CS21 PROJECT 01

ENGINEERING NOTEBOOK

4x4 and 9x9 Sudoku Solvers
Backtracking in MARS MIPS32

SECTION LAB4

Submitted to

Sir Ivan Carlo Balingit
Sir Wilson M. Tan

Prepared by

Yenzy Urson S. Hebron
202003090

APRIL 2022
A.Y. 2122.2

4x4 SUDOKU Solver

We begin by explaining how we approached the 4x4 Sudoku Solver problem.

Backtracking

The key algorithmic technique that we used for the 4x4 solver is called **backtracking**.

Backtracking is a more optimized version of brute-force search (BFS), however, as with the way I coded my Sudoku solvers, it still inherits many of the features of BFS, such as a potentially exhaustive approach.

Backtracking uses recursive function calls in order to explore the state space of a given problem. With each call, we assume small state changes, hoping that these changes would accumulate to a solution. As the exploration deepens, we check every state's validity according to some predefined constraints. If a state violates a constraint, we call that state a *dead-end*, and we backtrack by undoing our assumptions and returning from that call. If a state passes the constraints, that state is *potentially part of the solution*, and we explore the unexplored states adjacent to it until a proper solution is discovered.

Sudoku Constraints

In our case, the problem is made up of a 4x4 grid of numbers that is also partitioned into subgrids of 4 squares each. We have three main constraints:

1. Row Constraint: No two same numbers can exist within the same row.
2. Column Constraint: No two same numbers can exist within the same column.
3. Subgrid Constraint: No two same numbers can exist within the same subgrid.

Given the input grid with 0s denoting empty cells in it, it is fairly easy to see how backtracking can be applied to the Sudoku puzzle, wherein a state is a unique configuration of numbers in the grid especially pertaining to the empty cells, and a state change is a change in an assignment to an empty space in the grid. For every state change, we test the resulting state according to the three constraints we defined above, and proceed accordingly.

Caveats

However, I would like note this early on that as mentioned, my code is a backtracking algorithm in the purest sense, hence its BFS roots is still clearly pronounced, as we'll see later.

I did some optimizations here and there such as minimizing costly operations and avoiding register preservations as much as I can but overall, the code is just a plain application of backtracking, without any heuristics and all that fancy stuff.

Nonetheless, the code works, and for the most difficult test case I could find, the MARS MIPS version running in my marginalized hardware took 37 seconds at most to produce a solution (but the C implementation is always instantaneous in its computation).

Also, you may have noticed that I am emphasizing the assembly language as *MARS MIPS* instead of just *MIPS*. This is because we have reason to believe that MARS' Java emulation of MIPS causes a substantial degradation in performance of the MIPS code execution. Also, note that I'll be recording my test cases only after I rebooted the computer because as I've discovered, *harder* test cases end up becoming *impossible* test cases when processed after the computer has been turned on for a while, most probably due to the age of the machine or some other reason, confounded by subtle limitations of the MARS environment.

Let us now move on with the discussion.

High-level Pseudocode

I based my *high-level* pseudocode around the C language, which I also actually implemented as C later on for verification purposes. Note that we use zero-indexing.

Overview

1. Look for an empty cell 0. If non-empty cell move on to next cell.
2. If empty cell found, test values from 1 to 4.
3. For each test value, check constraints (Row, Column, and Subgrid).
4. If test value passed constraints, replace the current empty cell with that test value.
5. Then look for another empty cell and repeat the process (recursive call).
6. If the next recursive call fails (returns 0), backtrack (undo inserted test value, try another test value). Else if 1 is returned, we have found a valid solution by reaching our base case.
7. If all test values fail, return 0.
8. Return 1 on a base case, for example: Return 1 when grid bounds exceeded, which means that we've passed through all cells without failing a constraint test, meaning we've already discovered a solution.

More Detailed Pseudocode

void getGrid(int grid[]); // int grid[] ≡ int *grid, stores the puzzle

int sudoku(int grid[], int pos); // pos indicates the current grid position being processed by the solver according to the natural way we would label grid positions

void printGrid(int grid[]); // displays the solved Sudoku puzzle grid.

int main():

int grid[16] = {0}; // initialize a 1D array of 16 integer elements

getGrid(grid); // utility function to take the user input grid and represent it within the array *grid*, implemented arithmetically instead of string manipulation, remember that mod division by 10 gives the least significant digit (LSD) of a number, while flooring by 10 truncates the LSD of a number; apply necessary adjustments for array indexing.

sudoku(grid, 0); // solve puzzle, initialize solver at the first cell of the grid (topmost, leftmost cell)

// I first thought that this approach of using another parameter for the position instead of iterating through all the positions of the grid looking for a 0 or empty cell was a great decision, however I soon learned that such recursion is much more expensive compared to iterations especially with the save and restore tasks for each call. I discovered this by testing runtimes (and number of instructions) for hard cases, and the iterative version which I used with the 9x9 solver, is consistently faster than its recursive version. Note that we are talking about MARS MIPS here, C implementation is still instantaneous. But we stick with this for the 4x4 solver because it's generally a small state space and the improvement doesn't really matter. We only trade this for an iterative version in the 9x9 solver.

printGrid(grid); // utility function for displaying the solved Sudoku puzzle grid.

void getGrid(int grid[]):

int raw; // use as storage for user input while it's being processed

A: take user input, a string of *numbers*, for four times (4 rows) // user input is a *row* in the grid

B: initiate a for loop that would read the columns of the rows (its elements)

take LSD and store it in some variable, say *num* (num = raw % 10)

store obtained number in the right position in the current row, see actual code for adjustments

truncate LSD from *raw* (raw = raw // 10)

// if we haven't yet reached the *last cell* of row in raw, go back to B, else proceed

// if we haven't yet taken the last row from user input, go back to A, else proceed.

int RowColCheck(int grid[], int row, int col, int test); // check row and column constraints

int BoxCheck(int grid[], int row, int col, int test); // check subgrid constraints

int sudoku(int grid[], int pos):

compute row and col from position

base case: if position is already outside the grid, return 1, else proceed

// the position being outside the grid means that the sudoku solver has found a sequence of states that all passed the constraints, i.e. we have a found a solution, so we already return 1

// note that we only return 1 for the following two conditions: if the base case is reached, or if next state has its next state return 1, which by induction would mean that a solution or a sequence of valid states was found, and the *solution found* signal of 1 produced by the base case would be transmitted to the starting call as the recursions terminate.

if current position is empty, we can start verifying test values for that position, and insert the test value when it has passed all the necessary tests

// testing step

test values from 1 to 4 (the only valid values):

// check if state is valid

test Row and Column constraints for current value (call RowColCheck(grid, row, col, test)), this test was optimized in the 9x9 solver, specifically the row check

test Subgrid or Box constraint for current value (call BoxCheck(grid, row, col, test))

// recurse

if the two constraints are satisfied, insert test value to current position

recurse to explore the next state, using sudoku(grid, pos + 1)

if next state returns 1, that 1 is the *solution found* signal being propagated back from the base case, and this recursion must also return 1.

however, if we next state returns 0 instead, meaning we found a dead-end path so we **backtrack** and proceed with the next state, incrementing our test value

// we can call 0 a *dead-end signal*

if current position is not empty, that means we don't need to do any test here, so we skip the testing step and we proceed to check the next position and its corresponding states.

if next position returns 1, that's just the *solution found* signal as explained earlier, so we also return 1

and if we don't receive a return value of 1 from the next state, i.e. a *dead-end* signal is received, it means that the tests for the next states failed, and the state that we just returned from only lead to a dead-end. And if the current state also exhausts the test values without returning 1, then this state is part of the dead-end path, and we return 0 from this

// note that when we say *next state* here, we are pertaining to the state of the next position that was explored from the current position, I hope that's fair to say

int RowColCheck(int grid[], int row, int col, int test):

check row: idea is to first compute the starting position of the given row from the row argument, then check all elements of that row for any matches with the test value

// we increment the array index by 1 to go from left to right cell of that row; we set the loop to end when array index goes outside the row, i.e. all elements are tested, in which we'll then check the column, or when the loop has found a match, in which case we'll return a 0;

check column: current column's starting position is already encoded by the col argument, then check all elements of that column for any matches with the test value

// we increment the array index by 4 to go from top to bottom cell of that column; we set the loop to end when array index goes outside the column (bottommost "*column position*" would be 15), i.e. all elements are tested, in which we'll then return 1 having passed both row and col check, or when the loop has found a match, in which case we'll return a 0.

return 1 if all tests are passed

int BoxCheck(int grid[], int row, int col, int test):

// recall that the 4x4 Sudoku grid is partitioned into 4 smaller subgrids, each containing 4 cells

// subgrids are represented by the first cell in it as defined by its position in the overall grid, this is done to help make the matching test more easier

using the row and col arguments, identify which subgrid the current position we're testing belongs to; this can be done by simple comparisons

once the subgrid is identified, we proceed to turn the check the cells of the subgrid: the idea is to scan the subgrid column by column, from left to right, and this is done by simply allowing the selected subgrid position to increment until the last column, then array indices are adjusted to perform the matching test.

// again, return 0 on match, and return 1 on no match.

// the box check is optimized in the 9x9 solver first using double negation and De Morgan's Law on the logic, and when that proved too bulky for my MIPS skills, we just used simple for loops.

High-Level Code in C

For completeness, shown below is the C code that I first wrote to serve as basis for my MIPS code.

```
#include <stdio.h>

#define N 4         // not exactly used, just put here to show that we are
                    // dealing with a 4x4 Sudoku puzzle
int RowColCheck(int grid[], int row, int col, int test) {
    int j;
    // check row
    int rowpos = (row * 4);
    for (j = 0; j < 4; j++) {
        if (grid[rowpos + j] == test) {
            return 0;
        }
    }
```

```
    // check col
    for (j = col; j < 16; j += 4) {
        if (grid[j] == test) {
            return 0;
        }
    }
    return 1;
}

int BoxCheck(int grid[], int row, int col, int test) {
    int j;
    // check 2x2 square
    if (row <= 1 && col <= 1)
        j = 0;
    else if (row <= 1)
        j = 2;
    else if (row >= 2 && col <= 1)
        j = 8;
    else
        j = 10;

    int k = j + 1;
    for (; j <= k; j++) {
        if (grid[j] == test || grid[j + 4] == test) {
            return 0;
        }
    }
    return 1;
}

int sudoku(int grid[], int pos) {
    int row = pos/4;   // get row from position
    int col = pos%4;   // get col from position, equiv to col = pos - row * 4
    if (pos == 16) {
        return 1;
    }
    if (grid[pos] == 0) {
        // try test values
        for (int test = 1; test <= 4; test++) {
            // check validity of test value
            if (RowColCheck(grid, row, col, test) && BoxCheck(grid, row, col,
test)) {
                // if all tests passed, insert, then recurse
                grid[pos] = test;
                if (sudoku(grid, pos + 1) == 1) {
                    return 1;
                }
                grid[pos] = 0; // backtrack
            }
        }
```

```c
    }
    else if (sudoku(grid, pos + 1)) {
        return 1;
    }
    return 0;
}

int main() {
    int grid[16] = {0};   // grid with 16 positions, laid out as 1D array
    // utility function for getting grid
    int raw;
    for (int i = 0; i < 4; i++) {
        scanf("%d\n", &raw);
        for (int j = 3; j != -1; j--) { // store must be done in reverse...
            int num = raw % 10;         // index to represent grid accurately
            grid[j + (i * 4)] = num;
            raw = raw / 10;
        }
    }

    // solve grid, initialize solver at position 0 of grid
    sudoku(grid, 0);

    // utility function for printing
    for (int i = 0; i < 16; i++) {
        if (i % 4 == 0) {
            printf("\n");
        }
        printf("%d", grid[i]);
    }

    return 0;
}
```

Note that the above code for the 4x4 solver has important similarities and differences from the 9x9 solver, and I'll try my best to describe these later on.

MARS MIPS Assembly Code

I will explain the MIPS code in relation to its corresponding C code, however please first note the following:

As I was translating the C code into MIPS, I was applying small optimizations to it on the fly. So the resulting MIPS code isn't exactly faithful to the C code it was based on. For example, the C code has both utility functions getGrid and printGrid within the scope of the main function (so they're not really functions in that case), but in the MIPS code, I turned them into functions, a decision which I stuck to in the 9x9 solver. Another subtle difference is in the way inputs are taken, with the C code requiring some unrelated string in the input buffer to stop scanning for inputs, and the MIPS input syscall happily stopping taking inputs after the specified number of times it should do so. So there's some nuance here that must be enunciated, but I'm sure it's not really that hard to grasp.

Also, I challenged myself to also learn how to store local arrays and variables into the stack frame of the function call, further improving my handle of pointers. This is also an effort for my MIPS code to mimic the fact that the grid array in the C code is a local array stored in main. However, I have to deviate from this in the MIPS code of the 9x9 solver as it was giving me unnecessary hassle.

The Code | 202003090_4.asm

In the video documentation I'll be explaining the MIPS code as seen on MARS itself, but I'll attempt an in-depth explanation of this code on paper, possibly line-by-line. Note that the line numbers here thankfully correspond to the actual line numbers of the code in MARS (adjustments had to be done especially with the comments).

```
1   # CS 21 LAB4 -- S2 AY 2021-2022
2   # Yenzy Urson S. Hebron -- 04/18/2022
3   # 202003090_4.asm -- 4x4 Sudoku Solver
4   # New: grid & raw now stored in main stack frame, more faithful to translated C program
5   # Notes: getGrid is now implemented arithmetically to better handle 9 digit inputs.
6
7   #.include "macros.asm"
8   .macro do_syscall(%n)
9       li $v0, %n
10      syscall
11  .end_macro
12
13  .macro exit
14      do_syscall(10)
15  .end_macro
16
17  .macro read_int_to(%dest)
18      do_syscall(5)
19      move   %dest, $v0
20  .end_macro
21
22  # use int:1, float:2, double:3, str:4, char:11
23  # addu also supports immediates
24  .macro print_val(%val, %format)
25      addu   $a0, $0, %val
26      do_syscall(%format)
27  .end_macro
28
29  .text
30  # main stores grid just above $sp, and raw just above grid.
31  main:
32      # TODO: store grid in main stackframe to mimic typical C programs
```

```
    # (driver code in main)
    addi  $sp, $sp, -512
    sw $ra, 508($sp)
    sw $s0, 0($sp)

    addi  $s0, $sp, 0  # s0 = (int) grid[16] = {0}, 1D array with 16 integer elements
    move  $t0, $s0  # use t0 to run through grid and initialize all elements to 0
    addi  $t1, $s0, 64 # &(grid[15]) (last element of grid)
init:
    beq $t0, $t1, initdone
    sw $0, 0($t0)
    addi  $t0, $t0, 4
    j  init
initdone:

    # fill-up grid using getGrid(grid)
    move  $a0, $s0
    jal getGrid

    # solve puzzle
    move  $a0, $s0  # init solver as sudoku(grid, 0)
    li $a1, 0
    jal sudoku

    # print solved puzzle
    move  $a0, $s0
    jal printGrid

    lw $s0, 0($sp)
    lw $ra, 508($sp)
    addi  $sp, $sp, 512
    exit()

# getGrid(a0 = grid, a1 = raw)
getGrid:
    addi  $sp, $sp, -32
    sw $ra, 28($sp)
    sw $s0, 24($sp)
    sw $s1, 20($sp)

    # use (int) raw as "input buffer"

```

```
    move  $s0, $a0  # s0 = grid
    li $t0, 0    # t0 = i = 0
for1:
    beq $t0, 4, end1
    read_int_to($s1) # s1 = raw
    li $t1, 3    # t1 = j = 3
for2:
    beq $t1, -1, end2
    rem $t2, $s1, 10 # t2 = num
    sll $t3, $t0, 2  # i * 4
    add $t3, $t1, $t3 # j + (i * 4)
    sll $t3, $t3, 2  # byte offset
    add $t3, $s0, $t3 # t3 = (grid + j + (i * 4))
    sw $t2, 0($t3)  # grid[j + (i * 4)] = num
    div $s1, $s1, 10 # raw = raw // 10

    subi  $t1, $t1, 1  # j--
    j  for2
end2:
    addi  $t0, $t0, 1  # i++
    j  for1
end1:
    lw $s1, 20($sp)
    lw $s0, 24($sp)
    lw $ra, 28($sp)
    addi  $sp, $sp, 32
    jr $ra

# sudoku(a0 = grid, a1 = pos)
sudoku:
    addi  $sp, $sp, -32
    sw $ra, 28($sp)
    sw $s0, 24($sp)
    sw $s1, 20($sp)
    sw $s2, 16($sp)
    sw $s3, 12($sp)
    sw $s4, 8($sp)
    sw $s5, 4($sp)

    bne $a1, 16, notbase # immediately check base case
    li $v0, 1    # return 1 on base case
    jr $ra
```

```
notbase:
    move  $s0, $a0  # s0 = grid
    move  $s1, $a1  # s1 = pos

    srl  $s2, $s1, 2  # s2 = row = pos//4
    sll  $t0, $s2, 2  # t0 = row * 4
    sub  $s3, $s1, $t0# s3 = col = pos - row * 4

    sll  $t0, $s1, 2  # "pos * 4" (convert to word increment) <MIPS>
    add  $s4, $s0, $t0# (grid + pos) <C>
    lw $t0, 0($s4)  # t0 = *(grid + pos) <C>
    bnez  $t0, notempty

    li $s5, 1    # s5 = test value (aka test)
testval:
    bgt  $s5, 4, fail # exhaust test values (1 to 4)

    move  $a0, $s0  # a0 = grid
    move  $a1, $s2  # a1 = row
    move  $a2, $s3  # a2 = col
    move  $a3, $s5  # a3 = test
    jal  RowColCheck  # check if Row Col is safe
    beqz  $v0, unsafe
    jal  BoxCheck  # check if Box is safe
    beqz  $v0, unsafe

    sw $s5, 0($s4)  # *(grid + pos) = i (insert test val)

    move  $a0, $s0  # a0 = grid
    addi  $t1, $s1, 1  # t1 = pos + 1
    move  $a1, $t1  # a1 = pos + 1
    jal  sudoku
    beqz  $v0, backtrack  # go to backtrack
    li $v0, 1
    j  return1      # valid state </>

backtrack:
    sw $0, 0($s4)   # undo insertion
unsafe:
    addi  $s5, $s5, 1  # test++ (increment test value)
    j  testval      # loop back
```

```

notempty:		# elif (sudoku(grid, pos + 1)), check state of next pos
	move	$a0, $s0	# a0 = grid
	addi	$a1, $s1, 1	# a1 = pos + 1
	jal	sudoku
	beqz	$v0, fail
	li	$v0, 1		# valid state </>
	j	return1
fail:
	li	$v0, 0		# invalid state <X>

return1:
	lw	$s5, 4($sp)
	lw	$s4, 8($sp)
	lw	$s3, 12($sp)
	lw	$s2, 16($sp)
	lw	$s1, 20($sp)
	lw	$s0, 24($sp)
	lw	$ra, 28($sp)
	addi	$sp, $sp, 32
	jr	$ra

# RowColCheck(a0 = grid, a1 = row, a2 = col, a3 = test)
RowColCheck:
	addi	$sp, $sp, -32
	sw	$ra, 28($sp)
	sw	$s0, 24($sp)
	sw	$s1, 20($sp)

	move	$s0, $a0	# grid

	li	$t0, 0		# j = 0
	sll	$s1, $a1, 2	# rowpos = row * 4
rowchk:
	bge	$t0, 4, rowchkperfect
	add	$t1, $s1, $t0	# rowpos + j
	sll	$t1, $t1, 2	# "(rowpos + j) * 4" <pointer arithmetic>
	add	$t1, $s0, $t1	# grid + rowpos + j
	lw	$t2, 0($t1)	# *(grid + rowpos + j)
	bne	$t2, $a3, rowgood	# elif test dupe in row exist, row fail
	li	$v0, 0		# return 0
	j	RCret
```

```
rowgood:
    addi   $t0, $t0, 1   # j++ (go to next row entry)
    j   rowchk
rowchkperfect:

    move   $t0, $a2  # j = col
colchk:
    bge $t0, 16, colchkperfect
    sll $t1, $t0, 2  # "j * 4" <pointer arithmetic>
    add $t1, $s0, $t1# grid + j
    lw $t2, 0($t1)  # *(grid + j)
    bne $t2, $a3, colgood  # elif test dupe in col exist, col fail
    li $v0, 0    # return 0
    j   RCret     # fail
colgood:
    addi   $t0, $t0, 4   # j += 4 (go to next col entry)
    j   colchk
colchkperfect:

    li $v0, 1    # return 1

RCret:
    lw $s1, 20($sp)
    lw $s0, 24($sp)
    lw $ra, 28($sp)
    addi   $sp, $sp, 32
    jr $ra

# BoxCheck(a0 = grid, a1 = row, a2 = col, a3 = test)
BoxCheck:
    addi   $sp, $sp, -32
    sw $ra, 28($sp)
    sw $s0, 24($sp)
    sw $s1, 20($sp)

    # determine first square of box, set to s0 = j
    bgt $a1, 1, next1
    bgt $a2, 1, next1
    li $s0, 0
    j   jdone

next1: bgt $a1, 1, next2
```

```
    li $s0, 2
    j  jdone

next2: blt $a1, 2, next3
    bgt $a2, 1, next3
    li $s0, 8
    j  jdone

next3: li $s0, 10

    # s0 = j determined
jdone:
    addi  $s1, $s0, 1  # k = j + 1
boxchk:
    bgt $s0, $s1, boxchkperfect
    sll $t0, $s0, 2  # "j * 4" <pointer arithmetic>
    add $t0, $a0, $t0 # grid + j
    lw $t1, 0($t0)  # *(grid + j)
    seq $t1, $t1, $a3 # grid[j] == test

    addi  $t0, $s0, 4  # j + 4
    sll $t0, $t0, 2  # "(j + 4) * 4" <pointer arithmetic>
    add $t0, $a0, $t0 # grid + j + 4
    lw $t2, 0($t0)  # *(grid + j + 4)
    seq $t2, $t2, $a3 # grid[j + 4] == test

    or $t0, $t1, $t2 # (grid[j] == test || grid[j + 4] == test)
    beqz  $t0, boxgood
    li $v0, 0   # return 0
    j  boxret   # fail
boxgood:
    addi  $s0, $s0, 1  # j++
    j  boxchk

boxchkperfect:
    li $v0, 1   # return 1

boxret:
    lw $s1, 20($sp)
    lw $s0, 24($sp)
    lw $ra, 28($sp)
    addi  $sp, $sp, 32
```

```
285     jr $ra
286
287 #printGrid(a0 = grid)
288 printGrid:
289     addi    $sp, $sp, -32
290     sw $ra, 28($sp)
291     sw $s0, 24($sp)
292     sw $s1, 20($sp)
293
294     move    $s0, $a0  # s0 = grid
295     li $s1, 0    # i = 0
296 printer:
297     beq $s1, 16, printerdone
298     rem $t0, $s1, 4
299     bnez    $t0, nonewline
300     print_val('\n', 11)
301 nonewline:
302     sll $t1, $s1, 2  # convert to byte offset
303     add $t1, $s0, $t1# (grid + i)
304     lw $t1, 0($t1)  # *(grid + i)
305     print_val($t1, 1)   # print grid[i]
306     addi    $s1, $s1, 1
307     j   printer
308 printerdone:
309
310     lw $s1, 20($sp)
311     lw $s0, 24($sp)
312     lw $ra, 28($sp)
313     addi    $sp, $sp, 32
314     jr $ra
315
316 #.data
317 #grid: .space 64
318 #raw:  .space 12
```

In-Depth Explanation

I believe that the pseudocode was enough of an overview of what we intend to do, so I think it's better to just have a quick yet in-depth run-down of the functions in the MIPS 4x4 solver. Again, line numbers here is just the same as the line numbers in the actual program.

macros: Lines 8-27 shows the macros that we used for this program. They're pretty self-explanatory, so I'll just explain the ones I wrote myself.

- read_int_to(%dest) reads an integer from the user and stores it in the register given by %dest placeholder. We use this for the getGrid function.
- print_val(%val, %format) prints into the I/O screen the value stored at the register or immediate given by %val. Immediate is possible because MARS allows addu to accept an immediate operand. We use this for the printGrid function, both for printing the grid's integers (%format = 1) and for printing newline characters (%format = 11) to separate the rows of the grid.

main(): Line 31-64 contains the main function. For the 4x4 Solver, I seriously made this into a full-on function, complete with its own stack frame and register preservation.

- In the main stack frame, we store the grid array according to the HARRIS convention wherein it recommends local arrays and additional variables to be stored just above the decremented stack pointer $sp, with the contiguous array elements going into higher memory addresses away from $sp in order to preserve the convention of array indexing.
  - I also tried to do this with the 9x9 solver but I realized that it's taking away my focus from more important things in the code, so I just chucked the grid array into the .data segment as a global array.
  - We decrement $sp by 512 bytes in order to give enough room for the array. Actually, only 16x4 = 64 bytes are needed by the 4x4 Sudoku 1D array but I just used 512 bytes to prepare for the 81x4 = 324 bytes needed by the 9x9 Sudoku 1D array (which I ended up rewriting anyway).
  - We are working with the idea that integers take up 4 bytes.
- Line 38-46 initializes the grid array with 0 elements, just like what int grid[16] = {0} does in C. I know MARS already initializes all memory contents to 0 (if I'm not mistaken) and my getGrid function will eventually fill up empty cells with 0s but well I found that running through the grid filling it with 0s is a helpful visualization of the extent of the grid representation in memory.
- Line 49-50 calls utility function getGrid in order to take the user input. It takes the base address of the grid array as an argument so it stores the grid data in the local grid array of the main function.
- Line 53-55 calls sudoku to solve the puzzle obtained by getGrid. Base address of grid array also taken as an argument, along with an initial position of 0 in the grid.
- Line 58-59 calls utility function printGrid to display the solved puzzle. Again, base address of grid array is taken as an argument.

The above are largely similar to the C code except for the fact that the C code does not separate the utility functions into actual separate functions, which we did in MIPS.

Note: We check the *opposite logic* for conditionals. Also, *register preservation* now just implied.

getGrid(int grid[]): Lines 67-101 contains the getGrid function. Register save and restore at lines 68-71 and lines 97-101 respectively.

- grid base address moved to s0.
- We use raw as an *input buffer*, store it in s1.

- We use two for loops, one nested inside the other. Outer for loop takes the input and stores it in raw, which it passes to the inner for loop to work with.
    - To get LSD we use rem as num = raw % 10. We insert num to its correct position (indexing here optimized in 9x9 solver) then we truncate LSD from raw using integer division div.
    - *Story:* Also, actually I first used string manipulation for the getGrid() function in an earlier version of the solver but I realized that it wouldn't work as easier as with the 9x9 solver whose row strings wouldn't fit into a single word so I turned to arithmetic afterwards.
- After for loops are done, we return from getGrid. The grid array in the main stack frame now holds the desired user input.

Important: getGrid is the first time in which we demonstrate something called *convert to byte offset*, operations that I marked with either <pointer arithmetic> or <byte offset> or <"index * 4">, emphasis on the quote-unquote. We do this to accommodate the correct pointer arithmetic in MIPS considering the elemtype of the array. This conversion is easily handled by sll by 2.

At this point, *convert to byte offset* pointer arithmetic is now just **implied**.

sudoku(int grid[], int pos): Lines 104-180 contains the sudoku function. This function solves the puzzle.

- we immediately check the *solution found* base case, or the point in which pos, serving as index of grid array, exceeded the grid array (pos == 16, zero indexing). This means that all tests has been passed, allowing pos to get there.
- otherwise, we have to move on: grid base address moved to s0; pos moved to s1.
- for notbase case, we check if grid[pos] == 0. If so, we perform test values and checks, if not, we say that its notempty, and we move on to the next position. (I did a correction here)
- For the testing step, we first compute row and col from the given position, and I was able to do that with just srl, sll, and sub, not div needed.
    - Anyway, we then test values from 1 to 4:
        - First we call RowColCheck to check if Row and Column is safe for the test value.
        - We then call BoxCheck to check if the subgrid is safe for the test value.
        - If both tests are passed, we insert the test value as an assumption, and we proceed to the next position by recursively calling sudoku
            - If the recursive call fails, we backtrack by undoing the insertion, then returning 0.
            - If the recursive call is a success, that means this state is part of the solution, and we return 1.
        - If a constraint test is failed, the state with that test value inserted is a dead-end, and we proceed to test the next value.
    - Once all test values are exhausted, it means we ran out of possibly valid states to test, which means the current state is a dead-end, and we return 0.

Remark: Observe the label names. I tried to be as descriptive as possible when picking the labels.

RowColCheck(int grid[], int row, int col, int test): Lines 181-227 contains the RowColCheck function. We just pass row and col here so we don't have to compute them all the time.

- First we compute the starting position of the current row relative to the entire grid. We store this in rowpos. Line 191 does this using sll. However, for the 9x9 solver, sll isn't possible because 9 isn't a power of two, so we just use the difference of pos and col to give the starting rowpos.
- We then check the row by incrementing upon rowpos and comparing each row element with the test value. If pass, i.e. there is no match, proceed to column check; if fail, i.e. there is a match, return 0.
- We then check the column. It's a good thing that the parameter col is the same as the actual starting position of the column relative to the whole grid. If pass, return 1; if fail, return 0.

BoxCheck(int grid[], int row, int col, int test): Lines 229-285 contains the BoxChcek function. It is unfortunate that I named this BoxCheck instead of SubgridCheck but life goes on. Again, we just pass row and col here so we don't have to recompute them.

- Lines 236-249 First, using certain conditionals, we select the current subgrid using the row and col parameters. A subgrid is represented by its starting position or first cell (topmost, leftmost).
- Once subgrid is selected, we test the squares below the current position at the top row of the subgrid which we will be incrementing (twice in this case since subgrid is 2x2, and thrice for 3x4 subgrids of 9x9 Sudoku).
    - Testing the squares below means that we have to temporarily access positions below the current position, in this case we add 4 to the current position to get to the square below it in the column.
- If box test is passed, return 1; if box test is failed, return 0.

Remark: RowColCheck and BoxCheck returns are handled by beqz pseudoinstructions, i.e. skip return 1 if v0 = 0, and don't skip if v0 = 1.

printGrid(int grid[]): Lines 287-314 contains the printGrid utility function. This function receives the base address of the grid array stored within the main stack frame, and it works its way through the grid, printing each element it encounters and separating rows.

- To print each element, we use a for loop signified by the *printer* label, ending at the *printerdone* label. Printing will be done once the loop control variable i == 16, i.e. we've now printed the last element of the grid array and is now outside it (zero-indexing).
- We also take care to separate the rows of the grid, so we use the rem and bnez instructions of lines 298-299 to check if (i % 4 == 0), meaning we have to print a newline before printing the next integer.

Lines 316-318 are just vestigial lines that we're left over when we were still storing our grid array within the .data segment and using string manipulation on raw for taking input.

Sample Test Cases in MIPS

Here we present five (5) test cases (2 trivial cases and 3 difficult test cases) and their solutions. These test cases are generously provided by http://www.sudoku-download.net/sudoku_4x4.php. While the solutions are already available in the website, the solutions pasted here are copied directly from the MIPS

I/O itself, and we just compare with the website solution for verification. We show the actual execution during the video documentation.

Assumptions: The MIPS input can't take entire copy-pasted inputs compared to my C IDE, so we assume here for MIPS that we're just inputting test cases line by line. Also, we assume that all inputs are valid and solvable.

Trivial Test Cases		Difficult Test Cases		
TC 1 (solved grid) 2413 3142 4321 1234 Solution 1 2413 3142 4321 1234	TC 2 (empty grid) 0000 0000 0000 0000 Solution 2 1234 3412 2143 4321	TC 3 4003 0200 2040 0000 Solution 3 4123 3214 2341 1432	TC 4 0003 0300 4000 0010 Solution 4 1423 2341 4132 3214	TC 5 0000 4302 2004 0000 Solution 5 1243 4312 2134 3421

Notes: MARS MIPS and C execution times are all instantaneous. All test cases have unique solutions except for TC 2. The empty grid trivial case (TC 2) is a great way of verifying if our MIPS code is largely faithful to the C code it was based on and yes, after some testing, the MIPS code and the C code produce the same output for TC 2.

Summary

We approached the 4x4 Sudoku puzzle with a backtracking algorithm. Not much optimizations were done because of the already small state space. Even earlier revisions that hold costly operations were easily able to compute solutions.

And since 4 is a power of 2, we achieved "multiply by 2" operations using just sll in MIPS, which is not much of a burden on the execution so we left it that way. These will change to accommodate the larger state space of the 9x9 solver.

Regarding the functions we used, the code we wrote has 5 functions (excluding the driver function main) that work together to compute the solution.

1. getGrid – utility function to get the user input grid and store it in memory.
2. sudoku – function to solve the sudoku puzzle, relies on RowColCheck and BoxCheck functions for constraint tests.
3. RowColCheck – function called by sudoku to test if inserting a test value to an empty cell will be safe or valid as per the row and column constraints.
4. BoxCheck – function called by sudoku to test if inserting a test value to an empty cell will be safe or valid as per the subgrid constraint.
5. printGrid – usitility function used to display the solved Sudoku puzzle.

They are called in a similar sequence to the enumeration above. These will just repeat for the 9x9 solver.

Finally, in both translating from C to MIPS Assembly and in writing this documentation while examining the code, I also made tiny modifications. So there's that.

Remarks

For our main grid that shall receive our inputs and hold our states, we only treat it as a 1D array because MIPS pointer arithmetic for array indexing would be much easier to implement on a 1D array as opposed to a 2D array, so we only have to care about converting to byte offsets, not "nested offsets or indices" as we could call them.

9x9 SUDOKU Solver

Like with the 4x4 Sudoku Solver, the 9x9 Sudoku Solver utilizes backtracking as its main algorithmic technique. Understandably, it is also prefaced with Sudoku constraints.

Backtracking and Some Important Changes

We once again use backtracking for this, however, with more potentially empty cells, the state space is considerably larger than ever, so we have to do some optimizations to the code that we carried over from the 4x4 solver in order to keep the runtime bearable, especially by minimizing costly instructions such as mult and div and in minimizing stack preservations.

- To minimize mult and div instructions, we looked for and found ways to compute important values by just using row and col. For example,
  - We now compute rowpos within RowColCheck using pos (or i) and col passed from sudoku, so we use `rowpos = i - col` instead of `rowpos = row * 4`.
  - Moreover, we now just compute row = i/9 and col = i%9 (where i now stands in for position pos) in sudoku if the current position is actually empty to minimize unnecessary div instructions.
- To minimize stack preservations, we now just use a simple for loop to select the next empty cell for the current state.
  - This is because stack preservations in our case is actually much more expensive than for loop iterations because going to and from states require us to save and restore 7 registers given the way I wrote my code.
  - Relying on recursion to look for empty or 0 cells also mean that we need to call sudoku more, which means that we also need to pass more arguments into it repeatedly which also takes up overhead.
  - More specifically, stack preservation (save and restore) of my sudoku function needs 17 instructions while calling the sudoku function and the base case check requires 6 instructions at most. So for loop it is.
  - Nonetheless, the base case remains the same and is now packed neatly into the for loop.

Sudoku Constraints

Same constraints as before: Row, Column, and Subgrid constraints. However, the now 9x9 grid is subdivided into 9 3x3 subgrids, so we adjust the subgrid or box checker accordingly.

Caveat

I now want to emphasize an important caveat right now, my machine specs. I found out that even medium cases become hard test cases and hard test cases become impossible test cases when my machine attempts to solve them using MARS MIPS when it hasn't been rebooted for a while. So my machine suffers from some perf. degradation that I'm not really sure what is the cause, but it is probably it's aging hardware. I hope we keep that in mind when we start executing the test cases.

High-level Pseudocode

This is heavily similar to the 4x4 solver pseudocode so I won't dive into much detail.

void getGrid(int grid[]); // int grid[] ≡ int *grid, stores the puzzle

int sudoku(int grid[]); // *optimization:* removed pos parameter, use for loop instead inside sudoku to explore subsequent states especially for non-empty cell states.

void printGrid(int grid[]); // displays the solved Sudoku puzzle grid.

int main():

int grid[16] = {0}; // initialize a 1D array of 16 integer elements, globalized in MIPS by storing in .data segment

getGrid(grid); // utility function to take the user input grid and represent it within the array *grid*.

sudoku(grid); // solve puzzle. Now more iterative.

printGrid(grid); // utility function for displaying the solved Sudoku puzzle grid.

void getGrid(int grid[]):

int raw; // use as storage for user input while it's being processed

A: take user input, a string of *numbers*, for nine times (9 rows) // user input is a *row* in the grid

B: initiate a for loop that would read the columns of the rows (its elements)

take LSD and store it in some variable, say *num* (num = raw % 10)

store obtained number in the right position in the current row, see actual code for adjustments

truncate LSD from *raw* (raw = raw // 10)

// if we haven't yet reached the *last cell* of row in raw, go back to B, else proceed

// if we haven't yet taken the last row from user input, go back to A, else proceed.

int RowColCheck(int grid[], int row, int col, int test); // check row and column constraints

int BoxCheck(int grid[], int row, int col, int test); // check subgrid constraints

int sudoku(int grid[]):

for (i = 0; i < 81; i++):

// use a for loop to look for an empty cell in the current state. If empty cell is found, begin processing it.

if (grid[i] == 0):

compute row and col

for (test = 1; test <= 9; test++):

// begin testing values from 1 to 9 using another for loop

call RowColCheck and BoxCheck

if tests are passed, insert value as assumption

explore next state, return either 0 for dead-end path on next state or 1 if part of solution

else if a test is not found, backtrack, and try another test value

if all test values fail, it means the current state is a dead-end, return 0.

base case: reaching i == 81 terminates the for loop, which means we went through the entire grid without finding any empty cell and without any constraint violations, so return 1.

int RowColCheck(int grid[], int row, int col, int test):

check row: idea is to first compute the starting position of the given row from the row argument, then check all elements of that row for any matches with the test value

check column: current column's starting position is already encoded by the col argument, then check all elements of that column for any matches with the test value

return 1 if all tests are passed, else if a test is failed, return 0

int BoxCheck(int grid[], int row, int col, int test):

// 9x9 grid subdivided into 9 3x3 subgrids

using the row and col arguments, identify which subgrid the current position we're testing belongs to; this can be done by simple comparisons. Store this in j.

// process the identified subgrid

// idea is to do k++ and j+=9 three times, with k handling the "3 times constraint" and j handling the base position of the current row of the selected cell, this is obtained by incrementing by 9, leaping to the next row base position, then work on the three elements of that row using an inner loop.

for (k = 0; k <= 2; k++, j+=9):

for (l = 0; l <= 2; l++)

if (grid[j + l] == test)

// return 0 on match

return 0

// return 1 on no match

return 1

High-Level Code in C

Changes as stated above reflected here.

```
#include <stdio.h>

#define N 9
void print_grid(int grid[]);

int RowColCheck(int grid[], int i, int col, int test) {
    int j;
    // check row
    int rowpos = i - col;
    for (j = 0; j < 9; j++) {
        if (grid[rowpos + j] == test) {
            return 0;
        }
    }
    // check col
    for (j = col; j < 81; j += 9) {
        if (grid[j] == test) {
            return 0;
        }
    }
    return 1;
}

int BoxCheck(int grid[], int row, int col, int test) {
    int j;
    // check 3x3 square
    if (row < 6) {
        if (row < 3) {
            if (col < 6) {
                if (col < 3) {
                    j = 0;
                } else {
                    j = 3;
                }
            } else {
                j = 6;
            }
        } else {
            if (col < 6) {
                if (col < 3) {
                    j = 27;
                } else {
                    j = 30;
                }
            } else {
```

```
                j = 33;
            }
        }
    } else {
        if (col < 6) {
            if (col < 3) {
                j = 54;
            } else {
                j = 57;
            }
        } else {
            j = 60;
        }
    }

    for (int k = 0; k <= 2; k++, j+=9) {
        for (int l = 0; l <= 2; l++) {
            if (grid[j + l] == test) {
                return 0;
            }
        }
    }

    return 1;
}

int sudoku(int grid[]) {
    for (int i = 0; i < 81; i++) {
        if (grid[i] == 0) {
            int row = i/9;              // get row from i
            int col = i%9;              // get col from i
            // try test values
            for (int test = 1; test <= 9; test++) {
                // check validity of test value
                if (RowColCheck(grid, i, col, test) && BoxCheck(grid, row,
col, test)) {
                    // if all tests passed, insert, then recurse
                    grid[i] = test;
                    //print_grid(grid);
                    if (sudoku(grid)) {
                        return 1;
                    }
                    grid[i] = 0; // backtrack
                    //print_grid(grid);
                }
            }
            return 0;
        }
    }
    return 1;
}
```

```c
int main() {
    int grid[81] = {0};         // grid with 81 positions, laid out as 1D array
    // get grid
    int raw;
    for (int i = 0; i < 81; i += 9) {
        scanf("%d\n", &raw);
        for (int j = 8; j != -1; j--) {      // store in reverse index to...
            int num = raw % 10;         // ... represent grid accurately
            grid[i + j] = num;
            raw = raw / 10;
        }
    }

    // solve grid, initialize solver at position 0 of grid
    sudoku(grid);

    // utility function for printing
    print_grid(grid);
    return 0;
}


void print_grid(int grid[]) {
    for (int i = 0; i < 81; i++) {
        if (i % 9 == 0) {
            printf("\n");
        }
        printf("%d", grid[i]);
    }
    printf("\n");
}

```

MARS MIPS Assembly Code

We now store the grid array as a global object in the .data segment. However, we still use the grid array base address as a function argument in order to at maintain a degree of faithfulness to the translated C code. Changes as mentioned in *Backtracking and Some Important Changes* section are now present here.

The Code | 202003090_9.asm

```
# CS 21 LAB4 -- S2 AY 2021-2022
# Yenzy Urson S. Hebron -- 04/18/2022
# 202003090_9.asm -- 9x9 Sudoku Solver
# grid is now global

#.include "macros.asm"
.macro do_syscall(%n)
    li $v0, %n
    syscall
.end_macro

```

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

.macro read_int_to(%dest)
    do_syscall(5)
    move   %dest, $v0
.end_macro

# use int:1, float:2, double:3, str:4, char:11
# addu also supports immediates
.macro print_val(%val, %format)
    addu   $a0, $0, %val
    do_syscall(%format)
.end_macro

.text
main:
    # fill-up grid using getGrid()
    la $a0, grid
    jal getGrid

    # solve puzzle
    la $a0, grid
    jal sudoku

    # print solved puzzle
    la $a0, grid
    jal printGrid

    exit()

# getGrid(a0 = grid)
getGrid:
    addi   $sp, $sp, -32
    sw $ra, 28($sp)
    sw $s0, 24($sp)
    sw $s1, 20($sp)

    move   $s0, $a0
    # use (int) raw as "input buffer"
    li $t0, 0    # i = 0
```

```
for1:
    beq $t0, 81, end1 # i < 81
    read_int_to($s1) # raw
    li $t1, 8    # j = 8
for2:
    beq $t1, -1, end2 # j != -1
    rem $t2, $s1, 10 # raw % 10
    add $t3, $t0, $t1 # (i + j)
    sll $t3, $t3, 2  # (i + j) << 2
    add $t4, $s0, $t3 # grid + i + j
    sw $t2, 0($t4)  # grid[i + j] = num
    div $s1, $s1, 10 # raw // 10
    addi  $t1, $t1, -1 # j--
    j  for2
end2:
    addi  $t0, $t0, 9  # i += 9
    j  for1
end1:
    lw $s1, 20($sp)
    lw $s0, 24($sp)
    lw $ra, 28($sp)
    addi  $sp, $sp, 32
    jr $ra

# sudoku(a0 = grid)
sudoku:
    addi  $sp, $sp, -32
    sw $ra, 28($sp)
    sw $s0, 24($sp)
    sw $s1, 20($sp)
    sw $s2, 16($sp)
    sw $s3, 12($sp)
    sw $s4, 8($sp)
    sw $s5, 4($sp)

    move  $s0, $a0  # s0 = grid
    li $s1, 0    # s1 = i = 0

find0:
    beq $s1, 81, endfind0  # i < 81 (i == 81 is base case)

    sll $t0, $s1, 2  # "i * 4"
```

```
    add $s2, $s0, $t0 # s2 = grid + i (!)
    lw $t0, 0($s2)  # grid[i]
    bnez  $t0, notempty # grid[i] == 0

    li $t0, 9
    div $s1, $t0  # div i, 9
    mflo  $s3    # s3 = row = i//9
    mfhi  $s4    # s4 = col = i%9
    li $s5, 1    # s5 = test = 1
test:
    beq $s5, 10, endtest # test <= 9

    move  $a0, $s0  # grid
    move  $a1, $s1  # i (position)
    move  $a2, $s4  # col
    move  $a3, $s5  # test
    jal RowColCheck
    beqz  $v0, unsafe

    move  $a0, $s0  # grid
    move  $a1, $s3  # row
    move  $a2, $s4  # col
    move  $a3, $s5  # test
    jal BoxCheck
    beqz  $v0, unsafe

    sw $s5, 0($s2)  # grid[i] = test
    move  $a0, $s0
    jal sudoku
    beqz  $v0, backtrack
    li $v0, 1
    j  endfind0  # endSudoku

backtrack:
    sw $0, 0($s2)   # grid[i] = 0
unsafe:
    addi  $s5, $s5, 1  # test++
    j  test
notempty:
    addi  $s1, $s1, 1  # i++
    j  find0
endtest:
```

```
    li $v0, 0    # values exhausted, all fail
    j  endSudoku
endfind0:
    li $v0, 1    # reached end
endSudoku:
    lw $s5, 4($sp)
    lw $s4, 8($sp)
    lw $s3, 12($sp)
    lw $s2, 16($sp)
    lw $s1, 20($sp)
    lw $s0, 24($sp)
    lw $ra, 28($sp)
    addi  $sp, $sp, 32
    jr $ra

# RowColCheck(a0 = grid, a1 = i, a2 = col, a3 = test)
RowColCheck:
    addi  $sp, $sp, -32
    sw $ra, 28($sp)
    sw $s0, 24($sp)
    sw $s1, 20($sp)

    move  $s0, $a0  # s0 = grid
    sub$s1, $a1, $a2# rowpos = i - col

    li $t0, 0    # j = 0
rowchk:
    beq$t0, 9, rowchkperfect     # beq instead of bge
    add$t1, $s1, $t0# rowpos + j
    sll$t1, $t1, 2  # "(rowpos + j) * 4"
    add$t1, $s0, $t1# grid + rowpos + j
    lw $t2, 0($t1)  # grid[rowpos + j]
    bne$t2, $a3, rowgood  # elif test dupe in row exist, row fail
    li $v0, 0    # return 0
    j  RCret
rowgood:
    addi  $t0, $t0, 1  # j++ (go to next row entry)
    j  rowchk
rowchkperfect:

    move  $t0, $a2  # j = col
colchk:
```

```
    bge $t0, 81, colchkperfect
    sll $t1, $t0, 2  # "j * 4" <pointer arithmetic>
    add $t1, $s0, $t1 # grid + j
    lw  $t2, 0($t1)  # grid[j]
    bne $t2, $a3, colgood  # elif test dupe in col exist, col fail
    li  $v0, 0    # return 0
    j   RCret     # fail
colgood:
    addi   $t0, $t0, 9  # j += 9 (go to next col entry)
    j   colchk
colchkperfect:
    li  $v0, 1    # return 1
RCret:
    lw  $s1, 20($sp)
    lw  $s0, 24($sp)
    lw  $ra, 28($sp)
    addi   $sp, $sp, 32
    jr  $ra

# BoxCheck(a0 = grid, a1 = row, a2 = col, a3 = test)
BoxCheck:
    addi   $sp, $sp, -32
    sw  $ra, 28($sp)
    sw  $s0, 24($sp)
    sw  $s1, 20($sp)

    # determine which subgrid, set to s0 = j
    bge $a1, 6, next1
    bge $a1, 3, next2
    bge $a2, 6, next3
    bge $a2, 3, next4
    li  $s0, 0
    j   jdone
next4: li  $s0, 3
    j   jdone
next3: li  $s0, 6
    j   jdone
next2: bge $a2, 6, next5
    bge $a2, 3, next6
    li  $s0, 27
    j   jdone
next6: li  $s0, 30
```

```
    j   jdone
next5: li $s0, 33
    j   jdone
next1: bge $a2, 6, next7
    bge $a2, 3, next8
    li $s0, 54
    j   jdone
next8: li $s0, 57
    j   jdone
next7: li $s0, 60
    j   jdone
jdone: # note that $s0 = j
    li $t0, 0 # k = 0
boxchk:
    beq $t0, 3, boxgood
    li $t1, 0 # l = 0
for3:
    beq $t1, 3, nextrow
    add $t2, $s0, $t1
    sll $t2, $t2, 2
    add $t2, $a0, $t2
    lw $t3, 0($t2)
    bne $t3, $a3, good
    li $v0, 0
    j   boxret
good:
    addi   $t1, $t1, 1
    j   for3
nextrow:
    addi   $t0, $t0, 1
    addi   $s0, $s0, 9
    j   boxchk
boxgood:
    li $v0, 1    # return 1 when all checks passed
boxret:
    lw $s1, 20($sp)
    lw $s0, 24($sp)
    lw $ra, 28($sp)
    addi   $sp, $sp, 32
    jr $ra
```

```

#printGrid(a0 = grid)
printGrid:
    addi    $sp, $sp, -32
    sw $ra, 28($sp)
    sw $s0, 24($sp)
    sw $s1, 20($sp)

    la $s0, grid # s0 = grid
    li $s1, 0    # i = 0
printer:
    beq $s1, 81, printerdone
    rem $t0, $s1, 9
    bnez    $t0, nonewline
    print_val('\n', 11)
nonewline:
    sll $t1, $s1, 2  # convert to byte offset
    add $t1, $s0, $t1# (grid + i)
    lw $t1, 0($t1)   # *(grid + i)
    print_val($t1, 1)   # print grid[i]
    addi    $s1, $s1, 1
    j   printer
printerdone:

    lw $s1, 20($sp)
    lw $s0, 24($sp)
    lw $ra, 28($sp)
    addi    $sp, $sp, 32
    jr $ra

.data
grid:   .space 512
```

In-Depth Explanation

I'm not sure how in-depth we can still get because we're already 33 pages in, so let me just emphasize important changes and similarities from 4x4 solver to 9x9 solver again.

- Macros remain the same.
- .data segment now contains the grid array. Space of $2^5 = 512$ bytes is allocated, although we only really need $81 \times 4 = 324$ bytes. Just convention things.

- main function is noticeably smaller; we don't give main its own stack frame now since grid is global, also I removed the initialization of all the elements of the grid array to 0 because MARS already took care of that for us.
- getGrid is still essentially the same, just adjusted to work with nine rows each with 9 elements.
- sudoku has several adjustments as explained earlier:
    - Now use for loop instead of recursion to find 0s (Lines 92-140).
    - Base case condition is now i == 81 (Line 93), with i now serving as position variable (our main index).
    - Row and col now only computed if grid[i] == 0 (Lines 93-103).
- RowColCheck is essentially the same, only adjusted to accomodate 9 element rows and columns. For rowpos, it is now computed using rowpos = i – col instead of 4x4 solver's rowpos = row * 4 (would've been rowpos = row * 9 here). Lessens mult instructions. Hence i is now a parameter of RowColCheck, replacing row.
- BoxCheck has changed its conditionals for identifying which subgrid row and col parameters pertain to, subgrids still represented by their first position. More for loop is now used to test for matches, I believe this is faster because we need less logic for the test, traded with simple iterations.
- printGrid is essentially the same.

Sample Test Cases in MIPS

Trivial Test Cases		Difficult Test Cases		
TC 1 (solved grid) 974236158 638591742 125487936 316754289 742918563 589362417 867125394 253649871 491873625 Solution 1 974236158 638591742 125487936 316754289 742918563 589362417 867125394 253649871 491873625 Runtime: Instant	TC 2 (empty grid) 000000000 000000000 000000000 000000000 000000000 000000000 000000000 000000000 000000000 Solution 2 123456789 456789123 789123456 214365897 365897214 897214365 531642978 642978531 978531642 Runtime: Instant	TC 3* 450000000 002070630 000000028 000950000 086000200 020600750 000000476 070045000 008009000 Solution 3 453826197 892571634 167493528 714952863 586137249 329684751 935218476 671345982 248769315 Runtime: 4 s	TC 4** 800000000 003600000 070090200 050007000 000045700 000100030 001000068 008500010 090000400 Solution 4 812753649 943682175 675491283 154237896 369845721 287169534 521974368 438526917 796318452 Runtime: 35 s	TC 5*** 120400300 300010050 006000100 700090000 040603000 003002000 500080700 007000005 000000098 Solution 5 128465379 374219856 956837142 765198423 249673581 813542967 592386714 487921635 631754298 Runtime: 4 min

Remarks

- The empty grid test case (TC 2) is a great way of verifying if our MIPS code is largely faithful to the C code it was based on and yes, after some testing, we confirm that the MIPS code and the C code produce the same output for TC 2.
- TC 3 is a generic, expert-level Sudoku according to the website http://www.sudoku-download.net/sudoku_9x9.php.
- TC 4 is the hardest Sudoku in the world according to Finnish mathematician Arto Inkala. Probably hardest for a real person, not a machine. https://abcnews.go.com/blogs/headlines/2012/06/can-you-solve-the-hardest-ever-sudoku
- TC 5 is the hardest Sudoku among the puzzles rated by gsf's Sudoku q1 rating algorithm (2009). TC 5 has a difficulty rating of 99529, i.e. it offers great resistance to solving approaches. http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539.html#p65791.
- Runtimes are measured without much background programs during execution. **Test cases are run using the MARS GUI, not the command line which I hope I've done earlier.**

NOTE TO SELF (IMPORTANT): The TEST CASES are SIGNIFICANTLY FASTER when run using the command line! Using cmd, TC 4 was solved in 28 seconds, and TC 5 was solved in 3 minutes. Moreover, we can actually copy-paste the entire test case into the command line using cmd. Good to know!

Summary

We approached the 9x9 Sudoku puzzle with a backtracking algorithm. Optimizations were done to improve runtime for harder test cases. In some instances, the optimizations almost halve the runtime, as with TC 4 that went from 60 s to 35 s after a revision.

Regarding the functions we used, the code we wrote has 5 functions (excluding the driver function main) that work together to compute the solution.

6. getGrid – utility function to get the user input grid and store it in memory.
7. sudoku – function to solve the sudoku puzzle, relies on RowColCheck and BoxCheck functions for constraint tests.
8. RowColCheck – function called by sudoku to test if inserting a test value to an empty cell will be safe or valid as per the row and column constraints.
9. BoxCheck – function called by sudoku to test if inserting a test value to an empty cell will be safe or valid as per the subgrid constraint.
10. printGrid – usitlity function used to display the solved Sudoku puzzle.

They are called in a similar sequence to the enumeration above.

Finally, in both translating from C to MIPS Assembly and in writing this documentation while examining the code, I also made tiny modifications. So there may be tiny discrepancies between the actual code submitted and the code pasted in here.

That would be all, thank you for your attention and God Bless.

// END OF PROJECT 01 DOCUMENTATION //

Link to Video Demonstration (Google Drive)

https://drive.google.com/file/d/1k-E8vXAQEqSiizUXigCTDUmzAVF_FHYs/view?usp=sharing

Attributions

Cover image:

Abstract polygon vector created by Harryarts - www.freepik.com

Sudoku Test cases:

- http://www.sudoku-download.net/sudoku_4x4.php
- http://www.sudoku-download.net/sudoku_9x9.php
- https://abcnews.go.com/blogs/headlines/2012/06/can-you-solve-the-hardest-ever-sudoku
- http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539.html#p65791

Final Remarks

- **Test cases are run using the MARS GUI, not the command line which I hope I've done earlier.**
- **I also just learned how to pipe inputs into the program by reading from a text file, right after I finished my video documentation. It would've significantly improved my life. Oh my God my disappointment at myself XD.**
- **Anyway, life goes on!**