Implementation

Peg solitaire can be solved by a backtracking algorithm because it involves using a permutation of moves to reach an end goal. The process will be discussed along with the functions used in the following sections.

Constants

```
.eqv BOARD_W 7
.eqv BOARD_W2 14
.eqv BOARD_SIZE 49
.eqv PEG 111
# (char)111 = 'o'
.eqv HOLE 46
# (char)46 = '.'
.eqv PEG_LAST 79
# (char)79 = 'O'
.eqv HOLE_LAST 69
# (char)69 = 'E'
```

These are just constants used to make the code more robust to changes. The only thing to note here is that *BOARD*_W2 is just two times the *BOARD*_W.

Global Variables

```
.data
yes:            .asciiz "YES"           # string
no:             .asciiz "NO"            # string
arrow:          .asciiz "->"            # string
board:          .space BOARD_SIZE       # char[][]
                .space 1                # \n or \0 terminated
pegs:           .byte 0                 # int8          <= 49
end_coords:     .word 0                 # address
moves_size:     .byte 0                 # int8          < 49
moves:          .space 196              # int8[moves_size][x, y, p, q] x,y->p,q        <= 7
```

pegs stores the number of pegs currently in the board
end_coords stores the address of the position where the last peg should reside
moves_size stores the number of moves taken to reach the end goal if possible
moves stores the moves taken in reverse order

Main

```
# main #######################################################
main:           jal get_input
                jal init_board
                jal solve_board

                beqz $v0 then1
                print_string_label(yes)
                print_char('\n')
                jal print_moves         # print moves
                b end1
then1:          print_string_label(no)
end1:           li $v0  10              # end program
                syscall
```

We can see the overview of the program in the main. The *solve_board* function returns 1 or 0. If it is 1, it means that the board is solvable thus we print the moves; else, it is not solvable.

Input handling

```
# get_input ##################################################
get_input:      la $t0 board            # address start
                addi $t1 $t0 BOARD_SIZE # address end
get_input_loop: beq $t0 $t1 get_input_end
                li $v0 8                # read string
                move $a0 $t0            # pass address
                li $a1 BOARD_W          # pass size
                addi $a1 $a1 2
                syscall
                addi $t0 $t0 BOARD_W    # increment address by 7 bytes
                b get_input_loop
get_input_end:  jr $ra
```

Input is handled by taking the input line by line while saving it directly to the *board* global variable. Notice that it doesn't save registers to the stack. This is because this function does not use any preserved variables.

Initializing the board

```
# init_board ################################################
# t0 = current cell
# t1 = last cell + 1
# t2 = peg count
# t3 = cell value
# t4 = cell value to store
init_board:      move $a0 $zero          # reset parameter
                 la $t0 board            # address start
                 addi $t1 $t0 BOARD_SIZE # address end
                 move $t2 $zero          # peg count
init_board_loop:beq $t0 $t1 init_board_end
                 lb $t3 ($t0)            # load char
                 bne $t3 PEG init_board_1# if cell is PEG
                 addi $t2 $t2 1          # increment peg count
init_board_1:    bne $t3 PEG_LAST init_board_2# if cell is PEG_LAST
                 addi $t2 $t2 1          # increment peg count
                 sw $t0 end_coords       # store ending address
                 li $t4 PEG              # replace cell with PEG
                 sb $t4 ($t0)
init_board_2:    bne $t3 HOLE_LAST init_board_3# if cell is HOLE_LAST
                 sw $t0 end_coords       # store ending address
                 li $t4 HOLE             # replace cell with PEG
                 sb $t4 ($t0)
init_board_3:    addi $t0 $t0 1          # increment address by 1 byte
                 b init_board_loop
init_board_end: sb $t2 pegs             # store peg count
                 jr $ra
```

It initializes the board by setting the *pegs* global variable to the right number of pegs. It also checks where the last cell should be (i.e. where the 'E' or 'O' is) and stores the address of this cell to the *end_coords* global variable. Lastly, it also replaces the 'E' with '.' and the 'O' with 'o' to make the board consistent since we already saved the address of either 'E' or 'O'. Notice that it doesn't save registers to the stack. This is because this function does not use any preserved variables.

Solving the board

The code for solving the board is too long to include here. Thus, trivial parts are removed and replaced with a short description of what it does. Hopefully, this makes the code a lot more readable.

```
# solve_board ################################################
# v0 = (bool)solved
                # prologue ###################################
solve_board:    subiu $sp $sp 28
                sw $ra 4($sp)
                sw $s0 8($sp)
                sw $s1 12($sp)
                sw $s2 16($sp)
                sw $s3 20($sp)
                sw $s4 24($sp)
                sw $s5 28($sp)
                # base case ##################################
                lb $t0 pegs
                bne $t0 1 sb_body
                lw $t0 end_coords
                lb $t0 ($t0)
                bne $t0 PEG sb_body
                b sb_yes
                # body #######################################
# s0 = row start (this row start address)
# s1 = row end (last + 1 address)
# s2 = col start (current address)
# s3 = col end (next row start address)
# s4 = row index
# s5 = col index
sb_body:        la $s0 board            # row start
                addi $s1 $s0 BOARD_SIZE # row end
                move $s4 $zero          # row index
                move $s3 $s0
sb_row:         addi $s4 $s4 1          # increment row index
                move $s0 $s3            # update row
                beq $s0 $s1 sb_no
                move $s2 $s0            # col start
                addi $s3 $s2 BOARD_W    # col end
                move $s5 $zero          # col index
sb_col:         addi $s5 $s5 1          # increment col index
                beq $s2 $s3 sb_row
                # try moves ##################################
                lb $t0 ($s2)            # check if PEG
                bne $t0 PEG sb_skip_cell# if not a peg continue
.macro make_move (%next_offset, %next_next_offset)
                li $t0 HOLE             # execute move, remove peg from current cell
                sb $t0 0($s2)
                sb $t0 %next_offset($s2)# remove next peg
                li $t0 PEG              # place peg to landing cell
                sb $t0 %next_next_offset($s2)
                lb $t0 pegs             # decrement pegs
                subi $t0 $t0 1
                sb $t0 pegs
.end_macro
.macro reverse_move (%next_offset, %next_next_offset)
                li $t0 PEG              # reverse moves
                sb $t0 0($s2)
                sb $t0 %next_offset($s2)
                li $t0 HOLE
                sb $t0 %next_next_offset($s2)
                lb $t0 pegs             # increment pegs
                addi $t0 $t0 1
                sb $t0 pegs
.end_macro
```

for each direction:
  if move is valid:
    do move
    recursive call to *solve_board*
    if *solve_board* returns true:
      save move made in *moves* array
      increment *moves_size*
      return true
    else:
      reverse move
  else:
    continue

```
                ##############################################
sb_skip_cell:   addi $s2 $s2 1          # increment col
                j sb_col
sb_no:          li $v0 0                # set return to false
                b sb_epi
sb_yes:         li $v0 1                # set return to true
                # epilogue ###################################
sb_epi:         lw $ra 4($sp)
                lw $s0 8($sp)
                lw $s1 12($sp)
                lw $s2 16($sp)
                lw $s3 20($sp)
                lw $s4 24($sp)
                lw $s5 28($sp)
                addiu $sp $sp 28
                jr $ra
```

The *solve_board* function is a simple backtracking algorithm. It iterates over each cell in the board and checks each direction if a move is possible. If it is, it does the move and a recursive call to *solve_board*. It checks its return value. If it is true, it saves the move and return true also. Else, it reverses the move made and continue with the iteration over each cell. If there are no moves left, the function returns false to backtrack. Notice that with this algorithm, we save the moves in reverse order. This will have an effect to the *print_moves* function.

Printing the moves

```
# print_moves ##############################################
# t0 = start moves address      x,y->p,q
# t1 = end moves address

print_moves:    lb $t0 moves_size       # load moves_size
                sll $t0 $t0 2           # moves_size * 4
                la $t1 moves            # load moves address
                add $t0 $t0 $t1
pm_loop:        beq $t0 $t1 pm_end
                print_int_address(-4, $t0)      # print x
                print_char(',')
                print_int_address(-3, $t0)      # print y
                print_string_label(arrow)
                print_int_address(-2, $t0)      # print p
                print_char(',')
                print_int_address(-1, $t0)      # print q
                print_char('\n')
                subi $t0 $t0 4          # decrement move count
                b pm_loop
pm_end:         jr $ra
```

To print the moves taken, we should first take note that the *moves* array is the sequence of moves taken in reverse order. Thus, when printing, we start from the end of the array by adding the *moves_size* to the start address of our *moves* array. It then prints the moves accordingly line by line. Notice that it doesn't save registers to the stack. This is because this function does not use any preserved variables.