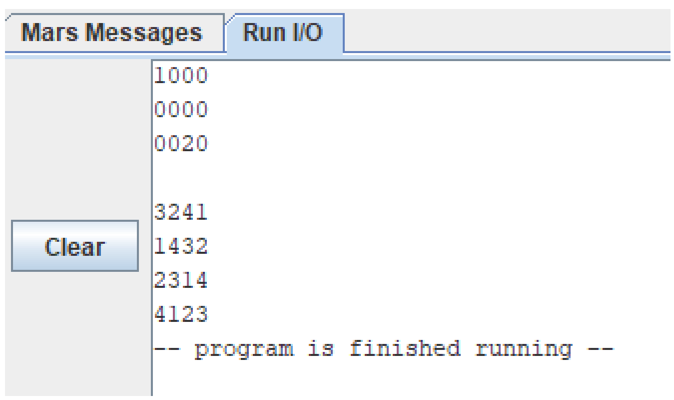
1

3			1
1			
		2	

1

3	2	4	1
1	4	3	2
2	3	1	4
4	1	2	3

Screenshot of output in console:



Test Case 2:

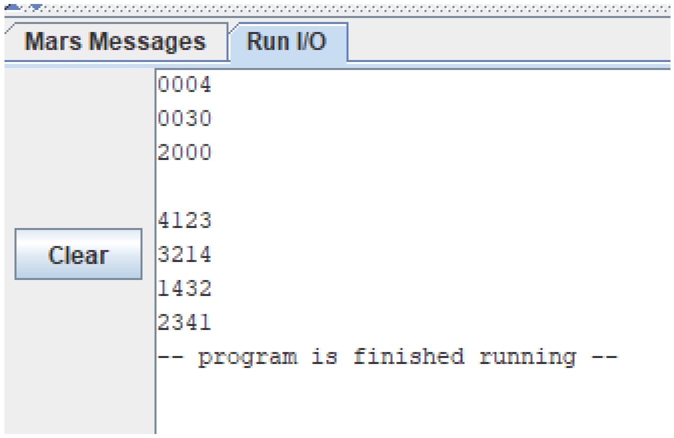
24

	1		
			4
		3	
2			

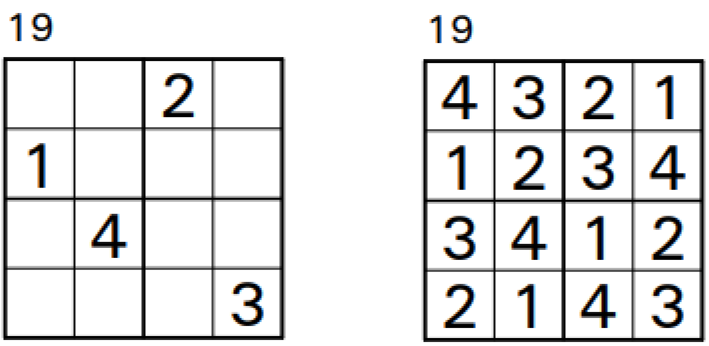
24

4	1	2	3
3	2	1	4
1	4	3	2
2	3	4	1

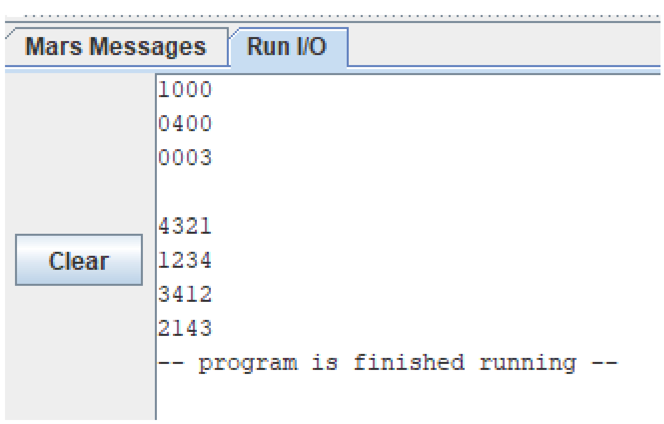
Screenshot of output in console:



Test Case 3:



Screenshot of output in console:



Other Sample Test Cases for 9x9

Here, we try three different test cases for the 9x9 Sudoku Solver program. Note that these test puzzles were taken from http://www.sudoku-download.net/sudoku_9x9.php

Test Case 1:

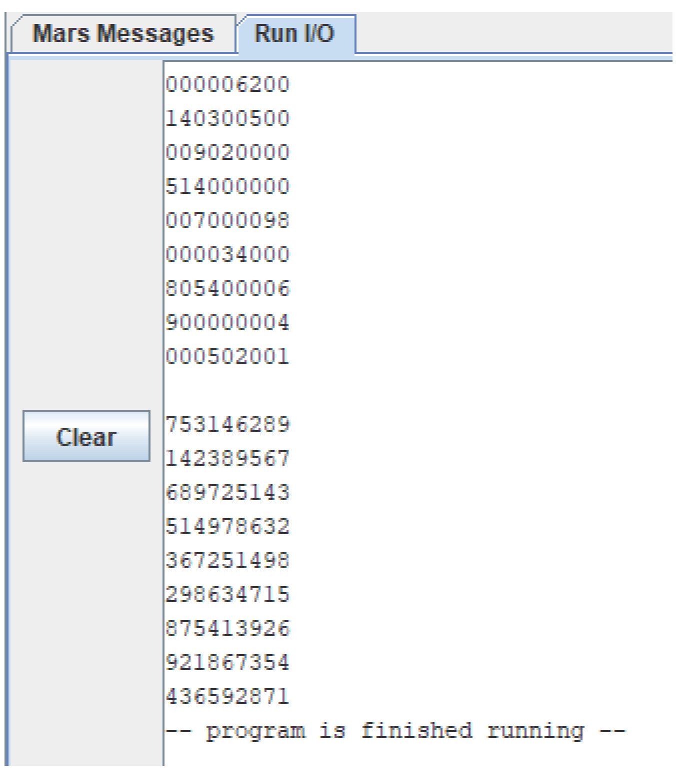
4

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

4

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

Screenshot of the output in console:



Test Case 2:

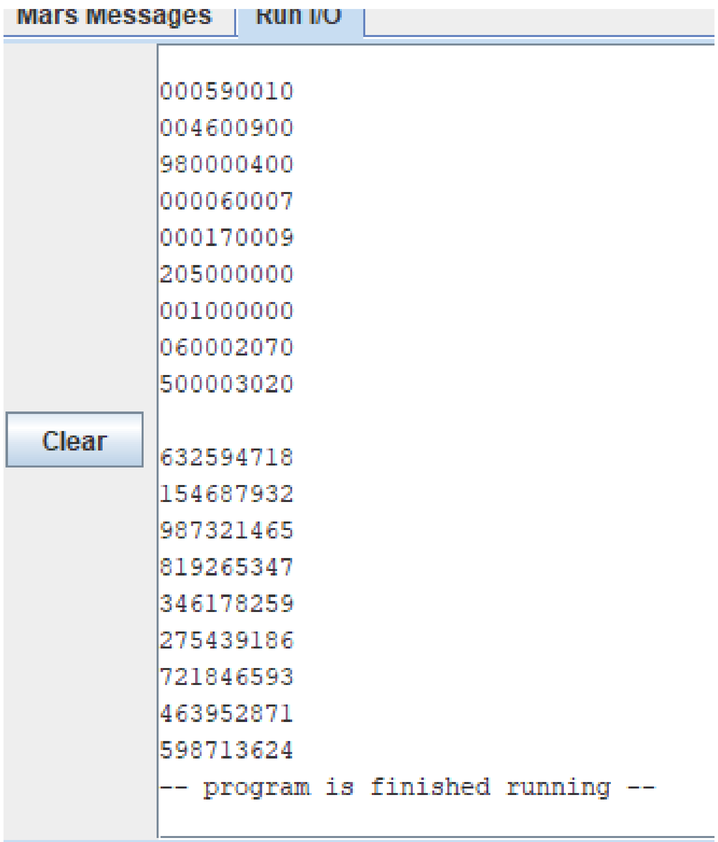
7

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

7

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

Screenshot of output in console:



Test Case 3:

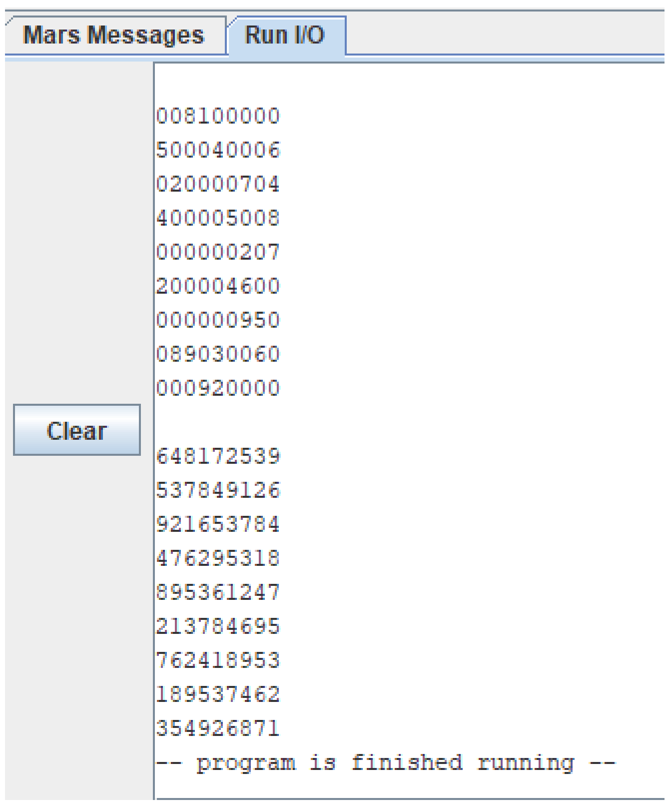
14

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

14

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

Screenshot of output in console:



Link to the Google Drive for the video demonstration:
https://drive.google.com/file/d/1yGBeRTgvylFmGdiNIf6Ub4VnU2gdBF1g/view?usp=sharing